INTERNATIONAL TELECOMMUNICATION UNION

ITU-T X.680-X.693

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

(2002)

SERIES X: DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS

OSI networking and system aspects

Information Technology

X.680-X.683: Abstract Syntax Notation One (ASN.1)

X.690-X.693: ASN.1 encoding rules

CAUTION !

PREPUBLISHED RECOMMENDATION

This prepublication is an unedited version of a recently approved Recommendation. It will be replaced by the published version after editing. Therefore, there will be differences between this prepublication and the published version.

ITU-T X-SERIES RECOMMENDATIONS

DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS

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.399
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.899
OPEN DISTRIBUTED PROCESSING	X.900–X.999

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

CONTENTS

Information technology – Abstract syntax notation one (ASN.1):

ITU-T Rec. X.680 | ISO/IEC 8824-1: Specification of basic notation

ITU-T Rec. X.681 | ISO/IEC 8824-2: Information object specification

ITU-T Rec. X.682 | ISO/IEC 8824-3: Constraint specification

ITU-T Rec. X.683 | ISO/IEC 8824-4: Parameterization of ASN.1 specifications

Information technology – ASN.1 encoding rules:

ITU-T Rec. X.690 | ISO/IEC 8825-1: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)

ITU-T Rec. X.691 | ISO/IEC 8825-2: Specification of Packed Encoding Rules (PER)

ITU-T Rec. X.692 | ISO/IEC 8825-3: Specification of Encoding Control Notation (ECN)

ITU-T Rec. X.693 | ISO/IEC 8825-4: XML encoding rules

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU [had/had not] received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

ITU-T Recommendation X.680
International Standard 8824-1

Information Technology –
Abstract Syntax Notation One (ASN.1):
Specification of basic notation

INTERNATIONAL STANDARD 8824-1

ITU-T RECOMMENDATION X.680

Summary

This Recommendation | International Standard provides a notation called Abstract Syntax Notation One (ASN.1) for defining the syntax of information data. It defines a number of simple data types and specifies a notation for referencing these types and for specifying values of these types.

The ASN.1 notations can be applied whenever it is necessary to define the abstract syntax of information without constraining in any way how the information is encoded for transmission.

Source

The ITU-T Recommendation X.680 was approved on the 13th of July 2002. The identical text is also published as ISO/IEC International Standard 8824-1.

CONTENTS

		Page
Introduction		vi
1	Scope	1
2	Normative references	1
2.1	Identical Recommendations | International Standards	1
2.2	Additional references	2
3	Definitions	2
3.1	Information object specification	2
3.2	Constraint specification	2
3.3	Parameterization of ASN.1 specification	2
3.4	Structure for identification of organizations	3
3.5	Universal Multiple-Octet Coded Character Set (UCS)	3
3.6	Additional definitions	3
4	Abbreviations	7
5	Notation	8
5.1	General	8
5.2	Productions	8
5.3	The alternative collections	8
5.4	Non-spacing indicator	9
5.5	Example of a production	9
5.6	Layout	9
5.7	Recursion	9
5.8	References to permitted sequences of lexical items	9
5.9	References to a lexical item	9
5.10	Short-hand notations	9
5.11	Value references and the typing of values	10
6	The ASN.1 model of type extension	10
7	Extensibility requirements on encoding rules	11
8	Tags	11
9	Use of the ASN.1 notation	12
10	The ASN.1 character set	13
11	ASN.1 lexical items	14
11.1	General rules	14
11.2	Type references	14
11.3	Identifiers	15
11.4	Value references	15
11.5	Module references	15
11.6	Comments	15
11.7	Empty lexical item	15
11.8	Numbers	16
11.9	Real numbers	16
11.10	Binary strings	16
11.11	XML binary string item	16
11.12	Hexadecimal strings	16
11.13	XML hexadecimal string item	16
11.14	Character strings	17
11.15	XML character string item	17
11.16	Assignment lexical item	19
11.17	Range separator	19

11.18 Ellipsis 20
11.19 Left version brackets 20
11.20 Right version brackets 20
11.21 XML end tag start item 20
11.22 XML single tag end item 20
11.23 XML boolean true item 20
11.24 XML boolean false item 20
11.25 XML tag names for ASN.1 types 21
11.26 Single character lexical items 21
11.27 Reserved words 22
12 Module definition 23
13 Referencing type and value definitions 26
14 Notation to support references to ASN.1 components 28
15 Assigning types and values 29
16 Definition of types and values 30
17 Notation for the boolean type 33
18 Notation for the integer type 33
19 Notation for the enumerated type 34
20 Notation for the real type 35
21 Notation for the bitstring type 36
22 Notation for the octetstring type 38
23 Notation for the null type 39
24 Notation for sequence types 39
25 Notation for sequence-of types 42
26 Notation for set types 44
27 Notation for set-of types 45
28 Notation for choice types 46
29 Notation for selection types 48
30 Notation for tagged types 48
31 Notation for the object identifier type 49
32 Notation for the relative object identifier type 51
33 Notation for the embedded-pdv type 52
34 Notation for the external type 54
35 The character string types 55
36 Notation for character string types 55
37 Definition of restricted character string types 56
38 Naming characters and collections defined in ISO/IEC 10646-1 60
39 Canonical order of characters 63
40 Definition of unrestricted character string types 64
41 Notation for types defined in clauses 42 to 44 65
42 Generalized time 65
43 Universal time 66
44 The object descriptor type 67
45 Constrained Types 67
46 Element set specification 68
47 Subtype elements 70
47.1 General 70

47.2 Single Value ... 71
47.3 Contained Subtype ... 71
47.4 Value Range ... 72
47.5 Size Constraint ... 72
47.6 Type Constraint ... 73
47.7 Permitted Alphabet ... 73
47.8 Inner Subtyping ... 73
47.9 Pattern constraint ... 74
48 The extension marker ... 75
49 The exception identifier ... 77
Annex A ASN.1 regular expressions ... 78
A.1 Definition ... 78
A.2 Metacharacters ... 78
Annex B Rules for Type and Value Compatibility ... 81
B.1 The need for the value mapping concept (Tutorial introduction) ... 81
B.2 Value mappings ... 83
B.3 Identical type definitions ... 84
B.4 Specification of value mappings ... 86
B.5 Additional value mappings defined for the character string types ... 86
B.6 Specific type and value compatibility requirements ... 87
B.7 Examples ... 88
Annex C Assigned object identifier values ... 90
C.1 Object identifiers assigned in this Recommendation | International Standard ... 90
C.2 Object identifiers in the ASN.1 and encoding rules standards ... 90
Annex D Assignment of object identifier component values ... 92
D.1 Root assignment of object identifier component values ... 92
D.2 ITU-T assignment of object identifier component values ... 92
D.3 ISO assignment of object identifier component values ... 93
D.4 Joint assignment of object identifier component values ... 93
Annex E Examples and hints ... 94
E.1 Example of a personnel record ... 94
E.1.1 Informal description of Personnel Record ... 94
E.1.2 ASN.1 description of the record structure ... 94
E.2 Guidelines for use of the notation ... 95
E.2.1 Boolean ... 96
E.2.2 Integer ... 96
E.2.3 Enumerated ... 96
E.2.4 Real ... 97
E.2.5 Bit string ... 98
E.2.6 Octet string ... 99
E.2.7 UniversalString, BMPString and UTF8String ... 100
E.2.10 Sequence and sequence-of ... 101
E.2.11 Set and set-of ... 103
E.2.12 Tagged ... 105
E.2.13 Choice ... 106
E.2.14 Selection type ... 108
E.2.16 Embedded-pdv ... 109
E.2.17 External ... 109
E.2.18 Instance-of ... 109
E.2.19 Relative Object Identifier ... 110
E.3 Identifying abstract syntaxes ... 110
E.4 Subtypes ... 111
Annex F Tutorial annex on ASN.1 character strings ... 114
F.1 Character string support in ASN.1 ... 114
F.2 The UniversalString, UTF8String and BMPString types ... 114
F.3 On ISO/IEC 10646-1 conformance requirements ... 115

F.4 Recommendations for ASN.1 users on ISO/IEC 10646-1 conformance 115
F.5 Adopted subsets as parameters of the abstract syntax 116
F.6 The CHARACTER STRING type 116
Annex G Tutorial annex on the ASN.1 model of type extension 117
G.1 Overview 117
G.2 Meaning of version numbers 118
G.3 Requirements on encoding rules 119
G.4 Combination of (possibly extensible) constraints 119
G.4.1 Model 119
G.4.2 Serial application of constraints 119
G.4.3 Use of set arithmetic 120
G.4.4 Use of the Contained Subtype notation 121
Annex H Summary of the ASN.1 notation 122

Introduction

This Recommendation | International Standard presents a standard notation for the definition of data types and values. A *data type* (or *type* for short) is a category of information (for example, numeric, textual, still image or video information). A *data value* (or *value* for short) is an instance of such a type. This Recommendation | International Standard defines several basic types and their corresponding values, and rules for combining them into more complex types and values.

In some protocol architectures, each message is specified as the binary value of a sequence of octets. However, standards-writers need to define quite complex data types to carry their messages, without concern for their binary representation. In order to specify these data types, they require a notation that does not necessarily determine the representation of each value. ASN.1 is such a notation. This notation is supplemented by the specification of one or more algorithms called *encoding rules* that determine the value of the octets that carry the application semantics (called the *transfer syntax*). ITU-T Rec.X.690 | ISO/IEC 8825-1, ITU-T Rec. X.691 | ISO/IEC 8825-2 and ITU-T Rec. X.693 | ISO/IEC 8825-4 specify three families of standardized encoding rules, called *Basic Encoding Rules (BER), Packed Encoding Rules (PER),* and *XML Encoding Rules (XER).*

Some users wish to redefine their legacy protocols using ASN.1, but cannot use standardized encoding rules because they need to retain their existing binary representations. Other users wish to have more complete control over the exact layout of the bits on the wire (the transfer syntax). These requirements are addressed by ITU-T Rec. X.692 | ISO/IEC 8825-3 which specifies an *Encoding Control Notation (ECN)* for ASN.1. ECN enables designers to formally specify the abstract syntax of a protocol using ASN.1, but to then (if they so wish) take complete or partial control of the bits on the wire by writing an accompanying ECN specification (which may reference standardized Encoding Rules for some parts of the encoding).

A very general technique for defining a complicated type at the abstract level is to define a small number of *simple types* by defining all possible values of the simple types, then combining these simple types in various ways. Some of the ways of defining new types are as follows:

a) given an (ordered) list of existing types, a value can be formed as an (ordered) sequence of values, one from each of the existing types; the collection of all possible values obtained in this way is a new type (if the existing types in the list are all distinct, this mechanism can be extended to allow omission of some values from the list);

b) given an unordered set of (distinct) existing types, a value can be formed as an (unordered) set of values, one from each of the existing types; the collection of all possible unordered sets of values obtained in this way is a new type (the mechanism can again be extended to allow omission of some values);

c) given a single existing type, a value can be formed as an (ordered) list or (unordered) set of zero, one or more values of the existing type; the collection of all possible lists or sets of values obtained in this way is a new type;

d) given a list of (distinct) types, a value can be chosen from any one of them; the set of all possible values obtained in this way is a new type;

e) given a type, a new type can be formed as a subset of it by using some structure or order relationship among the values.

An important aspect of combining types in this way is that encoding rules should recognize the combining constructs, providing unambiguous encodings of the collection of values of the basic types. Thus, every basic type defined using the notation specified in this Recommendation | International Standard is assigned a *tag* to aid in the unambiguous encoding of values.

Tags are mainly intended for machine use, and are not essential for the human notation defined in this Recommendation | International Standard. Where, however, it is necessary to require that certain types be distinct, this is expressed by requiring that they have distinct tags. The allocation of tags is therefore an important part of the use of this notation, but (since 1994) it is possible to specify the automatic allocation of tags.

NOTE – Within this Recommendation | International Standard, tag values are assigned to all simple types and construction mechanisms. The restrictions placed on the use of the notation ensure that tags can be used in transfer for unambiguous identification of values.

An ASN.1 specification will initially be produced with a set of fully defined ASN.1 types. At a later stage, however, it may be necessary to change those types (usually by the addition of extra components in a sequence or set type). If this is to be possible in such a way that implementations using the old type definitions can interwork with implementations using the new type definitions in a defined way, encoding rules need to provide appropriate support. The ASN.1 notation supports the inclusion of an *extension marker* on a number of types. This signals to encoding rules the intention of the

designer that this type is one of a series of related types (i.e., versions of the same initial type) called an *extension series*, and that the encoding rules are required to enable information transfer between implementations using different types that are related by being part of the same extension series.

Clauses 10 to 31 (inclusive) define the simple types supported by ASN.1, and specify the notation to be used for referencing simple types and for defining new types using them. Clauses 10 to 31 also specify notations to be used for specifying values of types defined using ASN.1. Two value notations are provided. The first is called the basic ASN.1 value notation, and has been part of the ASN.1 notation since its first introduction. The second is called the XML ASN.1 Value Notation, and provides a value notation using Extensible Markup Language (XML).

NOTE – The XML Value Notation provides a means of representing ASN.1 values using XML. Thus, an ASN.1 type definition also specifies the structure and content of an XML element. This makes ASN.1 a simple schema language for XML.

Clauses 33 to 34 (inclusive) define the types supported by ASN.1 for carrying within them the complete encoding of ASN.1 types.

Clauses 35 to 40 (inclusive) define the character string types.

Clauses 41 to 44 (inclusive) define certain types which are considered to be of general utility, but which require no additional encoding rules.

Clauses 45 to 47 (inclusive) define a notation which enables subtypes to be defined from the values of a parent type.

Clause 48 defines a notation which allows ASN.1 types specified in a "version 1" specification to be identified as likely to be extended in "version 2", and for additions made in subsequent versions to be separately listed and identified with their version number.

Clause 49 defines a notation which allows ASN.1 type definitions to contain an indication of the intended error handling if encodings are received for values which lie outside those specified in the current standardized definition.

Annex A forms an integral part of this Recommendation | International Standard, and specifies ASN.1 regular expressions.

Annex B forms an integral part of this Recommendation | International Standard, and specifies rules for type and value compatibility.

Annex C forms an integral part of this Recommendation | International Standard, and records object identifier and object descriptor values assigned in the ASN.1 series of Recommendations | International Standards.

Annex D does not form an integral part of this Recommendation | International Standard, and describes the top-level arcs of the registration tree for object identifiers.

Annex E does not form an integral part of this Recommendation | International Standard, and provides examples and hints on the use of the ASN.1 notation.

Annex F does not form an integral part of this Recommendation | International Standard, and provides a tutorial on ASN.1 character strings.

Annex G does not form an integral part of this Recommendation | International Standard, and provides a tutorial on the ASN.1 model of type extension.

Annex H does not form an integral part of this Recommendation | International Standard, and provides a summary of ASN.1 using the notation of clause 5.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

Information Technology – Abstract Syntax Notation One (ASN.1): Specification Of Basic Notation

1 Scope

This Recommendation | International Standard provides a standard notation called Abstract Syntax Notation One (ASN.1) that is used for the definition of data types, values, and constraints on data types.

This Recommendation | International Standard

- defines a number of simple types, with their tags, and specifies a notation for referencing these types and for specifying values of these types;
- defines mechanisms for constructing new types from more basic types, and specifies a notation for defining such types and assigning them tags, and for specifying values of these types;
- defines character sets (by reference to other Recommendations and/or International Standards) for use within ASN.1.

The ASN.1 notation can be applied whenever it is necessary to define the abstract syntax of information.

The ASN.1 notation is referenced by other standards which define encoding rules for the ASN.1 types.

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 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.

2.1 Identical Recommendations | International Standards

- ITU-T Recommendation X.660 (1992) | ISO/IEC 9834-1:1993, *Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: General procedures (plus amendments).*
- ITU-T Recommendation X.681 (2002) | ISO/IEC 8824-2:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.*
- ITU-T Recommendation X.682 (2002) | ISO/IEC 8824-3:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*
- ITU-T Recommendation X.683 (2002) | ISO/IEC 8824-4:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications.*
- ITU-T Recommendation X.690 (2002) | ISO/IEC 8825-1:2002, *Information technology – ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER).*
- ITU-T Recommendation X.691 (2002) | ISO/IEC 8825-2:2002, *Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER).*
- ITU-T Recommendation X.692 (2002) | ISO/IEC 8825-3:2002, *Information technology – ASN.1 encoding rules: Encoding Control Notation (ECN) for ASN.1.*

– ITU-T Recommendation X.693 (2002) | ISO/IEC 8825-4:2002, *Information technology – ASN.1 encoding rules: Specification of XML Encoding Rules (XER).*

2.2 Additional references

– ITU-R Rec. TF.460-5 (1997), *Standard-frequency and time-signal emissions.*

– CCITT Recommendation T.61 (1988), *Character repertoire and coded character sets for the international teletex service*.

– CCITT Recommendation T.100 (1988), *International information exchange for interactive videotex.*

– ITU-T Recommendation T.101 (1994), *International interworking for videotex services.*

– ISO *International Register of Coded Character Sets to be used with Escape Sequences*.

– ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange.*

– ISO/IEC 2022:1994, *Information technology – Character code structure and extension techniques*.

– ISO 6523:1984, *Data interchange – Structures for the identification of organizations.*

– ISO/IEC 7350:1991, *Information technology – Registration of repertoires of graphic characters from ISO 10367*.

– ISO 8601:1988, *Data elements and interchange formats – Information interchange – Representation of dates and times*.

– ISO/IEC 10646-1:2000, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*.

– The Unicode Standard, Version 3.2.0:2002. The Unicode Consortium. (Reading, MA, Addison-Wesley)

NOTE – The above reference is included because it provides names for control characters.

– W3C XML 1.0:2000, *Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, Copyright © [6 October 2000] World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University), http://www.w3.org/TR/2000/REC-xml-20001006.*

NOTE – The reference to a document within this Recommendation | International Standard does not give it, as a stand-alone document, the status of a Recommendation or International Standard.

3 Definitions

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

3.1 Information object specification

This Recommendation | International Standard uses the following terms defined in ITU-T Rec. X.681 | ISO/IEC 8824-2:

a) information object;

b) information object class;

c) information object set;

d) instance-of type;

e) object class field type.

3.2 Constraint specification

This Recommendation | International Standard uses the following terms defined in ITU-T Rec. X.682 | ISO/IEC 8824-3:

a) component relation constraint;

b) table constraint.

3.3 Parameterization of ASN.1 specification

This Recommendation | International Standard uses the following terms defined in ITU-T Rec. X.683 | ISO/IEC 8824-4:

a) parameterized type;

b) parameterized value.

3.4 Structure for identification of organizations

This Recommendation | International Standard uses the following terms defined in ISO 6523:

a) issuing organization;
b) organization code;
c) International Code Designator.

3.5 Universal Multiple-Octet Coded Character Set (UCS)

This Recommendation | International Standard uses the following terms defined in ISO/IEC 10646-1:

a) Basic Multilingual Plane (BMP);
b) cell;
c) combining character;
d) graphic symbol;
e) group;
f) limited subset;
g) plane;
h) row;
i) selected subset.

3.6 Additional definitions

3.6.1 abstract character: An abstract value which is used for the organization, control or representation of textual data.

NOTE – Annex F provides a more complete description of the term abstract character.

3.6.2 abstract value: A value whose definition is based only on the type used to carry some semantics, independently of how it is represented in any encoding.

NOTE – Examples of abstract values are the values of the integer type, the boolean type, a character string type, or of a type which is a sequence (or a choice) of an integer and a boolean.

3.6.3 ASN.1 character set: The set of characters, specified in clause 10, used in the ASN.1 notation.

3.6.4 ASN.1 specification: A collection of one or more ASN.1 modules.

3.6.5 associated type: A type which is used only for defining the value and subtype notation for a type.

NOTE – Associated types are defined in this Recommendation | International Standard when it is necessary to make it clear that there may be a significant difference between how the type is defined in ASN.1 and how it is encoded. Associated types do not appear in user specifications.

3.6.6 bitstring type: A simple type whose distinguished values are an ordered sequence of zero, one or more bits.

NOTE – Where there is a need to carry embedded encodings of an abstract value, the use of a bitstring (or an octetstring) type without a contents constraint (see ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 11) is deprecated. Otherwise, the use of the embedded-pdv type (see clause 33) provides a more flexible mechanism, allowing the announcement of the abstract syntax and of the encoding of the abstract value that is embedded.

3.6.7 boolean type: A simple type with two distinguished values.

3.6.8 character property: The set of information associated with a cell in a table defining a character repertoire.

NOTE – The information will normally include some or all of the following items:

a) a graphic symbol;
b) a character name;
c) the definition of functions associated with the character when used in particular environments;
d) whether it represents a digit;
e) an associated character differing only in (upper/lower) case.

3.6.9 character abstract syntax: Any abstract syntax whose values are specified as the set of character strings of zero, one or more characters from some specified collection of characters.

3.6.10 character repertoire: The characters in a character set without any implication on how such characters are encoded.

3.6.11 character string types: Simple types whose values are strings of characters from some defined character set.

3.6.12 character transfer syntax: Any transfer syntax for a character abstract syntax.

NOTE – ASN.1 does not support character transfer syntaxes which do not encode all character strings as an integral multiple of 8 bits.

3.6.13 choice types: Types defined by referencing a list of distinct types; each value of the choice type is derived from the value of one of the component types.

3.6.14 component type: One of the types referenced when defining a **CHOICE**, **SET**, **SEQUENCE**, **SET OF**, or **SEQUENCE OF**.

3.6.15 constraint: A notation which can be used in association with a type, to define a subtype of that type.

3.6.16 contents constraint: A constraint on a bit string or octet string type that specifies either that the contents are to be an encoding of a specified ASN.1 type, or that specified procedures are to be used to produce and process the contents.

3.6.17 control characters: Characters appearing in some character repertoires that have been given a name (and perhaps a defined function in relation to certain environments) but which have not been assigned a graphic symbol, and which are not spacing characters.

NOTE – HORIZONTAL TABULATION (9) and LINE FEED (10) are examples of control characters that have been assigned a formatting function in a printing environment. DATA LINK ESCAPE (16) is an example of a control character that has been assigned a function in a communication environment.

3.6.18 Coordinated Universal Time (UTC): The time scale maintained by the Bureau International de l'Heure (International Time Bureau) that forms the basis of a coordinated dissemination of standard frequencies and time signals.

NOTE 1 – The source of this definition is ITU-R Rec. TF.460-5. ITU-R has also defined the acronym for Coordinated Universal Time as UTC.

NOTE 2 – UTC and Greenwich Mean Time (GMT) are two alternative time standards which for most practical purposes determine the same time.

3.6.19 element: A value of a governing type or an information object of a governing information object class, distinguable from all other values of the same type or information objects of the same class, respectively.

3.6.20 element set: A set of elements, all of which are values of a governing type, or information objects of a governing class.

NOTE – Governing class is defined in ITU-T Rec. X.681 | ISO/IEC 8824-2, 3.4.7.

3.6.21 embedded-pdv type: A type whose set of values is formally the union of the sets of values in all possible abstract syntaxes. This type can be used in an ASN.1 specification that wishes to carry in its protocol an abstract value whose type may be defined externally to that ASN.1 specification. It carries an identification of the abstract syntax (the type) of the abstract value being carried, as well as an identification of the encoding rules used to encode that abstract value.

3.6.22 encoding: The bit-pattern resulting from the application of a set of encoding rules to an abstract value.

3.6.23 (ASN.1) encoding rules: Rules which specify the representation during transfer of the values of ASN.1 types. Encoding rules also enable the values to be recovered from the representation, given knowledge of the type.

NOTE – For the purpose of specifying encoding rules, the various referenced type (and value) notations, which can provide alternative notations for built-in types (and values), are not relevant.

3.6.24 enumerated types: Simple types whose values are given distinct identifiers as part of the type notation.

3.6.25 extension addition: One of the added notations in an extension series. For set, sequence and choice types, each extension addition is the addition of either a single extension addition group or a single component type. For enumerated types it is the addition of a single further enumeration. For a constraint it is the addition of (only) one subtype element.

NOTE – Extension additions are both textually ordered (following the extension marker) and logically ordered (having increasing enumeration values, and, in the case of **CHOICE** alternatives, increasing tags).

3.6.26 extension addition group: One or more components of a set, sequence or choice type grouped within version brackets. An extension addition group is used to clearly identify the components of a set, sequence or choice type that were added in a particular version of an ASN.1 module, and can identify that version with a simple integer.

3.6.27 extension addition type: A type contained within an extension addition group or a single component type that is itself an extension addition (in such a case it is not contained within an extension addition group).

3.6.28 extensible constraint: A subtype constraint with an extension marker at the outer level, or that is extensible through the use of set arithmetic with extensible sets of values.

3.6.29 extension insertion point (or insertion point): The location within a type definition where extension additions are inserted. This location is the end of the type notation of the immediately preceding type in the extension series if there is a single ellipsis in the type definition, or immediately before the second ellipsis if there is an extension marker pair in the definition of the type.

NOTE – There can be at most one insertion point within the components of any choice, sequence, or set type.

3.6.30 extension marker: A syntactic flag (an ellipsis) that is included in all types that form part of an extension series.

3.6.31 extension marker pair: A pair of extension markers between which extension additions are inserted.

3.6.32 extension-related: Two types that have the same extension root, where one was created by adding zero or more extension additions to the other.

3.6.33 extension root: An extensible type that is the first type in an extension series. It carries either the extension marker with no additional notation other than comments and white-space between the extension marker and the matching "}" or ")", or an extension marker pair with no additional notation other than a single comma, comments and white-space between the extension markers.

NOTE – Only an extension root can be the first type in an extension series.

3.6.34 extension series: A series of ASN.1 types which can be ordered in such a way that each successive type in the series is formed by the addition of text at the extension insertion point.

3.6.35 extensible type: A type with an extension marker, or to which an extensible constraint has been applied.

3.6.36 external reference: A type reference, value reference, information object class reference, information object reference, or information object set reference (which may be parameterized), that is defined in some other module than the one in which it is being referenced, and which is being referred to by prefixing the module name to the referenced item.

EXAMPLE – **`ModuleName.TypeReference`**

3.6.37 external type: A type which is a part of an ASN.1 specification that carries a value whose type may be defined externally to that ASN.1 specification. It also carries an identification of the type of the value being carried.

3.6.38 false: One of the distinguished values of the boolean type (see also "true").

3.6.39 governing (type); governor: A type definition or reference which affects the interpretation of a part of the ASN.1 syntax, requiring that part of the ASN.1 syntax to reference values in the governing type.

3.6.40 identical type definitions: Two instances of the ASN.1 "Type" production (see clause 16) are defined as identical type definitions if, after performing the transformations specified in Annex B, they are identical ordered lists of identical lexical items (see clause 11).

3.6.41 integer type: A simple type with distinguished values which are the positive and negative whole numbers, including zero (as a single value).

NOTE – When particular encoding rules limit the range of an integer, such limitations are chosen so as not to affect any user of ASN.1.

3.6.42 lexical item: A named sequence of characters from the ASN.1 character set, specified in clause 11, which is used in forming the ASN.1 notation.

3.6.43 module: One or more instances of the use of the ASN.1 notation for type, value, value set, information object class, information object, and information object set (as well as the parameterized variant of those), encapsulated using the ASN.1 module notation (see clause 12).

NOTE – The terms information object class (etc) are specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, and parameterization is specified in ITU-T X.683 | ISO/IEC 8824-4.

3.6.44 null type: A simple type consisting of a single value, also called null.

3.6.45 object: A well-defined piece of information, definition, or specification which requires a name in order to identify its use in an instance of communication.

NOTE – Such an object may be an information object as defined in ITU-T Rec. X.681 | ISO/IEC 8824-2.

3.6.46 object descriptor type: A type whose distinguished values are human-readable text providing a brief description of an object (see 3.6.45).

NOTE – An object descriptor value is usually associated with a single object. Only an object identifier value unambiguously identifies an object.

3.6.47 object identifier: A globally unique value associated with an object to unambiguously identify it.

3.6.48 object identifier type: A simple type whose values are the set of all object identifiers allocated in accordance with the rules of ITU-T Rec. X.660 | ISO/IEC 9834 series.

NOTE – The rules of ITU-T Rec. X.660 | ISO/IEC 9834-1 permit a wide range of authorities to independently associate object identifiers with objects.

3.6.49 octetstring type: A simple type whose distinguished values are an ordered sequence of zero, one or more octets, each octet being an ordered sequence of eight bits.

3.6.50 open systems interconnection: An architecture for computer communication which provides a number of terms which are used in this Recommendation | International Standard preceded by the abbreviation "OSI".

NOTE – The meaning of such terms can be obtained from the ITU-T Rec. X.200 series and equivalent ISO/IEC Standards if needed. The terms are only applicable if ASN.1 is used in an OSI environment.

3.6.51 open type notation: An ASN.1 notation used to denote a set of values from more than one ASN.1 type.

NOTE 1 – The term "open type" is used synonymously with "open type notation" in the body of this Recommendation | International Standard.

NOTE 2 – All ASN.1 encoding rules provide unambiguous encodings for the values of a single ASN.1 type. They do not necessarily provide unambiguous encodings for "open type notation", which carries values from ASN.1 types that are not normally determined at specification time. Knowledge of the type of the value being encoded in the "open type notation" is needed before the abstract value for that field can be unambiguously determined.

NOTE 3 – The only notation in this Recommendation | International Standard which is an open type notation is the "ObjectClassFieldType" specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, clause 14, where the "FieldName" denotes either a type field or a variable-type value field.

3.6.52 parent type (of a subtype): The type that is being constrained when defining a subtype, and which governs the subtype notation.

NOTE – The parent type may itself be a subtype of some other type.

3.6.53 production: A part of the formal notation (also called grammar or Backus-Naur Form, BNF) used to specify ASN.1.

3.6.54 real type: A simple type whose distinguished values (specified in clause 20) are members of the set of real numbers.

3.6.55 recursive definition (of a type): A set of ASN.1 definitions which cannot be reordered so that all types used in a construction are defined before the definition of the construction.

NOTE – Recursive definitions are allowed in ASN.1: the user of the notation has the responsibility for ensuring that those values (of the resulting types) which are used have a finite representation and that the value set associated with the type contains at least one value.

3.6.56 relative object identifier: A value which identifies an object by its position relative to some known object identifier (see 3.6.47).

3.6.57 relative object identifier type: A simple type whose values are the set of all possible relative object identifiers.

3.6.58 restricted character string type: A character string type whose characters are taken from a fixed character repertoire identified in the type specification.

3.6.59 selection types: Types defined by reference to a component type of a choice type, and whose values are precisely the values of that component type.

3.6.60 sequence types: Types defined by referencing a fixed, ordered list of types (some of which may be declared to be optional); each value of the sequence type is an ordered list of values, one from each component type.

NOTE – Where a component type is declared to be optional, a value of the sequence type need not contain a value of that component type.

3.6.61 sequence-of types: Types defined by referencing a single component type; each value in the sequence-of type is an ordered list of zero, one or more values of the component type.

3.6.62 serial application (of constraints): The application of a constraint to a parent type which is already constrained.

3.6.63 set arithmetic: The formation of new sets of values or information objects using the operations of union, intersection and set difference (use of **EXCEPT**) as specified in 46.2.

NOTE – The result of serial application of constraints is not covered by the term "set arithmetic".

3.6.64 set types: Types defined by referencing a fixed, unordered, list of types (some of which may be declared to be optional); each value in the set type is an unordered list of values, one from each component type.

NOTE – Where a component type is declared to be optional, a value of the set type need not contain a value of that component type.

3.6.65 set-of types: Types defined by referencing a single component type; each value in the set-of type is an unordered list of zero, one or more values of the component type.

3.6.66 simple types: Types defined by directly specifying the set of their values.

3.6.67 spacing character: A character in a character repertoire which is intended for inclusion with graphic characters in the printing of a character string but which is represented in the physical rendition by empty space; it is not normally considered to be a control character (see 3.6.17).

NOTE – There may be a single spacing character in the character repertoire, or there may be multiple spacing characters with varying widths.

3.6.68 subtype (of a parent type): A type whose values are a subset (or the complete set) of the values of some other type (the parent type).

3.6.69 tag: A type denotation which is associated with every ASN.1 type.

3.6.70 tagged types: A type defined by referencing a single existing type and a tag; the new type is isomorphic to the existing type, but is distinct from it.

3.6.71 tagging: Replacing the existing (possibly the default) tag of a type by a specified tag.

3.6.72 transfer syntax: The set of bit strings used to exchange the abstract values in an abstract syntax, usually obtained by application of encoding rules to an abstract syntax.

NOTE – The term "transfer syntax" is synonymous with "encoding".

3.6.73 true: One of the distinguished values of the boolean type (see also "false").

3.6.74 type: A named set of values.

3.6.75 type reference name: A name associated uniquely with a type within some context.

NOTE – Reference names are assigned to the types defined in this Recommendation | International Standard; these are universally available within ASN.1. Other reference names are defined in other Recommendations | International Standards, and are applicable only in the context of that Recommendation | International Standard.

3.6.76 unrestricted character string type: A type whose abstract values are values from a character abstract syntax, together with an identification of the character abstract syntax and of the character transfer syntax to be used in its encoding.

3.6.77 user (of ASN.1): The individual or organization that defines the abstract syntax of a particular piece of information using ASN.1.

3.6.78 value mapping: A 1-1 relationship between values in two types that enables a reference to one of those values to be used as a reference to the other value. This can, for example, be used in specifying subtypes and default values (see Annex B).

3.6.79 value reference name: A name associated uniquely with a value within some context.

3.6.80 value set: A collection of values of a type. Semantically equivalent to a subtype.

3.6.81 version brackets: A pair of adjacent left and right brackets ("**[[**" or "**]]**") used to delineate the start and end of an extension addition group. The pair of left brackets can optionally be followed by a number giving a version number for the extension addition group.

3.8.82 version number: A number which can be associated with a version bracket (see G.1.8).

NOTE – A version number cannot be added to an extension addition which is not part of an extension addition group, nor to extension additions to any type other than choice, sequence, or set.

3.6.83 white-space: Any formatting action that yields a space on a printed page, such as spaces or tabs.

4 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules of ASN.1
BMP	Basic Multilingual Plane
DCC	Data Country Code
DNIC	Data Network Identification Code
ECN	Encoding Control Notation of ASN.1
ICD	International Code Designator

IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union – Telecommunication Standardization Sector
OID	Object Identifier
OSI	Open Systems Interconnection
PER	Packed Encoding Rules of ASN.1
ROA	Recognized Operating Agency
UCS	Universal Multiple-Octet Coded Character Set
UTC	Coordinated Universal Time
XML	Extensible Markup Language

5 Notation

5.1 General

5.1.1 The ASN.1 notation consists of a sequence of characters from the ASN.1 character set specified in clause 10.

5.1.2 Each use of the ASN.1 notation contains characters from the ASN.1 character set grouped into lexical items. Clause 11 specifies all the sequences of characters forming lexical items, and names each item.

5.1.3 The ASN.1 notation is specified in clause 12 (and following clauses) by specifying and naming those sequences of lexical items which form valid instances of the ASN.1 notation, and by specifying the ASN.1 semantics of each sequence.

5.1.4 In order to specify the permitted sequences of lexical items, this Recommendation | International Standard uses a formal notation defined in the following subclauses.

5.2 Productions

5.2.1 All lexical items are named (see clause 11), and permitted sequences of lexical items are named.

5.2.2 A new (more complex) permitted sequence of lexical items is defined by means of a production. This uses the names of lexical items and of permitted sequences of lexical items and forms a new named permitted sequence of lexical items.

5.2.3 Each production consists of the following parts, on one or several lines, in order:

a) a name for the new permitted sequence of lexical items;

b) the characters

::=

c) one or more alternative sequences of lexical items, as defined in 5.3, separated by the character

|

5.2.4 A sequence of lexical items is present in the new permitted sequence of lexical items if it is present in one or more of the alternatives. The new permitted sequence of lexical items is referenced in this Recommendation | International Standard by the name in a) above.

NOTE – If the same sequence of lexical items appears in more than one alternative, any semantic ambiguity in the resulting notation is resolved by associated text.

5.3 The alternative collections

5.3.1 Each alternative in a production (see 5.2.c) is specified by a list of names. Each name is either the name of a lexical item, or is the name of a permitted sequence of lexical items defined and named by some other production.

5.3.2 The permitted sequence of lexical items defined by each alternative consists of all sequences obtained by taking any one of the sequences (or the lexical item) associated with the first name, in combination with (and followed by) any one of the sequences (or lexical item) associated with the second name, in combination with (and followed by) any one of the sequences (or lexical item) associated with the third name, and so on up to and including the last name (or lexical item) in the alternative.

5.4 Non-spacing indicator

If the non-spacing indicator "**&**" (AMPERSAND) is inserted between these items in production sequences, then the lexical item that precedes it and the lexical item that follows it shall not be separated by white-space.

NOTE – This indicator is only used in productions that describe the XML value notation. For example, it is used to specify that the lexical item "<" is to be immediately followed by an XML tag name.

5.5 Example of a production

5.5.1 The production:

```
ExampleProduction    ::=
        bstring                 |
        hstring                 |
        "{" IdentifierList "}"
```

associates the name "ExampleProduction" with the following sequences of lexical items:

a) any "bstring" (a lexical item); or

b) any "hstring" (a lexical item); or

c) any sequence of lexical items associated with "IdentifierList", preceded by a "**{**" and followed by a "**}**".

NOTE – "**{**" and "**}**" are the names of lexical items containing the single characters **{** and **}** (see 11.26).

5.5.2 In this example, "IdentifierList" would be defined by a further production, either before or after the production defining "ExampleProduction".

5.6 Layout

Each production used in this Recommendation | International Standard is preceded and followed by an empty line. Empty lines do not appear within productions. The production may be on a single line, or may be spread over several lines. Layout is not significant.

5.7 Recursion

The productions in this Recommendation | International Standard are frequently recursive. In this case the productions are to be continuously reapplied until no new sequences are generated.

NOTE – In many cases, such reapplication results in an infinite set of permitted sequences of lexical items. Some or all of the sequences in the set may themselves contain an unbounded number of lexical items. This is not an error.

5.8 References to permitted sequences of lexical items

This Recommendation | International Standard references a permitted sequence of lexical items (part of the ASN.1 notation) by referencing the name that appears before the "::=" in a production; the name is surrounded by the QUOTATION MARK (34) character (") to distinguish it from natural language text, unless it appears as part of a production.

5.9 References to a lexical item

This Recommendation | International Standard references a lexical item by using the name of the lexical item; when the name appears in natural language text, and could be confused with such text, then it is surrounded by the QUOTATION MARK (34) character (").

5.10 Short-hand notations

In order to make productions more concise and more readable, the following short-hand notations are used in the definition of permitted sequences of lexical items in this Recommendation | International Standard and also in ITU-T Rec. X.681 | ISO/IEC 8824-2, ITU-T Rec. X.682 | ISO/IEC 8824-3 and ITU-T Rec. X.683 | ISO/IEC 8824-4:

a) An asterisk (*) following two names, "A" and "B", denotes the "empty" lexical item (see 11.7), or one of the permitted sequences of lexical items associated with "A", or an alternating series of one of the sequences of lexical items associated with "A" and one of the sequences of lexical items associated with "B", both starting and finishing with one associated with "A". Thus:

C ::= A B *

is equivalent to:

C ::= D | empty
D ::= A | A B D

"D" being an auxiliary name not appearing elsewhere in the productions.

EXAMPLE – "C ::= A B *" is the shorthand notation for the following alternatives of C:

empty
A
A B A
A B A B A
A B A B A B A
...

b) A plus sign(+) is similar to the asterisk in a), except that the "empty" lexical item is excluded. Thus:

E ::= A B +

is equivalent to:

E ::= A | A B E

EXAMPLE – "E ::= A B +" is the shorthand notation for the following alternatives of E:

A
A B A
A B A B A
A B A B A B A
...

c) A question mark (?) following a name denotes either the "empty" lexical item (see 11.7) or a permitted sequence of lexical items associated with "A". Thus:

F ::= A ?

is equivalent to:

F ::= empty | A

NOTE – These short-hand notations take precedence over the juxtaposition of lexical items in production sequences (see 5.2.2).

5.11 Value references and the typing of values

5.11.1 The ASN.1 value assignment notation enables a name to be given to a value of a specified type. This name can be used wherever a reference to that value is needed. Annex B describes and specifies the value mapping mechanism that allows a value reference name for a value of one type to identify a value of a second (similar) type. Thus, a reference to the first value can be used wherever a reference to a value in the second type is required.

5.11.2 In the body of the ASN.1 standards normal English text is used to specify legality (or otherwise) of constructs where more than one type is involved. These legality specification generally require that two or more types be "compatible". For example, the type used in defining a value reference is required to be "compatible with" the governing type when the value reference is used. The normative Annex B uses the value mapping concept to give a precise statement about whether any given ASN.1 construct is legal or not.

6 The ASN.1 model of type extension

When decoding an extensible type, a decoder may detect:

a) the absence of expected extension additions in a sequence or set type; or

b) the presence of arbitrary unexpected extension additions above those defined (if any) in a sequence or set type, or of an unknown alternative in a choice type, or an unknown enumeration in an enumerated type, or of an unexpected length or value of a type whose constraint is extensible.

In formal terms, an abstract syntax defined by the extensible type **X** contains not only the values of type **X**, but also the values of all types that are extension-related to **X**. Thus, the decoding process never signals an error when either of the above situations (a or b) is detected. The action that is taken in each situation is determined by the ASN.1 specifier.

NOTE – Frequently the action will be to ignore the presence of unexpected additional extensions, and to use a default value or a "missing" indicator for expected extension additions that are absent.

Unexpected extension additions detected by a decoder in an extensible type can later be included in a subsequent encoding of that type (for transmission back to the sender, or to some third party), provided that the same transfer syntax is used on the subsequent transmission.

7 Extensibility requirements on encoding rules

NOTE – These requirements apply to standardized encoding rules. They do not apply to encoding rules defined using ECN (see ITU-T Rec. X.692 | ISO/IEC 8825-3).

7.1 All ASN.1 encoding rules shall allow the encoding of values of an extensible type **X** in such a way that they can be decoded using an extensible type **Y** that is extension-related to **X**. Further, the encoding rules shall allow the values that were decoded using **Y** to be re-encoded (using **Y**) and decoded using a third extensible type **Z** that is extension related to **Y** (and hence **X** also).

NOTE – Types **X**, **Y** and **Z** may appear in any order in the extension series.

If a value of an extensible type **X** is encoded and then relayed (directly or through a relaying application using extension-related type **Z**) to another application that decodes the value using extensible type **Y** that is extension-related to **X**, then the decoder using type **Y** obtains an abstract value composed of:

a) an abstract value of the extension root type;

b) an abstract value of each extension addition that is present in both **X** and **Y**;

c) delimited encoding for each extension addition (if any) that is in **X** but not in **Y**.

The encodings in c) shall be capable of being included in a later encoding of a value of **Y**, if so required by the application. That encoding shall be a valid encoding of a value of **X**.

Tutorial example: If system A is using an extensible root type (type **X**) that is a sequence type or a set type with an extension addition of an optional integer type, while system B is using an extension-related type (type **Y**) that has two extension additions where each is an optional integer type, then transmission by B of a value of **Y** which omits the integer value of the first extension addition and includes the second must not be confused by A with the presence of the first (only) extension addition of **X** that it knows about. Moreover, A must be able to re-encode the value of **X** with a value present for the first integer type, followed by the second integer value received from B, if so required by the application protocol.

7.2 All ASN.1 encoding rules shall specify the encoding and decoding of the value of an enumerated type and a choice type in such a way that if a transmitted value is in the set of extension additions held in common by the encoder and the decoder, then it is successfully decoded, otherwise it shall be possible for the decoder to delimit the encoding of it and to identify it as a value of an (unknown) extension addition.

7.3 All ASN.1 encoding rules shall specify the encoding and decoding of types with extensible constraints in such a way that if a transmitted value is in the set of extension additions held in common by the encoder and the decoder, then it is successfully decoded, otherwise it shall be possible for the decoder to delimit the encoding of and to identify it as a value of an (unknown) extension addition.

In all cases, the presence of extension additions shall not affect the ability to recognize later material when a type with an extension marker is nested inside some other type.

NOTE 1 – All variants of the Basic Encoding Rules of ASN.1 and the Packed Encoding Rules of ASN.1 satisfy all these requirements. Encoding rules defined using ECN do not necessarily satisfy all these requirements, but may do so.

NOTE 2 – PER and BER do not identify the version number in the encoding of an extension addition. Encodings specified using ECN may or may not provide such identification.

8 Tags

8.1 A tag is specified by giving a class and a number within the class. The class is one of:

– universal;

– application;

– private;

– context-specific.

8.2 The number is a non-negative integer, specified in decimal notation.

8.3 Restrictions on tags assigned by the user of ASN.1 are specified in clause 30.

NOTE – Clause 30 includes the restriction that users of this notation are not allowed to explicitly specify universal class tags in their ASN.1 specifications. There is no formal difference between use of tags from the other three classes. Where application class tags are employed, a private or context-specific class tag could generally be applied instead, as a matter of user choice and style. The presence of the three classes is largely for historical reasons, but guidance is given in E.2.12 on the way in which the classes are usually employed.

8.4 Table 1 summarizes the assignment of tags in the universal class which are specified in this Recommendation | International Standard.

Table 1 – Universal class tag assignments

UNIVERSAL 0	Reserved for use by the encoding rules
UNIVERSAL 1	Boolean type
UNIVERSAL 2	Integer type
UNIVERSAL 3	Bitstring type
UNIVERSAL 4	Octetstring type
UNIVERSAL 5	Null type
UNIVERSAL 6	Object identifier type
UNIVERSAL 7	Object descriptor type
UNIVERSAL 8	External type and Instance-of type
UNIVERSAL 9	Real type
UNIVERSAL 10	Enumerated type
UNIVERSAL 11	Embedded-pdv type
UNIVERSAL 12	UTF8String type
UNIVERSAL 13	Relative object identifier type
UNIVERSAL 14-15	Reserved for future editions of this Recommendation | International Standard
UNIVERSAL 16	Sequence and Sequence-of types
UNIVERSAL 17	Set and Set-of types
UNIVERSAL 18-22, 25-30	Character string types
UNIVERSAL 23-24	Time types
UNIVERSAL 31-...	Reserved for addenda to this Recommendation | International Standard

8.5 Some encoding rules require a canonical order for tags. To provide uniformity, a canonical order for tags is defined in 8.6.

8.6 The canonical order for tags is based on the outermost tag of each type and is defined as follows:

a) those elements or alternatives with universal class tags shall appear first, followed by those with application class tags, followed by those with context-specific tags, followed by those with private class tags;

b) within each class of tags, the elements or alternatives shall appear in ascending order of their tag numbers.

9 Use of the ASN.1 notation

9.1 The ASN.1 notation for a type definition shall be "Type" (see 16.1).

9.2 The ASN.1 notation for a value of a type shall be "Value" (see 16.7).

NOTE – It is not in general possible to interpret the value notation without knowledge of the type.

9.3 The ASN.1 notation for assigning a type to a type reference name shall be either "TypeAssignment" (see 15.1), "ValueSetTypeAssignment" (see 15.6), "ParameterizedTypeAssignment" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.2), or "ParameterizedValueSetTypeAssignment" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.2).

9.4 The ASN.1 notation for assigning a value to a value reference name shall be either "ValueAssignment" (see 15.2) or "ParameterizedValueAssignment" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.2).

9.5 The production alternatives of the notation "Assignment" shall only be used within the notation "ModuleDefinition" (except as specified in NOTE 2 of 12.1).

10 The ASN.1 character set

10.1 A lexical item shall consist of a sequence of the characters listed in Table 2 except as specified in 10.2 and 10.3. In Table 2, characters are identified by the names they are given in ISO/IEC 10646-1.

Table 2 – ASN.1 characters

A to Z	(LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z)
a to z	(LATIN SMALL LETTER A to LATIN SMALL LETTER Z)
0 to 9	(DIGIT ZERO to DIGIT 9)
!	(EXCLAMATION MARK)
"	(QUOTATION MARK)
&	(AMPERSAND)
'	(APOSTROPHE)
(	(LEFT PARENTHESIS)
)	(RIGHT PARENTHESIS)
*****	(ASTERISK)
,	(COMMA)
-	(HYPHEN-MINUS)**.**(FULL STOP)
/	(SOLIDUS)
:	(COLON)**;**(SEMICOLON)
<	(LESS-THAN SIGN)
=	(EQUALS SIGN)
>	(GREATER-THAN SIGN)
@	(COMMERCIAL AT)
[	(LEFT SQUARE BRACKET)
]	(RIGHT SQUARE BRACKET)
^	(CIRCUMFLEX ACCENT)
_	(LOW LINE)
{	(LEFT CURLY BRACKET)
|	(VERTICAL LINE)
}	(RIGHT CURLY BRACKET)

NOTE – Where equivalent derivative standards are developed by national standards bodies, additional characters may appear in the following lexical items:

- typereference (see 11.2);
- identifier (see 11.3);
- valuereference (see 11.4);
- modulereference (see 11.5).

When additional characters are introduced to accommodate a language in which the distinction between upper-case and lower-case letters is without meaning, the syntactic distinction achieved by dictating the case of the first character of certain of the above lexical items has to be achieved in some other way. This is to allow valid ASN.1 specifications to be written in various languages.

10.2 Where the notation is used to specify the value of a character string type, all characters for the defined character set can appear in the ASN.1 notation, surrounded by the QUOTATION MARK (34) characters (") (see 11.14).

10.3 Additional (arbitrary) graphic symbols may appear in the "comment" lexical item (see 11.6).

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

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

10.6 ASN.1 definitions can also contain white-space characters (see 11.1.6) between lexical items.

11 ASN.1 lexical items

11.1 General rules

11.1.1 The following subclauses specify the characters in lexical items. In each case the name of the lexical item is given, together with the definition of the character sequences which form the lexical item.

11.1.2 The lexical items specified in the subclauses of this clause 11 (except multiple-line "comment", "bstring", "hstring" and "cstring") shall not contain white-space (see 11.6, 11.10, 11.12 and 11.14).

11.1.3 The length of a line is not restricted.

11.1.4 Lexical items may be separated by one or more occurrences of white-space (see 11.1.6) or comments (see 11.6) except when the non-spacing indicator "**&**" (see 5.4) is used. Within an "XMLTypedValue" production (see 15.2), white-space may appear between lexical items, but the "comment" lexical item shall not be present.

NOTE – This is to avoid ambiguity resulting from the presence of adjacent hyphens or asterisk and solidus within an "xmlcstring" lexical item. Such characters never indicate the start of a "comment" lexical item when they appear within an "XMLTypedValue" production.

11.1.5 A lexical item shall be separated from a following lexical item by one or more instances of white-space or comment if the initial character (or characters) of the following lexical item is a permitted character (or characters) for inclusion at the end of the characters in the earlier lexical item.

11.1.6 This Recommendation | International Standard uses the terms "newline", and "white-space". In representing white-space and newline (end of line) in machine-readable specifications, any one or more of the following characters may be used in any combination (for each character, the character name and character code specified in The Unicode Standard are given):

For white-space:

HORIZONTAL TABULATION (9)

LINE FEED (10)

VERTICAL TABULATION (11)

FORM FEED (12)

CARRIAGE RETURN (13)

SPACE (32)

For newline:

LINE FEED (10)

VERTICAL TABULATION (11)

FORM FEED (12)

CARRIAGE RETURN (13)

NOTE – Any character or character sequence that is a valid newline is also a valid white-space.

11.2 Type references

Name of lexical item – typereference

11.2.1 A "typereference" shall consist of an arbitrary number (one or more) of letters, digits, and hyphens. The initial character shall be an upper-case letter. A hyphen shall not be the last character. A hyphen shall not be immediately followed by another hyphen.

NOTE – The rules concerning hyphen are designed to avoid ambiguity with (possibly following) comment.

11.2.2 A "typereference" shall not be one of the reserved character sequences listed in 11.27.

11.3 Identifiers

Name of lexical item – identifier

An "identifier" shall consist of an arbitrary number (one or more) of letters, digits, and hyphens. The initial character shall be a lower-case letter. A hyphen shall not be the last character. A hyphen shall not be immediately followed by another hyphen.

NOTE – The rules concerning hyphen are designed to avoid ambiguity with (possibly following) comment.

11.4 Value references

Name of lexical item – valuereference

A "valuereference" shall consist of the sequence of characters specified for an "identifier" in 11.3. In analysing an instance of use of this notation, a "valuereference" is distinguished from an "identifier" by the context in which it appears.

11.5 Module references

Name of lexical item – modulereference

A "modulereference" shall consist of the sequence of characters specified for a "typereference" in 11.2. In analysing an instance of use of this notation, a "modulereference" is distinguished from a "typereference" by the context in which it appears.

11.6 Comments

Name of lexical item – comment

11.6.1 A "comment" is not referenced in the definition of the ASN.1 notation. It may, however, appear at any time between other lexical items, and has no syntactic significance.

NOTE – Nonetheless, in the context of a Recommendation | International Standard that uses ASN.1, an ASN.1 comment may contain normative text related to the application semantics, or constraints on the syntax.

11.6.2 The lexical item "comment" can have two forms:

a) One-line comments which begin with "**--**" as defined in 11.6.3;

b) Multiple-line comments which begin with "**/*** " as defined in 11.6.4.

11.6.3 Whenever a "comment" begins with a pair of adjacent hyphens, it shall end with the next pair of adjacent hyphens or at the end of the line, whichever occurs first. A comment shall not contain a pair of adjacent hyphens other than the pair which starts it and the pair, if any, which ends it. If a comment beginning with "**--**" includes the adjacent characters "**/***" or "***/**", these have no special meaning and are considered part of the comment. The comment may include graphic symbols which are not in the character set specified in 10.1 (see 10.3).

11.6.4 Whenever a "comment" begins with "**/***", it shall end with a corresponding "***/**", whether this "***/**" is on the same line or not. If another "**/***" is found before a "***/**", then the comment terminates when a matching "***/**" has been found for each "**/***". If a comment beginning with "**/***" includes two adjacent hyphens "**--**", these hyphens have no special meaning and are considered part of the comment. The comment may include graphic symbols which are not in the character set specified in 10.1 (see 10.3).

NOTE – This allows the user to comment parts of an ASN.1 module that already contain comments (whether they begin with "**--**" or "**/***") by simply inserting "**/***" at the beginning of the part to be commented and "***/**" at its end, provided there are no character string values within the part to be commented out that contain "**/***" or "***/**".

11.7 Empty lexical item

Name of lexical item – empty

The "empty" item contains no characters. It is used in the notation of clause 5 when alternative sets of production sequences are specified, to indicate that absence of all alternatives is possible.

11.8 Numbers

Name of lexical item – number

A "number" shall consist of one or more digits. The first digit shall not be zero unless the "number" is a single digit.

NOTE – The "number" lexical item is always mapped to an integer value by interpreting it as decimal notation.

11.9 Real numbers

Name of lexical item - realnumber

A "realnumber" shall consist of an integer part that is a series of one or more digits, and optionally a decimal point (.). The decimal point can optionally be followed by a fractional part which is one or more digits. The integer part, decimal point or fractional part (whichever is last present) can optionally be followed by an e or E and an optionally-signed exponent which is one or more digits. The leading digit of the exponent shall not be zero unless the exponent is a single digit.

11.10 Binary strings

Name of lexical item – bstring

A "bstring" shall consist of an arbitrary number (possibly zero) of the characters:

0 1

possibly intermixed with white-space, preceded by an APOSTROPHE (39) character (**'**) and followed by the pair of characters:

'B

EXAMPLE – **'01101100'B**

Occurrences of white-space within a binary string lexical item have no significance.

11.11 XML binary string item

Name of item – xmlbstring

An "xmlbstring" shall consist of an arbitrary number (possibly zero) of zeros, ones or white-space. Any white-space characters that appear within a binary string item have no significance.

EXAMPLE – 01101100

This sequence of characters is also a valid instance of "xmlhstring" and "xmlcstring". In analysing an instance of use of this notation, an "xmlbstring" is distinguished from an "xmlhstring" or "xmlcstring" by the context in which it appears.

11.12 Hexadecimal strings

Name of lexical item – hstring

11.12.1 An "hstring" shall consist of an arbitrary number (possibly zero) of the characters:

A B C D E F 0 1 2 3 4 5 6 7 8 9

possibly intermixed with white-space, preceded by an APOSTROPHE (39) character (**'**) and followed by the pair of characters:

'H

EXAMPLE – **'AB0196'H**

Occurrences of white-space within a hexadecimal string lexical item have no significance.

11.12.2 Each character is used to denote the value of a semi-octet using a hexadecimal representation.

11.13 XML hexadecimal string item

Name of item – xmlhstring

11.13.1 An "xmlhstring" shall consist of an arbitrary number (possibly zero) of the characters:

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

or white-space. Any white-space characters that appear within a hexadecimal string item have no significance.

EXAMPLE – Ab0196

11.13.2 Each character is used to denote the value of a semi-octet using a hexadecimal representation.

11.13.3 Some instances of "xmlhstring" are also valid instances of "xmlbstring" and "xmlcstring". In analysing an instance of use of this notation, an "xmlhstring" is distinguished from an "xmlbstring" or "xmlcstring" by the context in which it appears.

11.14 Character strings

Name of lexical item – cstring

11.14.1 A "cstring" shall consist of an arbitrary number (possibly zero) of graphic symbols and spacing characters from the character set referenced by the character string type, preceded and followed by a QUOTATION MARK (34) character ("). If the character set includes a QUOTATION MARK (34) character, this character (if present in the character string being represented by the "cstring") shall be represented in the "cstring" by a pair of QUOTATION MARK (34) characters on the same line with no intervening spacing character. The "cstring" may span more than one line of text, in which case the character string being represented shall not include spacing characters in the position prior to or following the end of line in the "cstring". Any spacing characters that appear immediately prior to or following the end of line in the "cstring" have no significance.

NOTE 1 – The "cstring" can only be used to unambiguously represent (on a printed page) character strings for which every character in the string being represented has either been assigned a graphic symbol, or is a spacing character. Where a character string containing control characters needs to be denoted in a printed representation, alternative ASN.1 syntax is available (see clause 35).

NOTE 2 – The character string represented by a "cstring" consists of the characters associated with the graphic symbols and spacing characters. Spacing characters immediately preceding or following any end of line in the "cstring" are not part of the character string being represented (they are ignored). Where spacing characters are included in the "cstring", or where the graphic symbols in the character repertoire are not unambiguous in a printed representation, the character string denoted by "cstring" may be ambiguous in that printed representation.

EXAMPLE 1 – **"屎 屍 市 弒"**

EXAMPLE 2 – The "cstring":

```
"ABCDE   FGH
IJK""XYZ"
```

can be used to represent a character string value of type **IA5String**. The value represented consists of the characters:

```
ABCDE   FGHIJK"XYZ
```

where the precise number of spaces intended between **E** and **F** can be ambiguous in a printed representation if a proportional spacing font (such as is used above) is used in the printed specification, or if the character repertoire contains multiple spacing characters of different widths.

11.14.2 When a character is a combining character (see Annex F) it shall be denoted in a printed representation of the "cstring" as an individual character. It shall not be overprinted with the characters with which it combines. (This ensures that the order of combining characters in the string value is unambiguously defined in the printed version.)

EXAMPLE – Lower case "**e**" and the accent combining character are two characters in ISO/IEC 10646-1, and thus a corresponding "cstring" should be printed as two characters and not as the single character **é**.

11.15 XML character string item

Name of item – xmlcstring

11.15.1 An "xmlcstring" shall consist of an arbitrary number (possibly zero) of the following ISO/IEC 10646-1 characters:

a) HORIZONTAL TABULATION (9);

b) LINE FEED (10);

c) CARRIAGE RETURN (13);

d) any character whose ISO/IEC 10646-1 character code is in the range 32 (20 hex) to 55295 (D7FF hex), inclusive;

e) any character whose ISO/IEC 10646-1 character code is in the range 57344 (E000 hex) to 65533 (FFFD hex), inclusive;

f) any character whose ISO/IEC 10646-1 character code is in the range 65536 (10000 hex) to 1114111 (10FFFF hex), inclusive.

NOTE – Additional restrictions are imposed by the requirement that the "xmlcstring", in an instance of use, shall contain only characters permitted by the governing character string type.

11.15.2 The characters "&" (AMPERSAND), "<" (LESS-THAN SIGN) or ">" (GREATER-THAN SIGN) shall appear only as part of one of the character sequences specified in 11.15.4 or 11.15.5.

11.15.3 An "xmlcstring" is used to represent the value of a restricted character string (see 36.9), and can be used to represent all combinations of ISO/IEC 10646-1 characters, either directly, or by using the escape sequences specified below.

NOTE 1 – An "xmlcstring" cannot be used to represent characters that are not present in ISO/IEC 10646-1, such as some of the control characters which can appear in **`GeneralString`**, nor can it represent characters which might be defined with ISO/IEC 10646-1 character codes above 10FFFF hex.

NOTE 2 – The characters LINE FEED (10) and CARRIAGE RETURN (13) and the pair CARRIAGE RETURN + LINE FEED are not distinguished when processed by conforming XML processors.

11.15.4 If the characters "&" (AMPERSAND), "<" (LESS-THAN SIGN) or ">" (GREATER-THAN SIGN) are present in an abstract character string value being represented by "xmlcstring" (see 36.9), they shall be represented in the "xmlcstring" by either

a) the escape sequences specified in 11.15.8; or

b) the escape sequences "&", "<" or ">" respectively. These escape sequences shall not contain white-space (see 11.1.6).

11.15.5 If a character with an ISO/IEC 10646-1 character code in column 1 of Table 3 is present in the abstract character string value being represented by the "xmlcstring" (see 36.9), it shall be represented by the character sequence in column 2 of Table 3. These character sequences shall not contain white-space (see 11.1.6).

NOTE – This does not include characters with decimal character codes 9, 10, and 13, and all the letters in these character sequences are lowercase.

Table 3 – Escape sequences for control characters in an xmlcstring

ISO/IEC 10646-1 character code	xmlcstring representation
0 (0 hex)	<nul/>
1 (1 hex)	<soh/>
2 (2 hex)	<stx/>
3 (3 hex)	<etx/>
4 (4 hex)	<eot/>
5 (5 hex)	<enq/>
6 (6 hex)	<ack/>
7 (7 hex)	<bel/>
8 (8 hex)	<bs/>
11 (B hex)	<vt/>
12 (C hex)	<ff/>
14 (E hex)	<so/>
15 (F hex)	<si/>
16 (10 hex)	<dle/>
17 (11 hex)	<dc1/>
18 (12 hex)	<dc2/>
19 (13 hex)	<dc3/>
20 (14 hex)	<dc4/>

21 (15 hex)	<nak/>
22 (16 hex)	<syn/>
23 (17 hex)	<etb/>
24 (18 hex)	<can/>
25 (19 hex)	
26 (1A hex)	<sub/>
27 (1B hex)	<esc/>
28 (1C hex)	<is4/>
29 (1D hex)	<is3/>
30 (1E hex)	<is2/>
31 (1F hex)	<is1/>

11.15.6 When "xmlcstring" is used within an "XMLTypedValue" (see 15.2) forming part of an XER encoding (see ITU-T Rec. X.693 | ISO/IEC 8825-4), it may contain adjacent HYPHEN-MINUS (45) characters. When used within an instance of XML value notation in an ASN.1 module, it shall not contain two adjacent HYPHEN-MINUS characters. If this character sequence is present in an abstract character string value being represented by the "xmlcstring" in an ASN.1 module then at least one of the adjacent HYPHEN-MINUS characters shall be represented by the escape sequences specified in 11.15.8.

11.15.7 When "xmlcstring" is used within an "XMLTypedValue" forming part of an XER encoding (see ITU-T Rec. X.693 | ISO/IEC 8825-4), it may contain adjacent ASTERISK (42) and SOLIDUS (47) characters in any order. When used within an instance of XML value notation in an ASN.1 module, it shall not contain adjacent ASTERISK and SOLIDUS characters (in any order). If this character sequence is present in an abstract character string value being represented by the "xmlcstring" then at least one of the adjacent ASTERISK and SOLIDUS characters shall be represented by the escape sequences specified in 11.15.8.

11.15.8 Any character that can appear directly in an "xmlcstring" can also be represented in the "xmlcstring" by an escape sequence of the form "&#n;" (where n is the ISO/IEC 10646-1 character code in decimal notation) or of the form "&#xn;" (where n is the ISO/IEC 10646-1 character code in hexadecimal notation). These escape sequences shall not contain white-space (see 11.1.6).

NOTE 1 – Leading zeros are permitted in the decimal and hexadecimal values of "n" and both lowercase and uppercase letters "A" – "F" can be used in the hexadecimal value.

NOTE 2 – If the escape sequences "&#n" and "&#xn" are used for ISO/IEC 10646-1 characters which are not in the Basic Multilingual Plane (BMP), the value of "n" will be greater than 65535 (FFFF hex).

EXAMPLE – The "xmlcstring":

ABCDé FGHîJK&XYZ

can be used to represent a character string value of type UTF8String. The value represented consists of the characters:

ABCDé FGHîJK&XYZ

where the precise space characters between é and F can be ambiguous in print media if a proportional spacing font (such as above) is used in the specification.

11.16 Assignment lexical item

Name of lexical item – "**::=**"

This lexical item shall consist of the sequence of characters:

::=

NOTE – This sequence does not contain white-space (see 11.1.2).

11.17 Range separator

Name of lexical item – "**..**"

This lexical item shall consist of the sequence of characters:

..

NOTE – This sequence does not contain white-space (see 11.1.2).

11.18 Ellipsis

Name of lexical item – "**. . .**"

This lexical item shall consist of the sequence of characters:

. . .

NOTE – This sequence does not contain white-space (see 11.1.2).

11.19 Left version brackets

Name of lexical item – "**[[**"

This lexical item shall consist of the sequence of characters:

[[

NOTE – This sequence does not contain white-space (see 11.1.2).

11.20 Right version brackets

Name of lexical item – "**]]**"

This lexical item shall consist of the sequence of characters:

]]

NOTE – This sequence does not contain white-space (see 11.1.2).

11.21 XML end tag start item

Name of item – "</"

This item shall consist of the sequence of characters:

</

NOTE – This sequence does not contain any white-space characters (see 11.1.2).

11.22 XML single tag end item

Name of item – "/>"

This item shall consist of the sequence of characters:

/>

NOTE – This sequence does not contain any white-space characters (see 11.1.2).

11.23 XML boolean true item

Name of item – "true"

11.23.1 This item shall consist of the sequence of characters:

true

11.23.2 In analysing an instance of use of this notation, a "true" is distinguished from a "valuereference" or an "identifier" by the context in which it appears.

NOTE – This sequence does not contain any white-space characters (see 11.1.2).

11.24 XML boolean false item

Name of item – "false"

11.24.1 This item shall consist of the sequence of characters:

false

11.24.2 In analyzing an instance of use of this notation, a "false" is distinguished from a "valuereference" or an "identifier" by the context in which it appears.

NOTE – This sequence does not contain any white-space characters (see 11.1.2).

11.25 XML tag names for ASN.1 types

Name of item – xmlasn1typename

11.25.1 This Recommendation | International Standard uses the item "xmlasn1typename" when ASN.1 built-in types are to be used as XML tag names.

11.25.2 Table 4 lists the character sequences that are to form the "xmlasn1typename" for each of the ASN.1 built-in types listed in 16.2. The ASN.1 built-in type is identified in column 1 of Table 4 by its production name. The character sequence which shall be used for "xmlasn1typename" is identified in column 2 of Table 4, with no white-space before or after these character sequences.

11.25.3 The "xmlasn1typename" for the "UsefulType"s (see 41.1) shall be the "typereference" used in their definition.

11.25.4 The character sequence in the "xmlasn1typename" item for the "ObjectClassFieldType" and for the "InstanceOfType" are specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.1 and Annex C.

11.25.5 If the ASN.1 built-in type is a "TaggedType" then the type which determines the "xmlasn1typename" shall be "Type" in the "TaggedType" (see 30.1). If this is itself a "TaggedType", then this sub-clause 11.25.5 shall be recursively applied.

Table 4 – Characters in xmlasn1typename

ASN.1 type production name	Characters in xmlasn1typename
BitStringType	BIT_STRING
BooleanType	BOOLEAN
ChoiceType	CHOICE
EmbeddedPDVType	SEQUENCE
EnumeratedType	ENUMERATED
ExternalType	SEQUENCE
InstanceOfType	SEQUENCE
IntegerType	INTEGER
NullType	NULL
ObjectClassFieldType	*See ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.10 and 14.11*
ObjectIdentifierType	OBJECT_IDENTIFIER
OctetStringType	OCTET_STRING
RealType	REAL
RelativeOIDType	RELATIVE_OID
RestrictedCharacterStringType	*The type name (e.g.* `IA5String`*)*
SequenceType	SEQUENCE
SequenceOfType	SEQUENCE_OF
SetType	SET
SetOfType	SET_OF
TaggedType	*See 11.25.5*
UnrestrictedCharacterStringType	SEQUENCE

11.26 Single character lexical items

Names of lexical items –

"{"
"}"
"<"
">"

","
"."
"("
")"
"["
"]"
"-" (HYPEN-MINUS)
":"
"="
""" (QUOTATION MARK)
"'" (APOSTROPHE)
" " (SPACE)
";"
"@"
"|"
"!"
"^"

A lexical item with any of the names listed above shall consist of the single character without the quotation marks.

11.27 Reserved words

Names of reserved words –

ABSENT	**ENCODED**	**INTEGER**	**RELATIVE-OID**
ABSTRACT-SYNTAX	**END**	**INTERSECTION**	**SEQUENCE**
ALL	**ENUMERATED**	**ISO646String**	**SET**
APPLICATION	**EXCEPT**	**MAX**	**SIZE**
AUTOMATIC	**EXPLICIT**	**MIN**	**STRING**
BEGIN	**EXPORTS**	**MINUS-INFINITY**	**SYNTAX**
BIT	**EXTENSIBILITY**	**NULL**	**T61String**
BMPString	**EXTERNAL**	**NumericString**	**TAGS**
BOOLEAN	**FALSE**	**OBJECT**	**TeletexString**
BY	**FROM**	**ObjectDescriptor**	**TRUE**
CHARACTER	**GeneralizedTime**	**OCTET**	**TYPE-IDENTIFIER**
CHOICE	**GeneralString**	**OF**	**UNION**
CLASS	**GraphicString**	**OPTIONAL**	**UNIQUE**
COMPONENT	**IA5String**	**PATTERN**	**UNIVERSAL**
COMPONENTS	**IDENTIFIER**	**PDV**	**UniversalString**
CONSTRAINED	**IMPLICIT**	**PLUS-INFINITY**	**UTCTime**
CONTAINING	**IMPLIED**	**PRESENT**	**UTF8String**
DEFAULT	**IMPORTS**	**PrintableString**	**VideotexString**
DEFINITIONS	**INCLUDES**	**PRIVATE**	**VisibleString**
EMBEDDED	**INSTANCE**	**REAL**	**WITH**

Lexical items with the above names shall consist of the sequence of characters in the name, and are reserved character sequences.

NOTE 1 – White-space does not occur in these sequences.

NOTE 2 – The keywords **CLASS**, **CONSTRAINED**, **CONTAINING**, **ENCODED**, **INSTANCE**, **SYNTAX** and **UNIQUE** are not used in this Recommendation | International Standard; they are used in ITU-T Rec. X.681 | ISO/IEC 8824-2, ITU-T Rec. X.682 | ISO/IEC 8824-3 and ITU-T Rec. X.683 | ISO/IEC 8824-4.

12 Module definition

12.1 A "ModuleDefinition" is specified by the following productions:

```
ModuleDefinition ::=
        ModuleIdentifier
        DEFINITIONS
        TagDefault
        ExtensionDefault
        "::="
        BEGIN
        ModuleBody
        END

ModuleIdentifier ::=
        modulereference
        DefinitiveIdentifier

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

DefinitiveObjIdComponentList ::=
        DefinitiveObjIdComponent |
        DefinitiveObjIdComponent DefinitiveObjIdComponentList

DefinitiveObjIdComponent ::=
        NameForm |
        DefinitiveNumberForm |
        DefinitiveNameAndNumberForm

DefinitiveNumberForm ::= number

DefinitiveNameAndNumberForm ::= identifier "(" DefinitiveNumberForm ")"

TagDefault ::=
        EXPLICIT TAGS |
        IMPLICIT TAGS |
        AUTOMATIC TAGS |
        empty

ExtensionDefault ::=
        EXTENSIBILITY IMPLIED |
        empty

ModuleBody ::=
        Exports Imports AssignmentList |
        empty

Exports ::=
        EXPORTS SymbolsExported ";" |
        EXPORTS ALL ";" |
        empty

SymbolsExported ::=
        SymbolList |
        empty

Imports ::=
        IMPORTS SymbolsImported ";" |
        empty

SymbolsImported ::=
        SymbolsFromModuleList |
        empty

SymbolsFromModuleList ::=
        SymbolsFromModule |
```

```
                SymbolsFromModuleList  SymbolsFromModule

SymbolsFromModule ::=
                SymbolList FROM GlobalModuleReference

GlobalModuleReference ::=
                modulereference AssignedIdentifier

AssignedIdentifier ::=
                ObjectIdentifierValue   |
                DefinedValue            |
                empty

SymbolList ::=
                Symbol                  |
                SymbolList "," Symbol

Symbol ::=
                Reference               |
                ParameterizedReference

Reference ::=
                typereference           |
                valuereference          |
                objectclassreference    |
                objectreference         |
                objectsetreference

AssignmentList ::=
                Assignment              |
                AssignmentList Assignment

Assignment ::=
                TypeAssignment          |
                ValueAssignment         |
                XMLValueAssignment      |
                ValueSetTypeAssignment  |
                ObjectClassAssignment   |
                ObjectAssignment        |
                ObjectSetAssignment     |
                ParameterizedAssignment
```

NOTE 1 – The use of a "ParameterizedReference" in the "Exports" and "Imports" lists is specified in ITU-T Rec. X.683 | ISO/IEC 8824-4.

NOTE 2 – For examples (and for the definition in this Recommendation | International Standard of types with universal class tags), the "ModuleBody" can be used outside of a "ModuleDefinition".

NOTE 3 – "TypeAssignment", "ValueAssignment", "XMLValueAssignment" and "ValueSetTypeAssignment" productions are specified in clause 15.

NOTE 4 – The value of "TagDefault" for the module definition affects only those types defined explicitly in the module. It does not affect the interpretation of imported types.

NOTE 5 – The character semicolon does not appear in the assignment list specification or any of its subordinate productions, and is reserved for use by ASN.1 tool developers.

12.2 The "TagDefault" is taken as **EXPLICIT TAGS** if it is "empty".

NOTE – Clause 30 gives the meaning of **EXPLICIT TAGS**, **IMPLICIT TAGS**, and **AUTOMATIC TAGS**.

12.3 When the **AUTOMATIC TAGS** alternative of "TagDefault" is selected, automatic tagging is said to be selected for the module, otherwise it is said to be not selected. Automatic tagging is a syntactical transformation which is applied (with additional conditions) to the "ComponentTypeLists" and "AlternativeTypeLists" productions occurring within the definition of the module. This transformation is formally specified by 24.7 to 24.9, 26.3 and 28.2 to 28.5 regarding the notations for sequence types, set types and choice types, respectively.

12.4 The **EXTENSIBILITY IMPLIED** option is equivalent to the textual insertion of an extension marker ("**...**") in the definition of each type in the module for which it is permitted. The location of the implied extension marker is the last position in the type where an explicitly specified extension marker is allowed. The absence of **EXTENSIBILITY IMPLIED** means that extensibility is only provided for those types within the module where an extension marker is explicitly present.

NOTE – **EXTENSIBILITY IMPLIED** affects only types. It has no effect on object sets and subtype constraints.

12.5 The "modulereference" appearing in the "ModuleIdentifier" production is called the module name.

NOTE – The possibility of defining a single ASN.1 module by the use of several occurrences of "ModuleBody" assigned the same "modulereference" was (arguably) permitted in earlier specifications. It is not permitted by this Recommendation | International Standard.

12.6 Module names shall be used only once (except as specified in 12.9) within the sphere of interest of the definition of the module.

12.7 If the "DefinitiveIdentifier" is not empty, the denoted object identifier value unambiguously and uniquely identifies the module being defined. No defined value may be used in defining the object identifier value.

NOTE – The question of what changes to a module require a new "DefinitiveIdentifier" is not addressed in this Recommendation | International Standard.

12.8 If the "AssignedIdentifier" is not empty, the "ObjectIdentifierValue" and the "DefinedValue" alternatives unambiguously and uniquely identify the module from which reference names are being imported. When the "DefinedValue" alternative of "AssignedIdentifier" is used, it shall be a value of type object identifier. Each "valuereference" which textually appears within an "AssignedIdentifier" shall satisfy one of the following rules:

a) It is defined in the "AssignmentList" of the module being defined, and all "valuereference"s which textually appear on the right side of the assignment statement also satisfy this rule (rule "a") or the next rule (rule "b").

b) It appears as a "Symbol" in a "SymbolsFromModule" whose "AssignedIdentifier" does not textually contain any "valuereference"s.

NOTE – It is recommended that an object identifier be assigned so that others can unambiguously refer to the module.

12.9 The "GlobalModuleReference" in a "SymbolsFromModule" shall appear in the "ModuleDefinition" of another module, except that if it includes a non-empty "DefinitiveIdentifier", the "modulereference" may differ in the two cases.

NOTE – A different "modulereference" from that used in the other module should only be used when symbols are to be imported from two modules with the same name (the modules being named in disregard of 12.6). The use of alternative distinct names makes these names available for use in the body of the module (see 12.15).

12.10 When both a "modulereference" and a non-empty "AssignedIdentifier" are used in referencing a module, the latter shall be considered definitive.

12.11 When the referenced module has a non-empty "DefinitiveIdentifier", the "GlobalModuleReference" referencing that module shall not have an empty "AssignedIdentifier".

12.12 When the "SymbolsExported" alternative of "Exports" is selected:

a) each "Symbol" in "SymbolsExported" shall satisfy one and only one of the following conditions:

   i) is only defined in the module being constructed; or

   ii) appears exactly once in the "SymbolsImported" alternative of "Imports";

b) every "Symbol" to which reference from outside the module is appropriate shall be included in the "SymbolsExported" and only these "Symbol"s may be referenced from outside the module (subject to the relaxation specified in 12.13); and

c) if there are no such "Symbol"s, then the empty alternative of "SymbolsExported" (not of "Exports") shall be selected.

12.13 When either the "empty" alternative or the **EXPORTS ALL** alternative of "Exports" is selected, every "Symbol" defined in the module or imported by the module may be referenced from other modules subject to the restriction specified in 12.12a.

NOTE – The "empty" alternative of "Exports" is included for backwards compatibility.

12.14 Identifiers that appear in a "NamedNumberList", "Enumeration" or "NamedBitList" are implicitly exported if the typereference that defines them is exported or appears as a component (or subcomponent) within an exported type.

12.15 When the "SymbolsImported" alternative of "Imports" is selected:

a) Each "Symbol" in "SymbolsFromModule" shall either be defined in the module body, or be present in the "Imports" clause, of the module denoted by the "GlobalModuleReference" in "SymbolsFromModule". Importing a "Symbol" present in the "Imports" clause of the referenced module is only allowed if there is only one occurrence of the "Symbol" in that clause, and the "Symbol" is not defined in the referenced module.

NOTE 1 – 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) If the "SymbolsExported" alternative of "Exports" is selected in the definition of the module denoted by the "GlobalModuleReference" in "SymbolsFromModule" the "Symbol" shall appear in its "SymbolsExported".

c) Only those "Symbol"s that appear amongst the "SymbolList" of a "SymbolsFromModule" may appear as the symbol in any "External<X>Reference" which has the "modulereference" denoted by the "GlobalModuleReference" of that "SymbolsFromModule" (where <X> is "Value", "Type", "Object", "Objectclass", or "Objectset").

d) If there are no such "Symbol"s, then the "empty" alternative of "SymbolsImported" shall be selected.

NOTE 2 – An effect of c) and d) is that the statement **`IMPORTS;`** implies that the module cannot contain an "External<X>Reference".

e) All the "SymbolsFromModule" in the "SymbolsFromModuleList" shall include occurrences of "GlobalModuleReference" such that:

i) the "modulereference" in them are all different from each other 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.

12.16 When the "empty" alternative of "Imports" is selected, the module may still reference "Symbols" defined in other modules by means of an "External<X>Reference".

NOTE – The "empty" alternative of "Imports" is included for backwards compatibility.

12.17 Identifiers that appear in a "NamedNumberList", "Enumeration" or "NamedBitList" are implicitly imported if the typereference that defines them is imported or appears as a component (or subcomponent) within an imported type.

12.18 A "Symbol" in a "SymbolsFromModule" may appear in "ModuleBody" as a "Reference". The meaning associated with the "Symbol" is that which it has in the module denoted by the corresponding "GlobalModuleReference".

12.19 Where the "Symbol" also appears in an "AssignmentList" (deprecated), or appears in one or more other instances of "SymbolsFromModule", it shall only be used in an "External<X>Reference". Where it does not so appear, it shall be used directly as a "Reference".

12.20 The various alternatives for "Assignment" are defined in the following clauses in this Recommendation | International Standard, except as noted otherwise:

Assignment alternative	*Defining subclause*
"TypeAssignment"	15.1
"ValueAssignment"	15.2
"XMLValueAssignment"	15.2
"ValueSetTypeAssignment"	15.6
"ObjectClassAssignment"	ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.1
"ObjectAssignment"	ITU-T Rec. X.681 | ISO/IEC 8824-2, 11.1
"ObjectSetAssignment"	ITU-T Rec. X.681 | ISO/IEC 8824-2, 12.1
"ParameterizedAssignment"	ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.1

The first symbol of every "Assignment" is one of the alternatives of "Reference", denoting the reference name being defined. In no two assignments within an "AssignmentList" shall the reference names be the same.

13 Referencing type and value definitions

13.1 The defined type and value productions**:**

```
DefinedType ::=
        ExternalTypeReference   |
        Typereference           |
```

```
                ParameterizedType       |
                ParameterizedValueSetType

DefinedValue ::=
                ExternalValueReference  |
                Valuereference          |
                ParameterizedValue
```

specify the sequences which shall be used to reference type and value definitions. The type identified by a "ParameterizedType" and "ParameterizedValueSetType", and the value identified by a "ParameterizedValue" are specified in ITU-T Rec. X.683 | ISO/IEC 8824-4.

13.2 The "NonParameterizedTypeName" production:

```
NonParameterizedTypeName ::=
        ExternalTypeReference   |
        typereference           |
        xmlasn1typename
```

is used when an XML tag name is needed to represent an ASN.1 type.

13.3 The third alternative shall not be used as the "NonParameterizedTypeName" in the "XMLTypedValue" of "XMLValueAssignment" (see 15.2) or of "XMLOpenTypeFieldVal" (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.6) when the XML value notation is used in an ASN.1 module if the "xmlasn1typename" is "CHOICE", "ENUMERATED", "SEQUENCE", "SEQUENCE_OF", "SET" or "SET_OF".

NOTE – This restriction is imposed in XML value notation used in an ASN.1 module because these "xmlasn1typename"s do not define an ASN.1 type. The restriction is not present for use of this notation in encoding rules (such as XER, see ITU-T Rec. X.693 | ISO/IEC 8825-4) because XML tags formed from "xmlasn1typename"s are not used to determine the types that are being encoded.

13.4 Except as specified in 12.18, the "typereference", "valuereference", "ParameterizedType", "ParameterizedValueSetType" or "ParameterizedValue" alternatives shall not be used unless the reference is within the "ModuleBody" in which a type or value is assigned (see 15.1 and 15.2) to the "typereference" or "valuereference".

13.5 The "ExternalTypeReference" and "ExternalValueReference" shall not be used unless the corresponding "typereference" or "valuereference":

a) has been assigned a type or value respectively (see 15.1 and 15.2); or

b) are present in the "Imports" clause,

within the "ModuleBody" used to define the corresponding "modulereference". Referencing a name in the "Imports" clause of another module shall only be allowed if there is no more than one occurrence of the "Symbol" in that clause.

NOTE – This does not prohibit the same "Symbol" defined in two different modules from being imported into another module. However, if the same "Symbol" appears more than once in the **IMPORTS** clause of a module **A**, then that "Symbol" cannot be referenced using module **A** in an external reference.

13.6 An external reference shall be used in a module only to refer to a reference name which is defined in a different module, and is specified by the following productions:

```
ExternalTypeReference ::=
                modulereference
                "."
                typereference

ExternalValueReference ::=
                modulereference
                "."
                valuereference
```

NOTE – Additional external reference productions ("ExternalClassReference", "ExternalObjectReference" and "ExternalObjectSetReference") are specified in ITU-T Rec. X.681 | ISO/IEC 8824-2.

13.7 When the referencing module is defined using the "SymbolsImported" alternative of "Imports", the "modulereference" in the external reference shall appear in the "GlobalModuleReference" of exactly one of the "SymbolsFromModule" in the "SymbolsImported". When the referencing module is defined using the "empty" alternative of "Imports", the "modulereference" in the external reference shall appear in the "ModuleDefinition" of the module (different from the referencing module) where the "Reference" is defined.

13.8 Where a "DefinedType" is used as part of notation governed by a "Type" (for example, in a "SubtypeConstraint") then the "DefinedType" shall be compatible with the governing "Type" as specified in clause B.6.2.

13.9 Every occurrence within an ASN.1 specification of a "DefinedValue" is governed by a "Type", and that "DefinedValue" shall reference a value of a type that is compatible with the governing "Type" as specified in clause B.6.2.

14 Notation to support references to ASN.1 components

14.1 There is a requirement for formal reference to components of ASN.1 types, values, etc. for many purposes. One such instance is the need to write text to identify a specific type within some ASN.1 module. This clause defines a notation which can be used to provide such references.

14.2 The notation enables any component of a set or sequence type (which is either mandatorily or optionally present in the type) to be identified.

14.3 Any part of any ASN.1 type definition can be referenced by use of the "AbsoluteReference" syntactic construct:

```
AbsoluteReference ::= "@" ModuleIdentifier
                      "."
                      ItemSpec

ItemSpec ::=
                      typereference |
                      ItemId "." ComponentId

ItemId ::= ItemSpec

ComponentId ::=
                      identifier |
                      number |
                      "*"
```

NOTE – The AbsoluteReference production is not used elsewhere in this Recommendation | International Standard. It is provided for the purposes stated in 14.1.

14.4 The "ModuleIdentifier" identifies an ASN.1 module (see 12.1).

14.5 When the first altenative of "DefinitiveIdentifier" is used as part of the "ModuleIdentifier", the "DefinitiveIdentifier" unambiguously and uniquely identifies the module from which an name is being referenced.

14.6 The "typereference" references any ASN.1 type defined in the module identified by "ModuleIdentifier".

14.7 The "ComponentId" in each "ItemSpec" identifies a component of the type which has been identified by the "ItemId". It shall be the last "ComponentId" if the component it identifies is not a set, sequence, set-of, sequence-of, or choice type.

14.8 The "identifier" form of "ComponentId" can be used if the parent "ItemId" is a set or sequence type, and is required to be one of the "identifier"s of the "NamedType" in the "ComponentTypeLists" of that set or sequence. It can also be used if the "ItemId" identifies a choice type, and is then required to be one of the "identifier"s of a "NamedType" in the "AlternativeTypeLists" of that choice type. It cannot be used in any other circumstance.

14.9 The number form of "ComponentId" can be used only if the "ItemId" is a sequence-of or set-of type. The value of the number identifies the instance of the type in the sequence-of or set-of, with the value "1" identifying the first instance of the type. The value zero identifies a conceptual integer type component (not explicitly present in transfer) that contains a count of the number of instances of the type in the sequence-of or set-of that are present in the value of the enclosing type.

14.10 The "*****" form of "ComponentId" can be used only if the "ItemId" is a sequence-of or set-of. Any semantics associated with the use of the "*****" form of "ComponentId" apply to all components of the sequence-of and set-of.

NOTE – In the following example:

```
M DEFINITIONS ::= BEGIN
      T ::= SEQUENCE {
            a     BOOLEAN,
            b     SET OF INTEGER
```

```
        }
END
```

the components of "**T**" could be referenced by text outside an ASN.1 module (or in a comment), such as:

```
-- if (@M.T.b.0 is odd) then:
--        (@M.T.b.* shall be an odd integer)
```

which is used to state that if the number of components in **b** is odd, all components of **b** must be odd.

15 Assigning types and values

15.1 A "typereference" shall be assigned a type by the notation specified by the "TypeAssignment" production:

TypeAssignment ::=
typereference
"::="
Type

The "typereference" shall not be an ASN.1 reserved word (see 11.27).

15.2 A "valuereference" shall be assigned a value by the notation specified by either the "ValueAssignment" or "XMLValueAssignment" productions:

ValueAssignment ::=
valuereference
Type
"::="
Value

XMLValueAssignment ::=
valuereference
"::="
XMLTypedValue

XMLTypedValue ::=
"<" & NonParameterizedTypeName ">"
XMLValue
"</" & NonParameterizedTypeName ">" |
"<" & NonParameterizedTypeName "/>"

The value being assigned to the "valuereference" in the "ValueAssignment" is "Value", and is governed by "Type" and shall be a notation for a value of the type defined by "Type" (as specified in 15.3). The value being assigned to the "valuereference" in the "XMLValueAssignment" is "XMLValue" (see 16.7), and shall be a notation for a value of the type defined by "NonParameterizedTypeName" (as specified in 15.4). If this is the "xmlasn1typename" item, then it identifies the ASN.1 built-in type in the corresponding row of Table 4 (see also 13.3).

15.3 "Value" is a notation for a value of a type as specified in 16.7.

15.4 "XMLValue" is a notation for a value of a type if "XMLValue" is an "XMLBuiltinValue" notation for the type (see 16.10).

15.5 The second alternative of "XMLTypedValue" (use of an XML empty-element tag) can be used only if an instance of the "XMLValue" production is empty.

NOTE – If the "XMLValue" production was an "xmlcstring" containing only white-space, this would not be empty, and the second alternative could not be used.

15.6 A "typereference" can be assigned a value set by the notation specified by the "ValueSetTypeAssignment" production:

ValueSetTypeAssignment ::= typereference
Type
"::="
ValueSet

This notation assigns to "typereference" the type defined as a subtype of the type denoted by "Type" and which contains exactly the values which are specified in or allowed by "ValueSet". The "typereference" shall not be an ASN.1 reserved word (see 11.27), and may be referenced as a type. "ValueSet" is defined in 15.7.

15.7 A value set governed by some type shall be specified by the notation "ValueSet":

ValueSet ::= "{" ElementSetSpecs "}"

The value set comprises all of the values, of which there shall be at least one, specified by "ElementSetSpecs" (see clause 46).

15.8 The "ValueSetTypeAssignment" production expands into:

typereference
Type
"::="
"{" ElementSetSpecs "}"

For all purposes, including the application of encoding rules, this is defined to be exactly equivalent to the use of the production:

typereference
"::="
Type
"(" ElementSetSpecs ")"

with the same "Type" and "ElementSetSpecs" specifications.

16 Definition of types and values

16.1 A type shall be specified by the notation "Type":

Type ::= BuiltinType | ReferencedType | ConstrainedType

16.2 The built-in types of ASN.1 are specified by the notation "BuiltinType", defined as follows:

BuiltinType ::=
BitStringType |
BooleanType |
CharacterStringType |
ChoiceType |
EmbeddedPDVType |
EnumeratedType |
ExternalType |
InstanceOfType |
IntegerType |
NullType |
ObjectClassFieldType |
ObjectIdentifierType |
OctetStringType |
RealType |
RelativeOIDType |
SequenceType |
SequenceOfType |
SetType |
SetOfType |
TaggedType

The various "BuiltinType" notations are defined in the following clauses (in this Recommendation | International Standard unless otherwise stated):

BitStringType	21
BooleanType	17
CharacterStringType	36
ChoiceType	28
EmbeddedPDVType	33
EnumeratedType	19
ExternalType	34
InstanceOfType	ITU-T Rec. X.681 | ISO/IEC 8824-2, Annex C
IntegerType	18
NullType	23
ObjectClassFieldType	ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.1

ObjectIdentifierType	31
OctetStringType	22
RealType	20
RelativeOIDType	32
SequenceType	24
SequenceOfType	25
SetType	26
SetOfType	27
TaggedType	30

16.3 The referenced types of ASN.1 are specified by the notation "ReferencedType":

ReferencedType ::=
DefinedType |
UsefulType |
SelectionType |
TypeFromObject |
ValueSetFromObjects

The "ReferencedType" notation provides an alternative means of referring to some other type (and ultimately to a built-in type). The various "ReferencedType" notations, and the way in which the type to which they refer is determined, are specified in the following places in this Recommendation | International Standard unless otherwise stated):

DefinedType	13.1
UsefulType	41.1
SelectionType	29
TypeFromObject	ITU-T Rec. X.681 | ISO/IEC 8824-2, clause 15
ValueSetFromObjects	ITU-T Rec. X.681 | ISO/IEC 8824-2, clause 15

16.4 The "ConstrainedType" is defined in clause 45.

16.5 This Recommendation | International Standard requires the use of the notation "NamedType" in specifying the components of the set types, sequence types and choice types. The notation for "NamedType" is:

NamedType ::= identifier Type

16.6 The "identifier" is used to unambiguously refer to components of a set type, sequence type or choice type in the value notation, in inner subtype constraints and in component relation constraints (see ITU-T Rec. X.682 | ISO/IEC 8824-3). It is not part of the type, and has no effect on the type.

16.7 A value of some type shall be specified by the notation "Value" or by the notation "XMLValue":

Value ::=
BuiltinValue |
ReferencedValue |
ObjectClassFieldValue

XMLValue ::=
XMLBuiltinValue |
XMLObjectClassFieldValue

NOTE 1 – "ObjectClassFieldValue" and "XMLObjectClassFieldValue" are defined in ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.6.

NOTE 2 – "XMLValue" is only used in "XMLTypedValue".

16.8 If any part of the "XMLValue" production results in an XML start-tag immediately followed by an XML end-tag, possible separated by white-space inserted as permitted by 11.1.4 (for example, <field1></field1>), these two XML tags, and any intervening white-space, can be replaced by a single XML empty-element tag (<field1/>).

NOTE – If any white-space character, except white-space inserted as permitted by 11.1.4, is present between the final ">" character of the start tag and the initial "<" character of the end-tag, the condition above is not satisfied.

16.9 Values of the built-in types of ASN.1 can be specified by the notation "XMLBuiltinValue" (see 16.10) or "BuiltinValue", defined as follows:

BuiltinValue ::=
BitStringValue |
BooleanValue |
CharacterStringValue |
ChoiceValue |

EmbeddedPDVValue |
EnumeratedValue |
ExternalValue |
InstanceOfValue |
IntegerValue |
NullValue |
ObjectIdentifierValue |
OctetStringValue |
RealValue |
RelativeOIDValue |
SequenceValue |
SequenceOfValue |
SetValue |
SetOfValue |
TaggedValue

Each of the various "BuiltinValue" notations is defined in the same sub-clause as the corresponding "BuiltinType" notation, as listed in 16.2.

16.10 "XMLBuiltinValue" is defined as follows:

XMLBuiltinValue ::=
XMLBitStringValue |
XMLBooleanValue |
XMLCharacterStringValue |
XMLChoiceValue |
XMLEmbeddedPDVValue |
XMLEnumeratedValue |
XMLExternalValue |
XMLInstanceOfValue |
XMLIntegerValue |
XMLNullValue |
XMLObjectIdentifierValue |
XMLOctetStringValue |
XMLRealValue |
XMLRelativeOIDValue |
XMLSequenceValue |
XMLSequenceOfValue |
XMLSetValue |
XMLSetOfValue |
XMLTaggedValue

Each of the various "XMLBuiltinValue" notations is defined in the same clause as the corresponding "BuiltinType" notation, as listed in 16.2 above.

16.11 The referenced values of ASN.1 are specified by the notation "ReferencedValue":

ReferencedValue ::=
DefinedValue |
ValueFromObject

The "ReferencedValue" notation provides an alternative means of referring to some other value (and ultimately to a built-in value). The various "ReferencedValue" notations, and the way in which the value to which they refer is determined, are specified in the following places (in this Recommendation | International Standard unless otherwise stated):

DefinedValue	13.1
ValueFromObject	ITU-T Rec. X.681 | ISO/IEC 8824-2, clause 15

16.12 Regardless of whether or not a type is a "BuiltinType", "ReferencedType" or "ConstrainedType", its values can be specified by either a "BuiltinValue" or "ReferencedValue" of that type.

16.13 The value of a type referenced using the "NamedType" notation shall be defined by the notation "NamedValue", or when used as part of an "XMLValue", by the notation "XMLNamedValue". These productions are:

NamedValue ::= identifier Value

XMLNamedValue ::= "<" & identifier ">" XMLValue "</" & identifier ">"

where the "identifier" is the same as that used in the "NamedType" notation.

NOTE – The "identifier" is part of the notation, it does not form part of the value itself. It is used to unambiguously refer to the components of a set type, sequence type or choice type.

16.14 The implied (see 12.4) or explicit presence of an extension marker (see clause 6) in the definition of a type has no effect on the value notation. That is, the value notation for a type with an extension marker is exactly the same as if the extension marker was absent.

NOTE – Sub-clause 46.8 prohibits value notation used in a subtype constraint from referencing a value that is not in the extension root of the parent type.

17 Notation for the boolean type

17.1 The boolean type (see 3.6.7) shall be referenced by the notation "BooleanType":

BooleanType ::= BOOLEAN

17.2 The tag for types defined by this notation is universal class, number 1.

17.3 The value of a boolean type (see 3.6.73 and 3.6.38) shall be defined by the notation "BooleanValue", or when used as an "XMLValue", by the notation "XMLBooleanValue". These productions are:

BooleanValue ::= TRUE | FALSE

XMLBooleanValue ::=
"<" & "true" "/>" |
"<" & "false" "/>"

18 Notation for the integer type

18.1 The integer type (see 3.6.41) shall be referenced by the notation "IntegerType":

IntegerType ::=
INTEGER |
INTEGER "{" NamedNumberList "}"

NamedNumberList ::=
NamedNumber |
NamedNumberList "," NamedNumber

NamedNumber ::=
identifier "(" SignedNumber ")"|
identifier "(" DefinedValue ")"

SignedNumber ::=
number |
"-" number

18.2 The second alternative of "SignedNumber" shall not be used if the "number" is zero.

18.3 The "NamedNumberList" is not significant in the definition of a type. It is used solely in the value notation specified in 18.9.

18.4 The "valuereference" in "DefinedValue" shall be of type integer.

NOTE – Since an "identifier" cannot be used to specify the value associated with "NamedNumber", the "DefinedValue" can never be misinterpreted as an "IntegerValue". Therefore in the following case

```
a INTEGER ::= 1
T1 ::= INTEGER { a(2) }
T2 ::= INTEGER { a(3), b(a) }
c T2 ::= b
d T2 ::= a
```

c denotes the value 1, since it cannot be a reference to the second nor the third occurrence of **a**, and **d** denotes the value 3.

18.5 The value of each "SignedNumber" or "DefinedValue" appearing in the "NamedNumberList" shall be different, and represents a distinguished value of the integer type.

18.6 Each "identifier" appearing in the "NamedNumberList" shall be different.

18.7 The order of the "NamedNumber"s in the "NamedNumberList" is not significant.

18.8 The tag for types defined by this notation is universal class, number 2.

18.9 The value of an integer type shall be defined by the notation "IntegerValue", or when used as an "XMLValue", by the notation "XMLIntegerValue". These productions are:

IntegerValue ::=
SignedNumber |
identifier

XMLIntegerValue ::=
SignedNumber |
"<" & identifier "/>"

18.10 The "identifier" in "IntegerValue" and in "XMLIntegerValue" shall be one of the "identifier"s in the "IntegerType" with which the value is associated, and shall represent the corresponding number.

NOTE – When referencing an integer value for which an "identifier" has been defined, use of the "identifier" form of "IntegerValue" and "XMLIntegerValue" should be preferred.

18.11 Within an instance of value notation for an integer type with a "NamedNumberList", any occurrence of a name that is both an "identifier" from the "NamedNumberList" and a reference name shall be interpreted as the "identifier".

19 Notation for the enumerated type

19.1 The enumerated type (see 3.6.24) shall be referenced by the notation "EnumeratedType":

EnumeratedType ::=
ENUMERATED "{" Enumerations "}"

Enumerations ::=
RootEnumeration |
RootEnumeration "," "..." ExceptionSpec |
RootEnumeration "," "..." ExceptionSpec "," AdditionalEnumeration

RootEnumeration ::= Enumeration

AdditionalEnumeration ::= Enumeration

Enumeration ::=
EnumerationItem | EnumerationItem "," Enumeration

EnumerationItem ::=
identifier | NamedNumber

NOTE 1 – Each value of an "EnumeratedType" has an identifier which is associated with a distinct integer. However, the values themselves are not expected to have any integer semantics. Specifying the "NamedNumber" alternative of "EnumerationItem" provides control of the representation of the value in order to facilitate compatible extensions.

NOTE 2 – The numeric values inside the "NamedNumber"s in the "RootEnumeration" are not necessarily ordered or contiguous, and the numeric values inside the "NamedNumber"s in the "AdditionalEnumeration" are ordered but not necessarily contiguous.

19.2 For each "NamedNumber", the "identifier" and the "SignedNumber" shall be distinct from all other "identifier"s and "SignedNumber"s in the "Enumeration". Subclauses 18.2 and 18.4 also apply to each "NamedNumber".

19.3 Each "EnumerationItem" (in an "EnumeratedType") which is an "identifier" is successively assigned a distinct non-negative integer. For the "RootEnumeration", the successive integers starting with 0, but excluding any which are employed in "EnumerationItem"s which are "NamedNumber"s, are assigned.

NOTE – An integer value is associated with an "EnumerationItem" to assist in the definition of encoding rules. It is not otherwise used in the ASN.1 specification.

19.4 The value of each new "EnumerationItem" shall be greater than all previously defined "AdditionalEnumeration"s in the type.

19.5 When a "NamedNumber" is used in defining an "EnumerationItem" in the "AdditionalEnumeration", the value associated with it shall be different from the value of all previously defined "EnumerationItem"s (in this type) regardless of whether the previously defined "EnumerationItem"s occur in the enumeration root or not. For example:

```
A ::= ENUMERATED {a, b, ..., c(0)}   -- invalid, since both 'a' and 'c' equal 0
B ::= ENUMERATED {a, b, ..., c, d(2)}-- invalid, since both 'c' and 'd' equal 2
C ::= ENUMERATED {a, b(3), ..., c(1)}-- valid, 'c' = 1
D ::= ENUMERATED {a, b, ..., c(2)}   -- valid, 'c' = 2
```

19.6 The value associated with the first "EnumerationItem" in the "AdditionalEnumeration" alternative that is an "identifier" (not a "NamedNumber") shall be the smallest value for which an "EnumerationItem" is not defined in the "RootEnumeration" and all preceding "EnumerationItem"s in the "AdditionalEnumeration" (if any) are smaller. For example, the following are all valid:

```
A ::= ENUMERATED {a, b, ..., c}         -- c = 2
B ::= ENUMERATED {a, b, c(0), ..., d}   -- d = 3
C ::= ENUMERATED {a, b, ..., c(3), d}   -- d = 4
D ::= ENUMERATED {a, z(25), ..., d}     -- d = 1
```

19.7 The enumerated type has a tag which is universal class, number 10.

19.8 The value of an enumerated type shall be defined by the notation "EnumeratedValue", or when used as an "XMLValue", by the notation "XMLEnumeratedValue". These productions are:

EnumeratedValue ::= identifier

XMLEnumeratedValue ::= "<" & identifier "/>"

19.9 The "identifier" in "EnumeratedValue" and "XMLEnumeratedValue" shall be equal to that of an "identifier" in the "EnumeratedType" sequence with which the value is associated.

19.10 Within an instance of value notation for an enumerated type, any occurrence of a name that is both an "identifier" from the "Enumeration" and a reference name shall be interpreted as the "identifier".

20 Notation for the real type

20.1 The real type (see 3.6.54) shall be referenced by the notation "RealType":

RealType ::= `REAL`

20.2 The real type has a tag which is universal class, number 9.

20.3 The values of the real type are the values PLUS-INFINITY and MINUS-INFINITY together with the real numbers capable of being specified by the following formula involving three integers, M, B and E:

$$M \times B^{E}$$

where M is called the mantissa, B the base, and E the exponent.

20.4 The real type has an associated type which is used to give precision to the definition of the abstract values of the real type and is also used to support the value and subtype notations of the real type.

NOTE – Encoding rules may define a different type which is used to specify encodings, or may specify encodings without reference to the associated type. In particular, the encoding in BER and PER provides a Binary-Coded Decimal (BCD) encoding if "base" is 10, and an encoding which permits efficient transformation to and from hardware floating point representations if "base" is 2.

20.5 The associated type for value definition and subtyping purposes is (with normative comments):

```
SEQUENCE {
      mantissa   INTEGER,
      base  INTEGER (2|10),
      exponent   INTEGER
      -- The associated mathematical real number is "mantissa"
      -- multiplied by "base" raised to the power "exponent"
}
```

NOTE 1 – Non-zero values represented by "base" 2 and by "base" 10 are considered to be distinct abstract values even if they evaluate to the same real number value, and may carry different application semantics.

NOTE 2 – The notation **`REAL (WITH COMPONENTS { ... , base (10)})`** can be used to restrict the set of values to base 10 abstract values (and similarly for base 2 abstract values).

NOTE 3 – This type is capable of carrying an exact finite representation of any number which can be stored in typical floating point hardware, and of any number with a finite character-decimal representation.

20.6 The value of a real type shall be defined by the notation "RealValue", or when used in an "XMLValue", by the notation "XMLRealValue":

```
RealValue ::=
        NumericRealValue |
        SpecialRealValue

NumericRealValue ::=
        realnumber          |
        "-" realnumber      |
        SequenceValue       -- Value of the associated sequence type

SpecialRealValue ::=
        PLUS-INFINITY |
        MINUS-INFINITY
```

The second and third alternatives of "NumericRealValue" shall not be used for zero values.

```
XMLRealValue ::=
        XMLNumericRealValue | XMLSpecialRealValue

XMLNumericRealValue ::=
        realnumber          |
        "-" realnumber
```

The second alternative of "XMLNumericRealValue" shall not be used for zero values.

```
XMLSpecialRealValue ::=
        "<" & PLUS-INFINITY "/>" | "<" & MINUS-INFINITY "/>"
```

20.7 When the "realnumber" notation is used it identifies the corresponding "base" 10 abstract value. If the "RealType" is constrained to "base" 2, the "realnumber" identifies the "base" 2 abstract value corresponding either to the decimal value specified by the "realnumber" or to a locally-defined precision if an exact representation is not possible.

21 Notation for the bitstring type

21.1 The bitstring type (see 3.6.6) shall be referenced by the notation "BitStringType":

```
BitStringType ::=
        BIT STRING
        BIT STRING "{" NamedBitList "}"

NamedBitList ::=
        NamedBit            |
        NamedBitList "," NamedBit

NamedBit ::=
        identifier "(" number ")"  |
        identifier "(" DefinedValue ")"
```

21.2 The first bit in a bit string is called the leading bit. The final bit in a bit string is called the trailing bit.

NOTE – This terminology is used in specifying the value notation and in defining encoding rules.

21.3 The "DefinedValue" shall be a reference to a non-negative value of type integer.

21.4 The value of each "number" or "DefinedValue" appearing in the "NamedBitList" shall be different, and is the number of a distinguished bit in a bitstring value. The leading bit of the bit string is identified by the "number" zero, with succeeding bits having successive values.

21.5 Each "identifier" appearing in the "NamedBitList" shall be different.

NOTE 1 – The order of the "NamedBit" production sequences in the "NamedBitList" is not significant.

NOTE 2 – Since an "identifier" that appears within the "NamedBitList" cannot be used to specify the value associated with a "NamedBit", the "DefinedValue" can never be misinterpreted as an "IntegerValue". Therefore in the following case:

```
a  INTEGER ::= 1

T1 ::= INTEGER { a(2) }

T2 ::= BIT STRING { a(3), b(a) }
```

the last occurrence of **a** denotes the value 1, as it cannot be a reference to the second nor the third occurrence of **a**.

21.6 The presence of a "NamedBitList" has no effect on the set of abstract values of this type. Values containing 1 bits other than the named bits are permitted.

21.7 When a "NamedBitList" is used in defining a bitstring type ASN.1 encoding rules are free to add (or remove) arbitrarily any trailing 0 bits to (or from) values that are being encoded or decoded. Application designers should therefore ensure that different semantics are not associated with such values which differ only in the number of trailing 0 bits.

21.8 This type has a tag which is universal class, number 3.

21.9 The value of a bitstring type shall be defined by the notation "BitStringValue", or when used as an "XMLValue", by the notation "XMLBitStringValue". These productions are:

```
BitStringValue ::=
        bstring                 |
        hstring                 |
        "{" IdentifierList "}"  |
        "{" "}"                 |
        CONTAINING Value

IdentifierList ::=
            identifier   |
            IdentifierList "," identifier

XMLBitStringValue ::=
        XMLTypedValue       |
        Xmlbstring          |
        XMLIdentifierList   |
        empty

XMLIdentifierList ::=
        "<" & identifier "/>"      |
        XMLIdentifierList "<" & identifier "/>"
```

21.10 The "XMLTypedValue" alternative shall not be used unless the bitstring has a contents constraint which includes an ASN.1 type and does not include an **ENCODED BY**. If this alternative is used, the "XMLTypedValue" shall be a value of the ASN.1 type in the contents constraint.

21.11 The "XMLIdentifierList" alternative shall not be used unless the bitstring has a "NamedBitList".

21.12 Each "identifier" in "BitStringValue" or "XMLBitStringValue" shall be the same as an "identifier" in the "BitStringType" production sequence with which the value is associated.

21.13 The "empty" alternative denotes a bitstring with no bits.

21.14 If the bitstring has named bits, the "BitStringValue" or "XMLBitStringValue" notation denotes a bitstring value with ones in the bit positions specified by the numbers corresponding to the "identifier"s, and with all other bits zero.

NOTE – For a "BitStringType" that has a "NamedBitList", the "**{**" "**}**" production sequence in "BitStringValue" and the "empty" in "XMLBitStringValue" are used to denote the bitstring which contains no one bits.

21.15 When using the "bstring" or "xmlbstring" notation, the leading bit of the bitstring value is on the left, and the trailing bit of the bitstring value is on the right.

21.16 When using the "hstring" notation, the most significant bit of each hexadecimal digit corresponds to the leftmost bit in the bitstring.

NOTE – This notation does not, in any way, constrain the way encoding rules place a bitstring into octets for transfer.

21.17 The "hstring" notation shall not be used unless the bitstring value consists of a multiple of four bits.

EXAMPLE

'A98A'H

and

'1010100110001010'B

are alternative notations for the same bitstring value. If the type was defined using a "NamedBitList", the (single) trailing zero does not form part of the value, which is thus 15 bits in length. If the type was defined without a "NamedBitList", the trailing zero does form part of the value, which is thus 16 bits in length.

21.18 The **CONTAINING** alternative can only be used if there is a contents constraint on the bitstring type which includes **CONTAINING**. The "Value" shall then be value notation for a value of the "Type" in the "ContentsConstraint" (see ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 11).

NOTE – This value notation can never appear in a subtype constraint because ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 11.3 forbids further constraints after a "ContentsConstraint", and the above text forbids its use unless the governor has a "ContentsConstraint".

21.19 The **CONTAINING** alternative shall be used if there is a contents constraint on the bitstring type which does not contain **ENCODED BY**.

22 Notation for the octetstring type

22.1 The octetstring type (see 3.6.49) shall be referenced by the notation "OctetStringType":

OctetStringType ::= OCTET STRING

22.2 This type has a tag which is universal class, number 4.

22.3 The value of an octetstring type shall be defined by the notation "OctetStringValue", or when used as an "XMLValue", by the notation "XMLOctetStringValue". These productions are:

OctetStringValue ::=
bstring |
hstring |
CONTAINING Value

XMLOctetStringValue ::=
XMLTypedValue |
xmlhstring

22.4 The "XMLTypedValue" alternative shall not be used unless the octetstring has a contents constraint which includes an ASN.1 type and does not include an **ENCODED BY**. If this alternative is used, the "XMLTypedValue" shall be a value of the ASN.1 type in the contents constraint.

22.5 In specifying the encoding rules for an octetstring, the octets are referenced by the terms first octet and trailing octet, and the bits within an octet are referenced by the terms most significant bit and least significant bit.

22.6 When using the "bstring" notation, the left-most bit of the "bstring" notation shall be the most significant bit of the first octet of the octetstring value. If the "bstring" is not a multiple of eight bits, it shall be interpreted as if it contained additional zero trailing bits to make it the next multiple of eight.

22.7 When using the "hstring" or "xmlhstring" notation, the left-most hexadecimal digit shall be the most significant semi-octet of the first octet.

22.8 If the "hstring" is an odd number of hexadecimal digits, it shall be interpreted as if it contained a single additional trailing zero hexadecimal digit. The "xmlhstring" shall not be an odd number of hexadecimal digits.

22.9 The **CONTAINING** alternative can only be used if there is a contents constraint on the octetstring type which includes **CONTAINING**. The "Value" shall then be value notation for a value of the "Type" in the "ContentsConstraint" (see ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 11).

NOTE – This value notation can never appear in a subtype constraint because ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 11.3 forbids further constraints after a "ContentsConstraint", and the above text forbids its use unless the governor has a "ContentsConstraint".

22.10 The **CONTAINING** alternative shall be used if there is a contents constraint on the octetstring type which does not contain **ENCODED BY**.

23 Notation for the null type

23.1 The null type (see 3.6.44) shall be referenced by the notation "NullType":

NullType ::= NULL

23.2 This type has a tag which is universal class, number 5.

23.3 The value of a null type shall be referenced by the notation "NullValue", or when used as an "XMLValue", by the notation "XMLNullValue". These productions are:

NullValue ::= NULL

XMLNullValue ::= empty

24 Notation for sequence types

24.1 The notation for defining a sequence type (see 3.6.60) shall be the "SequenceType":

```
SequenceType ::=
        SEQUENCE "{" "}"                                                    |
        SEQUENCE "{" ExtensionAndException OptionalExtensionMarker "}"      |
        SEQUENCE "{" ComponentTypeLists "}"

ExtensionAndException ::= "..." | "..." ExceptionSpec

OptionalExtensionMarker ::= "," "..." | empty

ComponentTypeLists ::=
        RootComponentTypeList                                               |
        RootComponentTypeList "," ExtensionAndException ExtensionAdditions
            OptionalExtensionMarker                                         |
        RootComponentTypeList "," ExtensionAndException ExtensionAdditions
            ExtensionEndMarker "," RootComponentTypeList                    |
        ExtensionAndException ExtensionAdditions ExtensionEndMarker ","
            RootComponentTypeList                                           |
        ExtensionAndException ExtensionAdditions OptionalExtensionMarker

RootComponentTypeList ::= ComponentTypeList

ExtensionEndMarker ::= "," "..."

ExtensionAdditions ::=
        "," ExtensionAdditionList |
        empty

ExtensionAdditionList ::=
        ExtensionAddition |
        ExtensionAdditionList "," ExtensionAddition

ExtensionAddition ::=
        ComponentType |
        ExtensionAdditionGroup

ExtensionAdditionGroup ::= "[[" VersionNumber ComponentTypeList "]]"

VersionNumber ::= empty | number ":"

ComponentTypeList ::=
            ComponentType                   |
            ComponentTypeList "," ComponentType

ComponentType ::=
            NamedType                       |
            NamedType OPTIONAL              |
            NamedType DEFAULT Value         |
            COMPONENTS OF Type
```

24.2 When the "ComponentTypeLists" production occurs within the definition of a module for which automatic tagging is selected (see 12.3), and none of the occurrences of "NamedType" in any of the first three alternatives for "ComponentType" contains a "TaggedType", then automatic tagging transformation is selected for the entire "ComponentTypeLists", otherwise it is not.

NOTE 1 – The use of the "TaggedType" notation within the definition of the list of components for a sequence type gives control of tags to the specifier, as opposed to automatic assignment by the automatic tagging mechanism. Therefore, in the following case:

```
T ::= SEQUENCE { a INTEGER, b [1] BOOLEAN, c OCTET STRING }
```

no automatic tagging is applied to the list of components **a**, **b**, **c**, even if this definition of sequence type **T** occurs within a module for which automatic tagging is selected.

NOTE 2 – Only those occurrences of the "ComponentTypeLists" production appearing within a module where automatic tagging is selected are candidates for transformation by automatic tagging.

24.3 The decision to apply the automatic tagging transformation is taken individually for each occurrence of "ComponentTypeLists" and *prior* to the **COMPONENTS OF** transformation specified by 24.4. However, as specified in 24.7 to 24.9, the automatic tagging transformation (if applied) is applied *after* the **COMPONENTS OF** transformation.

NOTE – The effect of this is that the application of automatic tags is suppressed by tags explicitly present in the "ComponentTypeLists", but not by tags present in the "Type" following **COMPONENTS OF**.

24.4 "Type" in the "**COMPONENTS OF** Type" notation shall be a sequence type. The "**COMPONENTS OF** Type" notation shall be used to define the inclusion, at this point in the list of components, of all the component types of the referenced type, except for any extension marker and extension additions that may be present in the "Type". (Only the "RootComponentTypeList" of the "Type" in the "**COMPONENTS OF** Type" is included; extension markers and extension additions, if any, are ignored by the "**COMPONENTS OF** Type" notation.) Any subtype constraint applied to the referenced type is ignored by this transformation.

NOTE – This transformation is logically completed prior to the satisfaction of the requirements in the following subclauses.

24.5 The following subclauses each identify a series of occurrences of "ComponentType" in either the root or the extension additions or both. The rule of 24.5.1 shall apply to all such series.

24.5.1 Where there are one or more consecutive occurrences of "ComponentType" that are all marked **OPTIONAL** or **DEFAULT**, the tags of those "ComponentType"s and of any immediately following component type in the series shall be distinct (see clause 30). If automatic tagging was selected, the requirement that tags be distinct applies only after automatic tagging has been performed, and will always be satisfied.

24.5.2 Subclause 24.5.1 shall apply to the series of "ComponentType"s in the root.

24.5.3 Subclause 24.5.1 shall apply to the complete series of "ComponentType"s in the root or in the extension additions, in the textual order of their occurrence in the type definition (ignoring all version brackets and ellipsis notation). (See also 48.7.)

24.6 When the third or fourth alternative of "ComponentTypeLists" is used, all "ComponentType"s in extension additions shall have tags which are distinct from the tags of the textually following "ComponentType"s up to and including the first such "ComponentType" that is not marked **OPTIONAL** or **DEFAULT** in the trailing "RootComponentTypeList", if any. (See also 48.7.)

24.7 The automatic tagging transformation of an occurrence of "ComponentTypeLists" is logically performed *after* the transformation specified by 24.4, but only if 24.2 determines that it shall apply to that occurrence of "ComponentTypeLists". Automatic tagging transformation impacts each "ComponentType" of the "ComponentTypeLists" by replacing the "Type" originally in the "NamedType" production with a replacement "TaggedType" occurrence specified in 24.9.

24.8 If automatic tagging is in effect and the "ComponentType"s in the extension root have no tags, then no "ComponentType" within the "ExtensionAdditionList" shall be a "TaggedType".

24.9 If automatic tagging is in effect, the replacement "TaggedType" is specified as follows:

a) the replacement "TaggedType" notation uses the "Tag Type" alternative;

b) the "Class" of the replacement "TaggedType" is empty (i.e., tagging is context-specific);

c) the "ClassNumber" in the replacement "TaggedType" is tag value zero for the first "ComponentType" in the "RootComponentTypeList", one for the second, and so on, proceeding with increasing tag numbers;

d) the "ClassNumber" in the replacement "TaggedType" of the first "ComponentType" in the "ExtensionAdditionList" is zero if the "RootComponentTypeList" is missing, else it is one greater than the largest "ClassNumber" in the "RootComponentTypeList", with the next "ComponentType" in the "ExtensionAdditionList" having a "ClassNumber" one greater than the first, and so on, proceeding with increasing tag numbers;

e) the "Type" in the replacement "TaggedType" is the original "Type" being replaced.

NOTE 1 – The rules governing specification of implicit tagging or explicit tagging for replacement "TaggedType"s are provided by 30.6. Automatic tagging is always implicit tagging unless the "Type" is a choice type or an open type notation, or a "DummyReference" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.3), in which case it is explicit tagging.

NOTE 2 – Once 24.7 is satisfied, the tags of the components are completely determined, and are not modified even when the sequence type is referenced in the definition of a component within another "ComponentTypeLists" for which automatic tagging transformation applies. Thus, in the following case:

```
T ::= SEQUENCE { a   Ta,  b   Tb,  c   Tc }
E ::= SEQUENCE { f1  E1,  f2  T,   f3  E3 }
```

automatic tagging applied to the components of **E** never affects the tags attached to components **a**, **b** and **c** of **T**, whatever the tagging environment of **T**. If **T** is defined in an automatic tagging environment and **E** is not in an automatic tagging environment, automatic tagging is still applied to components **a**, **b** and **c** of **T**.

NOTE 3 – When a sequence type appears as the "Type" in "**COMPONENTS OF** Type", each occurrence of "ComponentType" in it is duplicated by the application of 24.4 prior to the possible application of automatic tagging to the referencing sequence type. Thus, in the following case:

```
T ::= SEQUENCE { a  Ta,  b  SEQUENCE { b1  T1,  b2  T2,  b3  T3},  c Tc }
W ::= SEQUENCE { x  Wx,  COMPONENTS OF  T, y  Wy }
```

the tags of **a**, **b**, and **c** within **T** need not be the same as the tags of **a**, **b**, and **c** within **W** if **W** has been defined in an automatic tagging environment, but the tags of **b1**, **b2** and **b3** are the same in both **T** and **W**. In other words, the automatic tagging transformation is only applied once to a given "ComponentTypeLists".

NOTE 4 – Subtyping has no impact on automatic tagging.

NOTE 5 – When automatic tagging is in place, insertion of new components at any location other than the extension insertion point (see 3.6.29) may result in changes to other components due to the side effect of modifying the tags thus causing interworking problems with an older version of the specification.

24.10 If **OPTIONAL** or **DEFAULT** are present, the corresponding value may be omitted from a value of the new type.

24.11 If **DEFAULT** occurs, the omission of a value for that type shall be exactly equivalent to the insertion of the value defined by "Value", which shall be a value notation for a value of the type defined by "Type" in the "NamedType" production sequence.

24.12 The value corresponding to an "ExtensionAdditionGroup" (all components together) is optional. However, if such a value is present, then the value corresponding to the components within the bracketed "ComponentTypeList" that are not marked **OPTIONAL** or **DEFAULT** shall be present.

24.13 The "identifier"s in all "NamedType" production sequences of the "ComponentTypeLists" (together with those obtained by expansion of **COMPONENTS OF**) shall all be distinct.

24.14 A value for a given extension addition type shall not be specified unless there are values specified for all extension addition types not marked **OPTIONAL** or **DEFAULT** that lie logically between the extension addition type and the extension root.

NOTE 1 – Where the type has grown from the extension root (version 1) through version 2 to version 3 by the addition of extension additions, the presence in an encoding of any addition from version 3 requires the presence of an encoding of all additions in version 2 that are not marked **OPTIONAL** or **DEFAULT**.

NOTE 2 – "ComponentType"s that are extension additions but not contained within an "ExtensionAdditionGroup" should always be encoded if they are not marked **OPTIONAL** or **DEFAULT**, except when the abstract value is being relayed from a sender that is using an earlier version of the abstract syntax in which the "ComponentType" is not defined.

NOTE 3 – Use of the "ExtensionAdditionGroup" production is recommended because:

a) it can result in more compact encodings depending on the encoding rules (e.g., PER);

b) the syntax is more precise in that it clearly indicates that a value of a type defined in the "ExtensionAdditionList" and not marked **OPTIONAL** or **DEFAULT** should always be present in an encoding if the extension addition group in which it is defined is encoded (compare with Note 1);

c) the syntax makes it clear which types in an "ExtensionAdditionList" must as a group be supported by an application.

24.15 A "VersionNumber" shall be used only if all "ExtensionAdditions"s within the module are "ExtensionAdditionGroup"s with "VersionNumber"s. The "number" in each "VersionNumber" of an "ExtensionAdditionGroup" shall be greater than or equal to two, and shall be greater than the "number" in any preceding "NumberedBracket" within an insertion point.

NOTE 1 – The convention used here is that the specification with no extension addition groups is version 1, thus the first added extension addition group will have a number greater than or equal to 2. Where a single "ExtensionAddition" is needed for an "ExtensionAdditions", an "ExtensionAdditionGroup" can be used with a single "ExtensionAddition".

NOTE 2 – The restrictions on use of "VersionNumber" apply only within a single module and impose no constraints on imported types.

24.16 All sequence types have a tag which is universal class, number 16.

NOTE – Sequence-of types have the same tag as sequence types (see 25.2).

24.17 The notation for defining a value of a sequence type shall be "SequenceValue", or when used as an "XMLValue", "XMLSequenceValue". These productions are:

SequenceValue ::=
"{" ComponentValueList "}" |
"{" "}"

ComponentValueList ::=
NamedValue |
ComponentValueList "," NamedValue

XMLSequenceValue ::=
XMLComponentValueList |
empty

XMLComponentValueList ::=
XMLNamedValue |
XMLComponentValueList XMLNamedValue

24.18 The "**{**" "**}**" or "empty" notation shall only be used if:

a) all "ComponentType" sequences in the "SequenceType" are marked **DEFAULT** or **OPTIONAL**, and all values are omitted; or

b) the type notation was **SEQUENCE{}**.

24.19 There shall be one "NamedValue" or "XMLNamedValue" for each "NamedType" in the "SequenceType" which is not marked **OPTIONAL** or **DEFAULT**, and the values shall be in the same order as the corresponding "NamedType" sequences.

25 Notation for sequence-of types

25.1 The notation for defining a sequence-of type (see 3.6.61) from another type shall be the "SequenceOfType".

SequenceOfType ::= SEQUENCE OF Type | SEQUENCE OF NamedType

NOTE – If an initial letter which is upper-case is needed for an XML tag name used in XML Value Notation for the "SequenceOfType", then the first alternative should be used. (The XML tag name is then formed from the name of the "Type".)

25.2 All sequence-of types have a tag which is universal class, number 16.

NOTE – Sequence types have the same tag as sequence-of types (see 24.16).

25.3 The notation for defining a value of a sequence-of type shall be the "SequenceOfValue", or when used as an "XMLValue", "XMLSequenceOfValue". These productions are:

SequenceOfValue ::=
"{" ValueList "}" |
"{" NamedValueList "}" |
"{" "}"

ValueList ::=
Value |
ValueList "," Value

NamedValueList ::=
NamedValue |
NamedValueList "," NamedValue

XMLSequenceOfValue ::=
XMLValueList |
XMLDelimitedItemList |
XMLSpaceSeparatedList |
empty

XMLValueList ::=
XMLValueOrEmpty |
XMLValueOrEmpty XMLValueList

```
XMLValueOrEmpty ::=
        XMLValue                        |
        "<" & NonParameterizedTypeName "/>"

XMLSpaceSeparatedList ::=
        XMLValueOrEmpty                 |
        XMLValueOrEmpty " " XMLSpaceSeparatedList

XMLDelimitedItemList ::=
        XMLDelimitedItem                |
        XMLDelimitedItem XMLDelimitedItemList

XMLDelimitedItem ::=
        "<" & NonParameterizedTypeName ">" XMLValue
            "</" & NonParameterizedTypeName ">" |
        "<" & identifier ">" XMLValue "</" & identifier ">"
```

The "**{**" "**}**" or "empty" notation is used when the "SequenceOfValue" or "XMLSequenceOfValue" is an empty list.

NOTE 1 – Semantic significance may be placed on the order of these values.

NOTE 2 – The "XMLSpaceSeparatedList" production is not used in this Recommendation | International Standard, and is not used in XML Value Notation. It is provided in order to allow specification of the use of "XMLSpaceSeparatedList" in encodings of the "IntegerType", "RealType", "ObjectIdentifierType", "RelativeOIDType", and the **GeneralizedTime** and **UTCTime** useful types. It is also possible to specify use of "XMLValueList" instead of "XMLDelimitedItemList" for some instances of "**SEQUENCE OF SEQUENCE**" and "**SEQUENCE OF SET**".

25.4 If the"XMLValue" for the component is "empty", then the second alternative of "XMLValueOrEmpty" shall be chosen to represent that value of the component.

25.5 The "XMLValueList" or "XMLDelimitedItemList" productions shall be used in accordance with column 2 of Table 5, where the "Type" of the component is listed in column 1.

Table 5 – "XMLSequenceOfValue" and "XMLSetOfValue" notation for ASN.1 types

ASN.1 type	XML value notation
BitStringType	XMLDelimitedItemList
BooleanType	XMLValueList
CharacterStringType	XMLDelimitedItemList
ChoiceType	XMLValueList
EmbeddedPDVType	XMLDelimitedItemList
EnumeratedType	XMLValueList
ExternalType	XMLDelimitedItemList
InstanceOfType	*See ITU-T Rec. X.681 | ISO/IEC 8824-2, C.9*
IntegerType	XMLDelimitedItemList
NullType	XMLValueList
ObjectClassFieldType	*See ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.10 and 14.11*
ObjectIdentifierType	XMLDelimitedItemList
OctetStringType	XMLDelimitedItemList
RealType	XMLDelimitedItemList
RelativeOIDType	XMLDelimitedItemList
SequenceType	XMLDelimitedItemList
SequenceOfType	XMLDelimitedItemList
SetType	XMLDelimitedItemList

SetOfType	XMLDelimitedItemList
TaggedType	*See 25.6*
UsefulType (GeneralizedTime)	XMLDelimitedItemList
UsefulType (UTCTime)	XMLDelimitedItemList
UsefulType (ObjectDescriptor)	XMLDelimitedItemList
TypeFromObject	*See ITU-T Rec. X.681 | ISO/IEC 8824-2, 15.6*
ValueSetFromObjects	*See ITU-T Rec. X.681 | ISO/IEC 8824-2, 15.6*

25.6 If the "Type" of the component is a "TaggedType" then the type which determines the "XMLSequenceOfValue" notation shall be the "Type" in the "TaggedType" (see 30.1). If this is itself a "TaggedType", then this sub-clause 25.6 shall be recursively applied.

25.7 If the "Type" of the component is a "ConstrainedType" then the type which determines the "XMLSequenceOfValue" notation shall be the "Type" in the "ConstrainedType" (see 45.1). If this is itself a "ConstrainedType", then this sub-clause 25.7 shall be recursively applied.

25.8 If the "Type" of the component is a "SelectionType" then the type which determines the "XMLSequenceOfValue" notation shall be the type referenced by the "SelectionType" (see clause 29).

25.9 The second alternative of "XMLDelimitedItem" shall be used if and only if the "SequenceOfType" contains an "identifier", and the "identifier" in the "XMLDelimitedItem" shall be that "identifier".

25.10 If the first alternative of "XMLDelimitedItem" is used, then if the component of the sequence-of type (after ignoring any tags) is a "typereference" or an "ExternalTypeReference", then the "NonParameterizedTypeName" shall be that "typereference" or "ExternalTypeReference", otherwise it shall be the "xmlasn1typename" specified in Table 4 corresponding to the built-in type of the component.

25.11 If the first alternative of "SequenceOfType" is used, then the first alternative of "SequenceOfValue" shall be used. Each "Value" in the "ValueList" of "SequenceOfValue", and each "XMLValue" in the alternatives of "XMLSequenceOfValue" shall be of the type specified in the "SequenceOfType".

25.12 If the second alternative of "SequenceOfType" is used, then the second alternative of "SequenceOfValue" shall be used, and each "NamedValue" in the "NamedValueList" shall contain a "Value" of the type specified in the "NamedType" of the "SequenceOfType". The "identifier" in the "NamedValue"s shall be the "identifier" in the "NamedType" of the "SequenceOfType".

26 Notation for set types

26.1 The notation for defining a set type (see 3.6.64) from other types shall be the "SetType":

```
SetType ::=
        SET "{" "}"                                                |
        SET "{" ExtensionAndException OptionalExtensionMarker "}"  |
        SET "{" ComponentTypeLists "}"
```

"ComponentTypeLists", "ExtensionAndException" and "OptionalExtensionMarker" are specified in 24.1.

26.2 "Type" in the "**COMPONENTS OF** Type" notation shall be a set type. The "**COMPONENTS OF** Type" notation shall be used to define the inclusion, at this point in the list of components, of all the component types of the referenced type, except for any extension marker and extension additions that may be present in the "Type". (Only the "RootComponentTypeList" of the "Type" in the "**COMPONENTS OF** Type" is included; extension markers and extension additions, if any, are ignored by the "**COMPONENTS OF** Type" notation.) Any subtype constraint applied to the referenced type is ignored by this transformation.

NOTE – This transformation is logically completed prior to the satisfaction of the requirements in the following subclauses.

26.3 The "ComponentType" types in a set type shall all have different tags (see clause 30). The tag of each new "ComponentType" added to the "ExtensionAdditions" shall be canonically greater (see 8.6) than those of the other components in the "ExtensionAdditions".

NOTE – Where the "TagDefault" for the module in which this notation appears is **AUTOMATIC TAGS**, this is achieved regardless of the actual "ComponentType"s, as a result of the application of 24.7. (See also 48.7.)

26.4 Subclauses 24.2 and 24.7 to 24.13 also apply to set types.

26.5 All set types have a tag which is universal class, number 17.

NOTE – Set-of types have the same tag as set types (see 27.2).

26.6 There shall be no semantics associated with the order of values in a set type.

26.7 The notation for defining the value of a set type shall be "SetValue", or when used as an "XMLValue", "XMLSetValue". These productions are:

```
SetValue ::=
        "{" ComponentValueList "}"   |
        "{" "}"

XMLSetValue ::=
        XMLComponentValueList        |
        empty
```

"ComponentValueList" and "XMLComponentValueList" are specified in 24.17.

26.8 The "SetValue" and "XMLSetValue" shall only be "**{**" "**}**" and "empty" respectively if:

a) all "ComponentType" sequences in the "SetType" are marked **DEFAULT** or **OPTIONAL**, and all values are omitted; or

b) the type notation was **SET{}**.

26.9 There shall be one "NamedValue" or "XMLNamedValue" for each "NamedType" in the "SetType" which is not marked **OPTIONAL** or **DEFAULT**.

NOTE – These "NamedValue"s or "XMLNamedValue"s may appear in any order.

27 Notation for set-of types

27.1 The notation for defining a set-of type (see 3.6.65) from another type shall be the "SetOfType":

```
SetOfType ::=
        SET OF Type          |
        SET OF NamedType
```

NOTE – If an initial letter which is upper-case is needed for an XML tag name used in XML Value Notation for the "SetOfType", then the first alternative should be used. (The XML tag name is then formed from the name of the "Type".)

27.2 All set-of types have a tag which is universal class, number 17.

NOTE – Set types have the same tag as set-of types (see 26.5).

27.3 The notation for defining a value of a set-of type shall be the "SetOfValue", or when used as an "XMLValue", "XMLSetOfValue". These productions are:

```
SetOfValue ::=
        "{" ValueList "}"            |
        "{" NamedValueList "}"       |
        "{" "}"

XMLSetOfValue ::=
        XMLValueList                 |
        XMLDelimitedItemList         |
        XMLSpaceSeparatedList        |
        empty
```

"ValueList", "NamedValueList" and the alternatives of "XMLSetOfValue" are specified in 25.3. The "**{**" "**}**" or "empty" notation is used when the "SetOfValue" or "XMLSetOfValue" is an empty list.

NOTE 1 – Semantic significance should not be placed on the order of these values.

NOTE 2 – Encoding rules are not required to preserve the order of these values.

NOTE 3 – The set-of type is not a mathematical set of values, thus, as an example, for **SET OF INTEGER** the values **{ 1 }** and **{ 1 1 }** are distinct.

27.4 If the first alternative of "SetOfType" is used, then the first alternative of "SetOfValue" shall be used. Each "Value" in the "ValueList" of "SetOfValue", and each "XMLValue" in the alternatives of "XMLSetOfValue" shall be of the type specified in the "SetOfType".

27.5 If the second alternative of "SetOfType" is used, then the second alternative of "SetOfValue" shall be used, and each "NamedValue" sequence in the "NamedValueList" shall contain a "Value" of the type specified in the "NamedType" of the "SetOfType". The "identifier" in the "NamedValue"s shall be the "identifier" in the "NamedType" of the "SetOfType".

28 Notation for choice types

28.1 The notation for defining a choice type (see 3.6.13) from other types shall be the "ChoiceType":

```
ChoiceType ::= CHOICE "{" AlternativeTypeLists "}"

AlternativeTypeLists ::=
        RootAlternativeTypeList                 |
        RootAlternativeTypeList "," 
            ExtensionAndException ExtensionAdditionAlternatives
            OptionalExtensionMarker

RootAlternativeTypeList ::= AlternativeTypeList

ExtensionAdditionAlternatives ::=
        "," ExtensionAdditionAlternativesList |
        empty

ExtensionAdditionAlternativesList ::=

        ExtensionAdditionAlternative |
        ExtensionAdditionAlternativesList "," ExtensionAdditionAlternative

ExtensionAdditionAlternative ::=
        ExtensionAdditionAlternativesGroup |
        NamedType

ExtensionAdditionAlternativesGroup ::= "[[" VersionNumber AlternativeTypeList "]]"

AlternativeTypeList ::=
        NamedType |
        AlternativeTypeList "," NamedType
```

NOTE – "`T ::= CHOICE { a A }`" and `A` are not the same type, and may be encoded differently by encoding rules.

28.2 When the "AlternativeTypeLists" production occurs within the definition of a module for which automatic tagging is selected (see 12.3), and none of the occurrences of "NamedType" in any "AlternativeTypeList" contains a "TaggedType", the automatic tagging transformation is selected for the entire "AlternativeTypeLists", otherwise it is not.

28.3 The types defined in the "AlternativeTypeList" productions in an "AlternativeTypeLists" shall have distinct tags (see clause 30, and 48.7). If automatic tagging was selected, the requirement that tags be distinct applies only after automatic tagging has been performed, and will always be satisfied.

28.4 If automatic tagging is in effect and the "NamedType"s in the extension root have no tags, then no "NamedType" within the "ExtensionAdditionAlternativesList" shall be a tagged type.

28.5 The automatic tagging transformation impacts each "NamedType" of the "AlternativeTypeLists" by replacing the "Type" originally in the "NamedType" production with a replacement "TaggedType". The replacement "TaggedType" is specified as follows:

a) the replacement "TaggedType" notation uses the "Tag Type" alternative;

b) the "Class" of the replacement "TaggedType" is empty (i.e., tagging is context-specific);

c) the "ClassNumber" in the replacement "TaggedType" is tag value zero for the first "NamedType" in the "RootAlternativeTypeList", one for the second, and so on, proceeding with increasing tag numbers;

d) the "ClassNumber" in the replacement "TaggedType" of the first "NamedType" in the "ExtensionAdditionAlternativesList" is one greater than the largest "ClassNumber" in the "RootAlternativeTypeList", with the next "NamedType" in the "ExtensionAdditionAlternativesList" having a "ClassNumber" one greater than the first, and so on, proceeding with increasing tag numbers;

e) the "Type" in the replacement "TaggedType" is the original "Type" being replaced.

NOTE 1 - The rules governing specification of implicit tagging or explicit tagging for replacement "TaggedType"s are provided by 30.6. Automatic tagging is always implicit tagging unless the "Type" is an untagged choice type or an untagged open type notation, or an untagged "DummyReference" (see ITU-T Rec. X683 | ISO/IEC 8824-4, 8.3), in which case it is explicit tagging.

NOTE 2 - Once automatic tagging has been applied, the tags of the components are completely determined, and are not modified even when the choice type is referenced in the definition of an alternative within another "AlternativeTypeLists" for which automatic tagging transformation applies. Thus, in the following case:

```
T ::= CHOICE { a Ta, b Tb, c Tc }
E ::= CHOICE {f1 E1, f2 T, f3 E3}
```

automatic tagging applied to the components of **E** never affects the tags attached to components **a**, **b** and **c** of **T**, whatever the tagging environment of **T**. If **T** is defined in an automatic tagging environment and **E** is not in an automatic tagging environment, automatic tagging is still applied to components **a**, **b** and **c** of **T**.

NOTE 3 - Subtyping does not affect automatic tagging.

NOTE 4 - When automatic tagging is in place, insertion of new alternatives at any location other than the extension insertion point (see 3.6.29) may result in changes to other alternatives due to the side effect of modifying the tags thus causing interworking problems with an older version of the specification.

28.5 "VersionNumber" is defined in 24.1, and the restrictions on consistent use of "VersionNumber" throughout a module that are specified in 24.15 shall apply to the use of "number"s within this production.

28.6 The tag of each new "NamedType" added to the "ExtensionAdditionAlternativesList" shall be canonically greater (see 8.6) than those of the other alternatives in the "ExtensionAdditionAlternativesList", and shall be the last "NamedType" in the "ExtensionAdditionAlternativesList".

28.7 The choice type contains values which do not all have the same tag. (The tag depends on the alternative which contributed the value to the choice type.)

28.8 When this type does not have an extension marker and is used in a place where this Recommendation | International Standard requires the use of types with distinct tags (see 28.3), all possible tags of values of the choice type shall be considered in such requirement. The following examples which assume that the "TagDefault" is not **AUTOMATIC TAGS** illustrate this requirement.

EXAMPLES

1
```
A ::= CHOICE {
      b          B,
      c          NULL}

B ::= CHOICE {
      d          [0] NULL,
      e          [1] NULL}
```

2
```
A ::= CHOICE {
      b          B,
      c          C}

B ::= CHOICE {
      d          [0] NULL,
      e          [1] NULL}

C ::= CHOICE {
      f          [2] NULL,
      g          [3] NULL}
```

3 **(INCORRECT)**
```
A ::= CHOICE {
      b          B,
      c          C}

B ::= CHOICE {
      d          [0] NULL,
      e          [1] NULL}

C ::= CHOICE {
      f          [0] NULL,
      g          [1] NULL}
```

Examples 1 and 2 are correct uses of the notation. Example 3 is incorrect without automatic tagging, as the tags for types **d** and **f** are identical, as well as for **e** and **g**.

28.9 The "identifier"s of all "NamedType"s in the "AlternativeTypeLists" shall differ from those of the other "NamedType"s in that list.

28.10 The notation for defining the value of a choice type shall be the "ChoiceValue", or when used as an "XMLValue", "XMLChoiceValue". These productions are:

ChoiceValue ::= identifier ":" Value

XMLChoiceValue ::= "<" & identifier ">" XMLValue "</" & identifier ">"

28.11 "Value" or "XMLValue" shall be a notation for a value of the type in the "AlternativeTypeLists" that is named by the "identifier".

29 Notation for selection types

29.1 The notation for defining a selection type (see 3.6.59) shall be "SelectionType":

SelectionType ::= identifier "<" Type

where "Type" denotes a choice type, and "identifier" is that of some "NamedType" appearing in the "AlternativeTypeLists" of the definition of that choice type.

29.2 When "Type" denotes a constrained type, the selection is performed on the parent type, ignoring any subtype constraint on the parent type.

29.3 Where the "SelectionType" is used as a "NamedType", the "identifier" of the "NamedType" is present, as well as the "identifier" of the "SelectionType".

29.4 Where the "SelectionType" is used as a "Type", the "identifier" is retained and the type denoted is that of the selected alternative.

29.5 The notation for a value of a selection type shall be the notation for a value of the type referenced by the "SelectionType".

30 Notation for tagged types

A tagged type (see 3.6.70) is a new type which is isomorphic with an old type, but which has a different tag. The tagged type is mainly of use where this Recommendation | International Standard requires the use of types with distinct tags (see 24.5 to 24.6, 26.3 and 28.3). The use of a "TagDefault" of **AUTOMATIC TAGS** in a module allows this to be accomplished without the explicit appearance of tagged type notation in that module.

NOTE – Where a protocol determines that values from several data types may be transmitted at any moment in time, distinct tags may be needed to enable the recipient to correctly decode the value.

30.1 The notation for a tagged type shall be "TaggedType":

```
TaggedType ::=
        Tag Type              |
        Tag IMPLICIT Type     |
        Tag EXPLICIT Type

Tag ::= "[" Class ClassNumber "]"

ClassNumber ::=
        number |
        DefinedValue

Class ::=
        UNIVERSAL       |
        APPLICATION     |
        PRIVATE         |
        empty
```

30.2 The "valuereference" in "DefinedValue" shall be of type integer, and assigned a non-negative value.

30.3 The new type is isomorphic with the old type, but has a tag with class "Class" and number "ClassNumber", except when "Class" is "empty", in which case the tag is context-specific class and number is "ClassNumber".

30.4 The "Class" shall not be **UNIVERSAL** except for types defined in this Recommendation | International Standard.

NOTE 1 – Use of universal class tags are agreed from time to time by ITU-T and ISO.

NOTE 2 – Subclause E.2.12 contains guidance and hints on stylistic use of tag classes.

30.5 All application of tags is either implicit tagging or explicit tagging. Implicit tagging indicates, for those encoding rules which provide the option, that explicit identification of the original tag of the "Type" in the "TaggedType" is not needed during transfer.

NOTE – It can be useful to retain the old tag where this was universal class, and hence unambiguously identifies the old type without knowledge of the ASN.1 definition of the new type. Minimum transfer octets is, however, normally achieved by the use of **IMPLICIT**. An example of an encoding using **IMPLICIT** is given in ITU-T Rec. X.690 | ISO/IEC 8825-1.

30.6 The tagging construction specifies explicit tagging if any of the following holds:

a) the "Tag **EXPLICIT** Type" alternative is used;

b) the "Tag Type" alternative is used and the value of "TagDefault" for the module is either **EXPLICIT TAGS** or is empty;

c) the "Tag Type" alternative is used and the value of "TagDefault" for the module is **IMPLICIT TAGS** or **AUTOMATIC TAGS**, but the type defined by "Type" is an untagged choice type, an untagged open type, or an untagged "DummyReference" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.3).

The tagging construction specifies implicit tagging otherwise.

30.7 If the "Class" is "empty", there are no restrictions on the use of "Tag", other than those implied by the requirement for distinct tags in 24.5 to 24.6, 26.3 and 28.3.

30.8 The **IMPLICIT** alternative shall not be used if the type defined by "Type" is an untagged choice type or an untagged open type or an untagged "DummyReference" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.3).

30.9 The notation for a value of a "TaggedType" shall be "TaggedValue", or when used as an "XMLValue", "XMLTaggedValue". These productions are:

TaggedValue ::= Value

XMLTaggedValue ::= XMLValue

where "Value" or "XMLValue" is a notation for a value of the "Type" in the "TaggedType".

NOTE – The "Tag" does not appear in this notation.

31 Notation for the object identifier type

31.1 The object identifier type (see 3.6.48) shall be referenced by the notation "ObjectIdentifierType":

```
ObjectIdentifierType ::=
                OBJECT IDENTIFIER
```

31.2 This type has a tag which is universal class, number 6.

31.3 The value notation for an object identifier shall be "ObjectIdentifierValue", or when used as an "XMLValue", "XMLObjectIdentifierValue". These productions are:

```
ObjectIdentifierValue ::=
        "{" ObjIdComponentsList "}"       |
        "{" DefinedValue ObjIdComponentsList "}"

ObjIdComponentsList ::=
        ObjIdComponents                   |
        ObjIdComponents ObjIdComponentsList

ObjIdComponents ::=
        NameForm                  |
        NumberForm                |
        NameAndNumberForm         |
        DefinedValue

NameForm ::= identifier

NumberForm ::= number | DefinedValue

NameAndNumberForm ::=
        identifier "(" NumberForm ")"
```

```
XMLObjectIdentifierValue ::=
        XMLObjIdComponentList

XMLObjIdComponentList ::=
        XMLObjIdComponent |
        XMLObjIdComponent & "." & XMLObjIdComponentList

XMLObjIdComponent ::=
        NameForm              |
        XMLNumberForm         |
        XMLNameAndNumberForm

XMLNumberForm ::= number

XMLNameAndNumberForm ::=
        identifier & "(" & XMLNumberForm & ")"
```

31.4 The "valuereference" in "DefinedValue" of "NumberForm" shall be of type integer, and assigned a non-negative value.

31.5 The "valuereference" in "DefinedValue" of "ObjectIdentifierValue" shall be of type object identifier.

31.6 The "DefinedValue" of "ObjIdComponents" shall be of type relative object identifier, and shall identify an ordered set of arcs from some starting node in the object identifier tree to some later node in the object identifier tree. The starting node is identified by the earlier "ObjIdComponents"s, and later "ObjIdComponents"s (if any) identify arcs from the later node. The starting node is required to be neither the root, nor a node immediately beneath the root.

NOTE – A relative object identifier value has to be associated with a specific object identifier value so as to unambiguously identify an object. Object identifier values are required (see 31.10) to have at least two components. This is why there is a restriction on the starting node.

31.7 The "NameForm" shall be used only for those object identifier components whose numeric value and identifier are specified in ITU-T Rec. X.660 | ISO/IEC 9834-1, Annexes A to C (see also Annex D of this Recommendation | International Standard), and shall be one of the identifiers specified in ITU-T Rec. X.660 | ISO/IEC 9834-1, Annexes A to C. Where ITU-T Rec. X.660 | ISO/IEC 9834-1 specifies synonymous identifiers, any synonym may be used with the same semantics. Where the same name is both an identifier specified in ITU-T Rec. X.660 | ISO/IEC 9834-1 and an ASN.1 value reference within the module containing the "NameForm", the name within the object identifier value shall be treated as an ITU-T Rec. X.660 | ISO/IEC 9834-1 identifier.

31.8 The "number" in the "NumberForm" and "XMLNumberForm" shall be the numeric value assigned to the object identifier component.

31.9 The "identifier" in the "NameAndNumberForm" and "XMLNameAndNumberForm" shall be specified when a numeric value is assigned to the object identifier component.

NOTE – The authorities allocating numeric values to object identifier components are identified in ITU-T Rec. X.660 | ISO/IEC 9834-1.

31.10 The semantics associated with an object identifier value are specified in ITU-T Rec. X.660 | ISO/IEC 9834-1.

NOTE – ITU-T Rec. X.660 | ISO/IEC 9834-1 requires that an object identifier value shall contain at least two arcs.

31.11 The significant part of the object identifier component is the "NameForm" or "NumberForm" or "XMLNumberForm" which it reduces to, and which provides the numeric value for the object identifier component. Except for the arcs specified in ITU-T Rec. X.660 | ISO/IEC 9834-1, Annexes A to C (see also Annex D of this Recommendation | International Standard), the numeric value of the object identifier component is always present in an instance of object identifier value notation.

31.12 Where the "ObjectIdentifierValue" includes a "DefinedValue" for an object identifier value, the list of object identifier components to which it refers is prefixed to the components explicitly present in the value.

NOTE – ITU-T Rec. X.660 | ISO/IEC 9834-1 recommends that whenever an object identifier value is assigned to identify an object, an object descriptor value is also assigned.

EXAMPLES

With identifiers assigned as specified in ITU-T Rec. X.660 | ISO/IEC 9834-1, the values:

```
{ iso standard 8571 pci (1) }
```

and

```
{ 1 0 8571 1 }
```

would each identify an object, **pci**, defined in ISO 8571, as would

```
iso.standard.8571.pci(1)
```

and

```
1.0.8571.1
```

in an "XMLObjectIdentifierValue".

With the following additional definition:

```
ftam OBJECT IDENTIFIER ::= { iso standard 8571 }
```

the following value is equivalent to those above:

```
{ ftam pci(1) }
```

32 Notation for the relative object identifier type

32.1 The relative object identifier type (see 3.6.57) shall be referenced by the notation "RelativeOIDType":

RelativeOIDType ::=
RELATIVE-OID

32.2 This type has a tag which is universal class, number 13.

32.3 The value notation for a relative object identifier shall be "RelativeOIDValue", or when used as "XMLValue", "XMLRelativeOIDValue". These productions are:

RelativeOIDValue ::=
"{" RelativeOIDComponentsList "}"

RelativeOIDComponentsList ::=
RelativeOIDComponents |
RelativeOIDComponents RelativeOIDComponentsList

RelativeOIDComponents ::=
NumberForm |
NameAndNumberForm |
DefinedValue

XMLRelativeOIDValue ::=
XMLRelativeOIDComponentList

XMLRelativeOIDComponentList ::=
XMLRelativeOIDComponent |
XMLRelativeOIDComponent & "." & XMLRelativeOIDComponentList

XMLRelativeOIDComponent ::=
XMLNumberForm |
XMLNameAndNumberForm

32.4 The productions "NumberForm", "NameAndNumberForm", "XMLNumberForm", "XMLNameAndNumberForm", and their semantics, are defined in sub-clauses 31.3 to 31.11.

32.5 The "DefinedValue" of "RelativeOIDComponents" shall be of type relative object identifier, and shall identify an ordered set of arcs from some starting node in the object identifier tree to some later node in the object identifier tree. The starting node is identified by the earlier "RelativeOIDComponents"s (if any), and later "RelativeOIDComponents"s (if any) identify arcs from the later nodes.

32.6 The first "RelativeOIDComponents" or "XMLRelativeOIDComponent" identifies one or more arcs from some starting node in the object identifier tree to some later node in the object identifier tree. The starting point can be defined by comments associated with the type definition. If there is no definition of the starting node within comments associated with the type definition, then it needs to be transmitted as an object identifier value in an instance of communication (see E.2.19). The starting node is required to be neither the root, nor a node immediately beneath the root.

NOTE – A relative object identifier value has to be associated with a specific object identifier value so as to unambiguously identify an object. Object identifier values are required (see 31.10) to have at least two components. This is why there is a restriction on the starting node.

EXAMPLE

With the following definitions:

```
thisUniversity  OBJECT IDENTIFIER       ::=
           {iso member-body country(29) universities(56) thisuni(32)}

firstgroup   RELATIVE-OID   ::=  {science-fac(4) maths-dept(3)}
```

or in XML value notation:

```
thisUniversity ::= <OBJECT_IDENTIFIER>1.2.29.56.32</OBJECT_IDENTIFIER>

firstgroup ::= <RELATIVE_OID>4.3</RELATIVE_OID>
```

the relative object identifier:

```
relOID   RELATIVE-OID  ::=  {firstgroup room(4) socket(6)}
```

or in XML value notation:

```
relOID ::= <RELATIVE_OID>4.3.4.6</RELATIVE_OID>
```

can be used instead of the **OBJECT IDENTIFIER** value **{1 2 29 56 32 4 3 4 6}** if the current root (known by the application or transmitted by the application) is **thisUniversity**.

33 Notation for the embedded-pdv type

33.1 The embedded-pdv type (see 3.6.21) shall be referenced by the notation "EmbeddedPDVType":

EmbeddedPDVType ::= EMBEDDED PDV

NOTE – The term "Embedded PDV" means an abstract value from a possibly different abstract syntax (essentially, the value and encoding of a message defined in a separate – and identified - protocol) that is embedded in a message. Historically, it meant "Embedded Presentation Data Value" from its use in the OSI Presentation Layer, but this expansion is not used today, and it should be interpreted as "embedded value".

33.2 This type has a tag which is universal class, number 11.

33.3 The type consists of values representing:

a) an encoding of a single data value that may, but need not, be the value of an ASN.1 type; and

b) identification (separately or together) of:

1) an abstract syntax; and

2) the transfer syntax.

NOTE 1 – The data value may be the value of an ASN.1 type, or may, for example, be the encoding of a still image or a moving picture. The identification consists of either one or two object identifiers, or (in an OSI environment) references an OSI presentation context identifier which specifies the abstract and transfer syntaxes.

NOTE 2 – The identification of the abstract syntax and/or the encoding may also be determined by the application designer as a fixed value, in which case it is not encoded in an instance of communication.

33.4 The embedded-pdv type has an associated type. This associated type is used to support the value and subtype notations of the embedded-pdv type.

33.5 The associated type for value definition and subtyping, assuming an automatic tagging environment, is (with normative comments):

```
SEQUENCE {
     identification                    CHOICE {
          syntaxes                     SEQUENCE {
               abstract                OBJECT IDENTIFIER,
               transfer                OBJECT IDENTIFIER }
```

```
        -- Abstract and transfer syntax object identifiers --,

        syntax                          OBJECT IDENTIFIER
        -- A single object identifier for identification of the abstract
        -- and transfer syntaxes --,

        presentation-context-id         INTEGER
        -- (Applicable only to OSI environments)
        -- The negotiated OSI presentation context identifies the
        -- abstract and transfer syntaxes --,

        context-negotiation             SEQUENCE {
              presentation-context-id   INTEGER,
              transfer-syntax           OBJECT IDENTIFIER }
        -- (Applicable only to OSI environments)
        -- Context-negotiation in progress, presentation-context-id
        -- identifies only the abstract syntax
        -- so the transfer syntax shall be specified --,

        transfer-syntax                 OBJECT IDENTIFIER
        -- The type of the value (for example, specification that it is
        -- the value of an ASN.1 type)
        -- is fixed by the application designer (and hence known to both
        -- sender and receiver). This
        -- case is provided primarily to support
        -- selective-field-encryption (or other encoding
        -- transformations) of an ASN.1 type --,

        fixed                           NULL
        -- The data value is the value of a fixed ASN.1 type (and hence
        -- known to both sender
        -- and receiver) -- },

    data-value-descriptor               ObjectDescriptor  OPTIONAL
    -- This provides human-readable identification of the class of the
    -- value --,
    data-value                          OCTET STRING }

( WITH COMPONENTS {
        ... ,
        data-value-descriptor  ABSENT } )
```

NOTE – The embedded-pdv type does not allow the inclusion of a **data-value-descriptor** value. However, the definition of the associated type provided here underlies the commonalities which exist between the embedded-pdv type, the external type and the unrestricted character string type.

33.6 The **presentation-context-id** alternative is only applicable in an OSI environment, when the integer value shall be an OSI presentation context identifier in the OSI defined context set. This alternative shall not be used during OSI context negotiation.

33.7 The **context-negotiation** alternative is only applicable in an OSI environment, and shall only be used during OSI context negotiation. The integer value shall be an OSI presentation context identifier proposed for addition to the OSI defined context set. The object identifier **transfer-syntax** shall identify a proposed transfer syntax for that OSI presentation context which is to be used to encode the value.

33.8 The notation for a value of the embedded-pdv type shall be the value notation for the associated type defined in 33.5, where the value of the **data-value** component of type **OCTET STRING** represents an encoding using the transfer syntax specified in **identification**.

EmbeddedPdvValue ::= SequenceValue -- *value of associated type defined in 33.5*

XMLEmbeddedPDVValue ::= XMLSequenceValue -- *value of associated type defined in 33.5*

EXAMPLE – If a single option is to be enforced, such as use of **syntaxes**, then this can be done by writing:

```
EMBEDDED PDV (WITH COMPONENTS {
                ... ,
                identification  (WITH COMPONENTS {
                      syntaxes  PRESENT } ) } )
```

34 Notation for the external type

34.1 The external type (see 3.6.37) shall be referenced by the notation "ExternalType":

ExternalType ::= EXTERNAL

34.2 This type has a tag which is universal class, number 8.

34.3 The type consists of values representing:

a) an encoding of a single data value that may, but need not, be the value of an ASN.1 type; and

b) identification of:

1) an abstract syntax; and

2) the transfer syntax; and

c) (optionally) an object descriptor which provides a human-readable description of the category of the data value. The optional object descriptor shall not be present unless explicitly permitted by comment associated with use of the "ExternalType" notation.

NOTE – Note 1 on 33.3 also applies to the external type.

34.4 The external type has an associated type. This type is used to give precision to the definition of the abstract values of the external type and is also used to support the value and subtype notations of the external type.

NOTE – Encoding rules may define a different type which is used to derive encodings, or may specify encodings without reference to any associated type. For example, the encoding in BER uses a different sequence type for historical reasons.

34.5 The associated type for value definition and subtyping, assuming an automatic tagging environment, is (with normative comments):

```
SEQUENCE {
     identification                         CHOICE {
          syntaxes                          SEQUENCE {
               abstract                     OBJECT IDENTIFIER,
               transfer                     OBJECT IDENTIFIER }
          -- Abstract and transfer syntax object identifiers --,

          syntax                            OBJECT IDENTIFIER
          -- A single object identifier for identification of the abstract
          -- and transfer syntaxes --,

          presentation-context-id           INTEGER
          -- (Applicable only to OSI environments)
          -- The negotiated OSI presentation context identifies the
          -- abstract and transfer syntaxes --,

          context-negotiation               SEQUENCE {
               presentation-context-id      INTEGER
               transfer-syntax              OBJECT IDENTIFIER }
          -- (Applicable only to OSI environments)
          -- Context-negotiation in progress, presentation-context-id
          -- identifies only the abstract syntax
          -- so the transfer syntax shall be specified --,

          transfer-syntax                   OBJECT IDENTIFIER
          -- The type of the value (for example, specification that it is
          -- the value of an ASN.1 type)
          -- is fixed by the application designer (and hence known to both
          -- sender and receiver). This
          -- case is provided primarily to support
          -- selective-field-encryption (or other encoding
          -- transformations) of an ASN.1 type --,

          fixed                             NULL
          -- The data value is the value of a fixed ASN.1 type (and hence
          -- known to both sender
          -- and receiver) -- },

     data-value-descriptor                  ObjectDescriptor  OPTIONAL
          -- This provides human-readable identification of the class of
          -- the value --,

     data-value                             OCTET STRING }
          ( WITH COMPONENTS {
               ... ,
```

```
identification  (WITH COMPONENTS {
      ... ,
      syntaxes                    ABSENT,
      transfer-syntax             ABSENT,
      fixed                       ABSENT } ) } )
```

NOTE – For historical reasons, the external type does not allow the **syntaxes**, **transfer-syntax** or **fixed** alternatives of **identification**. Application designers requiring these options should use the embedded-pdv type. The definition of the associated type provided here underlies the commonalities which exist between the external type, the unrestricted character string type and the embedded-pdv type.

34.6 The text of 33.6 and 33.7 also applies to the external type.

34.7 The notation for a value of the external type shall be the value notation for the associated type defined in 34.5, where the value of the **data-value** component of type **OCTET STRING** represents an encoding using the transfer syntax specified in **identification**.

ExternalValue ::= SequenceValue -- *value of associated type defined in 34.5*

XMLExternalValue ::= XMLSequenceValue -- *value of associated type defined in 34.5*

NOTE – For historical reasons, encoding rules are able to transfer embedded values in **EXTERNAL** whose encodings are not an exact multiple of eight bits. Such values cannot be represented in value notation using the above associated type.

35 The character string types

These types consist of strings of characters from some specified character repertoire. It is normal to define a character repertoire and its encoding by use of cells in one or more tables, each cell corresponding to a character in the repertoire. A graphic symbol and a character name are also usually assigned to each cell, although in some repertoires, cells are left empty, or have names but no shapes (examples of cells with names but no shape include control characters such as EOF in ISO/IEC 646 and spacing characters such as THIN-SPACE and EN-SPACE in ISO/IEC 10646-1).

In general, the information associated with a cell denotes a distinct abstract character in the repertoire even if that information is null (no graphic symbol or name is assigned to that cell).

The ASN.1 basic value notation for character string types has three variants (which can be combined), specified formally below:

a) A representation of the characters in the string using assigned graphic symbols, possibly including spacing characters; this is the "cstring" notation.

   NOTE 1 – Such a representation can be ambiguous in a printed representation when the same graphic symbol is used for more than one character in the repertoire.

   NOTE 2 – Such a representation can be ambiguous in a printed representation when spacing characters of different widths are present in the repertoire or the specification is printed with a proportional-spacing font.

b) A listing of the characters in the character string value by giving a series of ASN.1 value references that have been assigned the character; a set of such value references is defined in the module ASN1-CHARACTER-MODULE in clause 38 for the ISO/IEC 10646-1 character repertoire and for the IA5String character repertoire; this form is not available for other character repertoires unless the user assigns to such value references using the value notation described in a) above or c) below.

c) A listing of the characters in the character string value by identifying each abstract character by the position of its cell in the character repertoire table(s); this form is available only for **IA5String**, **UniversalString**, **UTF8String** and **BMPString**.

The ASN.1 XML value notation for character string types uses the "xmlcstring" notation, which includes the ability to use escape sequences for certain special characters, and for specification of characters using decimal or hexadecimal (see 11.15).

36 Notation for character string types

36.1 The notation for referencing a character string type (see 3.6.11) shall be:

```
CharacterStringType ::=
        RestrictedCharacterStringType |
        UnrestrictedCharacterStringType
```

"RestrictedCharacterStringType" is the notation for a restricted character string type and is defined in clause 37. "UnrestrictedCharacterStringType" is the notation for the unrestricted character string type and is defined in 40.1.

36.2 The tag of each restricted character string type is specified in 37.1. The tag of the unrestricted character string type is specified in 40.2.

36.3 The notation for a character string value shall be:

CharacterStringValue ::=
RestrictedCharacterStringValue |
UnrestrictedCharacterStringValue

XMLCharacterStringValue ::=
XMLRestrictedCharacterStringValue |
XMLUnrestrictedCharacterStringValue

"RestrictedCharacterStringValue" and "XMLRestrictedCharacterStringValue" are defined in 37.8 and 36.9 respectively. "UnrestrictedCharacterStringValue" and "XMLUnrestrictedCharacterStringValue" are notations for an unrestricted character string value and they are defined in 40.7.

37 Definition of restricted character string types

This clause defines types whose values are restricted to sequences of zero, one or more characters from some specified collection of characters. The notation for referencing a restricted character string type shall be "RestrictedCharacterStringType":

RestrictedCharacterStringType ::=
BMPString |
GeneralString |
GraphicString |
IA5String |
ISO646String |
NumericString |
PrintableString |
TeletexString |
T61String |
UniversalString |
UTF8String |
VideotexString |
VisibleString

Each "RestrictedCharacterStringType" alternative is defined by specifying:

a) the tag assigned to the type; and

b) a name (e.g., **NumericString**) by which the type is referenced; and

c) the characters in the collection of characters used in defining the type, by reference to a table listing the character graphics or by reference to a registration number in the ISO International Register of Coded Character Sets (see *ISO International Register of Coded Character Sets to be used with Escape Sequences*), or by reference to ISO/IEC 10646-1.

37.1 Table 6 lists the name by which each restricted character string type is referenced, the number of the universal class tag assigned to the type, the defining registration number or table, or the defining text clause, and, where necessary, identification of a Note relating to the entry in the table. Where a synonymous name is defined in the notation, this is listed in parentheses.

Table 6 – List of restricted character string types

Name for referencing the type	Universal class number	Defining registration number[a], table number, or ITU-T Rec. X.680 | ISO/IEC 8824-1 clause	Notes
UTF8String	12	Subclause 37.16	
NumericString	18	Table 7	Note 1
PrintableString	19	Table 8	Note 1
TeletexString (**T61String**)	20	6, 87, 102, 103, 106, 107, 126, 144, 150, 153, 156, 164, 165, 168 + SPACE + DELETE	Note 2
VideotexString	21	1, 13, 72, 73, 87, 89, 102, 108, 126, 128, 129, 144, 150, 153, 164, 165, 168 + SPACE + DELETE	Note 3
IA5String	22	1, 6 + SPACE + DELETE	
GraphicString	25	All G sets + SPACE	
VisibleString (**ISO646String**)	26	6 + SPACE	
GeneralString	27	All G and all C sets + SPACE + DELETE	
UniversalString	28	See 37.6	
BMPString	30	See 37.15	

a) The defining registration numbers are listed in ISO International Register of Coded Character Sets to be used with Escape Sequences.

NOTE 1 – The type-style, size, colour, intensity, or other display characteristics are not significant.

NOTE 2 – The entries corresponding to these registration numbers reference CCITT Rec. T.61 for rules concerning their use. Register entries 6 and 156 can be used instead of 102 and 103.

NOTE 3 – The entries corresponding to these registration numbers provide the functionality of CCITT Rec. T.100 and ITU-T Rec. T.101.

37.2 Table 7 lists the characters which can appear in the **NumericString** type and **NumericString** character abstract syntax.

Table 7 – NumericString

Name	Graphic
Digits	**0**, **1**, ... **9**
Space	(space)

37.3 The following object identifier and object descriptor values are assigned to identify and describe the **NumericString** character abstract syntax:

```
{ joint-iso-itu-t asn1(1) specification(0) characterStrings(1) numericString(0) }
```

and

```
"NumericString character abstract syntax"
```

NOTE 1 – This object identifier value can be used in **CHARACTER STRING** values and in other cases where there is a need to carry the identification of the character string type separate from the value.

NOTE 2 – A value of a **NumericString** character abstract syntax may be encoded by:

a) One of the rules given in ISO/IEC 10646-1 for encoding the abstract characters. In this case the character transfer syntax is identified by the object identifier associated with those rules in ISO/IEC 10646-1, Annex N.

b) The ASN.1 encoding rules for the built-in type **NumericString**. In this case the character transfer syntax is identified by the object identifier value **{joint-iso-itu-t asn1(1) basic-encoding(1)}**.

37.4 Table 8 lists the characters which can appear in the **PrintableString** type and **PrintableString** character abstract syntax.

Table 8 – PrintableString

Name	Graphic
Latin capital letters	**A**, **B**, ... **Z**
Latin small letters	**a**, **b**, ... **z**
Digits	**0**, **1**, ... **9**
SPACE	(space)
APOSTROPHE	**'**
LEFT PARENTHESIS	**(**
RIGHT PARENTHESIS	**)**
PLUS SIGN	**+**
COMMA	**,**
HYPHEN-MINUS	**-**
FULL STOP	**.**
SOLIDUS	**/**
COLON	**:**
EQUALS SIGN	**=**
QUESTION MARK	**?**

37.5 The following object identifier and object descriptor values are assigned to identify and describe the **PrintableString** character abstract syntax:

```
{ joint-iso-itu-t asn1(1) specification(0) characterStrings(1) printableString(1) }
```

and

```
"PrintableString character abstract syntax"
```

NOTE 1 – This object identifier value can be used in **CHARACTER STRING** values and in other cases where there is a need to carry the identification of the character string type separate from the value.

NOTE 2 – A value of a **PrintableString** character abstract syntax may be encoded by:

a) One of the rules given in ISO/IEC 10646-1 for encoding the abstract characters. In this case the character transfer syntax is identified by the object identifier associated with those rules in ISO/IEC 10646-1, Annex N.

b) The ASN.1 encoding rules for the built-in type **PrintableString**. In this case the character transfer syntax is identified by the object identifier **{ joint-iso-itu-t asn1(1) basic-encoding(1) }**.

37.6 The characters which can appear in the **UniversalString** type are any of the characters allowed by ISO/IEC 10646-1.

37.7 Use of this type invokes the conformance requirements specified in ISO/IEC 10646-1.

NOTE – Clause 38 of this Recommendation | International Standard defines an ASN.1 module containing a number of subtypes of this type for the "Collections of graphics characters for subsets" defined in ISO/IEC 10646-1, Annex A.

37.8 The "RestrictedCharacterStringValue" notation for the restricted character string types shall be "cstring" (see 11.14), "CharacterStringList", "Quadruple", or "Tuple". "Quadruple" is only capable of defining a character string of length one, and can only be used in value notation for **UniversalString**, **UTF8String** or **BMPString** types. "Tuple" is only capable of defining a character string of length one, and can only be used in value notation for **IA5String** types.

```
RestrictedCharacterStringValue ::=
        cstring                 |
        CharacterStringList     |
        Quadruple               |
        Tuple

CharacterStringList ::= "{" CharSyms "}"

CharSyms ::=
        CharsDefn               |
        CharSyms "," CharsDefn
```

```
CharsDefn ::=
        cstring         |
        Quadruple       |
        Tuple           |
        DefinedValue

Quadruple ::= "{" Group "," Plane "," Row "," Cell "}"

Group   ::= number

Plane   ::= number

Row     ::= number

Cell    ::= number

Tuple ::= "{" TableColumn "," TableRow "}"

TableColumn ::= number

TableRow ::= number
```

NOTE 1 – The "cstring" notation can only be used unambiguously on a medium capable of displaying the graphic symbols for the characters which are present in the value. Conversely, if the medium has no such capability, the only means of unambiguously specifying a character string value that uses such graphic symbols is by means of the "CharacterStringList" notation, and only if the type is **UniversalString**, **UTF8String**, **BMPString** or **IA5String**, and the "DefinedValue" alternative of "CharsDefn" is used (see 38.1.2).

NOTE 2 – Clause 38 defines a number of "valuereference"s which denote single characters (strings of size 1) of type **BMPString** (and hence **UniversalString** and **UTF8String**) and **IA5String**.

EXAMPLE – Suppose that one wishes to specify a value of "abcΣdef" for a **UniversalString** where the character "Σ" is not representable on the available medium, this value can also be expressed as:

```
IMPORTS BasicLatin, greekCapitalLetterSigma FROM ASN1-CHARACTER-MODULE
     { joint-iso-itu-t asn1(1) specification(0) modules(0) iso10646(0) };

MyAlphabet ::= UniversalString (FROM (BasicLatin | greekCapitalLetterSigma))

mystring MyAlphabet ::= { "abc" , greekCapitalLetterSigma , "def" }
```

NOTE 3 – When specifying the value of a **UniversalString**, **UTF8String** or **BMPString** type, the "cstring" notation should not be used unless ambiguities arising from different graphic characters with similar shapes have been resolved.

EXAMPLE – The following "cstring" notation should not be used because the graphic symbols 'H', 'O', 'P' and 'E' occur in the BASIC LATIN, CYRILLIC and BASIC GREEK alphabets and thus are ambiguous.

```
IMPORTS BasicLatin, Cyrillic, BasicGreek FROM ASN1-CHARACTER-MODULE
     { joint-iso-itu-t asn1(1) specification(0) modules(0) iso10646(0) };

MyAlphabet ::= UniversalString (FROM (BasicLatin | Cyrillic | BasicGreek))

mystring MyAlphabet ::= "HOPE"
```

An alternative unambiguous definition of **mystring** would be:

```
mystring  MyAlphabet(BasicLatin) ::= "HOPE"
```

Formally, **mystring** is a value reference to a value of a subset of **MyAlphabet**, but it can, by the value mapping rules of Annex B, be used wherever a value reference is needed to this value within **MyAlphabet**.

36.9 The "XMLRestrictedCharacterStringValue" notation is:

```
XMLRestrictedCharacterStringValue ::= xmlcstring
```

36.10 There are characters which cannot be directly represented in "xmlcstring". These shall be represented using the escape sequences specified in 11.15.

NOTE – If the restricted character string value contains characters which are not ISO/IEC 10646-1 characters specified in 11.15.1, these cannot be represented in "xmlcstring", and such values cannot be transferred using XML Encoding Rules (see ITU-T Rec. X.693 | ISO/IEC 8825-3).

37.11 The "DefinedValue" in "CharsDefn" shall be a reference to a value of that type.

37.12 The "number" in the "Plane", "Row" and "Cell" productions shall be less than 256, and in the "Group" production it shall be less than 128.

37.13 The "Group" specifies a group in the coding space of the UCS, the "Plane" specifies a plane within the group, the "Row" specifies a row within the plane, and the "Cell" specifies a cell within the row. The abstract character identified by this notation is the abstract character for the cell specified by the "Group", "Plane", "Row", and "Cell" values. In all cases, the set of permitted characters may be restricted by subtyping.

NOTE – Application designers should consider carefully the conformance implications when using open-ended character string types such as **GeneralString**, **GraphicString**, and **UniversalString** without the application of constraints. Careful text on conformance is also needed for bounded but large character string types such as **TeletexString**.

37.14 The "number" in the "TableColumn" production shall be in the range zero to seven, and the "number" in the "TableRow" production shall be in the range zero to fifteen. The "TableColumn" specifies a column and the "TableRow" specifies a row of a character code table in accordance with Figure 1 of ISO/IEC 2022. This notation is used only for **IA5String** when the code table contains Register Entry 1 in columns 0 and 1 and Register Entry 6 in columns 2 to 7 (see the *ISO International Register of Coded Character Sets to be used with Escape Sequences*).

37.15 **BMPString** is a subtype of **UniversalString** that has its own unique tag and contains only the characters in the Basic Multilingual Plane (those corresponding to the first 64K-2 cells, less cells whose encoding is used to address characters outside the Basic Multilingual Plane) of ISO/IEC 10646-1. It has an associated type defined as:

```
UniversalString (Bmp)
```

where **Bmp** is defined in the ASN.1 module **ASN1-CHARACTER-MODULE** (see clause 38) as the subtype of **UniversalString** corresponding to the "BMP" collection name defined in ISO/IEC 10646-1, Annex A.

NOTE 1 – Since **BMPString** is a built-in type, it is not defined in **ASN1-CHARACTER-MODULE**.

NOTE 2 – The purpose of defining **BMPString** as a built-in type is to enable encoding rules (such as BER) that do not take account of constraints to use 16-bit rather than 32-bit encodings.

NOTE 3 – In the value notation all **BMPString** values are valid **UniversalString** and **UTF8String** values.

37.16 **UTF8String** is synonymous with **UniversalString** at the abstract level and can be used wherever **UniversalString** is used (subject to rules requiring distinct tags) but has a different tag and is a distinct type.

NOTE – The encoding of **UTF8String** used by BER and PER is different from that of **UniversalString**, and for most text will be less verbose.

38 Naming characters and collections defined in ISO/IEC 10646-1

This clause specifies an ASN.1 built-in module which contains the definition of a value reference name for each character from ISO/IEC 10646-1, where each name references a **UniversalString** value of size 1. This module also contains the definition of a type reference name for each collection of characters from ISO/IEC 10646-1, where each name references a subset of the **UniversalString** type.

NOTE – These values are available for use in the value notation of the **UniversalString** type and types derived from it. All of the value and type references defined in the module specified in 38.1 are exported and must be imported by any module that uses them.

38.1 Specification of the ASN.1 Module "ASN1-CHARACTER-MODULE"

The module is not printed here in full. Instead, the means by which it is defined is specified.

38.1.1 The module begins as follows:

```
ASN1-CHARACTER-MODULE { joint-iso-itu-t asn1(1) specification(0) modules(0)
iso10646(0) }
      DEFINITIONS ::= BEGIN
      -- All of the value references and type references defined within this
      -- module are implicitly exported, and are available for import by any module.
      -- ISO/IEC 646 control characters:

      nul   IA5String ::= {0, 0}
      soh   IA5String ::= {0, 1}
      stx   IA5String ::= {0, 2}
      etx   IA5String ::= {0, 3}
      eot   IA5String ::= {0, 4}
      enq   IA5String ::= {0, 5}
      ack   IA5String ::= {0, 6}
      bel   IA5String ::= {0, 7}
      bs    IA5String ::= {0, 8}
      ht    IA5String ::= {0, 9}
      lf    IA5String ::= {0,10}
      vt    IA5String ::= {0,11}
      ff    IA5String ::= {0,12}
      cr    IA5String ::= {0,13}
      so    IA5String ::= {0,14}
      si    IA5String ::= {0,15}
      dle   IA5String ::= {1, 0}
      dc1   IA5String ::= {1, 1}
```

```
dc2   IA5String ::= {1, 2}
dc3   IA5String ::= {1, 3}
dc4   IA5String ::= {1, 4}
nak   IA5String ::= {1, 5}
syn   IA5String ::= {1, 6}
etb   IA5String ::= {1, 7}
can   IA5String ::= {1, 8}
em    IA5String ::= {1, 9}
sub   IA5String ::= {1,10}
esc   IA5String ::= {1,11}
is4   IA5String ::= {1,12}
is3   IA5String ::= {1,13}
is2   IA5String ::= {1,14}
is1   IA5String ::= {1,15}
del   IA5String ::= {7,15}
```

38.1.2 For each entry in each list of character names for the graphic characters (glyphs) shown in clauses 24 and 25 of ISO/IEC 10646-1, the module includes a statement of the form:

```
<namedcharacter> BMPString ::= <tablecell>
-- represents the character <iso10646name>, see ISO/IEC 10646-1
```

where:

a) <iso10646name> is the character name derived from one listed in ISO/IEC 10646-1;

b) <namedcharacter> is a string obtained by applying to <iso10646name> the procedures specified in 38.2;

c) <tablecell> is the glyph in the table cell in ISO/IEC 10646-1 corresponding to the list entry.

EXAMPLE

```
latinCapitalLetterA  BMPString  ::=  {0, 0, 0, 65}
-- represents the character LATIN CAPITAL LETTER A, see ISO/IEC 10646-1

greekCapitalLetterSigma BMPString  ::=  {0, 0, 3, 163}
-- represents the character GREEK CAPITAL LETTER SIGMA, see ISO/IEC 10646-1
```

38.1.3 For each name for a collection of graphic characters specified in ISO/IEC 10646-1, Annex A, a statement is included in the module of the form:

```
<namedcollectionstring> ::= BMPString
            (FROM (<alternativelist>))
      -- represents the collection of characters <collectionstring>,
      -- see ISO/IEC 10646-1.
```

where:

a) <collectionstring> is the name for the collection of characters assigned in ISO/IEC 10646-1;

b) <namedcollectionstring> is formed by applying to <collectionstring> the procedures of 38.3;

c) <alternativelist> is formed by using the <namedcharacter>s as generated in 38.2 for each of the characters specified by ISO/IEC 10646-1.

The resulting type reference, <namedcollectionstring>, forms a limited subset. (See the tutorial in Annex F.)

NOTE – A limited subset is a list of characters in a specified subset. Contrast this to a selected subset, which is a collection of characters listed in ISO/IEC 10646-1, Annex A, plus the BASIC LATIN collection.

EXAMPLE (partial)

```
space BMPString ::= {0, 0, 0, 32}
      exclamationMark BMPString ::= {0, 0, 0, 33}
      quotationMark BMPString ::= {0, 0, 0, 34}
      ...         -- and so on
      tilde BMPString ::= {0, 0, 0, 126}

BasicLatin ::= BMPString
            (FROM (space
            | exclamationMark
            | quotationMark
            | ...       -- and so on
            | tilde)
            )
```

```
-- represents the collection of characters BASIC LATIN, see ISO/IEC 10646-1.
-- The ellipsis in this example is used for brevity and means "and so on";
-- you cannot use this in an actual ASN.1 module.
```

38.1.4 ISO/IEC 10646-1 defines three levels of implementation. By default all types defined in **ASN1-CHARACTER-MODULE**, except for **Level1** and **Level2** conform to implementation level 3, since such types have no restriction on use of combining characters. **Level1** indicates that implementation level 1 is required, **Level2** indicates that implementation level 2 is required, and **Level3** indicates that implementation level 3 is required. Thus, the following are defined in **ASN1-CHARACTER-MODULE**:

```
Level1 ::= BMPString (FROM (ALL EXCEPT CombiningCharacters))

Level2 ::= BMPString (FROM (ALL EXCEPT CombiningCharactersType-2))

Level3 ::= BMPString
```

NOTE 1 – **CombiningCharacters** and **CombiningCharactersType-2** are the <namedcollectionstring>s corresponding to "COMBINING CHARACTERS" and "COMBINING CHARACTERS B-2", respectively, defined in ISO/IEC 10646-1, Annex A.

NOTE 2 – **Level1** and **Level2** will be used either following an "IntersectionMark" (see clause 46) or as the only constraint in a "ConstraintSpec". (See E.2.7.1 for an example.)

NOTE 3 – See F.2.5 for more information on this topic.

38.1.5 The module is terminated by the statement:

```
END
```

38.1.6 A user-defined equivalent of the example in 38.1.3 is:

```
BasicLatin ::= BMPString (FROM (space..tilde))
-- represents the collection of characters BASIC LATIN,
-- see ISO/IEC 10646-1.
```

38.2 A <namedcharacter> is the string obtained by taking an <iso10646name> (see 38.1.2) and applying the following algorithm:

a) each upper-case letter of the <iso10646name> is transformed into the corresponding lower-case letter, unless the upper-case letter is preceded by a SPACE, in which case the upper-case letter is kept unchanged;

b) each digit and each HYPHEN-MINUS is kept unchanged;

c) each SPACE is deleted.

NOTE – The above algorithm, taken in conjunction with the character naming guidelines in Annex K of ISO/IEC 10646-1 will always result in unambiguous value notation for every character name listed in ISO/IEC 10646-1.

EXAMPLE – The character from ISO/IEC 10646-1, row 0, cell 60, which is named "LESS-THAN SIGN" and has the graphic representation "<" can be referenced using the "DefinedValue" of:

```
less-thanSign
```

38.3 A <namedcollectionstring> is the string obtained by taking <collectionstring> and applying the following algorithm:

a) each upper-case letter of the ISO/IEC 10646-1 collection name is transformed into the corresponding lower-case letter, unless the upper-case letter is preceded by a SPACE or it is the first letter of the name, in which case the upper case letter is kept unchanged;

b) each digit and each HYPHEN-MINUS is kept unchanged;

c) each SPACE is deleted.

EXAMPLES

1 The collection identified in Annex A of ISO/IEC 10646-1 as:

BASIC LATIN

has the ASN.1 type reference

```
BasicLatin
```

2 A character string type consisting of the characters in the BASIC LATIN collection, together with the BASIC ARABIC collection, could be defined as follows:

```
My-Character-String ::= BMPString (FROM (BasicLatin | BasicArabic) )
```

NOTE – The above construction is necessary because the apparently simpler construction of:

```
My-Character-String ::= BMPString (BasicLatin | BasicArabic)
```

would allow only strings which were entirely BASIC LATIN or BASIC ARABIC but not a mixture of both.

39 Canonical order of characters

39.1 For the purpose of "ValueRange" subtyping and for possible use by encoding rules, a canonical ordering of characters is specified for **UniversalString**, **UTF8String**, **BMPString**, **NumericString**, **PrintableString**, **VisibleString**, and **IA5String**.

39.2 For the purpose of this clause only, a character is in one-to-one correspondence with a cell in a code table, whether that cell has been assigned a character name or shape, and whether it is a control character or printing character, combining or non-combining character.

39.3 The canonical order of an abstract character is defined by the canonical order of its value in the 32-bit representation of ISO/IEC 10646-1, with low numbers appearing first and high numbers appearing last in the canonical order.

39.4 Endpoints of "ValueRanges" within "PermittedAlphabet" notations (or individual characters) can be specified using either the ASN.1 value reference defined in the module **ASN1-CHARACTER-MODULE** or (where the graphic symbol is unambiguous in the context of the specification and the medium used to represent it) by giving the graphic symbol in a "cstring" (**ASN1-CHARACTER-MODULE** is defined in 38.1) , or by use of the "Quadruple" or "Tuple" notation of 37.8.

39.6 For **NumericString**, the canonical ordering, increasing from left to right, is defined (see Table 7 of 37.2) as:

(space) **0 1 2 3 4 5 6 7 8 9**

The entire character set contains precisely 11 characters. The endpoint of a "ValueRange" (or individual characters) can be specified using the graphic symbol in a "cstring".

NOTE – This order is the same as the order of the corresponding characters in the BASIC LATIN collection of ISO/IEC 10646-1.

39.7 For **PrintableString**, the canonical ordering, increasing from left to right and top to bottom, is defined (see Table 8 of 37.4) as:

(SPACE) (APOSTROPHE) (LEFT PARENTHESIS) (RIGHT PARENTHESIS) (PLUS SIGN) (COMMA) (HYPHEN-MINUS) (FULL STOP) (SOLIDUS) **0123456789** (COLON) (EQUAL SIGN) (QUESTION MARK) **ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz**

The entire character set contains precisely 74 characters. The endpoint of a "ValueRange" (or individual characters) can be specified using the graphic symbol in a "cstring".

NOTE – This order is the same as the order of the corresponding characters in the BASIC LATIN collection of ISO/IEC 10646-1.

39.8 For **VisibleString**, the canonical order of the cells is defined from the ISO 646 encoding (called ISO 646 ENCODING) as follows:

(ISO 646 ENCODING) - 32

NOTE – That is, the canonical order is the same as the characters in cells 2/0-7/14 of the ISO 646 code table.

The entire character set contains precisely 95 characters. The endpoint of a "ValueRange" (or individual characters) can be specified using the graphic symbol in a "cstring".

39.9 For **IA5String**, the canonical order of the cells is defined from the ISO 646 encoding as follows:

(ISO 646 ENCODING)

The entire character set contains precisely 128 characters. The endpoint of a "ValueRange" (or individual characters) can be specified using the graphic symbol in a "cstring" or an ISO 646 control character value reference defined in 38.1.1.

40 Definition of unrestricted character string types

This clause defines a type whose values are the values of any character abstract syntax. In an OSI environment, this abstract syntax may be part of the OSI defined context set. Otherwise, it is referenced directly for each instance of use of the unrestricted character string type.

NOTE 1 – A character abstract syntax (and one or more corresponding character transfer syntaxes) can be defined by any organization able to allocate ASN.1 **OBJECT IDENTIFIER**s.

NOTE 2 – Profiles produced by a community of interest will normally determine the character abstract syntaxes and character transfer syntaxes that are to be supported for specific instances or groups of instances of **CHARACTER STRING**. It will be usual in OSI applications to include reference to supported syntaxes in an OSI Protocol Implementation Conformance Statement.

40.1 The unrestricted character string type (see 3.6.76) shall be referenced by the notation "UnrestrictedCharacterStringType":

UnrestrictedCharacterStringType ::= CHARACTER STRING

40.2 This type has a tag which is universal class, number 29.

40.3 The type consists of values representing:

a) a character string value that may, but need not, be the value of an ASN.1 character string type; and

b) identification (separately or together) of:

1) a character abstract syntax; and

2) the character transfer syntax.

40.4 The unrestricted character string type has an associated type. This associated type is used to support its value and subtype notations.

40.5 The associated type for value definition and subtyping, assuming an automatic tagging environment, is (with normative comments):

```
SEQUENCE {
     identification                          CHOICE {
          syntaxes                           SEQUENCE {
               abstract                      OBJECT IDENTIFIER,
               transfer                      OBJECT IDENTIFIER }
               --  Abstract and transfer syntax object identifiers --,

          syntax                             OBJECT IDENTIFIER
               --  A single object identifier for identification of the
               -- abstract and transfer syntaxes --,
          presentation-context-id            INTEGER
               --  (Applicable only to OSI environments)
               --  The negotiated OSI presentation context identifies the
               -- abstract and transfer syntaxes  --,

          context-negotiation                SEQUENCE {
               presentation-context-id       INTEGER,
               transfer-syntax               OBJECT IDENTIFIER }
          -- (Applicable only to OSI environments)
          -- Context-negotiation in progress, presentation-context-id
          -- identifies only the
          -- abstract-syntax, so the transfer syntax shall be specified --,

          transfer-syntax                    OBJECT IDENTIFIER
          --  The type of the value (for example, specification that it is
          -- the value of an ASN.1 type) is fixed by the application
          -- designer (and hence known to both sender and receiver). This
          -- case is provided primarily to support
          -- selective-field-encryption (or other encoding
          --  transformations) of an ASN.1 type  --,

          fixed                              NULL
          -- The data value is the value of a fixed ASN.1 type (and hence
          -- known to both sender and receiver)  -- },

     data-value-descriptor                   ObjectDescriptor  OPTIONAL
          -- This provides human-readable identification of the class of
          -- the value  --,
     string-value                            OCTET STRING }
     ( WITH COMPONENTS {
          ... ,
          data-value-descriptor  ABSENT } )
```

NOTE – The unrestricted character string type does not allow the inclusion of a **data-value-descriptor** value together with the **identification**. However, the definition of the associated type provided here underlies the commonalities which exist between the embedded-pdv type, the external type and the unrestricted character string type.

40.6 The text of 33.6 and 33.7 also applies to the unrestricted character string type.

40.7 The value notation shall be the value notation for the associated type, where the value of the **string-value** component of type **OCTET STRING** represents an encoding using the transfer syntax specified in **identification**.

UnrestrictedCharacterStringValue ::= SequenceValue -- *value of associated type defined in 40.5*

XMLUnrestrictedCharacterStringValue ::=
XMLSequenceValue -- *value of associated type defined in 40.5*

40.8 An example of the unrestricted character string type is given in E.2.8.

41 Notation for types defined in clauses 42 to 44

41.1 The notation for referencing a type defined in clauses 42 to 44 shall be:

UsefulType ::= typereference

where "typereference" is one of those defined in clauses 42 to 44 using the ASN.1 notation.

41.2 The tag of each "UsefulType" is specified in clauses 42 to 44.

42 Generalized time

42.1 This type shall be referenced by the name:

GeneralizedTime

42.2 The type consists of values representing:

a) a calendar date, as defined in ISO 8601; and

b) a time of day, to any of the precisions defined in ISO 8601, except for the hours value 24 which shall not be used; and

c) the local time differential factor as defined in ISO 8601.

42.3 The type is defined, using ASN.1, as follows:

```
GeneralizedTime ::=    [UNIVERSAL 24] IMPLICIT VisibleString
```

with the values of the **VisibleString** restricted to strings of characters which are either

a) a string representing the calendar date, as specified in ISO 8601, with a four-digit representation of the year, a two-digit representation of the month and a two-digit representation of the day, without use of separators, followed by a string representing the time of day, as specified in ISO 8601, without separators other than decimal comma or decimal period (as provided for in ISO 8601), and with no terminating Z (as provided for in ISO 8601); or

b) the characters in a) above followed by an upper-case letter **Z**; or

c) the characters in a) above followed by a string representing a local time differential, as specified in ISO 8601, without separators.

In case a), the time shall represent the local time. In case b), the time shall represent coordinated universal time. In case c), the part of the string formed as in case a) represents the local time (t_1), and the time differential (t_2) enables coordinated universal time to be determined as follows:

coordinated universal time is $t_1 - t_2$

EXAMPLES

Case a)

"19851106210627.3"
local time 6 minutes, 27.3 seconds after 9 pm on 6 November 1985.

Case b)

"19851106210627.3Z"
coordinated universal time as above.

Case c)

"19851106210627.3-0500"
local time as in example a), with local time 5 hours retarded in relation to coordinated universal time.

42.4 The tag shall be as defined in 42.3.

42.5 The value notation shall be the value notation for the **VisibleString** defined in 42.3.

43 Universal time

43.1 This type shall be referenced by the name:

UTCTime

43.2 The type consists of values representing:

a) calendar date; and

b) time to a precision of one minute or one second; and

c) (optionally) a local time differential from coordinated universal time.

43.3 The type is defined, using ASN.1, as follows:

```
UTCTime ::= [UNIVERSAL 23] IMPLICIT VisibleString
```

with the values of the **VisibleString** restricted to strings of characters which are the juxtaposition of:

a) the six digits YYMMDD where YY is the two low-order digits of the Christian year, MM is the month (counting January as 01), and DD is the day of the month (01 to 31); and

b) either:

1) the four digits hhmm where hh is hour (00 to 23) and mm is minutes (00 to 59); or

2) the six digits hhmmss where hh and mm are as in 1) above, and ss is seconds (00 to 59); and

c) either:

1) the character **Z**; or

2) one of the characters **+** or **-**, followed by hhmm, where hh is hour and mm is minutes.

The alternatives in b) above allow varying precisions in the specification of the time.

In alternative c) 1), the time is coordinated universal time. In alternative c) 2), the time (t_1) specified by a) and b) above is the local time; the time differential (t_2) specified by c) 2) above enables the coordinated universal time to be determined as follows:

Coordinated universal time is $t_1 - t_2$

EXAMPLE 1 – If local time is 7am on 2 January 1982 and coordinated universal time is 12 noon on 2 January 1982, the value of **UTCTime** is either of:

– **"8201021200Z"**; or
– **"8201020700-0500"**.

EXAMPLE 2 – If local time is 7am on 2 January 2001 and coordinated universal time is 12 noon on 2 January 2001, the value of **UTCTime** is either of:

– **"0101021200Z"**; or
– **"0101020700-0500"**.

43.4 The tag shall be as defined in 43.3.

43.5 The value notation shall be the value notation for the **VisibleString** defined in 43.3.

44 The object descriptor type

44.1 This type shall be referenced by the name:

ObjectDescriptor

44.2 The type consists of human-readable text which serves to describe an object. The text is not an unambiguous identification of the object, but identical text for different objects is intended to be uncommon.

NOTE – It is recommended that an authority assigning values of type **OBJECT IDENTIFIER** to an object should also assign values of type **ObjectDescriptor** to that object.

44.3 The type is defined, using ASN.1, as follows:

```
ObjectDescriptor ::= [UNIVERSAL 7] IMPLICIT GraphicString
```

The **GraphicString** contains the text describing the object.

44.4 The tag shall be as defined in 44.3.

44.5 The value notation shall be the value notation for the **GraphicString** defined in 44.3.

45 Constrained Types

45.1 The "ConstrainedType" notation allows a constraint to be applied to a (parent) type, either to restrict its set of values to some subtype of the parent or (within a set or sequence type) to specify that component relations apply to values of the parent type and to values of some other component in the same set or sequence value. It also allows an exception identifier to be associated with a constraint.

ConstrainedType ::=
Type Constraint |
TypeWithConstraint

In the first alternative, the parent type is "Type", and the constraint is specified by "Constraint" as defined in 45.6. The second alternative is defined in 45.5.

45.2 When the "Constraint" notation follows a set-of or sequence-of type notation, it applies to the "Type" in the (innermost) set-of or sequence-of notation, not to the set-of or sequence-of type.

NOTE – For example, in the following the constraint **(SIZE(1..64))** applies to the **VisibleString**, not the **SEQUENCE OF**:

```
NamesOfMemberNations ::= SEQUENCE OF VisibleString (SIZE(1..64))
```

45.3 When the "Constraint" notation follows the selection type notation, it applies to the choice type, and not to the type of the selected alternative. Such a constraint is ignored (see 29.2).

NOTE – In the following example, the constraint **(WITH COMPONENTS {..., a ABSENT})** applies to the **CHOICE** type **T**, not to the selected **SEQUENCE** type, and has no effect on the values of **V**.

```
T ::= CHOICE {
      a SEQUENCE {
            a INTEGER OPTIONAL,
            b BOOLEAN
      },
      b NULL
}

V ::= a < T (WITH COMPONENTS {..., a ABSENT})
```

45.4 When the "Constraint" notation follows a "TaggedType" notation, the interpretation of the overall notation is the same regardless of whether the "TaggedType" or the "Type" is considered as the parent type.

45.5 As a consequence of the interpretation specified in 45.2, special notation is provided to allow a constraint to be applied to a set-of or sequence-of type. This is "TypeWithConstraint":

TypeWithConstraint ::=
SET Constraint OF Type |
SET SizeConstraint OF Type |
SEQUENCE Constraint OF Type |
SEQUENCE SizeConstraint OF Type

In the first and second alternatives the parent type is "**SET OF** Type", while in the third and fourth it is "**SEQUENCE OF** Type". In the first and third alternatives, the constraint is "Constraint" (see 45.6), while in the second and fourth it is "SizeConstraint" (see 47.5).

NOTE – Although the "Constraint" alternatives encompass the corresponding "SizeConstraint" alternatives, the "SizeConstraint" alternatives are provided for historical reasons.

45.6 A constraint is specified by the notation "Constraint":

Constraint ::= "(" ConstraintSpec ExceptionSpec ")"

ConstraintSpec ::=
SubtypeConstraint |
GeneralConstraint

"ExceptionSpec" is defined in clause 49. Unless it is used in conjunction with an "extension marker" (see clause 48), it shall only be present if the "ConstraintSpec" includes an occurrence of "DummyReference" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.3) or is a "UserDefinedConstraint" (see ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 9). The "GeneralConstraint" is defined in ITU-T Rec. X.682 | ISO/IEC 8824-3, 8.1.

45.7 The notation "SubtypeConstraint" is the general-purpose "ElementSetSpecs" notation (see clause 46):

SubtypeConstraint ::= ElementSetSpecs

In this context, the elements are values of the parent type (the governor of the element set is the parent type). There shall be at least one element in the set.

46 Element set specification

46.1 In some notations a set of elements of some identified type or information object class (the governor) can be specified. In such cases, the notation "ElementSetSpec" is used:

ElementSetSpecs ::=
RootElementSetSpec |
RootElementSetSpec "," "..." |
RootElementSetSpec "," "..." "," AdditionalElementSetSpec

RootElementSetSpec ::= ElementSetSpec

AdditionalElementSetSpec ::= ElementSetSpec

ElementSetSpec ::= Unions |
ALL Exclusions

Unions ::= Intersections |
UElems UnionMark Intersections

UElems ::= Unions

Intersections ::= IntersectionElements |
IElems IntersectionMark IntersectionElements

IElems ::= Intersections

IntersectionElements ::= Elements | Elems Exclusions

Elems ::= Elements

Exclusions ::= EXCEPT Elements

UnionMark ::= "|" | UNION

IntersectionMark ::= "^" | INTERSECTION

NOTE 1 – The caret character "^" and the word **INTERSECTION** are synonymous. The character "|" and the word **UNION** are synonymous. It is recommended that, as a stylistic matter, either the characters or the words be used throughout a user Specification. **EXCEPT** can be used with either style.

NOTE 2 – The order of precedence from highest to lowest is: **EXCEPT**, "^", "|". Notice that **ALL EXCEPT** is specified so that it cannot be interspersed with the other constraints without the use of parentheses around "**ALL EXCEPT** xxx".

NOTE 3 – Anywhere that "Elements" occurs, either a constraint without parentheses [e.g., **INTEGER (1..4)**] or a parenthesized subtype constraint [e.g., **INTEGER ((1..4 | 9))**] can appear.

NOTE 4 – Note that two **EXCEPT** operators must have either "|", "^", "(" or ")" separating them, so **(A EXCEPT B EXCEPT C)** is not permitted. This must be changed to **((A EXCEPT B) EXCEPT C)** or **(A EXCEPT (B EXCEPT C))**.

NOTE 5 – Note that **((A EXCEPT B) EXCEPT C)** is the same as **(A EXCEPT (B | C))**.

NOTE 6 – The elements that are referenced by "ElementSetSpecs" is the union of the elements referenced by the "RootElementSetSpec" and "AdditionalElementSetSpec" (when present).

NOTE 7 – When the elements are information objects (i.e., the governor is an information object class), the notation "ObjectSetElements" as defined in ITU-T Rec. X.681 | ISO/IEC 8824-2, 12.3 is used.

46.2 The elements forming the set are:

a) if the first alternative of the "ElementSetSpec" is selected, those specified in the "Unions" [see b)], otherwise all elements of the governor except those specified in the "Elements" notation of the "Exclusions";

b) if the first alternative of "Unions" is selected, then those specified in the "Intersections" [see c)], otherwise those specified at least once either in the "UElems" or "Intersections";

c) if the first alternative of "Intersections" is selected, those specified in the "IntersectionElements" [see d)], otherwise those specified by "IElems" which also are specified by "IntersectionElements";

d) if the first alternative of "IntersectionElements" is selected, those specified in the "Elements", otherwise those specified in the "Elems" except those specified in the "Exclusions".

46.3 The set of values is defined to be extensible if the following conditions hold:

a) for "Elements": there is an extension marker at the outer level;

NOTE – This applies even if all values of the parent are included in the root of the new constrained type.

b) for "Unions": at least one of the "UElems" is extensible;

c) for "Intersections": at least one of the "IElems" is extensible;

d) for "Exclusions": the set of elements preceding **EXCEPT** is extensible.

Otherwise, the set of values is not extensible (see also G.4).

46.4 If the set of values is extensible, the root values can be determined by performing the set arithmetic using only root values of the sets of values involved in the set arithmetic, as specified in 46.2. The extension additions can be determined by performing the set arithmetic using the root values augmented by the extension additions, for each set of values involved in the set arithmetic, and then excluding values that were determined to be root values.

46.5 The "Elements" notation is defined as follows:

```
Elements ::=
        SubtypeElements         |
        ObjectSetElements       |
        "(" ElementSetSpec ")"
```

The elements specified by this notation are:

a) As described in clause 47 below if the "SubtypeElements" alternative is used. This notation shall only be used when the governor is a type, and the actual type involved will further constrain the notational possibilities. In this context, the governor is referred to as the parent type.

b) As described in ITU-T Rec. X.681 | ISO/IEC 8824-2, 12.10, if the "ObjectSetElements" notation is used. This notation shall only be used when the governor is an information object class.

c) Those specified by the "ElementSetSpec" if the third alternative is used.

46.6 When performing set arithmetic within a subtype constraint or a value set when the governing type is extensible, only abstract values that are in the extension root of the governing type are used in the set arithmetic. In this case, all instances of value notation (including value references) used in set arithmetic are required to reference an abstract value of the extension root of the governing type. The end-points of a range constraint are required to reference values that are present in the extension root of the governing type, and the range specification as a whole references all (and only) those values in the range that are within the extension root of the governing type.

46.7 When performing set arithmetic involving information object sets, all information objects are used in the set arithmetic. If any of the information object sets contributing to the set arithmetic are extensible, or if there is an extension marker at the outermost level of an "ElementSetSpecs", the result of the set arithmetic is extensible.

46.8 If a subtype constraint is serially applied to a parent type which is extensible through the application of an extensible constraint, value notation used within it shall not reference values that are not in the extension root of the parent type. The result of the second (serially applied) constraint is defined to be the same as if the constraint had been applied to the parent type without its extension marker and possible extension additions.

EXAMPLE

```
Foo ::= INTEGER ( 1..6, ..., 73..80)

Bar ::= Foo (73) -- illegal

foo Foo ::= 73 -- legal since it is value notation for Foo, not part of a constraint
```

Bar is illegal since 73 is not in the extension root of **Foo**. If 73 had been in the extension root of **Foo**, the example would have been legal, and **Bar** would have contained the single value of 73.

47 Subtype elements

47.1 General

A number of different forms of notation for "SubtypeElements" are provided. They are identified below, and their syntax and semantics are defined in the following subclauses. Table 9 summarizes which notations can be applied to which parent types.

```
SubtypeElements ::=
        SingleValue             |
        ContainedSubtype        |
        ValueRange              |
        PermittedAlphabet       |
        SizeConstraint          |
        TypeConstraint          |
        InnerTypeConstraints    |
        PatternConstraint
```

Table 9 – Applicability of subtype value sets

Type (or derived from such a type by tagging or subtyping)	Single Value	Contained Subtype	Value Range	Size Constraint	Permitted Alphabet	Type constraint	Inner Subtyping	Pattern Constraint
Bit String	Yes	Yes	No	Yes	No	No	No	No
Boolean	Yes	Yes	No	No	No	No	No	No
Choice	Yes	Yes	No	No	No	No	Yes	No
Embedded-pdv	Yes	No	No	No	No	No	Yes	No
Enumerated	Yes	Yes	No	No	No	No	No	No
External	Yes	No	No	No	No	No	Yes	No
Instance-of	Yes	Yes	No	No	No	No	Yes	No
Integer	Yes	Yes	Yes	No	No	No	No	No
Null	Yes	Yes	No	No	No	No	No	No
Object class field type	Yes	Yes	No	No	No	No	No	No
Object Descriptor	Yes	Yes	No	Yes	Yes	No	No	No
Object Identifier	Yes	Yes	No	No	No	No	No	No
Octet String	Yes	Yes	No	Yes	No	No	No	No
open type	No	No	No	No	No	Yes	No	No
Real	Yes	Yes	Yes	No	No	No	Yes	No
Relative Object Identifier	Yes[b)]	Yes[b)]	No	No	No	No	No	No
Restricted Character String Types	Yes	Yes	Yes[a)]	Yes	Yes	No	No	Yes
Sequence	Yes	Yes	No	No	No	No	Yes	No
Sequence-of	Yes	Yes	No	Yes	No	No	Yes	No
Set	Yes	Yes	No	No	No	No	Yes	No
Set-of	Yes	Yes	No	Yes	No	No	Yes	No
Time Types	Yes	Yes	No	No	No	No	No	No
Unrestricted Character String Type	Yes	No	No	Yes	No	No	Yes	No

a) Allowed only within the "PermittedAlphabet" of **`BMPString`**, **`IA5String`**, **`NumericString`**, **`PrintableString`**, **`VisibleString`**, **`UTF8String`** and **`UniversalString`**.

b) The starting node for all relative object identifier types or values in constraints or valuesets shall be the same as the starting node for the governor.

47.2 Single value

47.2.1 The "SingleValue" notation shall be:

SingleValue ::= Value

where "Value" is the value notation for the parent type.

47.2.2 A "SingleValue" specifies the single value of the parent type specified by "Value".

47.3 Contained subtype

47.3.1 The "ContainedSubtype" notation shall be:

ContainedSubtype ::= Includes Type

Includes ::= `INCLUDES` | empty

The "empty" alternative of the "Includes" production shall not be used when "Type" in "ContainedSubtype" is the notation for the null type.

47.3.2 A "ContainedSubtype" specifies all of the values in the root of the parent type that are also in the root of "Type". "Type" is required to be derived from the same built-in type as the parent type.

47.3.3 The set of values referenced by an extensible "Type" used in a contained subtype constraint does not inherit the extension marker from the "Type". Any values in "Type" that are not in the extension root of that type are ignored, and do not contribute to the values of the constrained type.

NOTE – The use of an extensible "Type" does not in itself make the constrained type extensible.

47.4 Value range

47.4.1 The "ValueRange" notation shall be:

ValueRange ::= LowerEndpoint " . . " UpperEndpoint

47.4.2 A "ValueRange" specifies the values in a range of values which are designated by specifying the values of the endpoints of the range. This notation can only be applied to integer types, the "PermittedAlphabet" of certain restricted character string types (**`IA5String`**, **`NumericString`**, **`PrintableString`**, **`VisibleString`**, **`BMPString`**, **`UniversalString`** and **`UTF8String`** only) and real types. All values specified in the "ValueRange" are required to be in the root of the parent type.

NOTE – For the purpose of subtyping, **`PLUS-INFINITY`** exceeds all real values and **`MINUS-INFINITY`** is less than all real values.

47.4.3 Each endpoint of the range is either closed (in which case that endpoint is specified) or open (in which case the endpoint is not specified). When open, the specification of the endpoint includes a less-than symbol ("<"):

LowerEndpoint ::= LowerEndValue | LowerEndValue "<"

UpperEndpoint ::= UpperEndValue | "<" UpperEndValue

47.4.4 An endpoint may also be unspecified, in which case the range extends in that direction as far as the parent type allows:

LowerEndValue ::= Value | `MIN`

UpperEndValue ::= Value | `MAX`

NOTE – When a "ValueRange" is used as a "PermittedAlphabet" constraint, "LowerEndValue" and "UpperEndValue" shall be of size 1.

47.5 Size constraint

47.5.1 The "SizeConstraint" notation shall be:

SizeConstraint ::= `SIZE` Constraint

47.5.2 A "SizeConstraint" can only be applied to bit string types, octet string types, character string types, set-of types or sequence-of types.

47.5.3 The "Constraint" specifies the permitted integer values for the length of the specified values, and takes the form of any constraint which can be applied to the following parent type:

```
INTEGER (0 .. MAX)
```

The "Constraint" shall use the "SubtypeConstraint" alternative of "ConstraintSpec".

47.5.4 The unit of measure depends on the parent type, as follows:

Type	*Unit of measure*
bit string	bit
octet string	octet
character string	character
set-of	component value
sequence-of	component value

NOTE – The count of the number of characters specified in this subclause for determining the size of a character string value shall be clearly distinguished from a count of octets. The count of characters shall be interpreted according to the definition of the collection of characters used in the type, in particular, in relation to references to the standards, tables or registration numbers in a register which can appear in such a definition.

47.6 Type constraint

47.6.1 The "TypeConstraint" notation shall be:

TypeConstraint ::= Type

47.6.2 This notation is only applied to an open type notation and restricts the open type to values of "Type".

47.7 Permitted alphabet

47.7.1 The "PermittedAlphabet" notation shall be:

PermittedAlphabet ::= `FROM` Constraint

47.7.2 A "PermittedAlphabet" specifies all values which can be constructed using a sub-alphabet of the parent string. This notation can only be applied to restricted character string types.

47.7.3 The "Constraint" is any which could be applied to the parent type (see Table 9), except that it shall use the "SubtypeConstraint" alternative of "ConstraintSpec". The sub-alphabet includes precisely those characters which appear in one or more of the values of the parent string type which are allowed by the "Constraint".

47.7.4 If "Constraint" is extensible, then the set of values selected by the permitted alphabet constraint is extensible. The set of values in the root are those permitted by the root of "Constraint", and the extension additions are those values permitted by the root together with the extension-additions of "Constraint", excluding those values already in the root.

47.8 Inner subtyping

47.8.1 The "InnerTypeConstraints" notation shall be:

InnerTypeConstraints ::=
`WITH COMPONENT` SingleTypeConstraint |
`WITH COMPONENTS` MultipleTypeConstraints

47.8.2 An "InnerTypeConstraints" specifies only those values which satisfy a collection of constraints on the presence and/or values of the components of the parent type. A value of the parent type is not specified unless it satisfies all of the constraints expressed or implied (see 47.8.6). This notation can be applied to the set-of, sequence-of, set, sequence and choice types.

NOTE – An "InnerTypeConstraints" applied to a set or sequence type is ignored by the **`COMPONENTS OF`** transformation (see 24.4 and 26.2).

47.8.3 For the types which are defined in terms of a single other (inner) type (set-of and sequence-of), a constraint taking the form of a subtype value specification is provided. The notation for this is "SingleTypeConstraint":

SingleTypeConstraint ::= Constraint

The "Constraint" defines a subtype of the single other (inner) type. A value of the parent type is specified if and only if each inner value belongs to the subtype obtained by applying the "Constraint" to the inner type.

47.8.4 For the types which are defined in terms of multiple other (inner) types (choice, set, and sequence), a number of constraints on these inner types can be provided. The notation for this is "MultipleTypeConstraints":

MultipleTypeConstraints ::=
FullSpecification |
PartialSpecification

FullSpecification ::= "{" TypeConstraints "}"

PartialSpecification ::= "{" "..." "," TypeConstraints "}"

TypeConstraints ::=
NamedConstraint |
NamedConstraint "," TypeConstraints

NamedConstraint ::=
identifier ComponentConstraint

47.8.5 The "TypeConstraints" contains a list of constraints on the component types of the parent type. For a sequence type, the constraints must appear in order. The inner type to which the constraint applies is identified by means of its identifier. For a given component, there shall be at most one "NamedConstraint".

47.8.6 The "MultipleTypeConstraints" comprises either a "FullSpecification" or a "PartialSpecification". When "FullSpecification" is used, there is an implied presence constraint of **ABSENT** on all inner types which can be constrained to be absent (see 47.8.9) and which is not explicitly listed. Where "PartialSpecification" is employed, there are no implied constraints, and any inner type can be omitted from the list.

47.8.7 A particular inner type may be constrained in terms of its presence (in values of the parent type), its value, or both. The notation is "ComponentConstraint":

ComponentConstraint ::= ValueConstraint PresenceConstraint

47.8.8 A constraint on the value of an inner type is expressed by the notation "ValueConstraint":

ValueConstraint ::= Constraint | empty

The constraint is satisfied by a value of the parent type if and only if the inner value belongs to the subtype specified by the "Constraint" applied to the inner type.

47.8.9 A constraint on the presence of an inner type shall be expressed by the notation "PresenceConstraint":

PresenceConstraint ::= PRESENT | ABSENT | OPTIONAL | empty

The meaning of these alternatives, and the situations in which they are permitted are defined in 47.8.9.1 to 47.8.9.3.

47.8.9.1 If the parent type is a sequence or set, a component type marked **OPTIONAL** may be constrained to be **PRESENT** (in which case the constraint is satisfied if and only if the corresponding component value is present) or to be **ABSENT** (in which case the constraint is satisfied if and only if the corresponding component value is absent) or to be **OPTIONAL** (in which case no constraint is placed upon the presence of the corresponding component value).

47.8.9.2 If the parent type is a choice, a component type can be constrained to be **ABSENT** (in which case the constraint is satisfied if and only if the corresponding component type is not used in the value), or **PRESENT** (in which case the constraint is satisfied if and only if the corresponding component type is used in the value); there shall be at most one **PRESENT** keyword in a "MultipleTypeConstraints".

NOTE – See E.4.6 for a clarifying example.

47.8.9.3 The meaning of an empty "PresenceConstraint" depends on whether a "FullSpecification" or a "PartialSpecification" is being employed:

a) in a "FullSpecification", this is equivalent to a constraint of **PRESENT** for a set or sequence component marked **OPTIONAL** and imposes no further constraint otherwise;

b) in a "PartialSpecification", no constraint is imposed.

47.9 Pattern constraint

47.9.1 The "PatternConstraint" notation shall be:

PatternConstraint ::= PATTERN Value

47.9.2 "Value" shall be a "cstring" of type **UniversalString** (or a reference to such a character string) which contains an ASN.1 regular expression as defined in Annex A. The "PatternConstraint" selects those values of the parent type that satisfy the ASN.1 regular expression. The entire value shall satisfy the entire ASN.1 regular expression, i.e., the

"PatternConstraint" does not select values whose leading characters match the (entire) ASN.1 regular expression but which contain further trailing characters.

NOTE – "Value" is formally defined as a value of type **UniversalString**, but the sets of values of type **UniversalString** and **UTF8String** are the same (see 37.16). Thus a totally equivalent definition could have been to say that "Value" is a value of type **UTF8String**.

48 The extension marker

NOTE – Like the constraint notation in general, the extension marker has no effect on some encoding rules of ASN.1, such as the Basic Encoding Rules, but does on others, such as the Packed Encoding Rules. Its effect on encodings defined using ECN is determined by the ECN specification.

48.1 The extension marker, ellipsis, is an indication that extension additions are expected. It makes no statement as to how such additions should be handled other than that they shall not be treated as an error during the decoding process.

48.2 The joint use of the extension marker and an exception identifier (see clause 49) is both an indication that extension additions are expected and also provides a means for identifying the action to be taken by the application if there is a constraint violation. It is recommended that this notation be used in those situations where store and forward or any other form of relaying is in use, so as to indicate (for example) that any unrecognized extension additions are to be returned to the application for possible re-encoding and relaying.

48.3 The result of set arithmetic involving subtype constraints, value sets or information object sets that are extensible is specified in clause 46.

48.4 If a type defined with an extensible constraint is referenced in a "ContainedSubtype", the newly defined type does not inherit the extension marker or any of its extension additions (see 47.3.3). The newly defined type can be made extensible by including an extension marker at the outermost level in its "ElementSetSpecs" (see also 46.3). For example:

```
A ::= INTEGER (0..10, ..., 12)     -- A is extensible.
B ::= INTEGER (A)                  -- B is inextensible and is constrained to 0-10.
C ::= INTEGER (A, …)               -- C is extensible and is constrained to 0-10.
```

48.5 If a type defined with an extensible constraint is further constrained with an "ElementSetSpecs", the resulting type does not inherit the extension marker nor any extension additions that may be present in the former constraint (see 46.8). For example:

```
A ::= INTEGER (0..10, ...)   -- A is extensible.
B ::= A (2..5)               -- B is inextensible.
C ::= A                      -- C is extensible.
```

48.6 Components of a set, sequence or choice type that are constrained to be absent shall not be present, regardless of whether the set, sequence or choice type is an extensible type.

NOTE – Inner type constraints have no effect on extensibility.

For example:

```
A ::= SEQUENCE   {
      a     INTEGER
      b     BOOLEAN OPTIONAL,
      ...
}

B ::= A (WITH COMPONENTS {b ABSENT})    -- B is extensible, but 'b' shall
                                        -- not be present in any of its
                                        -- values.
```

48.7 Where this Recommendation | International Standard requires distinct tags (see 24.5 to 24.6, 26.3 and 28.3), the following transformation shall conceptually be applied before performing the check for tag uniqueness:

48.7.1 A new element or alternative (called the conceptually-added element, see 48.7.2) is conceptually added at the extension insertion point if:

a) there are no extension markers but extensibility is implied in the module heading, and then an extension marker is added and the new element is added as the first addition after that extension marker; or

b) there is a single extension marker in a **CHOICE** or **SEQUENCE** or **SET**, and then the new element is added at the end of the **CHOICE** or **SEQUENCE** or **SET** immediately prior to the closing brace; or

c) there are two extension markers in a **CHOICE** or **SEQUENCE** or **SET**, and then the new element is added immediately before the second extension marker.

48.7.2 This conceptually-added element is solely for the purposes of checking legality through the application of rules requiring distinct tags (see 24.5 to 24.6, 26.3 and 28.3). It is conceptually-added *after* the application of automatic tagging (if applicable) and the expansion of **COMPONENTS OF**.

48.7.3 The conceptually-added element is defined to have a tag which is distinct from the tag of all normal ASN.1 types, but which matches the tag of all such conceptually-added elements and matches the indeterminate tag of the open type, as specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.2, Note 2.

NOTE – The rules concerning tag uniqueness relating to the conceptually added element and to the open type, together with the rules requiring distinct tags (see 24.5 to 24.6, 26.3 and 28.3) are necessary and sufficient to ensure that:

a) any unknown extension addition can be unambiguously attributed to a single insertion point when a BER encoding is decoded; and

b) unknown extension additions can never be confused with **OPTIONAL** elements.

In PER the above rules are sufficient but are not necessary to ensure these properties. They are nonetheless imposed as rules of ASN.1 to ensure independence of the notation from encoding rules.

48.7.4 If, with these conceptually-added elements, the rules requiring distinct types are violated, then the specification has made illegal use of the extensibility notation.

NOTE – The purpose of the above rules is to make precise restrictions arising from the use of insertion points (particularly those which are not at the end of **SEQUENCE**s or **SET**s or **CHOICE**s). The restrictions are designed to ensure that in BER, DER and CER it is possible to attribute an unknown element received by a version 1 system unambiguously to a specific insertion point. This would be important if the exception handling of such added elements was different for different insertion points.

48.8 Examples

48.8.1 Example 1

```
A ::= SET {
      a     A,
      b     CHOICE {
            c     C,
            d     D,
            ...
      }
}
```

is legal, for there is no ambiguity as any added material must be part of **b**.

48.8.2 Example 2

```
A ::= SET {
      a     A,
      b     CHOICE {
            c     C,
            d     D,
            ...
      },
      ... ,
      d     D
}
```

is illegal, for added material may be part of **b**, or may be at the outer level of **A**, and a version 1 system cannot tell which.

48.8.3 Example 3

```
A ::= SET {
      a     A,
      b     CHOICE {
            c     C,
            ...
      } ,
      d     CHOICE {
            e     E,
            ...
      }
}
```

is also illegal, for added material may be part of **b** or **d**.

48.8.4 More complex examples can be constructed, with extensible choices inside extensible choices, or extensible choices within elements of a sequence marked **OPTIONAL** or **DEFAULT**, but the above rules are necessary and sufficient to ensure that an element not present in version 1 can be unambiguously attributed by a version 1 system to precisely one insertion point.

49 The exception identifier

49.1 In a complex ASN.1 specification, there are a number of places where it is specifically recognized that decoders have to handle material that is not completely specified in it. These cases arise in particular from use of a constraint that is defined using a parameter of the abstract syntax (see ITU-T Rec. X.683 | ISO/IEC 8824-4, clause 10).

49.2 In such cases, the application designer needs to identify the actions to be taken when some implementation-dependent constraint is violated. The exception identifier is provided as an unambiguous means of referring to parts of an ASN.1 specification in order to indicate the actions to be taken. The identifier consists of a "**!**" character, followed by an optional ASN.1 type and a value of that type. In the absence of the type, **INTEGER** is assumed as the type of the value.

49.3 If an "ExceptionSpec" is present, it indicates that there is text in the body of the standard saying how to handle the constraint violation associated with the "**!**" character. If it is absent, then the implementors will either need to identify text that describes the action that they are to take, or will take implementation-dependent action when a constraint violation occurs.

49.4 The "ExceptionSpec" notation is defined as follows:

```
ExceptionSpec ::= "!" ExceptionIdentification | empty

ExceptionIdentification ::=
                SignedNumber    |
                DefinedValue    |
                Type ":" Value
```

The first two alternatives denote exception identifiers of type integer. The third alternative denotes an exception identifier ("Value") of arbitrary type ("Type").

49.5 Where a type is constrained by multiple constraints, more than one of which has an exception identifier, the exception identifier in the outermost constraint shall be regarded as the exception identifier for that type.

49.6 Where an exception marker is present on types that are used in set arithmetic, the exception identifier is ignored and is not inherited by the type being constrained as a result of the set arithmetic.

Annex A

ASN.1 regular expressions

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

A.1 Definition

A.1.1 An ASN.1 regular expression is a pattern that describes a set of strings whose format conforms to this pattern. A regular expression is itself a string; it is constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions. The smallest expressions, which are (usually) made of one or two characters, are placeholders that stand for a set of characters.

The regular expressions presented here are very similar to those of scripting languages like Perl and to those of XML Schema, where some other examples of use can be found.

A.1.2 Most characters, including all letters and digits, are regular expressions that match themselves.

EXAMPLE

The regular expression **"fred"** matches only the string **"fred"**.

A.1.3 Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

A.2 Metacharacters

A.2.1 A metacharacter sequence (or metacharacter) is a set of one or more contiguous characters that have a special meaning in the context of a regular expression. The following list contains all of the metacharacter sequences. Their meaning is explained in the following clauses.

[ ]		Match any character in the set where ranges are denoted by "**-**". A "**^**" after the opening bracket complements the set which follows it.
{g**,**p**,**r**,**c**}**		Quadruple which identifies a character of ISO/IEC 10646-1 (see 37.8)
\N{name**}**		Match the named character (or any character of the named character set) as defined in 38.1
.		Match any character (unless it is one of the newline characters defined in 11.1.6)
\d		Match any digit (equivalent to "**[0-9]**")
\w		Match any alphanumeric character (equivalent to "**[a-zA-Z0-9]**")
\t		Match the HORIZONTAL TABULATION (9) character (see 11.1.6)
\n		Match any one of the newline characters defined in 11.1.6
\r		Match the CARRIAGE RETURN (13) character (see 11.1.6)
\s		Match any one of the white-space characters (see 11.1.6)
\b		Match a word boundary
****	(prefix)	Quote the next metacharacter and cause it to be interpreted literally
****		Match the REVERSE SOLIDUS (92) character "****"
""		Match the QUOTATION MARK (34) character (**"**)
|	(infix)	Alternative between two expressions
( )		Grouping of the enclosed expression
*****	(postfix)	Match the previous expression zero, one or several times
+	(postfix)	Match the previous expression one or several times
?	(postfix)	Match the previous expression once or not at all
#n	(postfix)	Match the previous expression exactly n times (where n is a single digit)
#(n**)**	(postfix)	Match the previous expression exactly n times
#(n**,)**	(postfix)	Match the previous expression at least n times
#(n**,**m**)**	(postfix)	Match the previous expression at least n but not more than m times
#(,m**)**	(postfix)	Match the previous expression not more than m times

NOTE 1 – The characters CIRCUMFLEX ACCENT (94) "**^**" and HYPHEN-MINUS (45) "**-**" are additional metacharacters in certain positions of the string defined in A.2.2.

NOTE 2 – The value in round brackets after a character name in this Annex is the decimal value of the character in ISO/IEC 10646-1.

NOTE 3 – This notation does not provide the metacharacters "**^**" and "**$**" to match the beginning and the end of a string respectively. Hence a string shall match a regular expression in its entirety except if the latter includes "**.***" at its beginning, at its end or at both sides.

NOTE 4 – The following metacharacter sequences cannot contain white-space (see 11.1.6) unless the white-space appears immediately prior to or following a newline:

{g,p,r,c**}**

\N{name**}**

#n

#(n**)**

#(n**,)**

#(n,m**)**

#(,m**)**

If a regular expression contains a newline, any spacing characters that appear immediately prior to or following the newline have no significance and match nothing (see 11.14.1).

A.2.2 A list of characters enclosed by "**[**" and "**]**" matches any single character in that list. If the first character of the list is the caret "**^**", then it matches any character which is not in the list. A range of characters may be specified by giving the first and last characters, separated by a hyphen (according to the order relation defined in 39.3). All metacharacter sequences, except "**]**" and "\", lose their special meaning inside a list. To include a literal CIRCUMFLEX ACCENT (94) "**^**", place it anywhere except in the first position or precede it with a backslash. To include a literal HYPHEN-MINUS (45) "**-**", place it first or last in the list, or precede it with a backslash. To include a literal CLOSING SQUARE BRACKET (93) "**]**", place it first. If the first character in the list is the caret "**^**", then the characters "**-**" and "**]**" also match themselves when they immediately follow that caret. The metacharacter sequences defined in A.2.3, A.2.4, A.2.6 and A.2.7 can be used between the square brackets where they keep their meaning.

EXAMPLES

The regular expression **"[0123456789]"**, or equivalently **"[0-9]"**, matches any single digit.

The regular expression **"[^0]"** matches any single character except **0**.

The regular expression **"[\d^.-]"** matches any single digit, a caret, a hyphen or a period.

A.2.3 To avoid any ambiguity between two ISO/IEC 10646-1 characters which have the same glyph, two notations are provided. A notation of the form **"{**group**,**plane**,**row**,**cell**}"** references a (single) character according to the "Quadruple" production defined in 37.8.

A.2.4 A notation of the form **"\N{**valuereference**}"** matches the referenced character if "valuereference" is a reference to a restricted character string value of size 1 (see clause 37) which is defined or imported in the current module. A notation of the form **"\N{**typereference**}"** matches any character of the referenced character set if "typereference" is a reference to a subtype of a "RestrictedCharacterStringType" which is defined in the current module, or is one of the "RestrictedCharacterStringType"s defined in clause 37.

NOTE – In particular, "valuereference" or "typereference" can be one of the references defined in the module **ASN1-CHARACTER-MODULE** (see 38.1) and imported into the current module (see 37.8).

EXAMPLES

The regular expression **"\N{greekCapitalLetterSigma}"** matches GREEK CAPITAL LETTER SIGMA.

The regular expression **"\N{BasicLatin}"** matches any (single) character of the BASIC LATIN character set.

"[\N{BasicLatin}\N{Cyrillic}\N{BasicGreek}]+", or equivalently **"(\N{BasicLatin}|\N{Cyrillic}|\N{BasicGreek})+"**, are regular expressions that match a string made of any (non null) number of characters from the three character sets specified.

A.2.5 The period "**.**" matches any single character, unless it is one of the newline characters defined in 11.1.6.

A.2.6 The symbol "**\d**" is a synonym for "**[0-9]**", i.e., it matches any single digit. The symbol "**\t**" matches the HORIZONTAL TABULATION (9) character. The symbol "**\w**" is a synonym for "**[a-zA-Z0-9]**", i.e., it matches any single (lower-case or upper-case) character or any single digit.

EXAMPLE

The regular expression **"\w+(\s\w+)*\."** matches a sentence made of at least one (alphanumeric) word. The words are separated by one white-space character as defined in 11.1.6. There is no white-space character before the ending period.

A.2.7 The symbol "**\r**" matches the CARRIAGE RETURN (13) character. The symbol "**\n**" matches any one of the newline characters defined in 11.1.6. The symbol "**\s**" matches any one of the white-space characters defined in 11.1.6. The symbol "**\b**" matches the empty string at the beginning or at the end of a word.

EXAMPLE

The regular expression **".*\bfred\b.*"** matches any string which includes the word **"fred"** (this word is not only a series of four characters; it is delimited). Hence it matches strings like **"fred"** or **"I am fred the first"**, but not strings like **"My name is freddy"** or **"I am afred I don't know how to spell 'afraid'!"**.

A.2.8 A character that normally functions as a metacharacter can be interpreted literally by prefixing it with a "\". If the regular expression includes a QUOTATION MARK (34), this character shall be represented by a pair of QUOTATION MARK characters.

EXAMPLES

The regular expression **"\."** matches the (single) string **"."**, but not any string of any single character.

The regular expression **""""** matches the string which contains a single QUOTATION MARK.

The regular expression **"\)"** matches the string **")"**.

The regular expression **"\a"** matches the character **"a"**.

NOTE – The fourth example shows that the backslash is allowed to precede characters that are not metacharacters, but this use is deprecated (because other metacharacters could be allowed in future versions of this Recommendation | International Standard).

A.2.9 Two or more regular expressions may be joined by the infix operator "**|**". The resulting regular expression matches any string matching either subexpression.

A.2.10 A regular expression may be followed by a repetition operator. If the operator is "**?**", the preceding item is optional and matched at most once. If the operator is "*****", the preceding item will be matched zero or more times. If the operator is "**+**", the preceding item will be matched one or more times. If the operator is of the form "**#(**n**)**", the preceding item is matched exactly n times; in this particular case, the parentheses can be omitted if n consists of one digit. If it is of the form "**#(**n**,)**", the item is matched n or more times. If it is of the form "**#(,**m**)**", the item is optional and is matched at most m times. Finally, if it is of the form "**#(**n**,**m**)**", the item is matched at least n times, but not more than m times.

NOTE – It is illegal to use the metacharacters "*****", "**+**", "**?**" or "**#**" as the first character of a regular expression. It is also illegal to use the metacharacters "**#**" or "**|**" as the last character of a regular expression.

EXAMPLES

A phone number like **"555-1212"** is matched by the regular expression **"\d#3-\d#4"**, or equivalently **"\d#(3)-\d#(4)"**.

A price in dollars like **"$12345.90"** is matched by the regular expression **"$\d#(1,)(\.\d#(1,2))?"**. Note that parentheses are requested after the "**#**" symbol when it is followed by a range.

A social security number like **"123-45-5678"** is matched by the regular expression **"\d#3-?\d#2-?\d#4"**.

A.2.11 Repetition (see A.2.10) takes precedence over concatenation (see A.1.3), which in turn takes precedence over alternation (see A.2.9). A whole subexpression may be enclosed in parentheses to override these precedence rules.

A.2.12 When a regular expression contains subexpressions in parentheses, each (non-quoted) opening parenthesis is successively assigned a distinct (strictly positive) integer from the left to the right of the regular expression. Each subexpression can then be referenced inside a comment with a notation like "*\1*", "*\2*" which uses the associated integer. The empty subexpression "**()**" is not permitted.

EXAMPLE

```
"((\d#2)(\d#2)(\d#4))"  -- \1 is a date in which \2 is the month, \3 the day
                        -- and \4 the year.
```

NOTE – There is a requirement for formal reference to subexpressions of a regular expression for many purposes. One such instance is the need to write text to document the regular expression within the ASN.1 module. This is a notation which can be used to provide such references. This notation is not used elsewhere in this Recommendation | International Standard.

Annex B

Rules for type and value Compatibility

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

This annex is expected to be mainly of use to tool builders to ensure that they interpret the language identically. It is present in order to clearly specify what is legal ASN.1 and what is not, and to be able to specify the precise value that any value reference name identifies, and the precise set of values that any type or value set reference name identifies. It is not intended to provide a definition of valid transformations of ASN.1 notations for any purpose other than those stated above.

B.1 The need for the value mapping concept (tutorial introduction)

B.1.1 Consider the following ASN.1 definitions:

```
A ::= INTEGER

B ::= [1] INTEGER

C ::= [2] INTEGER (0..6,...)

D ::= [2] INTEGER (0..6,...,7)

E ::= INTEGER (7..20)

F ::= INTEGER {red(0), white(1), blue(2), green(3), purple(4)}

a A ::= 3

b B ::= 4

c C ::= 5

d D ::= 6

e E ::= 7

f F ::= green
```

B.1.2 It is clear that the value references **a**, **b**, **c**, **d**, **e**, and **f** can be used in value notation governed by **A**, **B**, **C**, **D**, **E**, and **F**, respectively. For example:

```
W ::= SEQUENCE {w1  A  DEFAULT a}
```

and:

```
x A ::= a
```

and:

```
Y ::= A(1..a)
```

are all valid given the definitions in B.1.1. If, however, **A** above were replaced by **B**, or **C**, or **D**, or **E**, or **F**, would the resulting statements be illegal? Similarly, if the value reference **a** above were replaced in each of these cases by **b**, or **c**, or **d**, or **e**, or **f**, are the resulting statements legal?

B.1.3 A more sophisticated question would be to consider in each case replacement of the type reference by the explicit text to the right of its assignment. Consider for example:

```
f INTEGER {red(0), white(1), blue(2), green(3), purple(4)} ::= green

W ::= SEQUENCE {
      w1    INTEGER {red(0), white(1), blue(2), green(3),  purple(4)}
      DEFAULT f}

x INTEGER {red(0), white(1), blue(2), green(3),  purple(4)} ::= f

Y ::= INTEGER {red(0), white(1), blue(2), green(3),  purple(4)}(1..f)
```

Would the above be legal ASN.1?

B.1.4 Some of the above examples are cases which, even if legal (as most of them are – see later text), users would be ill-advised to write similar text, as they are at the least obscure and at worst confusing. However, there are frequent uses of a value reference to a value of some type (not necessarily just an **INTEGER** type) as the default value for that type

with tagging or subtyping applied in the governor. The *value mapping* concept is introduced in order to provide a clear and precise means of determining which constructs such as the above are legal.

B.1.5 Again, consider:

```
C ::= [2] INTEGER (0..6,...)
E ::= INTEGER (7..20)
F ::= INTEGER {red(0), white(1), blue(2), green(3), purple(4)}
```

In each case a new type is being created. For **F** we can clearly identify a 1-1 correspondence between the values in it and the values in the universal type **INTEGER**. In the case of **C** and **E**, we can clearly identify a 1-1 correspondence between the values in them and a subset of the values in the universal type **INTEGER**. We call this relationship a *value mapping* between values in the two types. Moreover, because values in **F**, **C**, and **E** all have (1-1) mappings to values of **INTEGER**, we can use these mappings to provide mappings between the values of **F**, **C**, and **E** themselves. This is illustrated for **F** and **C** in Figure B.1.

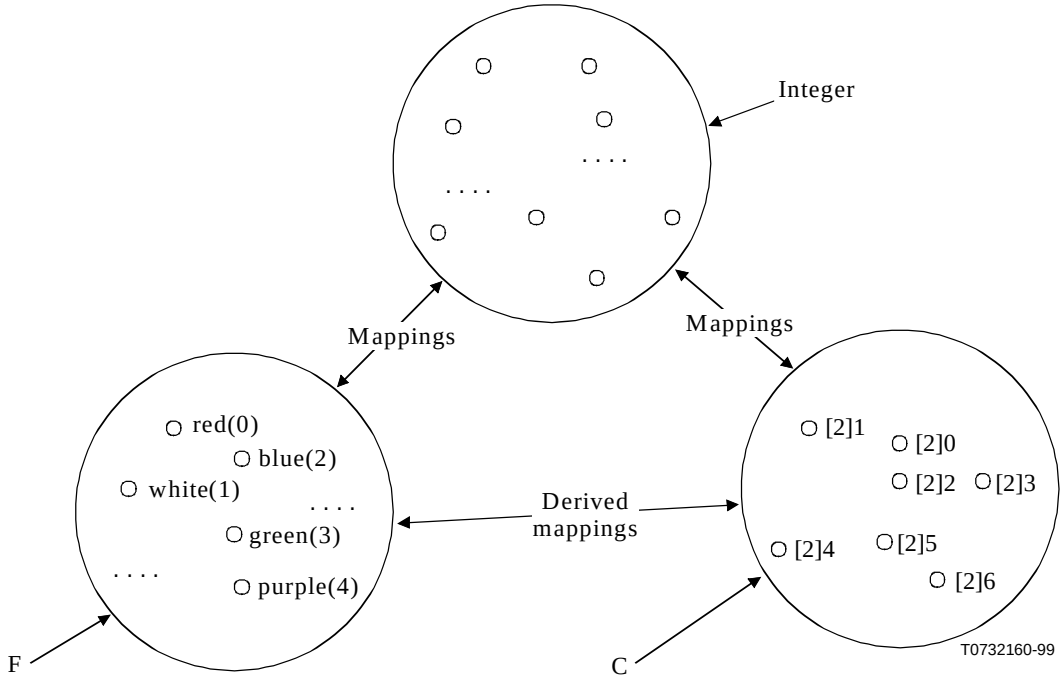


Figure B.1

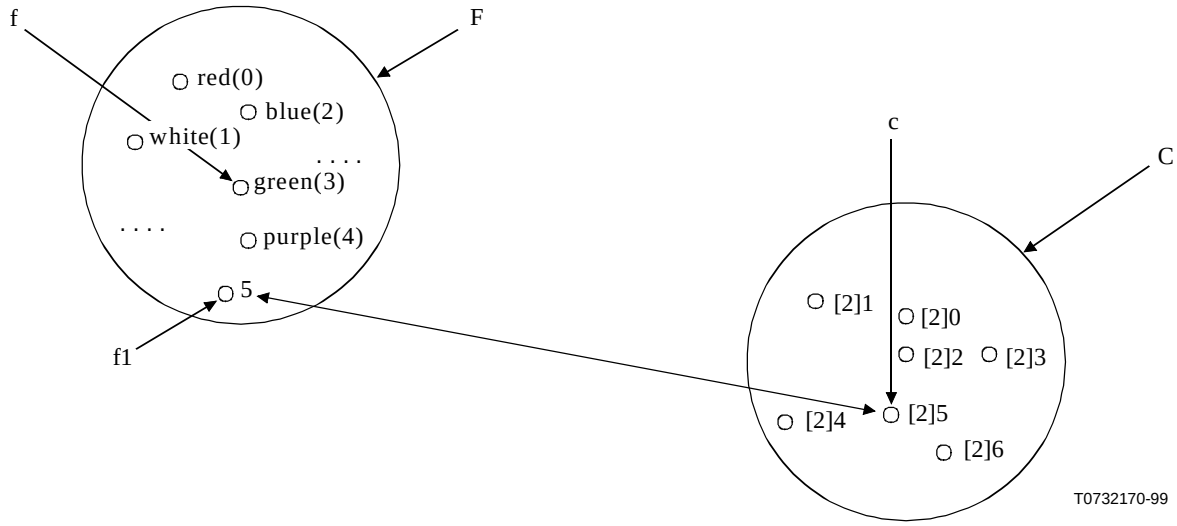
B.1.6 Now when we have a value reference such as:

```
c C ::= 5
```

to a value in **C** which is required in some context to identify a value in **F**, then, provided a value mapping exists between that value in **C** and a (single) value in **F**, we can (and do) define **c** to be a legal reference to the value in **F**. This is illustrated in Figure B.2, where the value reference **c** is used to identify a value in **F**, and can be used in place of a direct reference **f1** where we would otherwise have to define:

```
f1 F ::= 5
```

Figure B.2



B.1.7 It should be noted that in some cases there will be values in one type (7 to 20 in **A** of B.1.1 for example) that have value mappings to values in another type (7 to 20 in **E** of B.1.1 for example), but other values (21 upwards of **A**) that have no such mapping. A reference to such values in **A** would not provide a valid reference to a value in **E**. (In this example, the whole of **E** has a value mapping to a subset of **A**. In the general case, there may be a subset of values in both types that have mappings, with other values in both types that are unmapped.)

B.1.8 In the body of the ASN.1 standards, normal English text is used to specify legality in the above and similar cases. Subclause B.6 gives the precise requirements for legality and should be referenced whenever there is doubt about a complex construction.

NOTE – The fact that value mappings are defined to exist between two occurrences of the "Type" construct permits the use of value references established using one "Type" construct to identify values in another "Type" construct which is sufficiently similar. It allows dummy and actual parameters to be typed using two textually separate "Type" constructs without violating the rules for compatibility of dummy and actual parameters. It also allows fields of information object classes to be specified using one "Type" construct and the corresponding value in an information object to be specified using a distinct "Type" construct which is sufficiently similar. (These examples are not intended to be exhaustive.) It is, however, recommended that advantage be taken of this freedom only for simple cases such as **SEQUENCE OF INTEGER**, or **CHOICE {int INTEGER, id OBJECT IDENTIFIER}**, and not for more complex "Type" constructs.

B.2 Value mappings

B.2.1 The underlying model is of types, as non-overlapping containers, that contain values, with every occurrence of the ASN.1 "Type" construct defining a distinct new type (see Figures B.1 and B.2). This annex specifies when *value mappings* exist between such types, enabling a reference to a value in one type to be used where a reference to a value in some other type is needed.

Example: Consider:

```
X ::= INTEGER

Y ::= INTEGER
```

X and **Y** are type reference names (pointers) to two distinct types, but value mappings exist between these types, so any value reference to a value of **X** can be used when governed by **Y** (for example, following **DEFAULT**).

B.2.2 In the set of all possible ASN.1 values, a value mapping relates a pair of values. The whole set of value mappings is a mathematical relation. This relation possesses the following properties: it is reflexive (each ASN.1 value is related to itself), it is symmetric (if a value mapping is defined to exist from a value **x1** to a value **x2**, then there automatically exists a value mapping from **x2** to **x1**), and it is transitive (if there is a value mapping from a value **x1** to **x2**, and a value mapping from **x2** to **x3**, then there automatically exists a value mapping from **x1** to **x3**).

B.2.3 Furthermore, given any two types **X1** and **X2**, seen as sets of values, the set of value mappings from values in **X1** to values in **X2** is a one-to-one relation, that is, for all values **x1** in **X1**, and **x2** in **X2**, if there is a value mapping from **x1** to **x2**, then:

a) there is no value mapping from **x1** to another value in **X2** different from **x2**; and

b) there is no value mapping from any value in **X1** (other than **x1**) to **x2**.

B.2.4 Where a value mapping exists between a value **x1** and a value **x2**, a value reference to either one can automatically be used to reference the other if so required by some governing type.

NOTE – The fact that value mappings are defined to exist between values in some "Type" constructs is solely for the purpose of providing flexibility in the use of the ASN.1 notation. The existence of such mappings carries no implications whatsoever that the two types carry the same application semantics, but it is recommended that ASN.1 constructs which would be illegal without value mappings are used only if the corresponding types do indeed carry the same application semantics. Note that value mappings will frequently exist in any large specification between two types that are identical ASN.1 constructs, but which carry totally different application semantics, and where the existence of these value mappings is never used in determining the legality of the total specification.

B.3 Identical type definitions

B.3.1 The concept of identical type definitions is used to enable value mappings to be defined between two instances of "Type" which are either identical or sufficiently similar that one would normally expect their use to be interchangeable. In order to give precision to the meaning of "sufficiently similar", this subclause specifies a series of transformations which are applied to each of the instances of "Type" to produce a *normal form* for those instances of "Type". The two instances of "Type" are defined to be identical type definitions if, and only if, their normal forms are identical ordered lists of the same lexical items (see clause 11).

B.3.2 Each occurrence of "Type" in an ASN.1 specification is an ordered list of the lexical items defined in clause 11. The normal form is obtained by applying the transformations defined in B.3.2.1 to B.3.2.6 in that order.

B.3.2.1 All the comments (see 11.6) are removed.

B.3.2.2 The following transformations are not recursive and hence need only to be applied once, in any order:

a) For a type defined by a "ValueSetTypeAssignment", its definition is replaced by a "TypeAssignment" using the same "Type" and a subtype constraint which is the contents of the "ValueSet" as specified in 15.6.

b) For each integer type: the "NamedNumberList" (see 18.1), if any, is reordered so that the "identifier"s are in alphabetical order ("a" first, "z" last).

c) For each enumerated type: numbers are added, as specified in 19.3, to any "EnumerationItem" (see 19.1) that is an "identifier" (without a number); then the "RootEnumeration" is reordered so that the "identifiers" are in alphabetical order ("a" first, "z" last).

d) For each bitstring type: the "NamedBitList" (see 21.1), if any, is reordered so that the "identifiers" are in alphabetical order ("a" first, "z" last).

e) For each object identifier value: each "ObjIdComponents" is transformed into its corresponding "NumberForm" in accordance with the semantics of clause 31 (see the example in 31.12).

f) For each relative object identifier value (see 32.3): each "RelativeOIDComponents" is transformed into its corresponding "NumberForm" in accordance with the semantics of clause 32.

g) For sequence types (see clause 24) and set types (see clause 26): any extension of the form "ExtensionAndException", "ExtensionAdditions", is cut and pasted to the end of the "ComponentTypeLists"; "OptionalExtensionMarker", if present, is removed.

   If "TagDefault" is **IMPLICIT TAGS**, the keyword **IMPLICIT** is added to all instances of "Tag" (see clause 30) unless either:

   - it is already present; or
   - the reserved word **EXPLICIT** is present; or
   - the type being tagged is a **CHOICE** type or;
   - it is an open type.

   If "TagDefault" is **AUTOMATIC TAGS**, the decision on whether to apply automatic tagging is taken according to 24.2 (the automatic tagging will be performed later on).

   NOTE – Subclauses 24.3 and 26.2 specify that the presence of a "Tag" in a "ComponentType" which was inserted as a result of the replacement of "Components of Type" does not in itself prevent the automatic tagging transformation.

   If "ExtensionDefault" is **EXTENSIBILITY IMPLIED**, an ellipsis ("**...**") is added after the "ComponentTypeLists" if it is not present.

h) For choice type (see clause 28): "RootAlternativeTypeList" is reordered so that the identifiers of the "NameType"s are in alphabetical order ("a" first, "z" last). "OptionalExtensionMarker", if present, is

removed. If "TagDefault" is **IMPLICIT TAGS**, the keyword **IMPLICIT** is added to all instances of "Tags" (see clause 30) unless either:

- it is already present; or
- the reserved word **EXPLICIT** is present; or
- the type being tagged is a **CHOICE** type; or
- it is an open type.

If "TagDefault" is **AUTOMATIC TAGS**, the decision on whether to apply automatic tagging is taken according to 28.5 (the automatic tagging will be performed later on). If "ExtensionDefault" is **EXTENSIBILITY IMPLIED**, an ellipsis ("**...**") is added after the "AlternativeTypeLists" if it is not present.

B.3.2.3 The following transformations shall be applied recursively in the specified order, until a fix-point is reached:

a) For each object identifier value (see 31.3): if the value definition begins with a "DefinedValue", the "DefinedValue" is replaced by its definition.

b) For each relative object identifier value (see 32.3): if the value definition contains "DefinedValue"s, the "DefinedValue"s are replaced by their definition.

c) For sequence types and set types: all instances of "**COMPONENTS OF** Type" (see clause 24) are transformed according to clauses 24 and 26.

d) For sequence, set and choice types: if it has earlier been decided to tag automatically (see B.3.2.2 g) and h)), the automatic tagging is applied according to clauses 24, 26 and 28.

e) For selection type: the construction is replaced by the selected alternative according to clause 29.

f) All type references are replaced by their definitions according to the following rules:

- If the replacing type is a reference to the type being transformed, the type reference is replaced by a special item that matches no other item than itself.
- If the replacing type is a sequence-of type or a set-of type, the constraints following the replaced type, if any, are moved in front of the keyword **OF**.
- If the replaced type is a parameterized type or a parameterized value set (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.2), every "DummyReference" is replaced by the corresponding "ActualParameter".

g) All value references are replaced by their definitions; if the replaced value is a parameterized value (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.2), every "DummyReference" is replaced by the corresponding "ActualParameter".

NOTE – Before replacing any value reference, the procedures of this annex shall be applied to ensure that the value reference identifies, through value mappings or directly, a value in its governing type.

B.3.2.4 For set type: the "RootComponentTypeList" is reordered so that the "ComponentType"s are in alphabetical order ("a" first, "z" last).

B.3.2.5 The following transformations shall be applied to value definitions:

a) If an integer value is defined with an identifier, that identifier is replaced by the associated number.

b) If a bitstring value is defined using identifiers, it is replaced by the corresponding "bstring" with all trailing zero bits removed.

c) All white-space immediately before and after each newline (including the newline) in a "cstring" is removed.

d) All white-space in "bstring" and "hstring" is removed.

e) Each real value defined with base 2 is normalized so that the mantissa is odd, and each real value defined with base 10 is normalized so that the last digit of the mantissa is not 0.

f) Each **GeneralizedTime** and **UTCTime** value is replaced by a string which conforms to the rules used when encoding in DER and CER (see ITU-T Rec. X.690 | ISO/IEC 8825-1 11.7 and 11.8).

g) After applying c), each **UTF8String**, **NumericString**, **PrintableString**, **IA5String**, **VisibleString** (**ISO646String**), **BMPString** and **UniversalString** value is replaced by the equivalent value of type **UniversalString** written using the "Quadruple" notation (see clause 37.8).

B.3.2.6 Any occurrence of "realnumber" shall be transformed to a "base" 10 associated "SequenceValue". Any occurrence of the "RealValue" associated with "SequenceValue" shall be transformed to the associated "SequenceValue" of the same "base", such that the last digit of the mantissa is not zero.

B.3.3 If two instances of "Type", when transformed to their normal form, are identical lists of lexical items (see clause 11), then the two instances of "Type" are defined to be identical type definitions with the following exception: if an "objectclassreference" (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 7.1), an "objectreference" (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 7.2) or an "objectsetreference" (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 7.3) appears within the normalized form of the "Type", then the two types are not defined to be identical type definitions, and value mappings (see B.4 below) will not exist between them.

NOTE – This exception was inserted to avoid the need to provide transformation rules to normal form for elements of syntax concerned with information object class, information object, and information object set notation. Similarly, specification for the normalization of all value notation and of set arithmetic notation has not been included at this time. Should there prove to be a requirement for such specification, this could be provided in a future version of this Recommendation | International Standard. The concept of identical type definitions and of value mappings was introduced to ensure that simple ASN.1 constructs could be used either by using reference names or by copying text. It was felt unnecessary to provide this functionality for more complex instances of "Type" that included information object classes, etc.

B.4 Specification of value mappings

B.4.1 If two occurrences of "Type" are identical type definitions under the rules of B.3, then value mappings exist between every value of one type and the corresponding value of the other type.

B.4.2 For a type, **X1**, created from any type, **X2**, by tagging (see clause 30), value mappings are defined to exist between all the members of **X1** and the corresponding members of **X2**.

NOTE – Whilst value mappings are defined to exist between the values of **X1** and **X2** in B.4.2 above, and between the values of **X3** and **X4** in B.4.3, if such types are embedded in otherwise identical but distinct type definitions (such as **SEQUENCE** or **CHOICE** type definitions), the resulting type definitions (the **SEQUENCE** or **CHOICE** types) will not be identical type definitions, and there will be no value mappings between them.

B.4.3 For a type, **X3**, created by selecting values from any governing type, **X4**, by the element set construct or by subtyping, value mappings are defined to exist between the members of the new type and those members of the governing type that were selected by the element set or subtyping construct. The presence or absence of an extension marker has no effect on this rule.

B.4.4 Additional value mappings are specified in B.5 between some of the character string types.

B.4.5 A value mapping is defined to exist between all the values of any type defined as an integer type with named values and any integer type defined without named values, or with different named values, or with different names for named values, or both.

NOTE – The existence of the value mapping does not affect any scope rule requirements on the use of the names of named values. They can only be used in a scope governed by the type in which they are defined, or by a typereference name to that type.

B.4.6 A value mapping is defined to exist between all the values of any type defined as a bit string type with named bits and any bit string type defined without named bits, or with different named bits, or with different names for named bits, or both.

NOTE – The existence of the value mapping does not affect any scope rule requirements on the use of the names of named bits. They can only be used in a scope governed by the type in which they are defined, or by a typereference name to that type.

B.5 Additional value mappings defined for the character string types

B.5.1 There are two groups of restricted character string types, group A (see B.5.2) and group B (see B.5.3). Value mappings are defined to exist between all types in group A, and value references to values of these types can be used when governed by one of the other types. For the types in group B, value mappings never exist between these different types, nor between any type in group A and any type in group B.

B.5.2 Group A consists of:

UTF8String
NumericString
PrintableString
IA5String
VisibleString (**ISO646String**)
UniversalString
BMPString

B.5.3 Group B consists of:

TeletexString (**T61String**)
VideotexString
GraphicString
GeneralString

B.5.4 The value mappings in group A are specified by mapping the character string values of each type to **UniversalString**, then using the transitivity property of value mappings. To map values from one of the group A types to **UniversalString**, the string is replaced by a **UniversalString** of the same length with each character mapped as specified below.

B.5.5 Formally, the set of abstract values in **UTF8String** is the same set of abstract values that occur in **UniversalString** but with a different tag (see 37.16), and each abstract value in **UTF8String** is defined to map to the corresponding abstract value in **UniversalString**.

B.5.6 The glyphs (printed character shapes) for characters used to form the types **NumericString** and **PrintableString** have recognizable and unambiguous mappings to a subset of the glyphs assigned to the first 128 characters of ISO/IEC 10646-1. The mapping for these types is defined using this mapping of glyphs.

B.5.7 **IA5String** and **VisibleString** are mapped into **UniversalString** by mapping each character into the **UniversalString** character that has the identical (32-bit) value in the BER encoding of **UniversalString** as the (8-bit) value of the BER encoding of **IA5String** and **VisibleString**.

B.5.8 **BMPString** is formally a subset of **UniversalString**, and corresponding abstract values have value mappings.

B.6 Specific type and value compatibility requirements

This subclause uses the value mapping concept to provide precise text for the legality of certain ASN.1 constructs.

B.6.1 Any "Value" occurrence, *x-notation*, with a governing type, **Y**, identifies the value, *y-val*, in the governing type **Y** that has a value mapping to the value *x-val* specified by *x-notation*. It is a requirement that such a value exists.

For example, consider the occurrence of **x** in the last line of the following:

```
X   ::= [0] INTEGER (0..30)

x X ::= 29

Y   ::= [1] INTEGER (25..35)

Z1  ::= Y (x | 30)
```

These ASN.1 constructs are legal, and in the last assignment the *x-notation* **x** is referencing the *x-val* 29 in **X** and, through value mapping, identifies the *y-val* 29 in **Y**. The *x-notation* 30 is referencing the *y-val* 30 in **Y**, and **Z1** is the set of values 29 and 30. On the other hand, the assignment:

```
Z2 ::= Y (x | 20)
```

is illegal because there is no *y-val* to which the *x-notation* 20 can refer.

B.6.2 Any "Type" occurrence, *t-notation*, that has a governing type, **V**, identifies the complete set of values in the root of the governing type **V** that have value mappings to any of the values in the root of the "Type" *t-notation*. This set is required to contain at least one value.

For example, consider the occurrence of **W** in the last line of the following:

```
V  ::= [0] INTEGER (0..30)

W  ::= [1] INTEGER (25..35)

Y  ::= [2] INTEGER (31..35)

Z1 ::= V (W | 24)
```

W contributes values 25-30 to the set arithmetic resulting in **Z1** having the values 24-30. On the other hand, the assignment:

```
Z2 ::= V (Y | 24)
```

is illegal because there are no values in **Y** which map to a value in **V**.

B.6.3 The type of any value supplied as an actual parameter is required to have a value mapping from that value to one of the values in the type governing the dummy parameter, and it is a value of that governing type which is identified.

B.6.4 If a "Type" is supplied as an actual parameter for a dummy parameter which is a value set dummy parameter, then all values of that "Type" are required to have value mappings to values in the governor of the value set dummy parameter. The actual parameter selects the total set of values in the governor which have mappings to the "Type".

B.6.5 In specifying the type, **A**, of a dummy parameter that is a value or a value set parameter, it is an illegal specification unless for all values of **A**, and for every instance of use of **A** on the right-hand side of the assignment, that value of **A** can legally be applied in place of the dummy parameter.

B.7 Examples

B.7.1 This subclause provides examples to illustrate B.3 and B.4.

B.7.2 Example 1

```
X ::= SEQUENCE                  X1 ::=  SEQUENCE
      {name VisibleString,          {name VisibleString,
      age  INTEGER}                 -- comment --
                                          age  INTEGER}

X2 ::= [8] SEQUENCE             X3 ::=  SEQUENCE
      {name VisibleString,          {name VisibleString,
      age  INTEGER}                 age  AgeType}

                                AgeType ::= INTEGER
```

X, **X1**, **X2**, and **X3** are all identical type definitions. Differences of white-space and comment are not visible, nor does the use of the **AgeType** type reference in **X3** affect the type definition. Note, however, that if any of the identifiers for the elements of the sequence were changed, the types would cease to be identical definitions, and there would be no value mappings between them.

B.7.3 Example 2

```
B ::= SET                       B1 ::= SET
      {name VisibleString,              {age INTEGER,
      age  INTEGER}                     name VisibleString}
```

are identical type definitions provided neither is in a module with **AUTOMATIC TAGS** in the module header, otherwise they are not identical type definitions, and value mappings will not exist between them. Similar examples can be written using **CHOICE** and **ENUMERATED** (using the "identifier" form of "EnumerationItem").

B.7.4 Example 3

```
C ::= SET                       C1 ::= SET
      {name [0]VisibleString,           {name VisibleString,
      age INTEGER}                      age INTEGER (1..64)}
```

are not identical type definitions, nor are either of them identical type definitions to either of **B** or **B1**, and there are no value mappings between any of the values of **C** and **C1**, nor between either of them and either of **B** or **B1**.

B.7.5 Example 4

```
x  INTEGER { y (2) } ::= 3

z  INTEGER ::= x
```

is legal, and assigns the value 3 to **z** through the value mapping defined in B.4.5.

B.7.6 Example 5

```
b1 BIT STRING ::= '101'B

b2 BIT STRING {version1(0), version2(1), version3(2)} ::= b1
```

is legal, and assigns the value **{version1, version3}** to **b2**.

B.7.7 Example 6

With the definitions of B.1.1, **SEQUENCE** elements of the form:

```
X DEFAULT y
```

are legal, where **X** is any of **A**, **B**, **C**, **D**, **E**, or **F**, or any of the text to the right of the type assignments to these names, and **y** is any of **a**, **b**, **c**, **d**, **e**, or **f**, with the following exceptions: **E DEFAULT y** is illegal for all of **a**, **b**, **c**, **d**, **f**, and **C DEFAULT e** is illegal, because in these cases there are no value mappings available from the defaulting value reference into the type being defaulted.

Annex C

Assigned object identifier values

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

This annex records object identifier and object descriptor values assigned in the ASN.1 series of Recommendations | International Standards, and provides an ASN.1 module for use in referencing those object identifier values.

C.1 Object identifiers assigned in this Recommendation | International Standard

The following values are assigned in this Recommendation | International Standard:

Subclause	Object Identifier Value
37.3	{ joint-iso-itu-t asn1(1) specification(0) characterStrings(1) numericString(0) }
	Object Descriptor Value
	"NumericString ASN.1 type"
Subclause	**Object Identifier Value**
37.5	{ joint-iso-itu-t asn1(1) specification(0) characterStrings(1) printableString(1) }
	Object Descriptor Value
	"PrintableString ASN.1 type"
Subclause	**Object Identifier Value**
38.1	{ joint-iso-itu-t asn1(1) specification(0) modules(0) iso10646(0) }
	Object Descriptor Value
	"ASN.1 Character Module"
Subclause	**Object Identifier Value**
C.2	{ joint-iso-itu-t asn1(1) specification(0) modules(0) object-identifiers(1) }
	Object Descriptor Value
	"ASN.1 Object Identifier Module"

C.2 Object identifiers in the ASN.1 and encoding rules standards

This clause specifies an ASN.1 module which contains the definition of a value reference name for each object identifier value defined in the ASN.1 standards (ITU-T Rec. X.680 | ISO/IEC 8824-1 to ITU-T Rec. X.693 | ISO/IEC 8825-4).

NOTE – These values are available for use in the value notation of the OBJECT IDENTIFIER type and types derived from it. All of the value references defined in the module specified in this clause are exported and have to be imported by any module that wishes to use them.

```
ASN1-Object-Identifier-Module { joint-iso-itu-t asn1(1) specification(0) modules(0)
object-identifiers(1) }
      DEFINITIONS ::= BEGIN
      -- NumericString ASN.1 type (see 37.3) --

      numericString OBJECT IDENTIFIER ::=
                  { joint-iso-itu-t asn1(1) specification(0) characterStrings(1)
      numericString(0) }
      -- PrintableString ASN.1 type (see 37.5) --

      printableString OBJECT IDENTIFIER ::=
                  { joint-iso-itu-t asn1(1) specification(0) characterStrings(1)
      printableString(1) }
```

```
    -- ASN.1 Character Module (see 38.1) --

    asn1CharacterModule OBJECT IDENTIFIER ::=
                { joint-iso-itu-t asn1(1) specification(0) modules(0) iso10646(0) }
    -- ASN.1 Object Identifier Module (this module) --

    asn1ObjectIdentifierModule OBJECT IDENTIFIER ::=
                { joint-iso-itu-t asn1(1) specification(0) modules(0)
    object-identifiers(1) }
    -- BER encoding of a single ASN.1 type --

    ber OBJECT IDENTIFIER ::=
                { joint-iso-itu-t asn1(1) basic-encoding(1) }
    -- CER encoding of a single ASN.1 type --

    cer OBJECT IDENTIFIER ::=
                { joint-iso-itu-t asn1(1) ber-derived(2) canonical-encoding(0) }
    -- DER encoding of a single ASN.1 type --

    der OBJECT IDENTIFIER ::=
                { joint-iso-itu-t asn1(1) ber-derived(2) distinguished-encoding(1)
    }
    -- PER encoding of a single ASN.1 type (basic aligned) --

    perBasicAligned OBJECT IDENTIFIER ::=
                { joint-iso-itu-t asn1(1) packed-encoding(3) basic(0) aligned(0) }
    -- PER encoding of a single ASN.1 type (basic unaligned) --

    perBasicUnaligned OBJECT IDENTIFIER ::=
                { joint-iso-itu-t asn1(1) packed-encoding(3) basic(0) unaligned(1)
    }
    -- PER encoding of a single ASN.1 type (canonical aligned) --

    perCanonicalAligned OBJECT IDENTIFIER ::=
                { joint-iso-itu-t asn1(1) packed-encoding(3) canonical(1)
    aligned(0) }
    -- PER encoding of a single ASN.1 type (canonical unaligned) --

    perCanonicalUnaligned OBJECT IDENTIFIER ::=
                { joint-iso-itu-t asn1(1) packed-encoding(3) canonical(1)
    unaligned(1) }
    -- XER encoding of a single ASN.1 type (basic) --

    xerBasic OBJECT IDENTIFIER ::=
                {joint-iso-itu-t asn1(1) xml-encoding(5) basic(0) }
    -- XER encoding of a single ASN.1 type (canonical) --

    xerCanonical OBJECT IDENTIFIER ::=
                {joint-iso-itu-t asn1(1) xml-encoding(5) canonical(1) }

END -- ASN1-Object-Identifier-Module --
```

Annex D

Assignment of object identifier component values

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

This Annex describes the top-level arcs of the registration tree for object identifiers. No explanation is given on how new arcs are added, nor on the rules that registration authorities should follow. These are specified in ITU-T Rec. X.660 | ISO/IEC 9834-1.

D.1 Root assignment of object identifier component values

D.1.1 Three arcs are specified from the root node. The assignment of values and identifiers, and the authority for assignment of subsequent component values, are as follows:

Value	*Identifier*	*Authority for subsequent assignments*
0	**itu-t**	ITU-T (See D.2)
1	**iso**	ISO (See D.3)
2	**joint-iso-itu-t**	See D.4

D.1.2 The identifiers **itu-t**, **iso** and **joint-iso-itu-t**, assigned above, may each be used as a "NameForm" (see 31.3).

D.1.3 The identifiers **ccitt** and **joint-iso-ccitt** are synonyms for **itu-t** and **joint-iso-itu-t**, respectively, and thus may appear in object identifier values.

D.2 ITU-T assignment of object identifier component values

D.2.1 Five arcs are specified from the node identified by **itu-t**. The assignment of values and identifiers is:

Value	*Identifier*	*Authority for subsequent assignments*
0	**recommendation**	See D.2.2
1	**question**	See D.2.3
2	**administration**	See D.2.4
3	**network-operator**	See D.2.5
4	**identified-organization**	See D.2.6

These identifiers may be used as a "NameForm" (see 31.3).

D.2.2 The arcs below **recommendation** have the value **1** to **26** with assigned identifiers of **a** to **z**. Arcs below these have the numbers of ITU-T (and CCITT) Recommendations in the series identified by the letter. Arcs below this are determined as necessary by the ITU-T (and CCITT) Recommendations. The identifiers **a** to **z** may be used as a "NameForm".

D.2.3 The arcs below **question** have values corresponding to ITU-T Study Groups, qualified by the study period. The value is computed by the formula:

study group number + (period * 32)

where "period" has the value 0 for 1984-1988, 1 for 1988-1992, etc., and the multiplier is 32 decimal.

The arcs below each study group have the values corresponding to the questions assigned to that study group. Arcs below this are determined as necessary by the group (e.g., working party or special rapporteur group) assigned to study the question.

D.2.4 The arcs below **administration** have the values of X.121 DCCs. Arcs below this are determined as necessary by the Administration of the country identified by the X.121 DCC.

D.2.5 The arcs below **network-operator** have the value of X.121 DNICs. Arcs below this are determined as necessary by the Administration or ROA identified by the DNIC.

D.2.6 The arcs below **identified-organization** are assigned values by the ITU Telecommunication Standardization Bureau (TSB). Arcs below this are determined as necessary by the organizations identified by the value of the ITU.

NOTE – Organizations which may find this arc useful include:
– recognized operating agencies not operating a public data network;
– scientific and industrial organizations;
– regional standards organizations; and
– multi-national organizations.

D.3 ISO assignment of object identifier component values

D.3.1 Three arcs are specified from the node identified by "iso". The assignment of values and identifiers is:

Value	*Identifier*	*Authority for subsequent assignments*
0	**standard**	See D.3.2
2	**member-body**	See D.3.3
3	**identified-organization**	See D.3.4

These identifiers may be used as a "NameForm".

NOTE – The use of arc **registration-authority(1)** has been withdrawn.

D.3.2 The arcs below **standard** shall each have the value of the number of an International Standard. Where the International Standard is multi-part, there shall be an additional arc for the part number, unless this is specifically excluded in the text of the International Standard. Further arcs shall have values as defined in that International Standard.

D.3.3 The arcs immediately below **member-body** shall have values of a three digit numeric country code, as specified in ISO 3166, that identifies the ISO National Body in that country. The "NameForm" of object identifier component is not permitted with these identifiers.

D.3.4 The arcs immediately below **identified-organization** shall have values of an International Code Designator (ICD) allocated by the Registration Authority for ISO 6523 that identify an issuing organization specifically registered by that authority as allocating object identifier components. The arcs immediately below the ICD shall have values of an "organization code" allocated by the issuing organization in accordance with ISO 6523.

D.4 Joint assignment of object identifier component values

D.4.1 The arcs below **joint-iso-itu-t** have values which are assigned and agreed from time to time by a Registration Authority established by ISO/IEC and ITU-T to identify areas of joint ISO/IEC | ITU-T standardization activity, in accordance with ITU-T X.662 | ISO/IEC 9834-3.

Annex E

Examples and hints

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

This annex contains examples of the use of ASN.1 in the description of (hypothetical) data structures. It also contains hints, or guidelines, for the use of the various features of ASN.1. Unless otherwise stated, an environment of **AUTOMATIC TAGS** is assumed.

E.1 Example of a personnel record

The use of ASN.1 is illustrated by means of a simple, hypothetical personnel record.

E.1.1 Informal description of Personnel Record

The structure of the personnel record and its value for a particular individual are shown below.

Name:	John P Smith
Title:	Director
Employee Number:	51
Date of Hire:	17 September 1971
Name of Spouse:	Mary T Smith
Number of Children:	2
Child Information	
Name:	Ralph T Smith
Date of Birth	11 November 1957
Child Information	
Name:	Susan B Jones
Date of Birth	17 July 1959

E.1.2 ASN.1 description of the record structure

The structure of every personnel record is formally described below using the standard notation for data types.

```
PersonnelRecord ::= [APPLICATION 0] SET
{     name              Name,
      title             VisibleString,
      number            EmployeeNumber,
      dateOfHire        Date,
      nameOfSpouse      Name,
      children          SEQUENCE OF ChildInformation DEFAULT {}
}

ChildInformation ::= SET
{     name              Name,
      dateOfBirth       Date
}

Name ::= [APPLICATION 1] SEQUENCE
{     givenName         VisibleString,
      initial           VisibleString,
      familyName        VisibleString
}

EmployeeNumber ::= [APPLICATION 2] INTEGER

Date ::= [APPLICATION 3] VisibleString  --  YYYY MMDD
```

This example illustrates an aspect of the parsing of the ASN.1 syntax. The syntactic construct **DEFAULT** can only be applied to a component of a **SEQUENCE** or a **SET**, it cannot be applied to an element of a **SEQUENCE OF**. Thus, the **DEFAULT { }** in **PersonnelRecord** applies to **children**, not to **ChildInformation**.

E.1.3 ASN.1 description of a record value

The value of John Smith's personnel record is formally described below using the standard notation for data values.

```
{     name             {givenName "John", initial "P", familyName "Smith"},
      title            "Director",
      number           51,
      dateOfHire       "19710917",
      nameOfSpouse     {givenName "Mary", initial "T", familyName "Smith"},
      children
      {  {name  {givenName "Ralph", initial "T", familyName "Smith"} ,
            dateOfBirth "19571111"},
         {name  {givenName "Susan", initial "B", familyName "Jones"} ,
            dateOfBirth "19590717" }
      }
}
```

or in XML value notation:

```
person ::=
      <PersonnelRecord>
         <name>
            <givenName>John</givenName>
            <initial>P</initial>
            <familyName>Smith</familyName>
         </name>
         <title>Director</title>
         <number>51</number>
         <dateOfHire>19710917</dateOfHire>
         <nameOfSpouse>
            <givenName>Mary</givenName>
            <initial>T</initial>
            <familyName>Smith</familyName>
         </nameOfSpouse>
         <children>
            <ChildInformation>
               <name>
                  <givenName>Ralph</givenName>
                  <initial>T</initial>
                  <familyName>Smith</familyName>
               </name>
               <dateOfBirth>19571111</dateOfBirth>
            </ChildInformation>
            <ChildInformation>
               <name>
                  <givenName>Susan</givenName>
                  <initial>B</initial>
                  <familyName>Jones</familyName>
               </name>
               <dateOfBirth>19590717</dateOfBirth>
            </ChildInformation>
         </children>
      </PersonnelRecord>
```

E.2 Guidelines for use of the notation

The data types and formal notation defined by this Recommendation | International Standard are flexible, allowing a wide range of protocols to be designed using them. This flexibility, however, can sometimes lead to confusion, especially when the notation is approached for the first time. This annex attempts to minimize confusion by giving guidelines for, and examples of, the use of the notation. For each of the built-in data types, one or more usage guidelines are offered. The character string types (for example, **VisibleString**) and the types defined in clauses 42 to 44 are not dealt with here.

E.2.1 Boolean

E.2.1.1 Use a boolean type to model the values of a logical (that is, two-state) variable, for example, the answer to a yes-or-no question.

EXAMPLE

```
Employed ::= BOOLEAN
```

E.2.1.2 When assigning a reference name to a boolean type, choose one that describes the *true* state.

EXAMPLE

```
Married ::= BOOLEAN
```

not

```
MaritalStatus ::= BOOLEAN
```

E.2.2 Integer

E.2.2.1 Use an integer type to model the values (for all practical purposes, unlimited in magnitude) of a cardinal or integer variable.

EXAMPLE

```
CheckingAccountBalance ::= INTEGER -- in cents; negative means overdrawn.
balance CheckingAccountBalance ::= 0
```

or using XML value notation:

```
balance ::= <CheckingAccountBalance>0</CheckingAccountBalance>
```

E.2.2.2 Define the minimum and maximum allowed values of an integer type as named numbers.

EXAMPLE

```
DayOfTheMonth ::= INTEGER {first(1), last(31)}
today DayOfTheMonth ::= first
unknown DayOfTheMonth ::= 0
```

or using XML value notation:

```
today ::= <DayOfTheMonth><first/></DayOfTheMonth>
unknown ::= <DayOfTheMonth>0</DayOfTheMonth>
```

Note that the named numbers **first** and **last** were chosen because of their semantic significance to the reader, and does not exclude the possibility of **DayOfTheMonth** having other values which may be less than 1, greater than 31 or between 1 and 31.

To restrict the value of **DayOfTheMonth** to just **first** and **last**, one would write:

```
DayOfTheMonth ::= INTEGER {first(1), last(31)} (first | last)
```

and to restrict the value of the **DayOfTheMonth** to all values between 1 and 31, inclusive, one would write:

```
DayOfTheMonth ::= INTEGER {first(1), last(31)} (first .. last)
dayOfTheMonth DayOfTheMonth ::= 4
```

or using XML value notation:

```
dayOfTheMonth ::= <DayOfTheMonth>4</DayOfTheMonth>
```

E.2.3 Enumerated

E.2.3.1 Use an enumerated type to model the values of a variable with three or more states. Assign values starting with zero if their only constraint is distinctness.

EXAMPLE

```
DayOfTheWeek ::= ENUMERATED {sunday(0), monday(1), tuesday(2),
                             wednesday(3), thursday(4), friday(5), saturday(6)}

firstDay DayOfTheWeek ::= sunday
```

or using XML value notation:

```
firstDay ::= <DayOfTheWeek><sunday/></DayOfTheWeek>
```

Note that while the enumerations **sunday**, **monday**, etc., were chosen because of their semantic significance to the reader, **DayOfTheWeek** is restricted to assuming one of these values and no other. Further, only the name **sunday**, **monday**, etc., can be assigned to a value; the equivalent integer values are not allowed.

E.2.3.2 Use an extensible enumerated type to model the values of a variable that has just two states now, but that may have additional states in a future version of the protocol.

EXAMPLE

```
MaritalStatus ::= ENUMERATED {single, married}
          -- First version of MaritalStatus
```

in anticipation of

```
MaritalStatus ::= ENUMERATED {single, married, …, widowed}
          -- Second version of MaritalStatus
```

and later yet:

```
MaritalStatus ::= ENUMERATED {single, married, …, widowed, divorced}
          -- Third version of MaritalStatus
```

E.2.4 Real

E.2.4.1 Use a real type to model an approximate number.

EXAMPLE

```
AngleInRadians ::=  REAL

pi REAL ::= {mantissa 314159265358979323846264338327, base 10, exponent –30}
```

or using the alternate value notation for **REAL**:

```
pi   REAL ::= 3.14159265358979323846264338327
```

or using XML value notation:

```
pi ::=
     <REAL>
          3.14159265358979323846264338327
     </REAL>
```

E.2.4.2 Application designers may wish to ensure full interworking with real values despite differences in floating point hardware, and in implementation decisions to use (for example) single or double length floating point for an application. This can be achieved by the following:

```
App-X-Real ::= REAL (WITH COMPONENTS {
                     mantissa (–16777215..16777215),
                     base (2),
                     exponent (–125..128) } )
/*
  Senders shall not transmit values outside these ranges
  and conforming receivers shall be capable of receiving
  and processing all values in these ranges.
*/

girth App-X-Real ::= {mantissa 16, base 2, exponent 1}
```

or using XML value notation:

```
girth ::=
     <App-X-Real>
        32
```

```
</App-X-Real>
```

E.2.5 Bit string

E.2.5.1 Use a bit string type to model binary data whose format and length are unspecified, or specified elsewhere, and whose length in bits is not necessarily a multiple of eight.

EXAMPLE

```
G3FacsimilePage ::= BIT STRING
--  a sequence of bits conforming to Recommendation T.4.

image G3FacsimilePage ::= '100110100100001110110'B

trailer BIT STRING ::= '0123456789ABCDEF'H

body1 G3FacsimilePage ::= '1101'B

body2 G3FacsimilePage ::= '1101000'B
```

or using XML value notation:

```
image ::= <G3FacSimile>100110100100001110110</G3FacSimile>

trailer ::=
           <BIT_STRING>
                0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011
                1100 1101 1110 1111
           </BIT_STRING>

body1 ::= <G3FacSimile>1101</G3FacSimile>

body2 ::= <G3FacSimile>1101000</G3FacSimile>
```

Note that **body1** and **body2** are distinct abstract values because trailing 0 bits are significant (due to there being no "NamedBitList" in the definition of **G3FacsimilePage**).

E.2.5.2 Use a bit string type with a size constraint to model the values of a fixed sized bit field.

EXAMPLE

```
BitField ::= BIT STRING (SIZE (12))

map1 BitField ::= '100110100100'B

map2 BitField ::= '9A4'H

map3 BitField ::= '1001101001'B   --  Illegal - violates size constraint.
```

or using XML value notation:

```
map1 ::= <BitField>100110100100</BitField>
```

Note that **map1** and **map2** are the same abstract value, for the four trailing bits of **map2** are not significant.

E.2.5.3 Use a bit string type to model the values of a **bit map**, an ordered collection of logical variables indicating whether a particular condition holds for each of a correspondingly ordered collection of objects.

```
DaysOfTheWeek ::= BIT STRING {
           sunday(0), monday (1), tuesday(2),
           wednesday(3), thursday(4), friday(5),
           saturday(6) } (SIZE (0..7))

sunnyDaysLastWeek1 DaysOfTheWeek ::= {sunday, monday, wednesday}
     sunnyDaysLastWeek2 DaysOfTheWeek ::= '1101'B
     sunnyDaysLastWeek3 DaysOfTheWeek ::= '1101000'B

sunnyDaysLastWeek4 DaysOfTheWeek ::= '11010000'B -- Illegal
                                                  -- violates size constraint.
```

or using XML value notation:

```
sunnyDaysLastWeek1 ::=
     <DaysOfTheWeek>
         <sunday/><monday/><wednesday/>
     </DaysOfTheWeek>
```

```
sunnyDaysLastWeek2 ::= <DaysOfTheWeek>1101</DaysOfTheWeek>

sunnyDaysLastWeek3 ::= <DaysOfTheWeek>1101000</DaysOfTheWeek>
```

Note that if the bit string value is less than 7 bits long, then the missing bits indicate a cloudy day for those days, hence the first three values above have the same abstract value.

E.2.5.4 Use a bit string type to model the values of a *bit map*, a fixed-size ordered collection of logical variables indicating whether a particular condition holds for each of a correspondingly ordered collection of objects.

```
DaysOfTheWeek ::= BIT STRING {
            sunday(0), monday (1), tuesday(2),
            wednesday(3), thursday(4), friday(5),
            saturday(6) } (SIZE (7))
sunnyDaysLastWeek1 DaysOfTheWeek ::= {sunday, monday, wednesday}
sunnyDaysLastWeek2 DaysOfTheWeek ::= '1101'B -- Illegal
                                     -- violates size constraint.
sunnyDaysLastWeek3 DaysOfTheWeek ::= '1101000'B

sunnyDaysLastWeek4 DaysOfTheWeek ::= '11010000'B --  Illegal
                                         -- violates size constraint.
```

Note that the first and third values have the same abstract value.

E.2.5.5 Use a bit string type with named bits to model the values of a collection of related logical variables.

EXAMPLE

```
PersonalStatus ::= BIT STRING
            {married(0), employed(1), veteran(2), collegeGraduate(3)}

billClinton PersonalStatus ::= {married, employed, collegeGraduate}

hillaryClinton PersonalStatus ::= '110100'B
```

or using XML value notation:

```
billClinton ::=
      <PersonalStatus>
         <married/>
         <employed/>
         <collegeGraduate/>
      </PersonalStatus>

hillaryClinton ::= <PersonalStatus>110100</PersonalStatus>
```

Note that **billClinton** and **hillaryClinton** have the same abstract values.

E.2.6 Octet string

E.2.6.1 Use an octet string type to model binary data whose format and length are unspecified, or specified elsewhere, and whose length in bits is a multiple of eight.

EXAMPLE

```
G4FacsimileImage ::= OCTET STRING
--  a sequence of octets conforming to
--  Recommendations T.5 and T.6.

image G4FacsimilePage ::= '3FE2EBAD471005'H
```

or using XML value notation:

```
image ::= <G4FacSimileImage>3FE2EBAD471005</G4FacSimileImage>
```

E.2.6.2 Use a restricted character string type in preference to an octet string type, where an appropriate one is available.

EXAMPLE

```
Surname ::= PrintableString

president Surname ::= "Clinton"
```

or using XML value notation:

```
president ::= <Surname>Clinton</Surname>
```

E.2.7 UniversalString, BMPString and UTF8String

Use the **BMPString** type or the **UTF8String** type to model any string of information which consists solely of characters from the ISO/IEC 10646-1 Basic Multilingual Plane (BMP), and **UniversalString** or **UTF8String** to model any string which consists of ISO/IEC 10646-1 characters not confined to the BMP.

E.2.7.1 Use **Level1** or **Level2** to denote that the implementation level places restrictions on the use of combining characters.

EXAMPLE

```
RussianName ::= Cyrillic (Level1)
     -- RussianName uses no combining characters.

SaudiName ::= BasicArabic (SIZE (1..100) ^ Level2)
     --  SaudiName uses a subset of combining characters.
```

Representation of letter Σ:

```
greekCapitalLetterSigma BMPString ::= {0, 0, 3, 163}
```

or using XML value notation:

```
greekCapitalLetterSigma ::= <BMPString>&#x03a3;</BMPString>
```

Representation of string "f $\rightarrow \propto$":

```
rightwardsArrow UTF8String ::= {0, 0, 33, 146}

infinity UTF8String ::= {0, 0, 34, 30}

property UTF8String ::= {"f  ", rightwardsArrow, " ", infinity}
```

or using XML value notation:

```
property ::= <UTF8String>f &#x2192; &#x221E;</UTF8String>
```

E.2.7.2 A collection can be expanded to be a selected subset (i.e., include all characters in the BASIC LATIN collection) by use of the "UnionMark" (see clause 46).

EXAMPLE

```
KatakanaAndBasicLatin ::= UniversalString (FROM (Katakana | BasicLatin))
```

E.2.8 CHARACTER STRING

Use the unrestricted character string type to model any string of information which cannot be modelled using one of the restricted character string types. Be sure to specify the repertoire of characters and their coding into octets.

EXAMPLE

```
PackedBCDString ::= CHARACTER STRING (WITH COMPONENTS {
                                        identification  (WITH COMPONENTS {
                                        fixed  PRESENT })
     /* The abstract and transfer syntaxes shall be
        packedBCDString-AbstractSyntaxId and
        packedBCDString-TransferSyntaxId defined below.
     */
                              } )

     /* object identifier value for a character abstract syntax
        (character set) whose alphabet
        is the digits 0 through 9.
     */
     PackedBCDString-AbstractSyntaxId  OBJECT  IDENTIFIER ::=
          { joint-iso-itu-t asn1(1) examples(123) packedBCD(2) charSet(0) }

     /* object identifier value for a character transfer syntax that
        packs two digits per octet, each digit encoded as 0000 to
        1001, 1111₂ used for padding.
     */
     PackedBCDString-TransferSyntaxId  OBJECT IDENTIFIER ::=
          { joint-iso-itu-t asn1(1) examples(123) packedBCD(2)
               characterTransferSyntax(1) }
```

```
/* The encoding of PackedBCDString will contain only the defined
   encoding of the characters, with any necessary length field, and in
   the case of BER with a field carrying the tag. The object
   identifier values are not carried, as "fixed" has been specified.
*/
```

or using XML value notation:

```
packedBCDString-AbstractSyntaxId ::=
     <OBJECT_IDENTIFIER>
          joint-iso-itu-t.asn1(1).examples(123).packedBCD(2).charSet(0)
     </OBJECT_IDENTIFIER>


packedBCDString-TransferSyntaxId ::=
 <OBJECT_IDENTIFIER>
 joint-iso-itu-t.asn1(1).examples(123).packedBCD(2).characterTransferSyntax(1)
 </OBJECT_IDENTIFIER>
```

or

```
packedBCDString-AbstractSyntaxId ::=
     <OBJECT_IDENTIFIER>2.1.123.2.0</OBJECT_IDENTIFIER>

PackedBCDString-TransferSyntaxId ::=
     <OBJECT_IDENTIFIER>2.1.123.2.1</OBJECT_IDENTIFIER>
```

NOTE – Encoding rules do not necessarily encode values of the type **CHARACTER STRING** in a form that always includes the object identifier values, although they do guarantee that the abstract value is preserved in the encoding.

E.2.9 Null

Use a null type to indicate the effective absence of a component of a sequence.

EXAMPLE

```
PatientIdentifier ::= SEQUENCE {
     name        VisibleString,
     roomNumber        CHOICE {
          room        INTEGER,
          outPatient        NULL  --  if an out-patient  --
     }
}

lastPatient PatientIdentifier ::= {
     name  "Jane Doe",
     roomNumber  outPatient : NULL
}
```

or using XML value notation:

```
lastPatient ::=
     <PatientIdentifier>
        <name>Jane Doe</name>
        <roomNumber><outPatient/></roomNumber>
     </PatientIdentifier>
```

E.2.10 Sequence and sequence-of

E.2.10.1 Use a sequence-of type to model a collection of variables whose types are the same, whose number is large or unpredictable, and whose order is significant.

EXAMPLE

```
NamesOfMemberNations ::= SEQUENCE OF VisibleString
--  in alphabetical order

firstTwo  NamesOfMemberNations ::= {"Australia", "Austria"}
```

or, using the optional identifier:

```
NamesOfMemberNations2 ::= SEQUENCE OF memberNation VisibleString
--  in alphabetical order

firstTwo2  NamesOfMemberNations2 ::=
     {memberNation "Australia", memberNation "Austria"}
```

Using XML value notation, the above two values are as follows:

```
firstTwo ::=
      <NamesOfMemberNations>
          <VisibleString>Australia</VisibleString>
          <VisibleString>Austria</VisibleString>
      </NamesOfMemberNations>

firstTwo2 ::=
      <NamesOfMemberNations2>
          <memberNation>Australia</memberNation>
          <memberNation>Austria</memberNation>
      </NamesOfMemberNations2>
```

E.2.10.2 Use a sequence type to model a collection of variables whose types are the same, whose number is known and modest, and whose order is significant, provided that the make-up of the collection is unlikely to change from one version of the protocol to the next.

EXAMPLE

```
NamesOfOfficers ::= SEQUENCE {
            president                   VisibleString,
            vicePresident               VisibleString,
            secretary                   VisibleString}

acmeCorp NamesOfOfficers ::= {
            president                   "Jane Doe",
            vicePresident               "John Doe",
            secretary                   "Joe Doe"}
```

or using XML value notation:

```
acmeCorp ::=
      <NamesOfOfficers>
          <president>Jane Doe</president>
           <vicePresident>John Doe</vicePresident>
           <secretary>Joe Doe</secretary>
      </NamesOfOfficers>
```

E.2.10.3 Use an inextensible sequence type to model a collection of variables whose types differ, whose number is known and modest, and whose order is significant, provided that the make-up of the collection is unlikely to change from one version of the protocol to the next.

EXAMPLE

```
Credentials ::= SEQUENCE {
            userName                    VisibleString,
            password                    VisibleString,
            accountNumber               INTEGER}
```

E.2.10.4 Use an extensible sequence type to model a collection of variables whose order is significant, whose number currently is known and is modest, but which is expected to be increased:

EXAMPLE

```
Record ::= SEQUENCE {  -- First version of protocol containing "Record"
      userName                    VisibleString,
      password                    VisibleString,
      accountNumber               INTEGER,
      ...,
      ...
}
```

in anticipation of:

```
Record ::= SEQUENCE {  -- Second version of protocol containing "Record"
      userName                    VisibleString,
      password                    VisibleString,
      accountNumber               INTEGER,
      ...,
      [[2:              -- Extension addition added in protocol version 2
         lastLoggedIn             GeneralizedTime OPTIONAL,
         minutesLastLoggedIn      INTEGER
      ]],
      ...
```

```
}
```

and later yet (version 3 of the protocol made no additions to **Record**):

```
Record ::= SEQUENCE {  -- Third version of protocol containing "Record"
     userName                    VisibleString,
     password                    VisibleString,
     accountNumber               INTEGER,
     ...,
     [[2:                -- Extension addition added in protocol version 2
        lastLoggedIn             GeneralizedTime OPTIONAL,
        minutesLastLoggedIn      INTEGER
     ]],
     [[4:                -- Extension addition added in protocol version 3

        certificate              Certificate,
        thumb                    ThumbPrint OPTIONAL
     ]],
     ...
     }
```

E.2.11 Set and set-of

E.2.11.1 Use a set type to model a collection of variables whose number is known and modest and whose order is insignificant. If automatic tagging is not in effect, identify each variable by context-specifically tagging it as shown below. (With automatic tagging, the tags are not needed.)

EXAMPLE

```
UserName ::= SET {
           personalName                [0] VisibleString,
           organizationName            [1] VisibleString,
           countryName                 [2] VisibleString}

user UserName ::= {
           countryName                 "Nigeria",
           personalName                "Jonas Maruba",
           organizationName            "Meteorology, Ltd."}
```

or using XML value notation:

```
user ::=
     <UserName>
        <countryName>Nigeria</countryName>
        <personalName>Jonas Maruba</personalName>
        <organizationName>Meteorology, Ltd.</organizationName>
     </UserName>
```

E.2.11.2 Use a set type with **OPTIONAL** to model a collection of variables that is a (proper or improper) subset of another collection of variables whose number is known and reasonably small and whose order is insignificant. If automatic tagging is not in effect, identify each variable by context-specifically tagging it as shown below. (With automatic tagging, the tags are not needed.)

EXAMPLE

```
UserName ::= SET {
     personalName                     [0] VisibleString,
     organizationName                 [1] VisibleString OPTIONAL
     --  defaults to that of the local organization  -- ,
     countryName                      [2] VisibleString OPTIONAL
     --  defaults to that of the local country  --  }
```

E.2.11.3 Use an extensible set type to model a collection of variables whose make-up is likely to change from one version of the protocol to the next. The following assumes **AUTOMATIC TAGS** was specified in the module definition.

EXAMPLE

```
UserName ::= SET {
     personalName          VisibleString,-- First version of "UserName"
     organizationName      VisibleString OPTIONAL ,
     countryName           VisibleString OPTIONAL,
     ...,
     ...
}
```

```
user UserName ::= { personalName  "Jonas Maruba" }
```

or using XML value notation:

```
user ::=
     <UserName>
        <personalName>Jonas Maruba</personalName>
     </UserName>
```

in anticipation of:

```
UserName ::= SET {           -- Second version of "UserName"
     personalName                 VisibleString,
     organizationName             VisibleString OPTIONAL,
     countryName                  VisibleString OPTIONAL,
     ...,
     [[2:                    -- Extension addition added in protocol version 2
        internetEmailAddress      VisbleString,
        faxNumber                 VisibleString OPTIONAL
     ]],
     ...
}

user  UserName ::= {
     personalName                 "Jonas Maruba",
     internetEmailAddress         "jonas@meteor.ngo.com"
}
```

or using XML value notation:

```
 user ::=
<UserName>
   <personalName>Jonas Maruba</personalName>
   <internetEmailAddress>jonas@meteor.ngo.com</internetEmailAddress>
</UserName>
```

and later yet (versions 3 and 4 of the protocol made no additions to **UserName**):

```
UserName ::= SET {     -- Fifth version of protocol containing "UserName"
     personalName                 VisibleString,
     organizationName             VisibleString OPTIONAL,
     countryName                  VisibleString OPTIONAL,
     ...,
     [[2:                         -- Extension addition added in version 2
        internetEmailAddress      VisbleString,
        faxNumber                 VisibleString OPTIONAL
     ]],
     [[5:                         -- Extension addition added in version 5
     phoneNumber                  VisibleString OPTIONAL
     ]],
     ...
}

user  UserName ::= {
     personalName                 "Jonas Maruba",
     internetEmailAddress         "jonas@meteor.ngo.com"
}
```

or using XML value notation:

```
user ::=
     <UserName>
        <personalName>Jonas Maruba</personalName>
        <internetEmailAddress>jonas@meteor.ngo.com</internetEmailAddress>
     </UserName>
```

E.2.11.4 Use a set-of type to model a collection of variables whose types are the same and whose order is insignificant.

EXAMPLE

```
Keywords ::= SET OF VisibleString  --  in arbitrary order

someASN1Keywords Keywords ::= {"INTEGER", "BOOLEAN", "REAL"}
```

or, using the optional identifier:

```
Keywords2 ::= SET OF keyword VisibleString  --  in arbitrary order

someASN1Keywords2 Keywords2 ::= {keyword "INTEGER", keyword "BOOLEAN",
     keyword "REAL"}
```

Using XML value notation, the above two values are as follows:

```
someASN1Keywords ::=
     <Keywords>
        <VisibleString>INTEGER</VisibleString>
        <VisibleString>BOOLEAN</VisibleString>
        <VisibleString>REAL</VisibleString>
     </Keywords>

someASN1Keywords2 ::=
     <Keywords2>
        <keyword>INTEGER</keyword>
        <keyword>BOOLEAN</keyword>
        <keyword>REAL</keyword>
     </Keywords2>
```

E.2.12 Tagged

Prior to the introduction of the **AUTOMATIC TAGS** construct, ASN.1 specifications frequently contained tags. The following subclauses describe the way in which tagging was typically applied. With the introduction of **AUTOMATIC TAGS**, new ASN.1 specifications need make no use of the tag notation, although those modifying old notation may have to concern themselves with tags. New users of the ASN.1 notation are encouraged to use **AUTOMATIC TAGS** as this makes the notation more readable.

E.2.12.1 Universal class tags are used only within this Recommendation | International Standard. The notation **[UNIVERSAL 30]** (for example) is provided solely to enable precision in the definition of the "UsefulTypes" (see 41.1). It should not be used elsewhere.

E.2.12.2 A frequently encountered style for the use of tags is to assign an application class tag precisely once in the entire specification, using it to identify a type that finds wide, scattered, use within the specification. An application class tag is also frequently used (once only) to tag the types in the outermost **CHOICE** of an application, providing identification of individual messages by the application class tag. The following is an example use in the former case:

EXAMPLE

```
FileName ::= [APPLICATION 8] SEQUENCE {
           directoryName               VisibleString,
           directoryRelativeFileName   VisibleString}
```

E.2.12.3 Context-specific tagging is frequently applied in an algorithmic manner to all components of a **SET**, **SEQUENCE**, or **CHOICE**. Note, however, that the **AUTOMATIC TAGS** facility does this easily for you.

EXAMPLE

```
CustomerRecord ::= SET {
           name                        [0] VisibleString,
           mailingAddress              [1] VisibleString,
           accountNumber               [2] INTEGER,
           balanceDue                  [3] INTEGER  --  in cents  --}

CustomerAttribute ::= CHOICE {
           name                        [0] VisibleString,
           mailingAddress              [1] VisibleString,
           accountNumber               [2] INTEGER,
           balanceDue                  [3] INTEGER  --  in cents  --}
```

E.2.12.4 Private class tagging should normally not be used in internationally standardized specifications (although this cannot be prohibited). Applications produced by an enterprise will normally use application and context-specific tag classes. There may be occasional cases, however, where an enterprise-specific specification seeks to extend an internationally standardized specification, and in this case use of private class tags may give some benefits in partially protecting the enterprise-specific specification from changes to the internationally standardized specification.

EXAMPLE

```
AcmeBadgeNumber ::= [PRIVATE 2] INTEGER

badgeNumber AcmeBadgeNumber ::= 2345
```

or using XML value notation:

```
badgeNumber ::= <AcmeBadgeNumber>2345</AcmeBadgeNumber>
```

E.2.12.5 Textual use of **IMPLICIT** with every tag is generally found only in older specifications. BER produces a less compact representation when explicit tagging is used than when implicit tagging is used. PER produces the same compact encoding in both cases. With BER and explicit tagging, there is more visibility of the underlying type (**INTEGER**, **REAL**, **BOOLEAN**, etc.) in the encoded data. These guidelines use implicit tagging in the examples whenever it is legal to do so. This may, depending on the encoding rules, result in a compact representation, which is highly desirable in some applications. In other applications, compactness may be less important than, for example, the ability to carry out strong type-checking. In the latter case, explicit tagging can be used.

EXAMPLE

```
CustomerRecord ::= SET {
            name                        [0] IMPLICIT VisibleString,
            mailingAddress              [1] IMPLICIT VisibleString,
            accountNumber               [2] IMPLICIT INTEGER,
            balanceDue                  [3] IMPLICIT INTEGER  --  in cents  --
}

CustomerAttribute ::= CHOICE {
            name                        [0] IMPLICIT VisibleString,
            mailingAddress              [1] IMPLICIT VisibleString,
            accountNumber               [2] IMPLICIT INTEGER,
            balanceDue                  [3] IMPLICIT INTEGER  --  in cents  --
}
```

E.2.12.6 Guidance on use of tags in new ASN.1 specifications referencing this Recommendation | International Standard is quite simple: DON'T USE TAGS. Put **AUTOMATIC TAGS** in the module header, then forget about tags. If you need to add new components to the **SET**, **SEQUENCE** or **CHOICE** in a later version, add them to the end.

E.2.13 Choice

E.2.13.1 Use a **CHOICE** to model a variable that is selected from a collection of variables whose number are known and modest.

EXAMPLE

```
FileIdentifier ::= CHOICE {
            relativeName     VisibleString,
          --  name of file (for example, "MarchProgressReport")
            absoluteName     VisibleString,
          --  name of file and containing directory
          --  (for example, "<Williams>MarchProgressReport")
            serialNumber     INTEGER
          --  system-assigned identifier for file  --}

file FileIdentifier ::= serialNumber : 106448503
```

or using XML value notation:

```
fileIdentifier ::=
          <FileIdentifier>
             <serialNumber>106448503</serialNumber>
          </FileIdentifier>
```

E.2.13.2 Use an extensible **CHOICE** to model a variable that is selected from a collection of variables whose make-up is likely to change from one version of the protocol to the next.

EXAMPLE

```
FileIdentifier ::= CHOICE {             -- First version of FileIdentifier
            relativeName    VisibleString,

            absoluteName    VisibleString,

            ...,   ...
}
```

```
fileId1  FileIdentifier ::= relativeName : "MarchProgressReport.doc"
```

or using XML value notation:

```
fileId1 ::=
      <FileIdentifier>
         <relativeName>MarchProgressReport.doc</relativeName>
      </FileIdentifier>
```

in anticipation of:

```
FileIdentifier ::= CHOICE {       -- Second version of FileIdentifier
      relativeName     VisibleString,
      absoluteName     VisibleString,
      ...,
      serialNumber     INTEGER,    -- Extension addition added in version 2
      ...
}

fileId1  FileIdentifier ::= relativeName : "MarchProgressReport.doc"

fileId2  FileIdentifier ::= serialNumber : 214
```

or using XML value notation:

```
fileId1 ::=
      <FileIdentifier>
         <relativeName>MarchProgressReport.doc</relativeName>
      </FileIdentifier>

fileId2 ::=
      <FileIdentifier>
         <serialNumber>214</serialNumber>
      </FileIdentifier>
```

and later yet:

```
FileIdentifier ::= CHOICE {       -- Third version of FileIdentifier
      relativeName     VisibleString,
      absoluteName     VisibleString,
      ...,
      serialNumber     INTEGER,    -- Extension addition added in version 2
      [[                           -- Extension addition added in version 3
         vendorSpecificVendorExt,
         unidentified  NULL
      ]],
      ...
}

fileId1  FileIdentifier ::= relativeName : "MarchProgressReport.doc"

fileId2  FileIdentifier ::= serialNumber : 214

fileId3  FileIdentifier ::= unidentified : NULL
```

or using XML value notation:

```
fileId1 ::=
      <FileIdentifier>
         <relativeName>MarchProgressReport.doc</relativeName>
      </FileIdentifier>

fileId2 ::=
      <FileIdentifier>
         <serialNumber>214</serialNumber>
      </FileIdentifier>

fileId3 ::=
      <FileIdentifier>
         <unidentified/>
      </FileIdentifier>
```

E.2.13.3 Use an extensible **CHOICE** of only one type where the possibility is envisaged of more than one type being permitted in the future.

EXAMPLE

```
Greeting ::= CHOICE {                -- First version of "Greeting"
      postCard         VisibleString,
      ...,
      ...
}
```

in anticipation of:

```
Greeting ::= CHOICE {                -- Second version of "Greeting"
      postCard    VisibleString,
      ...,
      [[2:                           -- Extension addition added in version 2
         audio          Audio,
         video          Video
      ]],
      ...
}
```

E.2.13.4 Multiple colons are required when a choice value is nested within another choice value.

EXAMPLE

```
Greeting ::= [APPLICATION 12] CHOICE {
            postCard         VisibleString,
            recording        Voice }

Voice ::= CHOICE {
            english          OCTET STRING,
            swahili          OCTET STRING }

myGreeting Greeting ::= recording : english : '019838547E0'H
```

or using XML value notation:

```
myGreeting ::=
      <Greeting>
         <recording><english>019838547E0</english></recording>
      </Greeting>
```

E.2.14 Selection type

E.2.14.1 Use a selection type to model a variable whose type is that of some particular alternatives of a previously defined **CHOICE**.

E.2.14.2 Consider the definition:

```
FileAttribute ::= CHOICE {
            date-last-used   INTEGER,
            file-name        VisibleString}
```

then the following definition is possible:

```
AttributeList ::= SEQUENCE {
            first-attribute  date-last-used  < FileAttribute,
            second-attribute file-name < FileAttribute }
```

with a possible value notation of:

```
listOfAttributes AttributeList ::=  {
            first-attribute  27,
            second-attribute "PROGRAM" }
```

or using XML value notation:

```
listOfAttributes ::=
      <AttributeList>
         <first-attribute>27</first-attribute>
         <second-attribute>PROGRAM</second-attribute>
      </AttributeList>
```

E.2.15 Object class field type

E.2.15.1 Use an object class field type to identify a type defined by means of an information object class (see ITU-T Rec. X.681 | ISO/IEC 8824-2). For example, fields of the information object class **ATTRIBUTE** may be used in defining a type, **Attribute**.

EXAMPLE

```
ATTRIBUTE ::= CLASS {
      &AttributeType,
      &attributeId          OBJECT IDENTIFIER UNIQUE
            }

Attribute ::= SEQUENCE {
      attributeID       ATTRIBUTE.&attributeId,--  this is normally constrained.
      attributeValue    ATTRIBUTE.&AttributeType--  this is normally constrained.
            }
```

Both **ATTRIBUTE.&attributeId** and **ATTRIBUTE.&AttributeType** are object class field types, in that they are types defined by reference to an information object class (**ATTRIBUTE**). The type **ATTRIBUTE.&attributeId** is fixed because it is explicitly defined in **ATTRIBUTE** as an **OBJECT IDENTIFIER**. However, the type **ATTRIBUTE.&AttributeType** can carry a value of any type defined using ASN.1, since its type is not fixed in the definition of the information object class **ATTRIBUTE**. Notations that possess this property of being able to carry a value of any type are termed "open type notation", hence **ATTRIBUTE.&AttributeType** is an open type.

E.2.16 Embedded-pdv

E.2.16.1 Use an embedded-pdv type to model a variable whose type is unspecified, or specified elsewhere with no restriction on the notation used to specify the type.

EXAMPLE

```
FileContents ::= EMBEDDED PDV

DocumentList ::= SEQUENCE OF document EMBEDDED PDV
```

E.2.17 External

The external type is similar to the embedded-pdv type, but has fewer identification options. New specifications will generally prefer to use embedded-pdv because of its greater flexibility and the fact that some encoding rules encode its values more efficiently.

E.2.18 Instance-of

E.2.18.1 Use an instance-of to specify a type containing an object identifier field and an open type value whose type is determined by the object identifier. The instance-of type can only be used if the association between the object identifier value and the type is specified using an information object of a class derived from **TYPE-IDENTIFIER** (see ITU-T Rec. X.681 | ISO/IEC 8824-2, Annex A and Annex C).

EXAMPLE

```
ACCESS-CONTROL-CLASS ::= TYPE-IDENTIFIER

Get-Invoke ::= SEQUENCE {
      objectClass       ObjectClass,
      objectInstance    ObjectInstance,
      accessControl     INSTANCE OF ACCESS-CONTROL-CLASS,--  this is normally
                                                         --  constrained.
      attributeID       ATTRIBUTE.&attributeId
      }
```

Get-Invoke is then equivalent to:

```
Get-Invoke ::= SEQUENCE {
      objectClass       ObjectClass,
      objectInstance    ObjectInstance,
      accessControl     [UNIVERSAL 8] IMPLICIT SEQUENCE {
            type-id     ACCESS-CONTROL-CLASS.&id,--  this is normally
                                                 --  constrained.
            value       [0] ACCESS-CONTROL-CLASS.&Type--  this is normally
                                                      --  constrained.
      },
      attributeID ATTRIBUTE.&attributeId
}
```

The true utility of the instance-of type is not seen until it is constrained using an information object set, but such an example goes beyond the scope of this Recommendation | International Standard. See ITU-T Rec. X.682 | ISO/IEC 8824-3 for the definition of information object set, and Annex A of that Recommendation | International Standard for how to use an information object set to constrain an instance-of type.

E.2.19 Relative Object Identifier

E.2.19.1 Use a relative object identifier type to transmit object identifier values in a more compact form in contexts where the early part of the object identifier value is known. There are three situations that can arise:

a) The early part of the object identifier value is fixed for a given specification (it is an industry-specific standard, and all OIDs are relative to an OID allocated to the standardizing body. In this case, use:

```
RELATIVE-OID      --  The relative object identifier value is
                  --  relative to {iso identified-organization  set(22)}
```

b) The early part of the object identifier value is frequently a value that is known at specification time, but may occasionally be a more general value. In this case, use:

```
CHOICE
      {a    RELATIVE-OID    --  The value is relative to {1  3  22}--,
       b    OBJECT IDENTIFIER  -- Any object identifier value--}
```

c) The early part of the object identifier value is not known until communications time, but will frequently be common to many values that need to be sent, and quite often will be a value known at specification time. In this case use (for example):

```
SEQUENCE
      {oid-root   OBJECT IDENTIFIER DEFAULT {1  3  22},
       reloids    SEQUENCE OF RELATIVE-OID --relative to oid-root--}
```

E.3 Identifying abstract syntaxes

E.3.1 It is common for protocols to be defined by associating semantics with each of the values of a single ASN.1 type, typically a choice type. (This ASN.1 type is sometimes referred to informally as "the top-level type for the application".) This set of abstract values is formally called the abstract syntax for the application. An abstract syntax can be identified by giving it an abstract syntax name of ASN.1 type object identifier.

E.3.2 The assignment of an object identifier to an abstract syntax can be done using the built-in information object class **ABSTRACT-SYNTAX** which is defined in ITU-T Rec. X.681 | ISO/IEC 8824-2. This also serves to clearly identify the top-level type for the application.

E.3.3 The following is an example of text which might appear in an application specification:

EXAMPLE

```
Application-ASN1 DEFINITIONS ::=
      BEGIN
      EXPORTS Application-PDU;

      Application-PDU ::= CHOICE {
                  connect-pdu    ..... ,
                  data-pdu    CHOICE {
                        ..... ,
                        .....
                  },
                  .....
            }
            ......
      END

Abstract-Syntax-Module DEFINITIONS ::=
      BEGIN
      IMPORTS Application-PDU FROM Application-ASN1;

--  This application defines the following abstract syntax:

      Abstract-Syntax ABSTRACT-SYNTAX ::=
                  { Application-PDU  IDENTIFIED BY
                          application-abstract-syntax-object-id }

      application-abstract-syntax-object-id OBJECT IDENTIFIER ::=
                  {joint-iso-itu-t asn1(1) examples(123)
                        application-abstract-syntax(3) }
--  The corresponding object descriptor is:

      application-abstract-syntax-descriptor ObjectDescriptor ::=
                        "Example Application Abstract Syntax"
```

```
--  The ASN.1 object identifier and object descriptor values:
                --  encoding rule object identifier
                --  encoding rule object descriptor
--  assigned to encoding rules in ITU-T Rec. X.690 | ISO/IEC 8825-1
-- and ITU-T Rec. X.691 | ISO/IEC 8825-2 can be used as the transfer
-- syntax identifier in conjunction with this transfer syntax.

END
```

E.3.4 In order to ensure interworking, the standard may additionally identify a mandatory transfer syntax (typically one of those defined in the encoding rules of ITU-T Rec. X.690 | ISO/IEC 8825-1 or ITU-T Rec. X.691 | ISO/IEC 8825-2 or ITU-T Rec. X.692 | ISO/IEC 8825-3).

E.4 Subtypes

E.4.1 Use subtypes to limit the values of an existing type which are to be permitted in a particular situation.

EXAMPLES

```
AtomicNumber    ::=   INTEGER (1..104)

TouchToneString ::=  IA5String
                      (FROM ("0123456789" | "*" | "#")) (SIZE (1..63))

ParameterList   ::=  SET SIZE (1..63) OF Parameter

SmallPrime   ::=  INTEGER (2|3|5|7|11|13|17|19|23|29)
```

E.4.2 Use an extensible subtype constraint to model an **INTEGER** type whose set of permitted values is small and well defined, but which is expected to increase.

EXAMPLE

```
SmallPrime ::= INTEGER (2 | 3, ...)     -- First version of SmallPrime
```

in anticipation of:

```
SmallPrime ::= INTEGER (2 | 3, ..., 5 | 7 | 11)
-- Second version of SmallPrime
```

and later yet:

```
SmallPrime ::= INTEGER (2 | 3, ..., 5 | 7 | 11 | 13 | 17 | 19)
-- Third version of SmallPrime
```

NOTE – For certain types, some encoding rules (e.g. PER) provide a highly optimized encoding for subtype constraint extension root values (i.e. values appearing before the "**...**") and a less optimized encoding for subtype constraint extension addition values (i.e., values appearing after the "**...**"), while in some other encoding rules (e.g. BER) subtype constraints have no effect on the encoding.

E.4.3 Where two or more related types have significant commonality, consider explicitly defining their common parent as a type and use subtyping for the individual types. This approach makes clear the relationship and the commonality, and encourages (though does not force) this to continue as the types evolve. It thus facilitates the use of common implementation approaches to the handling of values of these types.

EXAMPLE

```
Envelope    ::=  SET {
                        typeA  TypeA,
                        typeB  TypeB   OPTIONAL,
                        typeC  TypeC   OPTIONAL}
             --  the common parent

ABEnvelope  ::=  Envelope (WITH  COMPONENTS
                        {... ,
                              typeB   PRESENT, typeC   ABSENT})
             --  where typeB must always appear and typeC must not

ACEnvelope  ::=  Envelope (WITH  COMPONENTS
                        {... ,
                              typeB   ABSENT, typeC   PRESENT})
             --  where typeC must always appear and typeB must not
```

The latter definitions could alternatively be expressed as:

```
ABEnvelope  ::= Envelope (WITH COMPONENTS {typeA, typeB})

ACEnvelope  ::= Envelope (WITH COMPONENTS {typeA, typeC})
```

The choice between the alternatives would be made upon such factors as the number of components in the parent type, and the number of those which are optional, the extent of the difference between the individual types, and the likely evolution strategy.

E.4.4 Use subtyping to partially define a value, for example, a protocol data unit to be tested for in a conformance test, where the test is concerned only with some components of the PDU.

EXAMPLE

Given:

```
PDU  ::=  SET
        {alpha          INTEGER,
         beta           IA5String  OPTIONAL,
         gamma          SEQUENCE OF Parameter,
         delta          BOOLEAN}
```

then in composing a test which requires the Boolean to be false and the integer to be negative, write:

```
TestPDU   ::=  PDU (WITH COMPONENTS
           {... ,
            delta (FALSE),
            alpha (MIN..<0)})
```

and if, further, the **IA5String**, **beta**, is to be present and either 5 or 12 characters in length, write:

```
FurtherTestPDU ::= TestPDU (WITH COMPONENTS {... , beta (SIZE (5|12)) PRESENT
} )
```

E.4.5 If a general-purpose data type has been defined as a **SEQUENCE OF**, use subtyping to define a restricted subtype of the general type.

EXAMPLE

```
Text-block ::=  SEQUENCE OF VisibleString

Address   ::=  Text-block (SIZE (1..6)) (WITH COMPONENT (SIZE (1..32)))
```

E.4.6 If a general-purpose data type had been defined as a **CHOICE**, use subtyping to define a restricted subtype of the general type.

EXAMPLE

```
Z ::= CHOICE {
           a          A,
           b          B,
           c          C,
           d          D,
           e          E
     }

V ::= Z (WITH COMPONENTS { ..., a  ABSENT,  b  ABSENT })--  'a' and 'b' must
                              -- be absent, either 'c',
                              --  'd' or 'e' may be present in a value.

W ::= Z (WITH COMPONENTS { ..., a  PRESENT })--  only 'a' can be present
                                    -- (see 47.8.9.2).

X ::= Z (WITH COMPONENTS { a  PRESENT })--  only 'a' can be present
                                    -- (see 47.8.9.2).

Y ::= Z (WITH COMPONENTS { a  ABSENT,  b,  c })--  'a', 'd' and 'e' must be
                                    -- absent, either 'b' or 'c' may
                                    -- be present in a value.
```

NOTE – **W** and **X** are semantically identical.

E.4.7 Use contained subtypes to form new subtypes from existing subtypes.

EXAMPLE

```
Months      ::= ENUMERATED {
                        january          (1),
```

```
                    february        (2),
                    march           (3),
                    april           (4),
                    may             (5),
                    june            (6),
                    july            (7),
                    august          (8),
                    september       (9),
                    october         (10),
                    november        (11),
                    december        (12) }

First-quarter ::= Months ( january | february | march )

Second-quarter ::= Months ( april | May | june )

Third-quarter ::= Months ( July | August | september )

Fourth-quarter ::= Months ( October | November | december )

First-half ::= Months ( First-quarter | Second-quarter )

Second-half ::= Months ( Third-quarter | Fourth-quarter )
```

Annex F

Tutorial annex on ASN.1 character strings

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

F.1 Character string support in ASN.1

F.1.1 There are four groups of character string support in ASN.1. The four groups are:

a) Character string types based on *ISO International Register of Coded Character Sets to be used with Escape Sequences* (that is, based on the structure of ISO/IEC 646) and the associated International Register of Coded Character Sets, and provided by the types **VisibleString**, **IA5String**, **TeletexString**, **VideotexString**, **GraphicString**, and **GeneralString**.

b) Character string types based on ISO/IEC 10646-1, and provided by subsetting the type **UniversalString**, **UTF8String** or **BMPString** with subsets defined in ISO/IEC 10646-1 or by using named characters.

c) Character string types providing a simple small collection of characters specified in this Recommendation | International Standard, and intended for specialized use; these are the **NumericString** and **PrintableString** types.

d) Use of the type **CHARACTER STRING**, with negotiation of the character set to be used (or announcement of the set being used); this permits an implementation to use any collection of characters and encodings for which **OBJECT IDENTIFIER**s have been assigned, including those of *ISO International Register of Coded Character Sets to be used with Escape Sequences*, ISO/IEC 7350, ISO/IEC 10646-1, and private collections of characters and encodings (profiles may impose requirements or restrictions on the character sets – the character abstract syntaxes – to be used).

F.2 The UniversalString, UTF8String and BMPString types

F.2.1 The **UniversalString** and **UTF8String** types carry any character from ISO/IEC 10646-1. The set of characters in ISO/IEC 10646-1 is generally too large for meaningful conformance to be required, and should normally be subsetted to a combination of the standard collections of characters in Annex A of ISO/IEC 10646-1.

F.2.2 The **BMPString** type carries any character from the Basic Multilingual Plan of ISO/IEC 10646-1. The Basic Multilingual Plane is normally subsetted to a combination of the standard collections of characters in Annex A of ISO/IEC 10646-1.

F.2.3 For the collections defined in Annex A of ISO/IEC 10646-1, there are type references defined in the built-in ASN.1 module **ASN1-CHARACTER-MODULE** (see clause 38). The "subtype constraint" mechanism allows new subtypes of **UniversalString** that are combinations of existing subtypes to be defined.

F.2.4 Examples of type references defined in **ASN1-CHARACTER-MODULE** and their corresponding ISO/IEC 10646-1 collection names are:

BasicLatin	BASIC LATIN
Latin-1Supplement	LATIN-1 SUPPLEMENT
LatinExtended-a	LATIN EXTENDED-A
LatinExtended-b	LATIN EXTENDED-B
IpaExtensions	IPA EXTENSIONS
SpacingModifierLetters	SPACING MODIFIER LETTERS
CombiningDiacriticalMarks	COMBINING DIACRITICAL MARKS

F.2.5 ISO/IEC 10646-1 specifies three "levels of implementation", and requires that all uses of ISO/IEC 10646-1 specify the implementation level.

The implementation level relates to the extent to which support is given for *combining characters* in the character repertoire, and hence, in ASN.1 terms, defines a subset of the **UniversalString** and **BMPString** restricted character string types.

In implementation level 1, combining characters are not allowed, and there is normally a one-to-one correspondence between abstract characters in ASN.1 character strings and printed characters in a physical rendition of the string.

In implementation level 2, certain combining characters (listed in ISO/IEC 10646-1, Annex B) are available for use, but there are others whose use is prohibited.

In implementation level 3, there are no restrictions on the use of combining characters.

F.2.6 A **BMPString** or **UniversalString** can be restricted to exclude all control functions by use of the subtype notation as follows:

```
VanillaBMPString ::= BMPString (FROM (ALL EXCEPT ({0,0,0,0}..{0,0,0,31} |
                                        {0,0,0,128}..{0,0,0,159})))
```

or equivalently:

```
C0 ::= BMPString (FROM ({0,0,0,0} .. {0,0,0,31}))   --C0 control functions
C1 ::= BMPString (FROM ({0,0,0,128} .. {0,0,0,159}))--C1 control functions
VanillaBMPString ::= BMPString (FROM (ALL EXCEPT (C0 | C1)))
```

F.3 On ISO/IEC 10646-1 conformance requirements

Use of **UniversalString**, **BMPString** or **UTF8String** (or subtypes of these) in an ASN.1 type definition requires that the conformance requirements of ISO/IEC 10646-1 be addressed.

These conformance requirements demand that implementors of a standard (X say) using such ASN.1 types provide (in the Protocol Implementation Conformance Statement) a statement of the adopted subset of ISO/IEC 10646-1 for their implementation of standard X, and of the level (support for combining characters) of the implementation.

The use of an ASN.1 subtype of **UniversalString**, **UTF8String** or **BMPString** in a specification requires that an implementation support all the ISO/IEC 10646-1 characters that are included in that ASN.1 subtype, and hence that (at least) those characters be present in the adopted subset for the implementation. It is also a requirement that the stated level be supported for all such ASN.1 subtypes.

NOTE – An ASN.1 specification (in the absence of parameters of the abstract syntax and exception specifications) determines both the (maximum) set of characters that can be transmitted and the (minimum) set of characters that have to be handled on receipt. The adopted set of ISO/IEC 10646-1 requires that characters beyond this set not be transmitted, and that all characters within this set be supported on receipt. The adopted set therefore needs to be precisely the set of all characters permitted by the ASN.1 specification. The case where a parameter of the abstract syntax is present is discussed below.

F.4 Recommendations for ASN.1 users on ISO/IEC 10646-1 conformance

Users of ASN.1 should make clear the set of ISO/IEC 10646-1 characters that will form the adopted subset of implementations (and the required implementation level) if the requirements of their standard are to be met.

This can conveniently be done by defining an ASN.1 subtype of **UniversalString**, **UTF8String** or **BMPString** that contains all the characters needed for the standard, and by restricting it to **Level1** or **Level2** if appropriate. A convenient name for this type might be **ISO-10646-String**.

EXAMPLE

```
ISO-10646-String ::= BMPString
     (FROM (Level2 INTERSECTION (BasicLatin UNION HebrewExtended UNION Hiragana)))
     --  This is the type that defines the minimum set of characters in
     --  the adopted subset for an implementation of this standard. The
     --  implementation level is required to be at least level 2.
```

In an OSI environment, the OSI Protocol Implementation Conformance Statement would then contain a simple statement that the adopted subset of ISO/IEC 10646-1 is the limited subset (and the level) defined by **ISO-10646-String**, and **ISO-10646-String** (possibly subtyped) would be used throughout the standard where ISO/IEC 10646-1 strings were to be included.

EXAMPLE CONFORMANCE STATEMENT

The adopted subset of ISO/IEC 10646-1 is the limited subset consisting of all the characters in the ASN.1 type **ISO-10646-String** defined in module <your module name goes here>, with an implementation level of 2.

EXAMPLE USE IN PROTOCOL

```
Message ::=  SEQUENCE {
     first-field ISO-10646-String,--  all characters in the adopted
                                  -- subset can appear
     second-field     ISO-10646-String
          (FROM (latinSmallLetterA .. latinSmallLetterZ)),--  lower case
                                                       -- latin letters only
```

```
        third-field ISO-10646-String
              (FROM (digitZero .. digitNine))--  digits only

    }
```

F.5 Adopted subsets as parameters of the abstract syntax

ISO/IEC 10646-1 requires that the adopted subset and level of an implementation be *explicitly* defined. Where an ASN.1 user does not wish to constrain the range of ISO/IEC 10646-1 characters in some part of the standard being defined, this can be expressed by defining **ISO-10646-String** (for example) as a subtype of **UniversalString**, **BMPString** or **UTF8String** with a subtype constraint consisting of (or including) **ImplementorsSubset** which is left as a parameter of the abstract syntax.

Users of ASN.1 are warned that in this case a conforming sender may transmit to a conforming receiver characters that cannot be handled by the receiver because they fall outside the (implementation-dependent) adopted subset or level of the receiver, and it is recommended that an exception-handling specification be included in the definition of **ISO-10646-String** in this case.

EXAMPLE

```
ISO-10646-String {UniversalString : ImplementorsSubset, ImplementationLevel} ::=
           UniversalString (FROM((ImplementorsSubset UNION BasicLatin)
                      INTERSECTION ImplementationLevel) !characterSetProblem)
-- The adopted subset of ISO/IEC 10646-1 shall include "BasicLatin", but
-- may also include any additional characters specified in
-- "ImplementorsSubset", which is a parameter of the abstract syntax.
-- "ImplementationLevel", which is a parameter of the abstract
-- syntax defines the implementation level. A conforming receiver must be
-- prepared to recieve characters outside of its adopted subset and
-- implementation level.  In this case the exception handling specified in
-- clause <add your clause number here> for "characterSetProblem" is
-- invoked. Note that this can never be invoked by a conforming
-- receiver if the actual characters used in an instance of communication
-- are restricted to "BasicLatin".

My-Level2-String ::= ISO-10646-String { { HebrewExtended UNION Hiragana }, Level2 }
```

F.6 The CHARACTER STRING type

F.6.1 The **CHARACTER STRING** type gives complete flexibility in the choice of character set and encoding method.

NOTE – Where a single connection provides end-to-end data transfer (no relaying), and the OSI protocols are in use, then negotiation of the character sets to be used and their encoding can be accomplished as part of the definition of the OSI presentation contexts for character abstract syntaxes. Otherwise, the abstract and transfer character syntaxes (character repertoire and encodings) are announced by a pair of object identifier values.

F.6.2 In formal terms, a character abstract syntax is an ordinary abstract syntax with some restrictions on the possible values (they are all character strings, and indeed are all the character strings formed from some collection of characters). Thus allocation of object identifier values for character abstract and transfer syntaxes is performed in the normal way.

F.6.3 The encoding of **CHARACTER STRING** announces the abstract and transfer syntax of the character repertoire in use (that is, character set and encoding). In OSI environments, negotiation of both these syntaxes is possible.

F.6.4 Character abstract syntaxes (and corresponding character transfer syntaxes) have been defined in a number of ITU-T Recommendations and International Standards, and additional character abstract syntaxes (and/or character transfer syntaxes) can be defined by any organization able to allocate object identifiers.

F.6.5 In ISO/IEC 10646-1, there is a character abstract syntax defined (and object identifiers assigned) for the entire collection of characters, for each of the defined collection of characters for subsets (BASIC LATIN, BASIC SYMBOLS, etc.), and for every possible combination of the defined collections of characters. There are also two character transfer syntaxes defined to identify the various options (particularly 16-bit and 32-bit) in ISO/IEC 10646-1.

Annex G

Tutorial annex on the ASN.1 model of type extension

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

G.1 Overview

G.1.1 It can happen that an ASN.1 type evolves over time from an **extension root** type by means of a series of extensions called **extension additions**.

G.1.2 An ASN.1 type available to a particular implementation may be the extension root type, or may be the extension root type plus one or more extension additions. Each such ASN.1 type that contains an extension addition also contains all previously defined extension additions.

G.1.3 The ASN.1 type definitions in this series are said to be **extension-related** (see 3.6.32 for a more precise definition of "extension-related"), and encoding rules are required to encode extension-related types in a such a way that if two systems are using two different types which are extension-related, transmissions between the two systems will successfully transfer the information content of those parts of the extension-related types that are common to the two systems. It is also required that those parts that are not common to both systems can be delimited and retransmitted (perhaps to a third party) on a subsequent transmission, provided the same transfer syntax is used.

NOTE – The sender may be using a type that is either earlier or later in the series of extension additions.

G.1.4 The series of types obtained by progressively adding to a root type is called an **extension series**. In order for encoding rules to make appropriate provision for transmissions of extension-related types (which may require more bits on the line), such types (including the extension root type) need to be syntactically flagged. The flag is an ellipsis (**...**), and is called an **extension marker**.

EXAMPLE

```
Extension root type    1st extension       2nd extension        3rd extension

A ::= SEQUENCE {      A ::= SEQUENCE {    A ::= SEQUENCE {     A ::= SEQUENCE {
      a INTEGER,            a INTEGER,          a INTEGER,           a INTEGER,
        ...                   ...,                ...,                 ...,
      }                       b BOOLEAN,          b BOOLEAN,           b BOOLEAN,
                              c INTEGER           c INTEGER,           c INTEGER,
                            }                     d SEQUENCE {         d SEQUENCE {
                                                    e INTEGER,           e INTEGER,
                                                      ...,                 ...,
                                                      ...,                 g BOOLEAN OPTIONAL,
                                                    f IA5String            h BMPString,
                                                  }                        ...,
                                                }                          f IA5String
                                                                         }
                                                                       }
```

G.1.5 All extension additions in sequence, set, and choice types are inserted between pairs of extension markers. A single extension marker is allowed if (in the extension root type) it appears as the last item in the type, in which case a matching extension marker is assumed to exist just before the closing brace of the type; in such cases all extension additions are inserted at the end of the type.

G.1.6 A type that has an extension marker can be nested inside a type that has none, or it can be nested within a type in an extension root, or it can be nested in an extension addition type. In such cases the extension series are treated independently, and the nested type with the extension marker has no impact on the type within which it is nested. Only one **extension insertion point** (the end of the type if a single extension marker is used, or just before the second extension marker if a pair of extension markers is used) can appear in any specific construct.

G.1.7 A new extension addition in the extension series is defined in terms of a single **extension addition group** (one or more types nested within "**[[**" "**]]**") or a single type added at the extension insertion point. In the following example the first extension defines an extension addition group where **b** and **c** must either be both present or both absent in a value of type **A**. The second extension defines a single component type, **d**, which may be absent in a value of type **A**. The third extension defines an extension addition group in which **h** must be present in a value of type **A** whenever the newly added extension addition group is present in a value.

EXAMPLE

```
Extension root type     1st extension       2nd extension          3rd extension

A ::= SEQUENCE {        A ::= SEQUENCE {    A ::= SEQUENCE {    A ::= SEQUENCE {
      a INTEGER,          a INTEGER,          a INTEGER,          a INTEGER,
        ...                 ...,                ...,                ...,
      }                   [[                  [[                  [[
                            b BOOLEAN,          b BOOLEAN,          b BOOLEAN,
                            c INTEGER           c INTEGER           c INTEGER
                          ]]                  ]],                 ]],
                        }                     d SEQUENCE {        d SEQUENCE {
                                                e INTEGER,          e INTEGER,
                                                  ...,                ...,
                                                  ...,              [[
                                                f IA5String         g BOOLEAN OPTIONAL,
                                              }                     h BMPString
                                            }                       ]],
                                                                      ...,
                                                                    f IA5String
                                                                  }
                                                                }
```

G.1.8 It is also possible to add the version number to version brackets, but only if it is present on all brackets within a module, and only if all extensions in the module are within version brackets. It is recommended that version numbers be used. The ability to omit numbers and version brackets is for historical reasons. (Version brackets and version numbers were not allowed in earlier versions of this Recommendation | International Standard.) (See also G.3.)

G.1.9 While the normal practice will be for extension additions to be added over time, the underlying ASN.1 model and specification does not involve time. Two types are extension-related if one can be "grown" from the other by extension additions. That is, one contains all the components of the other. There may be types that have to be "grown" in the opposite direction (although this is unlikely). It could even be that, over time, a type *starts* with a lot of extension additions which were progressively removed! All that ASN.1 and its encoding rules care about is whether a pair of type specifications are extension-related or not. If they are, then *all* ASN.1 encoding rules will ensure interworking between their users.

G.1.10 We start with a type and then decide whether we are going to want interworking with implementations of earlier versions if we later have to extend it. If so, we include the extension marker *now*. We can then add later extension additions to the type with defined handling of the extended values by earlier systems. It is, however, important to note that adding an extension marker to a type that was previously without one (or removing an extension marker) may prevent interworking.

NOTE – When ECN is used, it can be possible to add extensions in version 2 at places that did not have extension markers in version 1, and still retain interworking between versions 1 and 2.

G.1.11 Table G.1 shows the ASN.1 types that can form the extension root type of an ASN.1 extension series, and the nature of the single extension addition that is permitted for that type (multiple extension additions can of course be made in succession, or together as an extension group).

Table G.1 – Extension additions

Extension root type	Nature of extension addition
ENUMERATED	Addition of a single further enumeration at the end of the "AdditionalEnumeration"s, with an enumeration value greater than that of any enumeration already added.
SEQUENCE and **SET**	Addition of a single type or extension addition group to the end of the "ExtensionAdditionList". "ComponentType"s that are extension additions (not contained in an extension addition group) are not required to be marked **OPTIONAL** or **DEFAULT**, although this will often be the case.
CHOICE	Addition of a single "NamedType" to the end of the "ExtensionAdditionAlternativesList"
Constraint notation	Addition of a single "AdditionalElementSetSpec" to the "ElementSetSpecs" notation

G.2 Meaning of version numbers

G.2.1 Version numbers are not used in BER or PER encodings. Their use (if any) in ECN encodings is determined by the ECN specification.

G.2.2 Version numbers are most useful when they relate to the means of decoding a complete PDU, not to an individual type. Where a type which is used as a component of several protocols and hence contributes to different complete PDUs, an addition to that type will normally require that the version number for all the PDUs to which it contributes be incremented.

G.2.3 When used to provide interworking between deployed systems, version numbers should be used on extension addition groups in such a way that deployed systems have knowledge of the syntax and semantics for all extension addition groups with a given version number (no matter where they appear within the protocol), and of all extension addition groups with an earlier version number. ECN specifiers will normally assume that version numbers have been allocated (to all parts of types to which ECN is applied) in accordance with this principle.

G.3 Requirements on encoding rules

G.3.1 An abstract syntax can be defined as the values of a single ASN.1 type that is an extensible type. It then contains all the values that can be obtained by the addition or removal of extension-additions. Such an abstract syntax is called an extension-related abstract syntax.

G.3.2 A set of well-formed encoding rules for an extension-related abstract syntax satisfies the additional requirements stated in G.3.3 to G.3.5.

NOTE – All ASN.1 encoding rules satisfy these requirements.

G.3.3 The definition of the procedures for transforming an abstract value into an encoding for transfer, and for transforming a received encoding into an abstract value shall recognize the possibility that the sender and receiver are using abstract syntaxes that are not identical, but are extension-related.

G.3.4 In this case, the encoding rules shall ensure that where the sender has a type specification that is earlier in the extension series than that of the receiver, values of the sender shall be transferred in such a way that the receiver can determine that extension additions are not present.

G.3.5 The encoding rules shall ensure that where the sender has a type specification that is later in the extension series than that of the receiver, transfer of values of that type to the receiver shall be possible.

G.4 Combination of (possibly extensible) constraints

G.4.1 Model

G.4.1.1 The basic ASN.1 model for applying constraints is simple: A type is a set of abstract values, and a constraint applied to it selects a subset of those abstract values. If the unconstrained type was not extensible, then the resulting type is defined to be extensible if and only if the applied constraint is defined to be extensible.

G.4.1.2 Even in this simple case, there is one feature to clarify: A type may be formally extensible, even though there can never be any extension additions. Consider:

```
A ::= INTEGER (MIN .. MAX, ... , 1..10)
```

As with many examples in this annex, this is something that nobody would ever write, but which tool vendors have to write code for because the ASN.1 standard has been left simple and general, and this example is therefore legal ASN.1. In this example, A is formally an extensible **INTEGER**, with the full range of integer values in the root.

G.4.1.3 Complexities arise from three main sources:

- The application of a constraint to a type that has already had an extensible constraint applied to it (serial application of constraints - see G.4.2).
- The combination of extensible constraints using **UNION** and **INTERSECTION** and **EXCEPT** (set arithmetic - see G.4.3).
- The use of a typereference (a contained subtype) in the set arithmetic of a constraint, when the typereference de-references to an extensible type (perhaps with actual extension additions - see G.4.4).

G.4.2 Serial application of constraints

G.4.2.1 Serial application of constraints occurs when a type is constrained (in an assignment to a typereference) and the typereference is subsequently used with a further constraint applied to it.

G.4.2.2 It can also, but less commonly, occur when a type has multiple constraints directly applied to it in a serial fashion. This latter form is used for many of the examples in this annex (for simplicity of exposition), but the case where a typereference links the two (or more) constraints is the form in which serial application normally occurs in real specifications.

G.4.2.3 There are two key points in the serial application of constraints:

– If a constrained type is extensible (and perhaps extended), the "extensible" flag and all extension additions are discarded if a further constraint is subsequently serially applied. The extensibility of a constrained type (and any extension additions) depends solely on the last constraint that is applied, which can reference only values in the root of the type that is being further constrained (the parent type). Values included in the root or the extension additions of the resulting type can only be values that are in the root of the parent type.

– The serial application of constraints is (for complex cases) not the same as a set arithmetic intersection, even when there is no extensibility involved. Firstly, the environment in which **MIN** and **MAX** are interpreted, and secondly the abstract values that can be referenced in the second constraint are very different in serial application from the situation where the two constraints are specified as an intersection of values from a common parent.

NOTE – Use of a range such as **20..28** in a constraint on an integer type is legal if (and only if) both **20** and **28** are in the (root of the) parent type, but the values referenced by this range specification are only those in (the root of) the parent. So if the parent has already been constrained to exclude the values **24** and **25**, the range **20..28** is referencing only **20** to **23** and **26** to **28**.

Here are some examples:

```
A1 ::= INTEGER (1..32, ... , 33..128)
      -- A1 is extensible, and contains values 1 to 128 with 1 to 32
      -- in the root and 33 to 128 as extension additions.

B1 ::= A1 (1..128)
      -- or equivalently

B1 ::= INTEGER (1..32, ... , 33..128) (1..128)
      -- These are illegal, as 128 is not in the parent, which
      -- lost its extension additions when it was further constrained

B2 ::= A1 (1..16)
      -- This is legal. B2 is not extensible, and contains 1 to 16.

A2 ::= INTEGER (1..32) (MIN .. 63)
      -- MIN is 1, and 63 is illegal

A3 ::= INTEGER ( (1..32) INTERSECTION (MIN..63) )
      -- This is legal.  MIN is minus infinity and A3 contains 1 to 32
```

G.4.3 Use of set arithmetic

G.4.3.1 The results are largely intuitive, and obey the normal mathematical rules for intersection, union and set difference (**EXCEPT**). In particular, both intersection and union are commutative, that is:

```
( <some set 1 of values> INTERSECTION <some set 2 of values> )
```

is the same as

```
( <some set 2 of values> INTERSECTION <some set 1 of values> )
```

similarly for **UNION**.

G.4.3.2 The commutativity is true, no matter what sets of values are extensible, and no matter what extension additions are present.

G.4.3.3 Misunderstandings can arise if an intersection makes it impossible for extension addition values ever to occur. This is similar to the case of **INTEGER (MIN..MAX, ...)**.

G.4.3.4 For example:

```
A ::= INTEGER ((1..256, ... , B) INTERSECTION (1..256))
-- A always contains (only) the values 1..256, no matter what values
-- B contains, but is nonetheless formally extensible
```

G.4.3.5 It is also important to remember that while parents lose their extensibility and extension additions when further constrained, and contained subtypes lose their extensibility and extension additions, sets of values directly specified in set arithmetic lose neither their extensibility nor their extension additions.

G.4.3.6 The rules for extensibility of sets of values produced by set arithmetic are clearly stated in 46.3 and 46.4, and do not depend on whether the set arithmetic makes actual extension additions possible or not.

G.4.3.7 The rules are summarized here for completeness, using **E** to denote a set of values with the "extensible" flag set and **N** to denote a set values which are formally non-extensible. The values in the root of each set are denoted by **R**, and the extension additions (if any) by **X**, and the contents of the result are shown for each case.

NOTE 1 – For the purposes of this annex and for simplicity of exposition, if a set of values is not extensible, we describe all its values as root values.

NOTE 2 – It is an illegal specification if the root of any resulting set of values used in a serially applied constraint is empty.

NOTE 3 – To avoid verbosity below, "Extensions" is used in place of the more correct "Extension additions".

G.4.3.8 The rules are:

```
N1 INTERSECTION N2 => N
        Root: R1 INTERSECTION R2
N1 INTERSECTION E2 => E
        Root: R1 INTERSECTION R2, Extensions: R1 INTERSECTION X2
E1 INTERSECTION E2 => E
        Root: R1 INTERSECTION R2, Extensions: ((R1 UNION X1)
                                              INTERSECTION
                                              (R2 UNION X2))
                                              EXCEPT
                                              (R1 INTERSECTION R2)
N1 UNION N2 => N
        Root: R1 UNION R2
N1 UNION E2 => E
        Root: R1 UNION R2, Extensions: X2
E1 UNION E2 => E
        Root: R1 UNION R2, Extensions: (R1 UNION X1 UNION R2 UNION X2)
                                       EXCEPT
                                       (R1 UNION R2)
N1 EXCEPT N2 => N
        Root: R1 EXCEPT R2
N1 EXCEPT E2 => N
        Root: R1 EXCEPT R2
E1 EXCEPT N2 => E
        Root: R1 EXCEPT R2, Extensions: (X1 EXCEPT R2)
                                        EXCEPT
                                        (R1 EXCEPT R2)
E1 EXCEPT E2 => E
        Root: R1 EXCEPT R2, Extensions: (X1 EXCEPT (R2 UNION X2) )
                                        EXCEPT
                                        (R1 EXCEPT R2)
N1 ... N2 => E
        Root: R1, Extensions: R2 EXCEPT R1
E1 ... N2 => E
        Root: R1, Extensions: X1 UNION R2
                              EXCEPT
                              R1
N1 ... E2 => E
        Root: R1, Extensions: R2 UNION X2
                              EXCEPT
                              R1
E1 ... E2 => E
        Root: R1, Extensions: X1 UNION R2 UNION E2
                              EXCEPT
                              R1
```

NOTE – If the result of set arithmetic on extensible sets of values does not have actual extension additions, or even can never have actual extension additions (no matter what extension additions are added to the extensible inputs), the result is still formally defined to be extensible for results **E** above.

G.4.4 Use of the Contained Subtype notation

A contained subtype may or may not be extensible, but when it is used in set arithmetic it is always treated as not extensible, and all its extension additions are discarded.

Annex H

Summary of the ASN.1 notation

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

The following lexical items are defined in clause 11:

typereference
identifier
valuereference
modulereference
comment
empty
number
realnumber
bstring
hstring
cstring
xmlbstring
xmlhstring
xmlcstring
xmlasn1typename
"true"
"false"
"::="
"**[[**"
"**]]**"
"**..**"
"**...**"
"</"
"/>"
"**{**"
"**}**"
"<"
">"
","
"."
"**(**"
"**)**"
"["
"]"
"-"
"**:**"
"="
""**"**" (QUOTATION MARK)
"**'**" (APOSTROPHE)
" " (SPACE)
";"
"@"
"|"
"**!**"
"**^**"
ABSENT
ABSTRACT-SYNTAX
ALL
APPLICATION
AUTOMATIC
BEGIN
BIT
BMPString
BOOLEAN
BY
CHARACTER
CHOICE
CLASS
COMPONENT
COMPONENTS
CONSTRAINED
CONTAINING
DEFAULT
DEFINITIONS
EMBEDDED
ENCODED
END
ENUMERATED
EXCEPT
EXPLICIT
EXPORTS
EXTENSIBILITY
EXTERNAL
FALSE
FROM
GeneralizedTime
GeneralString
GraphicString
IA5String
IDENTIFIER
IMPLICIT
IMPLIED
IMPORTS
INCLUDES
INSTANCE
INTEGER
INTERSECTION
ISO646String
MAX
MIN
MINUS-INFINITY
NULL
NumericString
OBJECT
ObjectDescriptor
OCTET
OF
OPTIONAL
PATTERN
PDV
PLUS-INFINITY
PRESENT
PrintableString
PRIVATE
REAL
RELATIVE-OID
SEQUENCE
SET
SIZE
STRING
SYNTAX

T61String

TAGS

TeletexString

TRUE

TYPE-IDENTIFIER

UNION

UNIQUE

UNIVERSAL

UniversalString

UTCTime

UTF8String

VideotexString

VisibleString

WITH

The following productions are used in this Recommendation | International Standard, with the above lexical items as terminal symbols:

```
ModuleDefinition ::= ModuleIdentifier
                     DEFINITIONS
                     TagDefault
                     ExtensionDefault
                     "::="
                     BEGIN
                     ModuleBody
                     END

ModuleIdentifier ::= modulereference
                         DefinitiveIdentifier

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

DefinitiveObjIdComponentList ::=
            DefinitiveObjIdComponent     |
            DefinitiveObjIdComponent DefinitiveObjIdComponentList

DefinitiveObjIdComponent ::=
            NameForm                 |
            DefinitiveNumberForm     |
            DefinitiveNameAndNumberForm

DefinitiveNumberForm     ::= number

DefinitiveNameAndNumberForm ::= identifier "(" DefinitiveNumberForm ")"

TagDefault ::=
            EXPLICIT TAGS    |
            IMPLICIT TAGS    |
            AUTOMATIC TAGS   |
            empty

ExtensionDefault ::=
        EXTENSIBILITY IMPLIED | empty

ModuleBody ::= Exports Imports AssignmentList |
                         empty

Exports ::=      EXPORTS SymbolsExported ";" |
                         EXPORTS ALL ";" |
                         empty

SymbolsExported ::= SymbolList |
                         empty

Imports ::=      IMPORTS SymbolsImported ";" |
                         empty

SymbolsImported ::= SymbolsFromModuleList |
                         empty

SymbolsFromModuleList ::=
                         SymbolsFromModule |
                         SymbolsFromModuleList  SymbolsFromModule

SymbolsFromModule ::=    SymbolList FROM GlobalModuleReference

GlobalModuleReference ::= modulereference AssignedIdentifier

AssignedIdentifier ::= ObjectIdentifierValue |
                         DefinedValue |
                         empty
```

```
SymbolList   ::= Symbol | SymbolList "," Symbol

Symbol  ::= Reference | ParameterizedReference

Reference ::=
                typereference                |
                valuereference               |
                objectclassreference         |
                objectreference              |
                objectsetreference

AssignmentList ::=          Assignment  | AssignmentList Assignment

Assignment ::=
                TypeAssignment               |
                ValueAssignment              |
                XMLValueAssignment           |
                ValueSetTypeAssignment       |
                ObjectClassAssignment        |
                ObjectAssignment             |
                ObjectSetAssignment          |
                ParameterizedAssignment

DefinedType ::=
                ExternalTypeReference        |
                typereference                |
                ParameterizedType            |
                ParameterizedValueSetType

ExternalTypeReference ::=
                    modulereference
                    "."
                    typereference

NonParameterizedTypeName ::=
            Externaltypereference        |
            typereference                |
            xmlasn1typename

DefinedValue ::=
                ExternalValueReference       |
                valuereference               |
                ParameterizedValue

ExternalValueReference ::=
                    modulereference
                    "."
                    valuereference

AbsoluteReference ::= "@" ModuleIdentifier
                "."
                ItemSpec

ItemSpec ::=
                typereference |
                ItemId "." ComponentId

ItemId ::= ItemSpec

ComponentId ::=
                identifier | number | "*"

TypeAssignment ::=          typereference
                                "::="
                                Type

ValueAssignment ::=   valuereference
                            Type
```

```
                                    ":::="
                                    Value

XMLValueAssignment ::=
        valuereference
        ":::="
        XMLTypedValue

XMLTypedValue ::=
        "<" & NonParameterizedTypeName ">"
        XMLValue
        "</" & NonParameterizedTypeName ">"|
        "<" & NonParameterizedTypeName "/>"

ValueSetTypeAssignment ::=          typereference
                                    Type
                                    ":::="
                                    ValueSet

ValueSet ::= "{" ElementSetSpecs "}"

Type ::= BuiltinType | ReferencedType | ConstrainedType

BuiltinType ::=
        BitStringType           |
        BooleanType             |
        CharacterStringType     |
        ChoiceType              |
        EmbeddedPDVType         |
        EnumeratedType          |
        ExternalType            |
        InstanceOfType          |
        IntegerType             |
        NullType                |
        ObjectClassFieldType    |
        ObjectIdentifierType    |
        OctetStringType         |
        RealType                |
        RelativeOIDType         |
        SequenceType            |
        SequenceOfType          |
        SetType                 |
        SetOfType               |
        TaggedType

NamedType ::= identifier   Type

ReferencedType ::=
        DefinedType         |
        UsefulType          |
        SelectionType       |
        TypeFromObject      |
        ValueSetFromObjects

Value  ::=  BuiltinValue | ReferencedValue | ObjectClassFieldValue

XMLValue ::= XMLBuiltinValue | XMLObjectClassFieldValue

BuiltinValue ::=
        BitStringValue          |
        BooleanValue            |
        CharacterStringValue    |
        ChoiceValue             |
        EmbeddedPDVValue        |
        EnumeratedValue         |
        ExternalValue           |
```

```
        InstanceOfValue         |
        IntegerValue            |
        NullValue               |
        ObjectIdentifierValue   |
        OctetStringValue        |
        RealValue               |
        RelativeOIDValue        |
        SequenceValue           |
        SequenceOfValue         |
        SetValue                |
        SetOfValue              |
        TaggedValue

XMLBuiltinValue ::=
        XMLBitStringValue           |
        XMLBooleanValue             |
        XMLCharacterStringValue     |
        XMLChoiceValue              |
        XMLEmbeddedPDVValue         |
        XMLEnumeratedValue          |
        XMLExternalValue            |
        XMLInstanceOfValue          |
        XMLIntegerValue             |
        XMLNullValue                |
        XMLObjectIdentifierValue    |
        XMLOctetStringValue         |
        XMLRealValue                |
        XMLRelativeOIDValue         |
        XMLSequenceValue            |
        XMLSequenceOfValue          |
        XMLSetValue                 |
        XMLSetOfValue               |
        XMLTaggedValue

ReferencedValue ::=
        DefinedValue            |
        ValueFromObject

NamedValue ::= identifier Value

XMLNamedValue ::=
        "<" & identifier ">" XMLValue "</" & identifier ">"

BooleanType ::=BOOLEAN

BooleanValue::= TRUE | FALSE

XMLBooleanValue ::=
        "<" & "true" "/>" |
        "<" & "false" "/>"

IntegerType ::=
                INTEGER         |
                INTEGER "{" NamedNumberList "}"

NamedNumberList ::=
                NamedNumber         |
                NamedNumberList "," NamedNumber

NamedNumber ::=
                identifier "(" SignedNumber ")"|
                identifier "(" DefinedValue ")"
```

```
SignedNumber ::= number | "-" number

IntegerValue ::=  SignedNumber | identifier

XMLIntegerValue ::=
        SignedNumber     |
        "<" & identifier "/>"

EnumeratedType ::=
            ENUMERATED "{" Enumerations "}"

Enumerations ::= RootEnumeration |
            RootEnumeration  ","  "..." ExceptionSpec |
            RootEnumeration  ","  "..." ExceptionSpec  ","  AdditionalEnumeration

RootEnumeration ::= Enumeration

AdditionalEnumeration ::= Enumeration

Enumeration ::=
            EnumerationItem | EnumerationItem "," Enumeration

EnumerationItem ::=
            identifier | NamedNumber

EnumeratedValue  ::=
            identifier

XMLEnumeratedValue ::= "<" & identifier "/>"

RealType ::= REAL

RealValue ::=
            NumericRealValue | SpecialRealValue

NumericRealValue ::=
            realnumber |
            "-" realnumber |
            SequenceValue              -- Value of the associated sequence type

SpecialRealValue ::=
            PLUS-INFINITY | MINUS-INFINITY

XMLRealValue ::=
        XMLNumericRealValue | XMLSpecialRealValue

XMLNumericRealValue ::=
        realnumber    |
        "-" realnumber

XMLSpecialRealValue ::=
        "<" & PLUS-INFINITY "/>" | "<" & MINUS-INFINITY "/>"

BitStringType   ::=         BIT STRING | BIT STRING "{" NamedBitList "}"

NamedBitList::=       NamedBit | NamedBitList "," NamedBit

NamedBit    ::=       identifier "(" number ")" |
                        identifier "(" DefinedValue ")"

BitStringValue  ::=         bstring | hstring | "{" IdentifierList "}" | "{"  "}" | CONTAINING Value

IdentifierList  ::=     identifier | IdentifierList "," identifier

XMLBitStringValue ::=
        XMLTypedValue       |
        xmlbstring          |
        XMLIdentifierList   |
        empty

XMLIdentifierList ::=
        "<" & identifier "/>"                |
```

```
        XMLIdentifierList "<" & identifier "/>"

OctetStringType ::= OCTET STRING

OctetStringValue ::= bstring | hstring | CONTAINING Value

XMLOctetStringValue ::=
        XMLTypedValue      |
        xmlhstring

NullType    ::= NULL

NullValue   ::= NULL

XMLNullValue ::= empty

SequenceType ::=
            SEQUENCE "{" "}"                                                    |
            SEQUENCE "{" ExtensionAndException OptionalExtensionMarker "}"|
            SEQUENCE "{" ComponentTypeLists "}"

ExtensionAndException ::= "..." | "..." ExceptionSpec

OptionalExtensionMarker ::= "," "..." | empty

ComponentTypeLists ::=
            RootComponentTypeList                                               |
            RootComponentTypeList "," ExtensionAndException ExtensionAdditions
                OptionalExtensionMarker                                         |
            RootComponentTypeList "," ExtensionAndException ExtensionAdditions
                ExtensionEndMarker "," RootComponentTypeList                    |
            ExtensionAndException ExtensionAdditions ExensionEndMarker ","
                RootComponentTypeList                                           |
            ExtensionAndException ExtensionAdditions OptionalExtensionMarker

RootComponentTypeList ::= ComponentTypeList

ExtensionEndMarker ::= "," "..."

ExtensionAdditions ::= "," ExtensionAdditionList | empty

ExtensionAdditionList ::= ExtensionAddition |
            ExtensionAdditionList "," ExtensionAddition

ExtensionAddition ::= ComponentType | ExtensionAdditionGroup

ExtensionAdditionGroup ::= "[[" VersionNumber ComponentTypeList "]]"

VersionNumber ::= empty | number ":"

ComponentTypeList ::=           ComponentType |
                                ComponentTypeList "," ComponentType

ComponentType ::=
            NamedType                   |
            NamedType OPTIONAL          |
            NamedType DEFAULT Value     |
            COMPONENTS OF Type

SequenceValue   ::=  "{" ComponentValueList "}" | "{" "}"

ComponentValueList ::=      NamedValue|
                        ComponentValueList "," NamedValue

XMLSequenceValue ::=
        XMLComponentValueList |
        empty

XMLComponentValueList ::=
        XMLNamedValue                   |
        XMLComponentValueList XMLNamedValue
```

```
SequenceOfType  ::=  SEQUENCE OF Type

SequenceOfValue  ::=  "{" ValueList "}" | "{"  "}"

ValueList   ::=     Value | ValueList "," Value

XMLSequenceOfValue ::=
        XMLValueList                |
        XMLDelimitedItemList        |
        XMLSpaceSeparatedList       |
        empty

XMLValueList ::=
        XMLValueOrEmpty |
        XMLValueOrEmpty XMLValueList

XMLValueOrEmpty ::=
        XMLValue |
        "<" & NonParameterizedTypeName "/>"

XMLSpaceSeparatedList ::=
        XMLValueOrEmpty |
        XMLValueOrEmpty " " XMLSpaceSeparatedList

XMLDelimitedItemList ::=
        XMLDelimitedItem                        |
        XMLDelimitedItem XMLDelimitedItemList

XMLDelimitedItem ::=
        "<" & NonParameterizedTypeName ">" XMLValue
            "</" & NonParameterizedTypeName ">" |
        "<" & identifier ">" XMLValue "</" & identifier ">"

SetType ::= SET "{"  "}" |
                SET "{" ExtensionAndException OptionalExtensionMarker "}" |
                SET "{" ComponentTypeLists "}"

SetValue    ::=    "{" ComponentValueList "}" | "{"  "}"

XMLSetValue ::= XMLComponentValueList    |    empty

SetOfType   ::=   SET OF Type

SetOfValue   ::= "{" ValueList "}"  |  "{"  "}"

XMLSetOfValue ::=
            XMLValueList                |
            XMLDelimitedItemList        |
            XMLSpaceSeparatedList       |
            empty

ChoiceType   ::= CHOICE "{" AlternativeTypeLists "}"

AlternativeTypeLists ::=
            RootAlternativeTypeList |
            RootAlternativeTypeList  ","
                    ExtensionAndException  ExtensionAdditionAlternatives
    OptionalExtensionMarker

RootAlternativeTypeList  ::=  AlternativeTypeList

ExtensionAdditionAlternatives  ::=  ","  ExtensionAdditionAlternativesList | empty

ExtensionAdditionAlternativesList  ::= ExtensionAdditionAlternative |
            ExtensionAdditionAlternativesList  ","  ExtensionAdditionAlternative
```

```
ExtensionAdditionAlternative ::= ExtensionAdditionAlternativesGroup | NamedType

ExtensionAdditionAlternativesGroup ::= "[[" VersionNumber AlternativeTypeList "]]"

AlternativeTypeList ::=          NamedType |
                                 AlternativeTypeList "," NamedType

ChoiceValue ::=      identifier ":" Value

XMLChoiceValue ::= "<" & identifier ">" XMLValue "</" & identifier ">"

SelectionType ::=        identifier "<" Type

TaggedType ::= Tag Type |
               Tag IMPLICIT Type |
               Tag EXPLICIT Type

Tag    ::=           "[" Class ClassNumber "]"

ClassNumber ::= number | DefinedValue

Class ::= UNIVERSAL     |
          APPLICATION |
          PRIVATE     |
          empty

TaggedValue ::= Value

XMLTaggedValue ::= XMLValue

EmbeddedPDVType ::=          EMBEDDED PDV

EmbeddedPDVValue ::=         SequenceValue

XMLEmbeddedPdvValue ::= XMLSequenceValue

ExternalType ::= EXTERNAL

ExternalValue ::= SequenceValue

XMLExternalValue ::= XMLSequenceValue

ObjectIdentifierType ::=         OBJECT IDENTIFIER

ObjectIdentifierValue ::=        "{" ObjIdComponentsList "}" |
                                 "{" DefinedValue ObjIdComponentsList "}"

ObjIdComponentsList ::=          ObjIdComponents    |
                                 ObjIdComponents ObjIdComponentsList

ObjIdComponents ::= NameForm       |
                    NumberForm     |
                    NameAndNumberForm|
                    DefinedValue

NameForm    ::=      identifier

NumberForm    ::=        number | DefinedValue

NameAndNumberForm ::=      identifier "(" NumberForm ")"

XMLObjectIdentifierValue ::=
        XMLObjIdComponentList

XMLObjIdComponentList ::=
        XMLObjIdComponent |
        XMLObjIdComponent & "." & XMLObjIdComponentList

XMLObjIdComponent ::=
        NameForm |
        XMLNumberForm |
        XMLNameAndNumberForm
```

```
XMLNumberForm ::= number

XMLNameAndNumberForm ::=
        identifier & "(" & XMLNumberForm & ")"

RelativeOIDType ::=
                RELATIVE-OID

RelativeOIDValue ::=
            "{" RelativeOIDComponentsList "}"

RelativeOIDComponentsList ::=
            RelativeOIDComponents         |
            RelativeOIDComponents RelativeOIDComponentsList

RelativeOIDComponents ::=     NumberForm     |
                              NameAndNumberForm|
                              DefinedValue

XMLRelativeOIDValue ::=
        XMLRelativeOIDComponentList

XMLRelativeOIDComponentList ::=
        XMLRelativeOIDComponent |
        XMLRelativeOIDComponent & "." & XMLRelativeOIDComponentList

XMLRelativeOIDComponent ::=
        XMLNumberForm |
        XMLNameAndNumberForm

CharacterStringType ::= RestrictedCharacterStringType | UnrestrictedCharacterStringType

RestrictedCharacterStringType ::=
        BMPString   |
                    GeneralString          |
                    GraphicString          |
                    IA5String              |
                    ISO646String           |
                    NumericString          |
                    PrintableString        |
                    TeletexString          |
                    T61String              |
                    UniversalString        |
                    UTF8String             |
                    VideotexString         |
                    VisibleString
```

RestrictedCharacterStringValue ::= cstring | CharacterStringList | Quadruple | Tuple

CharacterStringList ::= "{" CharSyms "}"

CharSyms ::= CharsDefn | CharSyms "," CharsDefn

CharsDefn ::= cstring | Quadruple | Tuple | DefinedValue

Quadruple ::= "{" Group "," Plane "," Row "," Cell "}"

Group ::= number

Plane ::= number

Row ::= number

Cell ::= number

Tuple ::= "{" TableColumn "," TableRow "}"

TableColumn ::= number

TableRow ::= number

XMLRestrictedCharacterStringValue ::= xmlcstring

UnrestrictedCharacterStringType ::= `CHARACTER STRING`

CharacterStringValue ::= RestrictedCharacterStringValue | UnrestrictedCharacterStringValue

XMLCharacterStringValue ::=
XMLRestrictedCharacterStringValue |
XMLUnrestrictedCharacterStringValue

UnrestrictedCharacterStringValue ::= SequenceValue

XMLUnrestrictedCharacterStringValue ::= XMLSequenceValue

UsefulType ::= typereference

The following character string types are defined in 37.1:

`NumericString`	`VisibleString`
`PrintableString`	`ISO646String`
`TeletexString`	`IA5String`
`T61String`	`GraphicString`
`VideotexString`	`GeneralString`
`UniversalString`	`BMPString`

The following useful types are defined in clauses 42 to 44:

`GeneralizedTime`

`UTCTime`

`ObjectDescriptor`

The following productions are used in clauses 45 to 47:

ConstrainedType ::=
Type Constraint |
TypeWithConstraint

TypeWithConstraint ::=
`SET` Constraint `OF` Type |
`SET` SizeConstraint `OF` Type |
`SEQUENCE` Constraint `OF` Type |
`SEQUENCE` SizeConstraint `OF` Type

```
Constraint ::= "(" ConstraintSpec  ExceptionSpec ")"

ConstraintSpec ::=
        SubtypeConstraint |
        GeneralConstraint

ExceptionSpec ::= "!"  ExceptionIdentification | empty

ExceptionIdentification ::= SignedNumber     |
                    DefinedValue |
                    Type ":" Value

SubtypeConstraint ::= ElementSetSpecs

ElementSetSpecs ::=
                RootElementSetSpec                  |
                RootElementSetSpec  ","  "..."     |
                RootElementSetSpec  ","  "..."  ","  AdditionalElementSetSpec

RootElementSetSpec ::= ElementSetSpec

AdditionalElementSetSpec ::= ElementSetSpec

ElementSetSpec ::= Unions | ALL  Exclusions

Unions ::= Intersections |
                UElems UnionMark Intersections

UElems ::= Unions

Intersections ::= IntersectionElements |
                IElems IntersectionMark IntersectionElements

IElems ::= Intersections

IntersectionElements ::= Elements | Elems Exclusions

Elems ::= Elements

Exclusions ::= EXCEPT Elements

UnionMark  ::=  "|"    |    UNION

IntersectionMark  ::=  "^" |     INTERSECTION

Elements  ::=
        SubtypeElements         |
        ObjectSetElements       |
        "(" ElementSetSpec ")"

SubtypeElements ::=
        SingleValue                 |
        ContainedSubtype            |
        ValueRange                  |
        PermittedAlphabet           |
        SizeConstraint              |
        TypeConstraint              |
        InnerTypeConstraints        |
        PatternConstraint
```

SingleValue ::= Value

ContainedSubtype ::= Includes Type

Includes ::= `INCLUDES` | empty

ValueRange ::= LowerEndpoint ".." UpperEndpoint

LowerEndpoint ::= LowerEndValue | LowerEndValue "<"

UpperEndpoint ::= UpperEndValue | "<" UpperEndValue

LowerEndValue ::= Value | `MIN`

UpperEndValue ::= Value | MAX

SizeConstraint ::= `SIZE` Constraint

PermittedAlphabet ::= `FROM` Constraint

TypeConstraint ::= Type

InnerTypeConstraints ::=
`WITH COMPONENT` SingleTypeConstraint |
`WITH COMPONENTS` MultipleTypeConstraints

SingleTypeConstraint::= Constraint

MultipleTypeConstraints ::= FullSpecification | PartialSpecification

FullSpecification ::= "{" TypeConstraints "}"

PartialSpecification ::= "{" "..." "," TypeConstraints "}"

TypeConstraints ::=
NamedConstraint |
NamedConstraint "," TypeConstraints

NamedConstraint ::=
identifier ComponentConstraint

ComponentConstraint ::= ValueConstraint PresenceConstraint

ValueConstraint ::= Constraint | empty

PresenceConstraint ::= `PRESENT` | `ABSENT` | `OPTIONAL` | empty

PatternConstraint ::= `PATTERN` Value

ITU-T Recommendation X.681
International Standard 8824-2

Information Technology –
Abstract Syntax Notation One (ASN.1):
Information Object Specification

INTERNATIONAL STANDARD 8824-2

ITU-T RECOMMENDATION X.681

Information Technology – Abstract Syntax Notation One (ASN.1): Information Object Specification

Summary

This Recommendation | International Standard provides the ASN.1 notation which allows information object classes as well as individual information objects and sets thereof to be defined and given reference names. An information object class defines the form of a conceptual table (an information object set) with one column for each field in the information object class, and with each complete row defining an information object.

Source

The ITU-T Recommendation X.681 was approved on the 13th of July 2002. The identical text is also published as ISO/IEC International Standard 8824-2.

CONTENTS

Page

Introduction ... iii

1 Scope ... 1

2 Normative references ... 1
- 2.1 Identical Recommendations | International Standards ... 1

3 Definitions ... 1
- 3.1 Specification of basic notation ... 1
- 3.2 Constraint specification ... 1
- 3.3 Parameterization of ASN.1 specification ... 1
- 3.4 Additional definitions ... 2

4 Abbreviations ... 3

5 Convention ... 3

6 Notation ... 3
- 6.1 Assignments ... 3
- 6.2 Types ... 3
- 6.3 Values ... 4
- 6.4 Elements ... 4

7 ASN.1 lexical items ... 4
- 7.1 Information object class references ... 4
- 7.2 Information object references ... 4
- 7.3 Information object set references ... 4
- 7.4 Type field references ... 4
- 7.5 Value field references ... 4
- 7.6 Value set field references ... 5
- 7.7 Object field references ... 5
- 7.8 Object set field references ... 5
- 7.9 Word ... 5
- 7.10 Additional keywords ... 5

8 Referencing definitions ... 5

9 Information object class definition and assignment ... 6

10 Syntax List ... 9

11 Information object definition and assignment ... 11

12 Information object set definition and assignment ... 13

13 Associated tables ... 15

14 Notation for the object class field type ... 15

15 Information from objects ... 17

Annex A The TYPE-IDENTIFIER information object class ... 20

Annex B Abstract syntax definitions ... 21

Annex C The instance-of type ... 22

Annex D Examples ... 24
- D.1 Example usage of simplified OPERATION class ... 24
- D.2 Example usage of "ObjectClassFieldType" ... 25
- D.3 Illustrate usage of objects and object sets ... 25

Annex E Tutorial annex on the ASN.1 model of object set extension ... 27

Annex F Summary of the notation ... 28

Introduction

An application designer frequently needs to design a protocol which will work with any of a number of instances of some class of information objects, where instances of the class may be defined by a variety of other bodies, and may be added to over time. Examples of such information object classes are the "operations" of Remote Operating Service (ROS) and the "attributes" of the OSI Directory.

This Recommendation | International Standard provides notation which allows information object classes as well as individual information objects and information object sets thereof to be defined and given reference names.

An information object class is characterized by the kinds of fields possessed by its instances. A field may contain:

- an arbitrary type (a type field); or
- a single value of a specified type (a fixed-type value field); or
- a single value of a type specified in a (named) type field (a variable-type value field);
- a non-empty set of values of a specified type (a fixed-type value set field); or
- a non-empty set of values of a type specified in a (named) type field (a variable-type value set field); or
- a single information object from a specified information object class (an object field);
- an information object set from a specified information object class (an object set field).

A fixed-type value field of an information object class may be selected to provide unique identification of information objects in that class. This is called the identifier field for that class. Values of the identifier field, if supplied, are required to be unique within any information object set that is defined for that class. They may, but need not, serve to unambiguously identify information objects of that class within some broader scope, particularly by the use of object identifier as the type of the identifier field.

An information object class is defined by specifying:

- the names of the fields;
- for each field, the form of that field (type, fixed-type value, variable-type value, fixed-type value set, variable-type value set, object, or object set);
- optionality and default settings of fields;
- which field, if any, is the identifier field.

An individual information object in the class is defined by providing the necessary information for each field.

The notation defined herein permits an ASN.1 type to be specified by reference to a field of some information object class – the object class field type. In ITU-T Rec. X.682 | ISO/IEC 8824-3, notation is provided to enable this type to be restricted by reference to some specific information object set.

It can be useful to consider the definition of an information object class as defining the form of an underlying conceptual table (the associated table) with one column for each field, and with a completed row defining an information object. The form of the table (determined by the information object class specification) determines the sort of information to be collected and used to complete some protocol specification. The underlying conceptual table provides the link between those specifying information objects of that class and the protocol which needs that information to complete its specification. Typically, the actual information object set used to complete a particular protocol specification will be a parameter of that protocol (see ITU-T Rec. X.683 | ISO/IEC 8824-4).

The "InformationFromObjects" notation referencing a specific object or object set (probably a parameter) can be used to extract information from cells of conceptual tables.

This Recommendation | International Standard:

- Specifies a notation for defining an information object class, and for identifying it with a reference name (see clause 9).
- Specifies a notation by which the definer of an information object class can provide a defined syntax for the definition of information objects of that class; a default notation is provided for classes for which no defined syntax has been defined (see clause 10).
- Specifies a notation for defining an information object, and for assigning it to a reference name (see clause 11), and provides analogous notation for an object set (see clause 12).
- Defines the "associated table" for an object or object set of a class (see clause 13).

– Specifies notation for the object class field type and its values (see clause 14).

NOTE – These constructs enable an ASN.1 type to be specified using a named field of a named information object class. Constraints on that type to restrict it to values related to a specific information object set appear in ITU-T Rec. X.682 | ISO/IEC 8824-3.

– Specifies notation for extracting information from objects (see clause 15).

The set of information objects used in defining an object set may be partially or entirely unknown at the time of definition of an ASN.1 specification. Such cases occur, for example, in network management where the set of managed objects varies while the network manager is executing. This Recommendation | International Standard specifies the rules for inclusion of an *extension marker* in the definition of object sets to signal to implementors the intention of the designer that the contents of the object set is not fully defined in the ASN.1 specification. When an object set is defined with an extension marker, the implementor must provide means, possibly outside the scope of ASN.1, for dynamically adding objects to the object set and removing previously added objects from the object set.

Annex A, which is an integral part of this Recommendation | International Standard, specifies the information object class whose object class reference is **TYPE-IDENTIFIER**. This is the simplest useful class, with just two fields, an identifier field of type object identifier, and a single type field which defines the ASN.1 type for carrying all information concerning any particular object in the class. It is defined herein because of the widespread use of information objects of this form.

Annex B, which is an integral part of this Recommendation | International Standard, specifies the notation for defining an abstract syntax (composed of the set of values of a single ASN.1 type) by the definition of an appropriate information object.

Annex C, which is an integral part of this Recommendation | International Standard, specifies the notation for the instance-of type (the **INSTANCE OF** notation).

Annex D, which is not an integral part of this Recommendation | International Standard, provides examples on how to use the notation described in this Recommendation | International Standard.

Annex E, which is not an integral part of this Recommendation | International Standard, provides a summary of the ASN.1 model of object set extension.

Annex F, which is not an integral part of this Recommendation | International Standard, provides a summary of the notation defined herein.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

Information Technology – Abstract Syntax Notation One (ASN.1): Information Object Specification

1 Scope

This Recommendation | International Standard is part of Abstract Syntax Notation One (ASN.1) and provides notation for specifying information object classes, information objects and information object sets.

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 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.

2.1 Identical Recommendations | International Standards

- ITU-T Recommendation X.680 (2002) | ISO/IEC 8824-1:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.*
- ITU-T Recommendation X.682 (2002) | ISO/IEC 8824-3:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*
- ITU-T Recommendation X.683 (2002) | ISO/IEC 8824-4:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications.*

3 Definitions

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

3.1 Specification of basic notation

This Recommendation | International Standard uses the terms defined in ITU-T Rec. X.680 | ISO/IEC 8824-1.

3.2 Constraint specification

This Recommendation | International Standard uses the following terms defined in ITU-T Rec. X.682 | ISO/IEC 8824-3:

- table constraint.

3.3 Parameterization of ASN.1 specification

This Recommendation | International Standard uses the following terms defined in ITU-T Rec. X.683 | ISO/IEC 8824-4:

a) parameterized type;

b) parameterized value.

3.4 Additional definitions

3.4.1 associated table: (For some information object or information object set) an abstract table, derivable from the object or object set by flattening the hierarchical structure resulting from the presence of link fields (see 3.4.15).

NOTE – An associated table can be used to determine the precise nature of some constraint (see ITU-T Rec. X.682 | ISO/IEC 8824-3) which has been applied using an object set.

3.4.2 default syntax: The notation which shall be used for defining information objects of classes whose definers have not provided a defined syntax (see example 11.10).

3.4.3 defined syntax: A notation, provided by the definer of a class, which allows information objects of that class to be defined in a user-friendly manner.

NOTE – For example, the defined syntax for the class **OPERATION** might allow instances of the class to be defined by the word **ARGUMENT** followed by **&ArgumentType**, then the word **RESULT** followed by the **&ResultType**, then the word **CODE** followed by **&operationCode** (see example 11.11).

3.4.4 extensible object set: An object set with an extension marker or which has been defined using set arithmetic with object sets that are extensible.

3.4.5 field: A component of an information object class. Each field is a type field, a fixed-type value field, a variable-type value field, a fixed-type value set field, a variable-type value set field, an information object field or an information object set field.

3.4.6 field name: A name which identifies a field of some class; either the class which specifies the field directly, in which case the name is a primitive field name, or a class which has a chain of link fields to that in which the field is actually specified (see 9.13 and 9.14).

3.4.7 governing (class); governor: An information object class definition or reference which affects the interpretation of a part of the ASN.1 syntax, requiring it to reference or to specify information objects of the governing class.

3.4.8 identifier field: A fixed-type value field of a class, selected to provide unique identification of information objects in that class. Values of the identifier field, if supplied, are required to be unambiguous within any information object set that is defined for that class. They may, but need not, serve to unambiguously identify information objects of that class within some broader scope.

NOTE 1 – The identifier field has a fixed ASN.1 type, and values of that type can be carried in protocol to identify information objects within the class.

NOTE 2 – The scope within which the identifier is unambiguous is that of an information object set. It could, however, also be made unambiguous within any given abstract syntax, or within an entire application context, or could even be global across all classes and all application contexts by use of the object identifier type for the identifier field.

3.4.9 information object: An instance of some information object class, being composed of a set of fields which conform to the field specifications of the class.

NOTE – For example, one specific instance of the information object class **OPERATION** (mentioned in the example in 3.4.10) might be **invertMatrix**, which has an **&ArgumentType** field containing the type **Matrix**, a **&ResultType** field also containing the type **Matrix**, and an **&operationCode** field containing the value 7 (see example in 10.13).

3.4.10 information object class (class): A set of fields, forming a template for the definition of a potentially unbounded collection of information objects, the instances of the class.

NOTE – For example, an information object class **OPERATION** might be defined to correspond to the **operation** concept of Remote Operations (ROS). Each of the various named field specifications would then correspond to some aspect which can vary from one operation instance to another. Thus, there could be **&ArgumentType**, **&ResultType**, and **&operationCode** fields, the first two specifying type fields and the third specifying a value field.

3.4.11 information object field: A field which contains an information object of some specified class.

3.4.12 information object set: A non-empty set of information objects, all defined using the same information object class reference name.

NOTE – For example, one information object set, **MatrixOperations**, of the class **OPERATION** (used in the example in 3.4.10) might contain **invertMatrix** (mentioned in 3.4.9) together with other related operations, such as **addMatrices**, **multiplyMatrices**, etc. Such an object set might be used in defining an abstract syntax that makes provision for the invocation and result reporting of all of these operations (see example in 12.11).

3.4.13 information object set field: A field which contains an information object set of some specified class.

3.4.14 instance-of type: A type, defined by referencing an information object class which associates object identifiers with types.

3.4.15 link field: An object or object set field.

3.4.16 object class field type: A type specified by reference to some field of an information object class. In ITU-T Rec. X.682 | ISO/IEC 8824-3, notation is provided to enable this type to be restricted by reference to an information object set of the class.

3.4.17 primitive field name: The name specified directly in an information object class definition without use of a link field.

3.4.18 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 information object class is permitted. Recursive definition of an information object or an information object set is forbidden by 11.2 and 12.2 respectively.

3.4.19 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 information object class (including an encoding structure) is permitted. Recursive instantiation of an information object or an information object set is forbidden by 11.2 and 12.2 respectively.

3.4.20 type field: A field which contains an arbitrary type.

3.4.21 value field: A field which contains a value. Such a field is either of fixed-type or of variable-type. In the former case the type of the value is fixed by the field specification. In the latter case the type of the value is contained in some (specific) type field of the same information object.

3.4.22 value set field: A field which contains a non-empty set of values of some type. Such a field is either of fixed-type or of variable-type. In the former case the type of the values is fixed by the field specification. In the latter case the type of the values is contained in some (specific) type field of the same information object.

NOTE – The set of values in a value set field for an information object constitutes a subtype of the specified type.

4 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviation applies:

ASN.1 Abstract Syntax Notation One

5 Convention

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

6 Notation

This clause summarizes the notation defined in this Recommendation | International Standard.

6.1 Assignments

The following notations which can be used as alternatives for "Assignment" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 12) are defined in this Recommendation | International Standard:

- ObjectClassAssignment (see 9.1);
- ObjectAssignment (see 11.1);
- ObjectSetAssignment (see 12.1).

6.2 Types

6.2.1 The following notations which can be used as alternatives for "BuiltinType" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 16.2) are defined in this Recommendation | International Standard:

- ObjectClassFieldType (see 14.1);
- InstanceOfType (see Annex C).

6.2.2 The following notations which can be used as alternatives for "ReferencedType" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 16.3) are defined in this Recommendation | International Standard:

- TypeFromObject (see clause 15);
- ValueSetFromObjects (see clause 15).

6.3 Values

6.3.1 The following notation which can be used as an alternative for "Value" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 16.7) is defined in this Recommendation | International Standard:

– ObjectClassFieldValue (see 14.6);

6.3.2 The following notation which can be used as an alternative for "BuiltinValue" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 16.9) is defined in this Recommendation | International Standard:

– InstanceOfValue (see Annex C).

6.3.3 The following notation which can be used as an alternative for "ReferencedValue" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 16.11) is defined in this Recommendation | International Standard:

– ValueFromObject (see clause 15).

6.4 Elements

6.4.1 The following notation which can be used as an alternative for "Elements" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 46.5) is defined in this Recommendation | International Standard:

– ObjectSetElements (see 12.10).

7 ASN.1 lexical items

In addition to the lexical items specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 11, this Recommendation | International Standard makes use of the lexical items specified in the following subclauses. The general rules applicable to these lexical items are as defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.1. These new lexical items make use of the ASN.1 character set, as specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 10, and in addition the character ampersand ("**&**").

NOTE – The Note in ITU-T Rec. X.680 | ISO/IEC 8824-1, 10.1, also applies to the lexical items specified in 7.1 to 7.9 below.

7.1 Information object class references

Name of lexical item – objectclassreference

An "objectclassreference" shall consist of a sequence of characters as specified for a "typereference" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.2, except that no lower-case letters shall be included.

7.2 Information object references

Name of lexical item – objectreference

An "objectreference" shall consist of a sequence of characters as specified for a "valuereference" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.4.

7.3 Information object set references

Name of lexical item – objectsetreference

An "objectsetreference" shall consist of a sequence of characters as specified for a "typereference" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.2.

7.4 Type field references

Name of lexical item – typefieldreference

A "typefieldreference" shall consist of an ampersand ("**&**") immediately followed by a sequence of characters as specified for a "typereference" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.2.

7.5 Value field references

Name of lexical item – valuefieldreference

A "valuefieldreference" shall consist of an ampersand ("**&**") immediately followed by a sequence of characters as specified for a "valuereference" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.4.

7.6 Value set field references

Name of lexical item – valuesetfieldreference

A "valuesetfieldreference" shall consist of an ampersand ("**&**") immediately followed by a sequence of characters as specified for a "typereference" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.2.

7.7 Object field references

Name of lexical item – objectfieldreference

An "objectfieldreference" shall consist of an ampersand ("**&**") immediately followed by a sequence of characters as specified for an "objectreference" in 7.2.

7.8 Object set field references

Name of lexical item – objectsetfieldreference

An "objectsetfieldreference" shall consist of an ampersand ("**&**") immediately followed by a sequence of characters as specified for an "objectsetreference" in 7.3.

7.9 Word

Name of lexical item – word

A "word" shall consist of a sequence of characters as specified for a "typereference" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.2, except that no lower-case letters or digits shall be included.

7.10 Additional keywords

The names **CLASS**, **INSTANCE**, **SYNTAX** and **UNIQUE** are listed in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.27 as reserved words.

8 Referencing definitions

8.1 The constructs:

```
DefinedObjectClass ::=
        ExternalObjectClassReference   |
        objectclassreference           |
        UsefulObjectClassReference

DefinedObject ::=
        ExternalObjectReference |
        objectreference

DefinedObjectSet ::=
        ExternalObjectSetReference |
        objectsetreference
```

are used to reference class, information object, and information object set definitions, respectively.

8.2 References to information objects and information object sets have a governing class. It is a requirement that the referenced information object and the information objects in the referenced information object set shall be of the governing class. There is no equivalent of "value mappings" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, Annex B) specified for information objects, so the above statement means that the information object or information object set must be defined using the same information object class reference as is used as the governor (or one obtained from it by simple reference assignment). Two identical (but textually distinct) instances of the information object class notation do not identify the same information object class for the purposes of this requirement.

8.3 Except as specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 12.15, the "objectclassreference", "objectreference", and "objectsetreference" alternatives shall only be used within the module in which a class or information object or information object set is assigned (see 9.1, 11.1 and 12.1) to that reference.

The "ExternalObjectClassReference", "ExternalObjectReference", and "ExternalObjectSetReference" alternatives are defined as follows:

ExternalObjectClassReference ::=
modulereference
"."
objectclassreference

ExternalObjectReference ::=
modulereference
"."
objectreference

ExternalObjectSetReference ::=
modulereference
"."
objectsetreference

These alternatives shall not be used unless the corresponding "objectclassreference", "objectreference", or "objectsetreference" has been assigned a class or information object or information object set (see 9.1, 11.1 and 12.1) within the module (different from the referencing module) identified by the corresponding "modulereference". It is that class or information object or information object set respectively which is referenced.

8.4 The "UsefulObjectClassReference" alternative of "DefinedObjectClass" is defined as follows:

UsefulObjectClassReference ::= `TYPE-IDENTIFIER` | `ABSTRACT-SYNTAX`

of which the first alternative is specified in Annex A, and the second in Annex B.

NOTE – The names **`TYPE-IDENTIFIER`** and **`ABSTRACT-SYNTAX`** are listed in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.27 as reserved words.

9 Information object class definition and assignment

9.1 The construct "ObjectClassAssignment" is used to assign an information object class to a reference name ("objectclassreference"). This construct is one of the alternatives for "Assignment" in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 12, and is defined as follows:

ObjectClassAssignment ::=
objectclassreference
":::="
ObjectClass

9.2 The information object class is that defined by the construct "ObjectClass":

ObjectClass ::=
DefinedObjectClass |
ObjectClassDefn |
ParameterizedObjectClass

If the "ObjectClass" is a:

a) "DefinedObjectClass", then the class definition is the same as that of the class referred to;

b) "ObjectClassDefn", then the class is defined as described in 9.3;

c) "ParameterizedObjectClass", then the class is defined as described in ITU-T Rec. X.683 | ISO/IEC 8824-4, 9.2.

9.3 Every class is ultimately defined by an "ObjectClassDefn":

ObjectClassDefn ::=
`CLASS`
"{" FieldSpec "," + "}"
WithSyntaxSpec?

WithSyntaxSpec ::= `WITH SYNTAX` SyntaxList

This notation allows the definer of a class to provide the named field specifications, each of which is a "FieldSpec", as defined in 9.4. Optionally, the definer can provide an information object definition syntax ("SyntaxList"), as defined in 10.5. The definer of the class may also specify semantics associated with the definition of the class.

9.4 Each "FieldSpec" specifies and names one of the fields which shall or may be associated with instances of the class:

```
FieldSpec ::=
        TypeFieldSpec                   |
        FixedTypeValueFieldSpec         |
        VariableTypeValueFieldSpec      |
        FixedTypeValueSetFieldSpec      |
        VariableTypeValueSetFieldSpec   |
        ObjectFieldSpec                 |
        ObjectSetFieldSpec
```

The various alternatives for "FieldSpec" are specified in the following subclauses.

9.5 A "TypeFieldSpec" specifies that the field is a type field (see 3.4.20):

```
TypeFieldSpec ::=
        typefieldreference
        TypeOptionalitySpec?

TypeOptionalitySpec ::= OPTIONAL | DEFAULT Type
```

The name of the field is "typefieldreference". If the "TypeOptionalitySpec" is absent, all information object definitions for that class are required to include a specification of a type for that field. If **OPTIONAL** is present, then the field can be left undefined. If **DEFAULT** is present, then the following "Type" provides the default setting for the field if it is omitted in a definition.

9.6 A "FixedTypeValueFieldSpec" specifies that the field is a fixed-type value field (see 3.4.21):

```
FixedTypeValueFieldSpec ::=
        valuefieldreference
        Type
        UNIQUE ?
        ValueOptionalitySpec ?

ValueOptionalitySpec ::= OPTIONAL | DEFAULT Value
```

The name of the field is "valuefieldreference". The "Type" construct specifies the type of the value contained in the field. The "ValueOptionalitySpec", if present, specifies that the value may be omitted in an information object definition, or, in the **DEFAULT** case, that omission produces the following "Value", which shall be of that type. The presence of the keyword **UNIQUE** specifies that this field is an identifier field as defined in 3.4.8 (see also ITU-T Rec. X.682 | ISO/IEC 8824-3, 10.20). If the keyword is present, the "ValueOptionalitySpec" shall not be "**DEFAULT** Value".

9.7 Where a value is assigned for an identifier field, that value is required to be unambiguous within any defined information object set.

9.8 A "VariableTypeValueFieldSpec" specifies that the field is a variable-type value field (see 3.4.21):

```
VariableTypeValueFieldSpec ::=
        valuefieldreference
        FieldName
        ValueOptionalitySpec?
```

The name of the field is "valuefieldreference". The "FieldName" (see 9.14), which is relative to the class being specified, shall be that of a type field; the type field which is either in the same information object as the value field, or is linked by the chain of object fields whose references appear in the "FieldName", will contain the type of the value. (All link fields whose field references appear in the "FieldName" shall be object fields.) The "ValueOptionalitySpec", if present, specifies that the value may be omitted in an information object definition, or, in the **DEFAULT** case, that omission produces the following "Value". The "ValueOptionalitySpec" shall be such that:

a) if the type field denoted by the "FieldName" has a "TypeOptionalitySpec" of **OPTIONAL**, then the "ValueOptionalitySpec" shall also be **OPTIONAL**; and

b) if the "ValueOptionalitySpec" is "**DEFAULT** Value", then the type field denoted by the "FieldName" shall have a "TypeOptionalitySpec" of "**DEFAULT** Type", and "Value" shall be a value of that type.

9.9 A "FixedTypeValueSetFieldSpec" specifies that the field is a fixed-type value set field (see 3.4.22):

```
FixedTypeValueSetFieldSpec ::=
        valuesetfieldreference
        Type
        ValueSetOptionalitySpec ?
```

ValueSetOptionalitySpec ::= OPTIONAL | DEFAULT ValueSet

NOTE – "ValueSet" is defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 15.6 and 15.7, and allows the explicit listing (in curly braces) of the set of values, or the use of a "typereference" for a subtype of the "Type".

The name of the field is "valuesetfieldreference". The "Type" construct specifies the type of the values contained in the field. The "ValueSetOptionalitySpec", if present, specifies that the field may be unspecified in information object definition, or, in the **DEFAULT** case, that omission produces the following "ValueSet", which shall be a subtype of that type.

9.10 A "VariableTypeValueSetFieldSpec" specifies that the field is a variable-type value set field (see 3.4.22):

VariableTypeValueSetFieldSpec ::=
valuesetfieldreference
FieldName
ValueSetOptionalitySpec?

The name of the field is "valuesetfieldreference". The "FieldName" (see 9.14), which is relative to the class being specified, shall be that of a type field; the type field which is either in the same information object as the value set field, or is linked by the chain of object fields whose references appear in the "FieldName", will contain the type of the values. (All link fields whose field references appear in the "FieldName" shall be object fields.) The "ValueSetOptionalitySpec", if present, specifies that the value set may be omitted in an information object definition, or, in the **DEFAULT** case, that omission produces the following "ValueSet". The "ValueSetOptionalitySpec" shall be such that:

a) if the type field denoted by the "FieldName" has a "TypeOptionalitySpec" of **OPTIONAL**, then the "ValueSetOptionalitySpec" shall also be **OPTIONAL**; and

b) if the "ValueSetOptionalitySpec" is "**DEFAULT** ValueSet", then the type field denoted by the "FieldName" shall have a "TypeOptionalitySpec" of "**DEFAULT** Type", and "ValueSet" shall be a subtype of that type.

9.11 An "ObjectFieldSpec" specifies that the field is an information object field (see 3.4.11):

ObjectFieldSpec ::=
objectfieldreference
DefinedObjectClass
ObjectOptionalitySpec ?

ObjectOptionalitySpec ::= OPTIONAL | DEFAULT Object

The name of the field is "objectfieldreference". The "DefinedObjectClass" references the class of the object contained in the field (which may be the "ObjectClass" currently being defined). The "ObjectOptionalitySpec", if present, specifies that the field may be unspecified in an information object definition, or, in the **DEFAULT** case, that omission produces the following "Object" (see 11.3) which shall be of the "DefinedObjectClass".

9.12 An "ObjectSetFieldSpec" specifies that the field is an information object set field (see 3.4.13):

ObjectSetFieldSpec ::=
objectsetfieldreference
DefinedObjectClass
ObjectSetOptionalitySpec ?

ObjectSetOptionalitySpec ::= OPTIONAL | DEFAULT ObjectSet

The name of the field is "objectsetfieldreference". The "DefinedObjectClass" references the class of the objects contained in the field. The "ObjectSetOptionalitySpec", if present, specifies that the field may be unspecified in an information object definition, or, in the **DEFAULT** case, that omission produces the following "ObjectSet" (see 12.3), all of whose objects shall be of "DefinedObjectClass".

9.13 The construct "PrimitiveFieldName" is used to identify a field relative to the class containing its specification:

PrimitiveFieldName ::=
typefieldreference |
valuefieldreference |
valuesetfieldreference |
objectfieldreference |
objectsetfieldreference

The names of all of the fields specified in the class definition shall be distinct.

9.14 The construct "FieldName" is used to identify a field relative to some class which either contains the field specification directly or which has a chain of link fields to the containing class. The chain is indicated by a list of "PrimitiveFieldName"s separated by periods.

FieldName ::= PrimitiveFieldName " . " +

9.15 If there is any chain (of length one or more) of specifications of link fields (see 3.4.15) such that:

a) the first is in the class which is being defined and is not the field being defined; and

b) each subsequent one is a field of the class used in defining the previous; and

c) the last is defined using the class which is being defined,

then at least one of the field specifications shall have an "ObjectOptionalitySpec" or "ObjectSetOptionalitySpec" (as appropriate).

NOTE – This is to prevent recursive information object class definitions (which are in general permitted) with no finite representation for an information object of that recursive class.

9.16 Examples

An expanded version of the information object class described informally as an example in 3.4.10 could be defined as follows:

```
OPERATION ::= CLASS
      {
            &ArgumentType          OPTIONAL,
            &ResultType            OPTIONAL,
            &Errors                ERROR OPTIONAL,
            &Linked                OPERATION OPTIONAL,
            &resultReturned        BOOLEAN DEFAULT TRUE,
            &code                  INTEGER UNIQUE
      }

ERROR ::= CLASS
      {
            &ParameterType   OPTIONAL,
            &code            INTEGER UNIQUE
      }
```

NOTE 1 – This example is based upon the operation and error concepts of the Remote Operations standard, but simplified for the present purposes.

NOTE 2 – The fields specified for this class include two type fields (**&ArgumentType** and **&ResultType**) two object set fields (**&Errors** and **&Linked**) and two value fields (**&resultReturned** and **&code**) the latter being an identifier field.

NOTE 3 – Any information object set made up of **OPERATION**s must be such that no two objects in the set have the same value for the **&code** field. (The same applies to object sets of **ERROR**s.)

NOTE 4 – The **OPERATION** information object class includes a chain of link fields as described in 9.15 above. The chain is of length one and is formed by the **&Linked** field, which is specified (recursively) by means of **OPERATION**. However, this is quite valid, because the field is designated **OPTIONAL** (see 9.15).

NOTE 5 – Neither of these examples includes a "WithSyntaxSpec". However, corresponding examples which do are provided in 10.13.

10 Syntax List

10.1 It is frequently the case that a single specification defines an information object class, for which many other independent specifications separately define information objects. It can be appropriate for the definer of the class to provide a user-friendly notation for the definition of information objects in that class.

10.2 This clause specifies a notation by which the specifier of an information object class defines the class-specific defined syntax for the specification of information objects of that class.

10.3 The notation is the syntactic construct "SyntaxList", which occurs in the syntactic construct "ObjectClassDefn" (see 9.3).

10.4 A "SyntaxList" specifies the syntax for the definition of a single information object of the class being defined. The syntax appears as the "DefinedSyntax" in the following subclause.

NOTE – It is a property of this specification that the end of any syntactic construct defined by a "SyntaxList" (an instance of "DefinedSyntax") can be determined by:

a) ignoring ASN.1 comments;

b) treating character string values as lexical tokens;

c) expecting an initial "**{**", matching nested "**{**" and "**}**", and terminating on an unmatched "**}**".

10.5 The "SyntaxList" specifies the sequence of "DefinedSyntaxToken" that is to appear in the "DefinedSyntax" (see 11.6):

SyntaxList ::= "{" TokenOrGroupSpec empty + "}"

TokenOrGroupSpec ::= RequiredToken | OptionalGroup

OptionalGroup ::= "[" TokenOrGroupSpec empty + "]"

RequiredToken ::=
Literal |
PrimitiveFieldName

NOTE 1 – The writer of "SyntaxList" is not given the full power of BNF. Roughly, the notational power is equivalent to that commonly used in specifying command line syntaxes for command interpreters. The list of possible "RequiredToken"s are given in the order they are permitted; one or more consecutive tokens can be made optional by enclosing them in square brackets.

NOTE 2 – When parsing a "SyntaxList", any occurrence of "**[[**" (or "**]]**") is not interpreted as the lexical items "**[[**" (or "**]]**" respectively) defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.19 and 11.20, but as two lexical items "**[**" and "**[**" (or "**]**" and "**]**" respectively).

10.6 A "word" token used as a "Literal" shall not be one of the following:

BIT
BOOLEAN
CHARACTER
CHOICE
EMBEDDED
END
ENUMERATED
EXTERNAL
FALSE
INSTANCE
INTEGER
INTERSECTION
MINUS-INFINITY
NULL
OBJECT
OCTET
PLUS-INFINITY
REAL
RELATIVE-OID
SEQUENCE
SET
TRUE
UNION

NOTE – This list comprises only and all those ASN.1 reserved words which can appear as the first lexical item of a "Type", "Value", "ValueSet", "Object" or "ObjectSet", and also the reserved word **END**. Use of other ASN.1 reserved words does not cause ambiguity and is permitted. Where the defined syntax is used in an environment in which a "word" is also a "typereference" or "objectsetreference", the use as a "word" takes precedence.

10.7 A "Literal" specifies the actual inclusion of that "Literal", which is either a "word" or a comma (","), at that position in the defined syntax:

Literal ::=
word |
","

10.8 Each "PrimitiveFieldName" specifies the inclusion (at that position in the new syntax) of a "Setting" (see 11.7) for the corresponding field.

10.9 Each "PrimitiveFieldName" of the information object class shall appear precisely once.

10.10 When, in the parse process, an "OptionalGroup" is encountered, and the following lexical item is syntactically acceptable as the first lexical item in the optional group, then that group is assumed to be present. If it is not syntactically acceptable as the first lexical item in the optional group, then that group is assumed to be absent.

NOTE – In order to avoid unexpected effects, designers should normally make the first lexical item in an optional group a "Literal".

10.11 An instance of use of the "DefinedSyntax" is invalid unless it specifies all mandatory fields for the information object class.

10.12 In order to ensure easy parsing of the new syntax and to prevent abuses, the following additional restrictions are placed on the definer of new syntax:

a) Every "OptionalGroup" is required to have at least one "PrimitiveFieldName" or "OptionalGroup" within it.

NOTE 1 – This is to help prevent the apparent collection of information which is not reflected in any field of the information object.

b) The use of "OptionalGroup"s shall be such that at no time in the parsing process can a "Setting" appear that could potentially be a setting for more than one "FieldName".

c) If an "OptionalGroup" starts with a "Literal", then the first token following the "OptionalGroup" shall also be a "Literal" and shall be different from the first "Literal" of all immediately preceding "OptionalGroup"s,

while the following restriction is placed upon the user of the "DefinedSyntax":

d) Whenever a "Literal" is present in a "DefinedSyntax" that occurs in an "OptionalGroup" a "Setting" for a "PrimitiveFieldName" in that "OptionalGroup" shall also be present.

NOTE 2 – This is to help prevent the apparent collection of information which is not reflected in any field of the information object.

NOTE 3 – The following example is a legal syntax but restriction d) prevents the user from writing **LITERAL** without following it by one or both of the optional groups:

```
[LITERAL  [A &field]  [B &field2]]
```

10.13 Examples

The examples of class definitions from 9.16 above can be equipped with defined syntax to provide a "user-friendly" way of defining instances of the classes (this defined syntax is used in the example in 11.11):

```
OPERATION ::= CLASS
      {
            &ArgumentType                 OPTIONAL,
            &ResultType                   OPTIONAL,
            &Errors                       ERROR OPTIONAL,
            &Linked                       OPERATION OPTIONAL,
            &resultReturned               BOOLEAN DEFAULT TRUE,
            &operationCode                INTEGER UNIQUE
      }
      WITH SYNTAX
      {
            [ARGUMENT                     &ArgumentType]
            [RESULT                       &ResultType]
            [RETURN RESULT                &resultReturned]
            [ERRORS                       &Errors]
            [LINKED                       &Linked]
            CODE                          &operationCode
      }

ERROR ::= CLASS
      {
            &ParameterType                OPTIONAL,
            &errorCode                    INTEGER UNIQUE
      }
      WITH SYNTAX
      {
            [PARAMETER                    &ParameterType]
            CODE                          &errorCode
      }
```

11 Information object definition and assignment

11.1 The syntactic construct "ObjectAssignment" is used to assign an information object of a specified class to a reference name ("objectreference"). This construct is one of the alternatives for "Assignment" in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 12, and is defined as follows:

```
ObjectAssignment ::=
      objectreference
      DefinedObjectClass
      "::="
      Object
```

11.2 There shall be no recursive definition (see 3.4.18) of an "objectreference", and there shall be no recursive instantiation (see 3.4.19) of an "objectreference".

11.3 The information object, which shall be of the class referenced by "DefinedObjectClass", is that defined by the construct "Object":

Object ::=
DefinedObject |
ObjectDefn |
ObjectFromObject |
ParameterizedObject

If the "Object" is a:

a) "DefinedObject", then the object is the same as that referred to;

b) "ObjectDefn", then the object is as specified in 11.4;

c) "ObjectFromObject", then the object is as specified in clause 15;

d) "ParameterizedObject", then the object is defined as specified in ITU-T Rec. X.683 | ISO/IEC 8824-4, 9.2.

11.4 Every information object is ultimately defined by an "ObjectDefn":

ObjectDefn ::=
DefaultSyntax |
DefinedSyntax

The "ObjectDefn" shall be "DefaultSyntax" (see 11.5) if the class definition does not include a "WithSyntaxSpec" and shall be "DefinedSyntax" (see 11.6) if it does include one.

11.5 The "DefaultSyntax" construct is defined as follows:

DefaultSyntax ::= "{" FieldSetting "," * "}"

FieldSetting ::= PrimitiveFieldName Setting

There shall be precisely one "FieldSetting" for each "FieldSpec" in the class definition which is not **OPTIONAL** and does not have a **DEFAULT**, and at most one "FieldSetting" for each other "FieldSpec". The "FieldSetting"s can appear in any order. The "PrimitiveFieldName" in each "FieldSetting" shall be the name of the corresponding "FieldSpec". The construct "Setting" is specified in 11.7.

11.6 The "DefinedSyntax" construct is defined as follows:

DefinedSyntax ::= "{" DefinedSyntaxToken empty * "}"

DefinedSyntaxToken ::=
Literal |
Setting

The "SyntaxList" in the "WithSyntaxSpec" (see clause 10) determines the sequence of "DefinedSyntaxToken"s that are to appear in the "DefinedSyntax". The construct "Setting" is specified in 11.7; each occurrence specifies the setting for some field of the information object. The construct "Literal" is defined in 10.7; "Literal"s are present for human readability.

11.7 A "Setting" specifies the setting of some field within an information object being defined:

Setting ::=
Type |
Value |
ValueSet |
Object |
ObjectSet

If the field is:

a) a type field, the "Type" alternative;

b) a value field, the "Value" alternative;

c) a value set field, the "ValueSet" alternative;

d) an information object field, the "Object" alternative;

e) an information object set field, the "ObjectSet" alternative,

shall be selected.

NOTE – The setting is further restricted as described in the appropriate subclause of 9.5 to 9.12 above, and 11.8 to 11.9.

11.8 A setting of a variable-type value field shall be a value of the type specified by the appropriate type field of the same or linked object (that is, the value notation for an open type is not employed).

11.9 A setting of a variable-type value set field shall be a value set of the type specified by the appropriate type field of the same or linked object (that is, the value notation for an open type is not employed).

11.10 Examples (Default Syntax)

Given the information object class definitions of 9.16 above (which do not include a "WithSyntaxSpec") instances of the classes are defined using the "DefaultSyntax". For example (an expanded version of the example given in 3.4.9):

```
invertMatrix OPERATION ::=
      {
            &ArgumentType               Matrix,
            &ResultType                 Matrix,
            &Errors                     {determinantIsZero},
            &operationCode              7
      }

determinantIsZero ERROR ::=
      {
            &errorCode                  1
      }
```

11.11 Examples (Defined Syntax)

In 10.13, the example classes are provided "WithSyntaxSpec" and thus, instances of the classes are defined using the "DefinedSyntax". The examples of 11.10 would be written thus:

```
invertMatrix OPERATION ::=
      {
            ARGUMENT                    Matrix
            RESULT                      Matrix
            ERRORS                      {determinantIsZero}
            CODE                        7
      }

determinantIsZero ERROR ::=
      {
            CODE                        1
      }
```

12 Information object set definition and assignment

12.1 The syntactic construct "ObjectSetAssignment" is used to assign a set of information objects of a specified class to a reference name ("objectsetreference"). This construct is one of the alternatives for "Assignment" in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 12, and is defined as follows:

```
ObjectSetAssignment ::=
      objectsetreference
      DefinedObjectClass
      "::="
      ObjectSet
```

12.2 There shall be no recursive definition (see 3.4.18) of an "objectsetreference", and there shall be no recursive instantiation (see 3.4.19) of an "objectsetreference".

12.3 The information object set, which shall be of the class referenced by "DefinedObjectClass", is defined by the construct "ObjectSet":

```
ObjectSet ::= "{" ObjectSetSpec "}"

ObjectSetSpec ::=
      RootElementSetSpec                        |
      RootElementSetSpec "," "..."              |
      "..."                                     |
      "..." "," AdditionalElementSetSpec        |
      RootElementSetSpec "," "..." "," AdditionalElementSetSpec
```

"RootElementSetSpec" and "AdditionalElementSetSpec" are specified in ITU-T Rec. X.680 | ISO/IEC 8824-1 and enable an information object set to be specified in terms of information objects or sets thereof of the governing class. There shall be at least one information object in the set unless the third alternative ("**...**") of "ObjectSetSpec" is specified. In the latter case, the presence of the ellipses is an indication that the object set is initially empty but will have objects dynamically added to it by the application program.

NOTE 1 – The elements that are referenced by "ObjectSetSpec" are the union of the elements referenced by the "RootElementSetSpec" and "AdditionalElementSetSpec".

NOTE 2 – Unlike extensible types such as **SET** or **SEQUENCE**, or extensible subtype constraints, which are static in respect to the set of "understood" values being set for each version of the ASN.1 specification, an extensible object set can grow and contract dynamically within a given version. Indeed, it may expand and contract within a given instance of use of an application program as it dynamically defines or undefines objects (see Annex E for further discussion).

12.4 The result of set arithmetic involving information object sets that are extensible is specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 46.

12.5 If an extensible information object set, **A**, is referenced in the definition of another object set, **B**, its extension marker and its extensions are inherited by **B**.

12.6 If a "ValueSetFromObjects" (see clause 15) is defined using an extensible information object set, the resulting value set does not inherit the extension marker from that information object set.

12.7 If a type is constrained by a table constraint (see ITU-T Rec. X.682 | ISO/IEC 8824-3, 10.3) and the object set referenced in the table constraint is extensible, the type does not inherit the extension marker from the object set. If the type is meant to be extensible, then an extension marker shall be explicitly added to its "ElementSetSpecs".

12.8 If a type is constrained by an information object set that is not extensible, then a conforming implementation shall support all the information objects in that set, and shall not generate encodings using information objects not in the set.

12.9 If a type is constrained by an information object set that is extensible, then a conforming implementation may choose to support an information object in either the root or the extensions as a local decision on each instance of the constrained type. It shall not generate encodings using other information objects with values for any **UNIQUE** field which are those of objects in either the root or the extensions of the extensible information object set, but may otherwise generate encodings for any information object of the required class.

12.10 The notation for "ObjectSetElements" is as follows:

```
ObjectSetElements ::=
        Object                  |
        DefinedObjectSet        |
        ObjectSetFromObjects    |
        ParameterizedObjectSet
```

The elements specified by this notation are determined by which alternative is employed, as follows:

a) If the "Object" alternative is used, then only the object so designated is specified. That object shall be of the governing class.

b) If any of the remaining alternatives is used, then all of the objects of the set so designated are specified. The objects shall be of the governing class. If the "DefinedObjectSet" alternative is used, the object set is that referred to. If the "ObjectSetFromObjects" alternative is used then the object set is as specified in clause 15. If the "ParameterizedObjectSet" alternative is used, then the object set is as specified in ITU-T Rec. X.683 | ISO/IEC 8824-4, 9.2.

12.11 Example

The information object set described informally in the Note in 3.4.12 can be specified as follows:

```
MatrixOperations OPERATION ::=
      {
            invertMatrix |
            addMatrices |
            subtractMatrices |
            multiplyMatrices
      }
```

13 Associated tables

13.1 Every information object or information object set can be viewed as a table: its associated table. Each cell of the associated table corresponds to the setting of some field of an information object, or is empty. The set of columns of the associated table is determined by the class to which the object or objects belong; the set of rows, however, is determined by the object or objects involved.

13.2 Given the definition of a class, the set of columns is determined as follows:

a) There is one column for each field specification in the class definition. Each such column is named by the corresponding "PrimitiveFieldName".

b) There is an additional set of columns corresponding to each link field specification. This set of columns is that determined by the application of these rules for the governing class of the link field, except that their names are prefixed by the "PrimitiveFieldName" of the link field, and a period (".").

NOTE – These rules are recursive, and are such that if a class is directly or indirectly self-referential the set of columns is not finite. This is not prohibited.

13.3 Given an information object of some class, the associated table is that which would result from applying 13.4 to the object set containing just that object.

13.4 Given an information object set of some class, the set of rows in the associated table are those which would result from performing the following recursive procedure:

a) Start with one row for each object in the object set. In each such row, the cells in the columns named by "PrimitiveFieldName"s will correspond to the setting of the appropriate field in the object, while all other cells will be empty.

b) For each link field appearing in some row in the set:

1) Generate the (subordinate) associated table of the contents of the link field.

2) Next, replace the row in which the link field appears by a collection of rows, one for each row of the subordinate associated table. Each of the rows in this collection is the same as that being replaced, except that the cells from the selected row of the subordinate associated table are used to fill the corresponding cells, hitherto empty, whose "FieldName"s are prefixed by the link field's "PrimitiveFieldName".

NOTE – These rules are recursive, and are such that if an information object is directly or indirectly self-referential, the procedure will not terminate. This is not prohibited. In practice it is only necessary to know the contents of cells with names of a finite length, and a bounded procedure can be devised for this.

13.5 Examples of valid "FieldName"s

The following "FieldName"s are among those which are valid for the associated table for information objects or information object sets of class **OPERATION** (as defined in 10.13):

```
&ArgumentType
&Errors.&Parameter
&Errors.&errorCode
&Linked.&ArgumentType
&Linked.&Linked.&operationCode
&Linked.&Linked.&Linked.&Linked.&Linked.&Errors.&errorCode
```

Because the class **OPERATION** is self-referential (through the **&Linked** field), the number of columns is not finite.

14 Notation for the object 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, 14.2 to 14.5 specify the type that is referenced.

14.1 The notation for an object class field type (see 3.4.16) shall be "ObjectClassFieldType":

```
ObjectClassFieldType ::=
        DefinedObjectClass
        "."
        FieldName
```

where the "FieldName" is as specified in 9.14 relative to the class identified by the "DefinedObjectClass".

14.2 For a type field, the notation defines an open type, that is, one whose set of values is the complete set of all possible values that can be specified using ASN.1. The specification of constraints using a corresponding information object set (see ITU-T Rec. X.682 | ISO/IEC 8824-3) may restrict this type to a specific type. The following constraints on the use of this notation apply when the "FieldName" references a type field:

a) This notation shall not be used directly or indirectly in the definition of the type of a value or value set field of an information object class.

b) This notation has an indeterminate tag and thus cannot be used where a tag distinct from that of some other type is required.

NOTE 1 – This restriction can normally be avoided by (explicitly) tagging the type.

NOTE 2 – Notwithstanding the statement in ITU-T Rec. X.680 | ISO/IEC 8824-1, 48.7.3 that the conceptually added element for an extension marker has a tag that is distinct from the tag of all known ASN.1 types, the open type shall not be used where it is required to have a tag that is distinct from that of the conceptually added element.

c) This notation shall not be implicitly tagged.

NOTE 3 – The reason for this is that when this open type is restricted to a particular type that type may be a choice type.

d) Encoding rules are required to encode the value assigned to a component defined in this way in such a way that a receiver can successfully determine the abstract values corresponding to all other parts of the construction in which the component is embedded without any knowledge of the actual type of this component.

NOTE 4 – This "Type" construct will commonly be constrained by use of an information object set and the "AtNotation", as specified in ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 10. Users of ASN.1 are, however, cautioned that use of this notation without the application of a constraint can lead to ambiguity in implementation requirements, and should normally be avoided.

14.3 For a fixed-type value or a fixed type value set field, the notation denotes the "Type" that appears in the specification of that field in the definition of the information object class.

14.4 For a variable-type value or a variable-type value set field, the notation defines an open type. Its use is subject to the same restrictions as specified in 14.2.

14.5 This notation is not permitted if the field is an object field or an object set field.

14.6 The notation for defining a value of this type shall be "ObjectClassFieldValue", or when used in an "XMLTypedValue", an "XMLObjectClassFieldValue":

ObjectClassFieldValue ::=
OpenTypeFieldVal |
FixedTypeFieldVal

OpenTypeFieldVal ::= Type ":" Value

FixedTypeFieldVal ::= BuiltinValue | ReferencedValue

XMLObjectClassFieldValue ::=
XMLOpenTypeFieldVal |
XMLFixedTypeFieldVal

XMLOpenTypeFieldVal ::= XMLTypedValue

XMLFixedTypeFieldVal ::= XMLBuiltinValue

14.7 For a fixed-type value or value set field defined by an "ObjectClassFieldType", the "FixedTypeFieldVal" or "XMLFixedTypeFieldVal" shall be used, and shall be a value of the "Type" specified in the definition of the information object class.

14.8 For a type field or a variable-type value or value set field defined by an "ObjectClassFieldType", the "OpenTypeFieldVal" shall be used in any "Value". The "Type" in the "OpenTypeFieldVal" shall be any ASN.1 type, and the "Value" shall be any value of that type.

14.9 For a type field or a variable-type value or value set field defined by an "ObjectClassFieldType", the "XMLOpenTypeFieldVal" shall be used in any "XMLValue". When used in an ASN.1 module, the type identified by the XMLTypedValue shall be any ASN.1 type (but see ITU-T Rec. X.680 | ISO/IEC 8824-1, 13.3) and the "XMLValue" in the "XMLTypedValue" shall be any value of that type.

NOTE – When the notation is used as specified in ITU-T Rec. X.693 | ISO/IEC 8825-4, 8.3.1, the type of the "XMLTypedValue" in an "XMLOpenTypeFieldVal" is identified by the protocol (for example, by a component relation constraint), the "NonParameterizedTypeName" in the "XMLTypedValue" is derived from this, and the "XMLValue" is a value of this type.

14.10 The character sequence in the "xmlasn1typename" item for the XML value notation (for an "ObjectClassFieldType") which is a "XMLFixedTypeFieldVal" shall be the character sequence for the "Type" specified in the information object class. The XML value notation for sequence-of and set-of (see ITU-T Rec. X.680 | ISO/IEC 8824-1, Table 5) shall be determined by the "Type" specified in the information object class.

14.11 The character sequence in the "xmlasn1typename" item for the XML value notation (for an "ObjectClassFieldType") which is an "XMLOpenTypeFieldVal" will never be required. The XML value notation for sequence-of and set-of (see ITU-T Rec. X.680 | ISO/IEC 8824-1, Table 5) shall be "XMLDelimitedItemList".

14.12 For an "XMLOpenTypeFieldVal", if the "Type" specified in the information object (after ignoring any tags) is a "typereference" or an "ExternalTypeReference", then the "NonParameterizedTypeName" shall be that "typereference" or "ExternalTypeReference", otherwise it shall be the "xmlasn1typename" specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, Table 4 corresponding to the built-in type specified in the information object.

14.13 Example usage of "ObjectClassFieldType"

Each of the following examples is based on the example in 10.13 and shows (a) a possible "ObjectClassFieldType", (b) the type to which the example type (a) is equivalent (when used unconstrained), and (c) the notation for an example value of that type.

1 (a) **`OPERATION.&operationCode`**
(b) **`INTEGER`**
(c) **`7`**

2 (a) **`OPERATION.&ArgumentType`**
(b) *open type*
(c)
```
Matrix:
{{1, 0, 0, 0},
 {0, 1, 0, 0},
 {0, 0, 1, 0},
 {0, 0, 0, 1}}
```

3 (a) **`OPERATION.&Linked.&Linked.&Errors.&errorCode`**
(b) **`INTEGER`**
(c) **`1`**

4 (a) **`OPERATION.&Linked.&ArgumentType`**
(b) *open type*
(c) **`UniversalString:{planckConstant, " and ", hamiltonOperator}`**

15 Information from objects

15.1 Information from the column of the associated table for an object or an object set can be referenced by the various cases of the "InformationFromObjects" notation:

```
InformationFromObjects ::=
        ValueFromObject                 |
        ValueSetFromObjects             |
        TypeFromObject                  |
        ObjectFromObject                |
        ObjectSetFromObjects

ValueFromObject ::=
        ReferencedObjects
        "."
        FieldName

ValueSetFromObjects ::=
        ReferencedObjects
        "."
        FieldName

TypeFromObject ::=
        ReferencedObjects
        "."
        FieldName
```

ObjectFromObject ::=
ReferencedObjects
"."
FieldName

ObjectSetFromObjects ::=
ReferencedObjects
"."
FieldName

ReferencedObjects ::=
DefinedObject |
ParameterizedObject |
DefinedObjectSet |
ParameterizedObjectSet

NOTE – The production "InformationFromObjects" is provided to aid understanding and for use in the English text. It is not referenced elsewhere in this Recommendation | Internation Standard.

15.2 This notation references the total contents of the referenced column of the associated table for the "ReferencedObjects".

15.3 Depending on the form of the "ReferencedObjects" and the "FieldName", this notation can denote a value, a value set, a type, an object, or an object set. These five cases are denoted by the constructs "ValueFromObject", "ValueSetFromObjects", "TypeFromObject", "ObjectFromObject", and "ObjectSetFromObjects" respectively. Each of these constructs is a special case of "InformationFromObjects".

15.4 The "InformationFromObjects" production can be divided into two parts. The first part is formed by deleting the final (or only) "PrimitiveFieldName" and its preceding period. If the first part denotes an object or an object set, then 15.5 to 15.12 apply. Otherwise the notation is illegal. The second part is the final (or only) "PrimitiveFieldName".

NOTE – (Tutorial) Given the following definition:

`obj.&a.&b.&c.&d`

the first part in the definition is **`obj.&a.&b.&c`** and the second part is **`&d`**.

15.5 The first column of Table 1 indicates the first part defined in 15.4. The second column indicates the second part defined in 15.4. The third column indicates which (if any) of the five cases of "InformationFromObjects" (listed in 15.3) applies.

Table 1 – Permissable cases of "InformationFromObjects"

The first part of InformationFromObjects	The second part of InformationFromObjects	Construct
object	fixed-type value field	"ValueFromObject"
	variable-type value field	"ValueFromObject"
	fixed-type value set field	"ValueSetFromObjects"
	variable-type value set field	not permitted
	type field	"TypeFromObject"
	object field	"ObjectFromObject"
	object set field	"ObjectSetFromObject"
object set	fixed-type value field	"ValueSetFromObjects"
	variable-type value field	not permitted
	fixed-type value set field	"ValueSetFromObjects"
	variable-type value set field	not permitted
	type field	not permitted
	object field	"ObjectSetFromObjects"
	object set field	"ObjectSetFromObjects"

15.6 For a "TypeFromObject" and a "ValueSetFromObjects", the XML value notation for sequence-of and set-of (see ITU-T Rec. X.680 | ISO/IEC 8824-1, Table 5) shall be determined by the "Type" specified in the information object(s).

15.7 If the first part references an object and the second part references a fixed-type value set field, the "ValueSetFromObjects" is equivalent to a type with a SimpleTableConstraint. The type is "<ClassName>.<FieldName>" where "<ClassName>" is the Information Object Class of the object, and "<FieldName>" is the field referenced by the second part. The SimpleTableConstraint consists of an object set containing only the object referenced by the first part. The object set is not extensible.

15.8 If the first part references an object set and the second part references a fixed-type value field or a fixed-type value set field, the "ValueSetFromObjects" is equivalent to a type with a "SimpleTableConstraint". The type is "<ClassName>.<FieldName>" where "<ClassName>" is the information object class of the object set referenced by the first part, and "<FieldName>" is the field referenced by the second part. The "SimpleTableConstraint" consists of the object set referenced by the first part.

15.9 A "ValueSetFromObjects" can be defined using an information object set that is initially empty but extensible. Such an information object set shall have at least one object in it whenever a value set defined in terms of it is used by an application.

15.10 If object sets are involved and the final "PrimitiveFieldName" identifies an object set field, then "ObjectSetFromObjects" is the union of the selected object sets.

15.11 As shown in Table 1, the notation is not permitted if an object set is involved and the final "PrimitiveFieldName" identifies a variable-type value or value set field or a type field.

15.12 Use of this notation is not permitted if all cells in the column being referenced are empty, except where it is used to directly define a field of an information object which is **`OPTIONAL`** (or **`DEFAULT`**), which results in the field becoming empty (or defaults).

15.13 Example information from objects

Given the definitions in the examples of 11.10, 11.11 and 12.11, the following constructs (in the left column) are valid, and can be used as equivalent to the expression in the right column.

"ValueFromObject"

`invertMatrix.&operationCode`	**`7`**
`determinantIsZero.&errorCode`	**`1`**

"TypeFromObject"

`invertMatrix.&ArgumentType`	**`Matrix`**

"ValueSetFromObjects"

`invertMatrix.&Errors.&errorCode`	**`{ 1 }`**
`MatrixOperations.&operationCode`	**`{7 |`** *and others* **`}`**

"ObjectSetFromObjects"

`invertMatrix.&Errors`	**`{determinantIsZero}`**
`MatrixOperations.&Errors`	**`{determinantIsZero |`** *and others* **`}`**

Annex A

The TYPE-IDENTIFIER information object class

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

A.1 This annex specifies a useful information object class, with class reference **TYPE-IDENTIFIER**.

NOTE – This information object class is the simplest useful class, having just two fields, an identifier field of type **OBJECT IDENTIFIER**, and a type field which defines the ASN.1 type for carrying all information concerning any particular object in the class. It is defined in this Recommendation | International Standard because of the widespread use of information objects of this form.

A.2 The **TYPE-IDENTIFIER** information object class is defined as:

```
TYPE-IDENTIFIER ::= CLASS
      {
            &id OBJECT IDENTIFIER UNIQUE,
            &Type
      }
      WITH SYNTAX {&Type IDENTIFIED BY &id}
```

A.3 This class is defined as a "useful" information object class, and is available in any module without the necessity for importing it.

A.4 Example

The body of a Message Handling System (MHS) communication can be defined as:

```
MHS-BODY-CLASS ::= TYPE-IDENTIFIER

g4FaxBody MHS-BODY-CLASS ::=
            {BIT STRING IDENTIFIED BY {mhsbody 3}}
```

A protocol designer would typically define a component to carry an **MHS-BODY-CLASS** by specifying the type **INSTANCE OF MHS-BODY-CLASS** defined in C.10.

Annex B

Abstract syntax definitions

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

B.1 This annex specifies a useful information object class, **ABSTRACT-SYNTAX**, for defining abstract syntaxes.

NOTE – It is recommended that an instance of this information object class be defined whenever an abstract syntax is defined as the values of a single ASN.1 type.

B.2 The **ABSTRACT-SYNTAX** information object class is defined as:

```
ABSTRACT-SYNTAX ::= CLASS
      {
            &id         OBJECT IDENTIFIER UNIQUE,
            &Type,
            &property   BIT STRING {handles-invalid-encodings(0)}  DEFAULT {}
      }
      WITH SYNTAX {
            &Type  IDENTIFIED  BY  &id  [HAS  PROPERTY  &property]
      }
```

The **&id** field of each **ABSTRACT-SYNTAX** is the abstract syntax name, while the **&Type** field contains the single ASN.1 type whose values make up the abstract syntax. The property **handles-invalid-encodings** indicates that the invalid encodings are not to be treated as an error during the decoding process, and the decision on how to treat such invalid encodings is left up to the application.

B.3 This information object class is defined as being "useful" because it is of general utility, and is available in any module without the necessity for importing it.

B.4 Example

If an ASN.1 type has been defined called **XXX-PDU**, then an abstract syntax can be specified which contains all the values of **XXX-PDU** by the notation:

```
xxx-Abstract-Syntax ABSTRACT-SYNTAX ::=
            { XXX-PDU IDENTIFIED BY {xxx 5} }
```

See ITU-T Rec. X.680 | ISO/IEC 8824-1, E.3, for a detailed example of use of the **ABSTRACT-SYNTAX** information object class.

B.5 It will frequently be the case that an abstract syntax will be defined in terms of a parameterized type (as defined in ITU-T Rec. X.683 | ISO/IEC 8824-4), for example with parameters providing bounds on some components of the protocol. Such parameters, subject to restrictions specified in ITU-T Rec. X.683 | ISO/IEC 8824-4, clause 10, may be resolved at the time of abstract syntax definition, or may be carried forward as parameters of the abstract syntax.

Annex C

The instance-of type

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

C.1 This annex specifies type and value notation for the instance-of types (see 3.4.14). Such types are capable of carrying any value from any information object in an information object class defined to be of class **TYPE-IDENTIFIER** (see Annex A) using an information object class assignment (the information object class reference is specified as part of this notation).

C.2 The "InstanceOfType" notation is referenced in ITU-T Rec. X.680 | ISO/IEC 8824-1, 16.2, as one of the notations that produce a "Type", and is defined as:

InstanceOfType ::= INSTANCE OF DefinedObjectClass

NOTE – ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 10, specifies the way in which this type can be constrained by applying a "table constraint", restricting the values of the type to those representing some specific information object set of the class.

C.3 This notation specifies a type which carries the **&id** field (an **OBJECT IDENTIFIER**) and a value of the **&Type** field from any instance of the "DefinedObjectClass".

NOTE – This construct will normally be constrained by an object set which will usually be (but is not necessarily) a dummy reference name as defined in ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.3-8.11, with the actual object set defined elsewhere.

C.4 All instance-of types have a tag which is universal class, number 8.

NOTE – This is the same universal tag as for external type, and use of the instance-of type can be bit-compatible with the external type when the basic encoding rules for ASN.1 are in use.

C.5 The instance-of type has an associated sequence type which is used for defining values and subtypes of the instance-of type.

NOTE – Where this type is constrained by the constraint notation of ITU-T Rec. X.682 | ISO/IEC 8824-3, the associated sequence type is also constrained. The constraints on the associated sequence type resulting from a constraint on the instance-of type are specified in ITU-T Rec. X.682 | ISO/IEC 8824-3, Annex A.

C.6 The associated sequence type is assumed to be defined within an environment in which **EXPLICIT TAGS** is in force.

C.7 The associated sequence type shall be:

```
SEQUENCE
      {
            type-id          <DefinedObjectClass>.&id,
            value            [0] <DefinedObjectClass>.&Type
      }
```

where "<DefinedObjectClass>" is replaced by the particular "DefinedObjectClass" used in the "InstanceOfType" notation.

C.8 The value notation "InstanceOfValue" and "XMLInstanceOfValue" for an "InstanceOfType" notation shall be the value notation for the associated sequence type.

InstanceOfValue ::= Value

XMLInstanceOfValue ::= XMLValue

C.9 The XML value notation for sequence-of and set-of (see ITU-T Rec. X.680 | ISO/IEC 8824-1, Table 5) shall be "XMLDelimitedItemList".

C.10 Example

An example, building on the example given in A.4, is as follows:

The type:

```
INSTANCE OF MHS-BODY-CLASS
```

has an associated sequence type of:

```
SEQUENCE
      {
            type-id     MHS-BODY-CLASS.&id,
            value [0]   MHS-BODY-CLASS.&Type
      }
```

An example of the application of a table constraint to this type can be found in ITU-T Rec. X.682 | ISO/IEC 8824-3, Annex A.

Annex D

Examples

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

D.1 Example usage of simplified OPERATION class

Given the following simple definition of the **OPERATION** and **ERROR** information object classes:

```
OPERATION ::= CLASS
      {
            &ArgumentType                OPTIONAL,
            &ResultType                  OPTIONAL,
            &Errors                      ERROR OPTIONAL,
            &Linked                      OPERATION OPTIONAL,
            &resultReturned              BOOLEAN DEFAULT TRUE,
            &operationCode               INTEGER UNIQUE
      }

WITH SYNTAX
      {
            [ARGUMENT                    &ArgumentType]
            [RESULT                      &ResultType]
            [RETURN RESULT               &resultReturned]
            [ERRORS                      &Errors]
            [LINKED                      &Linked]
            CODE                         &operationCode
      }

ERROR ::= CLASS
      {
            &ParameterType               OPTIONAL,
            &errorCode                   INTEGER UNIQUE
      }

WITH SYNTAX
      {
            [PARAMETER                   &ParameterType]
            CODE                         &errorCode
      }
```

We can define the following object set that contains two **OPERATION** objects:

```
My-Operations OPERATION ::= { operationA | operationB }

operationA OPERATION ::= {
            ARGUMENT    INTEGER
            ERRORS      { { PARAMETER INTEGER CODE 1000 } | { CODE 1001 } }
            CODE              1
      }

operationB OPERATION ::= {
            ARGUMENT    IA5String
            RESULT      BOOLEAN
            ERRORS      { { CODE 1002 } | { PARAMETER IA5String CODE 1003 } }
            CODE        2
      }
```

Extraction of the set of the **ERROR** objects from the object set above is done as follows:

```
My-OperationErrors ERROR ::= { My-Operations.&Errors }
```

The resulting object set is:

```
My-OperationErrors ERROR ::= {
            { PARAMETER INTEGER  CODE 1000 } |
            { CODE 1001 } |
            { CODE 1002 } |
            { PARAMETER IA5String  CODE 1003 }
      }
```

Extraction of the set of error codes of the errors of the operations is done as follows:

```
My-OperationErrorCodes INTEGER ::= { My-Operations.&Errors.&errorCode }
```

The resulting value set is:

```
My-OperationErrorCodes INTEGER ::= { 1000 | 1001 | 1002 | 1003 }
```

D.2 Example usage of "ObjectClassFieldType"

The "ObjectClassFieldType" can be used in specification of types, for example:

```
-- "ObjectClassFieldType"s are extracted from this class.
-- Only the first five fields can be used in the extraction.

EXAMPLE-CLASS ::= CLASS {
        &TypeField                  OPTIONAL,
        &fixedTypeValueField        INTEGER   OPTIONAL,
        &variableTypeValueField     &TypeField   OPTIONAL,
        &FixedTypeValueSetField     INTEGER   OPTIONAL,
        &VariableTypeValueSetField  &TypeField   OPTIONAL,
        &objectField                SIMPLE-CLASS   OPTIONAL,
        &ObjectSetField             SIMPLE-CLASS   OPTIONAL
    }

WITH SYNTAX {
        [TYPE-FIELD                 &TypeField]
        [FIXED-TYPE-VALUE-FIELD     &fixedTypeValueField]
        [VARIABLE-TYPE-VALUE-FIELD  &variableTypeValueField]
        [FIXED-TYPE-VALUE-SET-FIELD &FixedTypeValueSetField]
        [VARIABLE-TYPE-VALUE-SET-FIELD&VariableTypeValueSetField]
        [OBJECT-FIELD               &objectField]
        [OBJECT-SET-FIELD           &ObjectSetField]
    }

SIMPLE-CLASS ::= CLASS {
        &value     INTEGER
    }

WITH SYNTAX {
        &value
    }
-- This type contains components which are specified using
-- "ObjectClassFieldType" notation.  In case of type fields and
-- variable-type value and value set fields the resulting
-- component type is an open type. In case of fixed-type value and
-- value set fields the resulting component type is "INTEGER".
-- NOTE – Constraints are omitted from all the following  uses of
-- "ObjectClassFieldType"; you normally will use constraints when
-- referencing an "ObjectClassFieldType".

ExampleType ::= SEQUENCE {
        openTypeComponent1     EXAMPLE-CLASS.&TypeField,
        integerComponent1      EXAMPLE-CLASS.&fixedTypeValueField,
        openTypeComponent2     EXAMPLE-CLASS.&variableTypeValueField,
        integerComponent2      EXAMPLE-CLASS.&FixedTypeValueSetField,
        openTypeComponent3     EXAMPLE-CLASS.&VariableTypeValueSetField
    }

exampleValue ExampleType ::= {
        openTypeComponent1     BOOLEAN : TRUE,
        integerComponent1      123,
        openTypeComponent2     IA5String : "abcdef",
        integerComponent2      456,
        openTypeComponent3     BIT STRING : '0101010101'B
    }
```

D.3 Illustrate usage of objects and object sets

The following uses the object class defined in D.2:

```
objectA EXAMPLE-CLASS ::= {
        FIXED-TYPE-VALUE-FIELD       123
        FIXED-TYPE-VALUE-SET-FIELD   { 1 | 2 | 3 }
        OBJECT-FIELD                 { 1 }
        OBJECT-SET-FIELD             { { 2 } | { 3 } }
    }
```

```
objectB EXAMPLE-CLASS ::= {
          TYPE-FIELD                    IA5String
          FIXED-TYPE-VALUE-FIELD        456
          VARIABLE-TYPE-VALUE-FIELD     "abc"
          VARIABLE-TYPE-VALUE-SET-FIELD{ "d" | "e" | "f" }
     }
-- The following object set contains two defined objects and one
-- builtin object.

ObjectSet EXAMPLE-CLASS ::= {
          objectA |
          objectB |
          {
               TYPE-FIELD             INTEGER
               FIXED-TYPE-VALUE-FIELD 789
               VARIABLE-TYPE-VALUE-SET-FIELD{ 4 | 5 | 6 }
          }
     }
-- The following definitions extract information from the objects and
-- the object set.

integerValue INTEGER ::= objectA.&fixedTypeValueField

stringValue IA5String ::= objectB.&variableTypeValueField

IntegerValueSetFromObjectA INTEGER ::= { objectA.&FixedTypeValueSetField }

StringType ::= objectB.&TypeField

objectFromObjectA SIMPLE-CLASS ::= objectA.&objectField

ObjectSetFromObjectA SIMPLE-CLASS ::= { objectA.&ObjectSetField }

SetOfValuesInObjectSet INTEGER ::= { ObjectSet.&fixedTypeValueField }

SetOfValueSetsInObjectSet INTEGER ::= { ObjectSet.&FixedTypeValueSetField }

SetOfObjectsInObjectSet SIMPLE-CLASS ::= { ObjectSet.&objectField }

SetOfObjectSetsInObjectSet SIMPLE-CLASS ::= { ObjectSet.&ObjectSetField }
```

Annex E

Tutorial annex on the ASN.1 model of object set extension

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

E.1 An ASN.1 specification can define information object sets and such object sets can be marked extensible by means of an extension marker or by the inclusion of extensible object sets using set arithmetic. Use of an extension marker with object sets differs from such use with types in that it specifies that an application can dynamically add/remove objects to/from the object set in an instance of communication.

E.2 Table and component relation constraints which are not satisfied are not in themselves considered errors if the constraining object set is extensible. In such cases, it is not an error if a value of a **UNIQUE** field is not found in the object set, but if it is found, then it is an error if the constraint imposed on the referencing type is not satisfied.

Annex F

Summary of the notation

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

The following lexical items are defined in clause 7:

objectclassreference
objectreference
objectsetreference
typefieldreference
valuefieldreference
valuesetfieldreference
objectfieldreference
objectsetfieldreference
word
CLASS
INSTANCE
SYNTAX
UNIQUE

The following lexical items are defined in ITU-T Rec. X.680 | ISO/IEC 8824-1 and used in this Recommendation | International Standard:

empty
modulereference
xmlasn1typename
"::="
"{"
"}"
","
"."
"["
"]"
":"
DEFAULT
OF
OPTIONAL
WITH

The following productions are defined in ITU-T Rec. X.680 | ISO/IEC 8824-1 and are used in this Recommendation | International Standard:

BuiltinValue
ElementSetSpec
NonParameterizedTypeName
ReferencedValue
Type
Value
ValueSet
XMLBuiltinValue
XMLTypedValue
XMLValue

The following productions are defined in ITU-T Rec. X.683 | ISO/IEC 8824-4 and are used in this Recommendation | International Standard:

ParameterizedObjectClass
ParameterizedObjectSet
ParameterizedObject

The following productions are defined in this Recommendation | International Standard:

DefinedObjectClass ::=
ExternalObjectClassReference | objectclassreference | UsefulObjectClassReference

ExternalObjectClassReference ::= modulereference "." objectclassreference

UsefulObjectClassReference ::=
TYPE-IDENTIFIER |
ABSTRACT-SYNTAX

ObjectClassAssignment ::= objectclassreference "::=" ObjectClass

ObjectClass ::= DefinedObjectClass | ObjectClassDefn | ParameterizedObjectClass

ObjectClassDefn ::= CLASS "{" FieldSpec "," + "}" WithSyntaxSpec?

FieldSpec ::=
TypeFieldSpec |
FixedTypeValueFieldSpec |
VariableTypeValueFieldSpec |
FixedTypeValueSetFieldSpec |
VariableTypeValueSetFieldSpec |
ObjectFieldSpec |
ObjectSetFieldSpec

PrimitiveFieldName ::=
typefieldreference |
valuefieldreference |
valuesetfieldreference |
objectfieldreference |
objectsetfieldreference

FieldName ::= PrimitiveFieldName "." +

TypeFieldSpec ::= typefieldreference TypeOptionalitySpec?

TypeOptionalitySpec ::= `OPTIONAL` | `DEFAULT` Type

FixedTypeValueFieldSpec ::= valuefieldreference Type `UNIQUE` ? ValueOptionalitySpec ?

ValueOptionalitySpec ::= `OPTIONAL` | `DEFAULT` Value

VariableTypeValueFieldSpec ::= valuefieldreference FieldName ValueOptionalitySpec ?

FixedTypeValueSetFieldSpec ::= valuesetfieldreference Type ValueSetOptionalitySpec ?

ValueSetOptionalitySpec ::= `OPTIONAL` | `DEFAULT` ValueSet

VariableTypeValueSetFieldSpec ::= valuesetfieldreference FieldName ValueSetOptionalitySpec?

ObjectFieldSpec ::= objectfieldreference DefinedObjectClass ObjectOptionalitySpec?

ObjectOptionalitySpec ::= `OPTIONAL` | `DEFAULT` Object

ObjectSetFieldSpec ::= objectsetfieldreference DefinedObjectClass ObjectSetOptionalitySpec ?

ObjectSetOptionalitySpec ::= `OPTIONAL` | `DEFAULT` ObjectSet

WithSyntaxSpec ::= `WITH SYNTAX` SyntaxList

SyntaxList ::= "`{`" TokenOrGroupSpec empty + "`}`"

TokenOrGroupSpec ::= RequiredToken | OptionalGroup

OptionalGroup ::= "[" TokenOrGroupSpec empty + "]"

RequiredToken ::= Literal | PrimitiveFieldName

Literal ::= word | ","

DefinedObject ::= ExternalObjectReference | objectreference

ExternalObjectReference ::= modulereference "." objectreference

ObjectAssignment ::= objectreference DefinedObjectClass ":=" Object

Object ::= DefinedObject | ObjectDefn | ObjectFromObject | ParameterizedObject

ObjectDefn ::= DefaultSyntax | DefinedSyntax

DefaultSyntax ::= "`{`" FieldSetting "," * "`}`"

FieldSetting ::= PrimitiveFieldName Setting

DefinedSyntax ::= "`{`" DefinedSyntaxToken empty * "`}`"

DefinedSyntaxToken ::= Literal | Setting

Setting ::= Type | Value | ValueSet | Object | ObjectSet

DefinedObjectSet ::= ExternalObjectSetReference | objectsetreference

ExternalObjectSetReference ::= modulereference "." objectsetreference

ObjectSetAssignment ::= objectsetreference DefinedObjectClass "::=" ObjectSet

ObjectSet ::= "`{`" ObjectSetSpec "`}`"

ObjectSetSpec ::=

```
RootElementSetSpec                          |
RootElementSetSpec "," "..."                |
"..."                                       |
"..." "," AdditionalElementSetSpec          |
RootElementSetSpec "," "..." "," AdditionalElementSetSpec
```

ObjectSetElements ::=

Object | DefinedObjectSet | ObjectSetFromObjects | ParameterizedObjectSet

ObjectClassFieldType ::= DefinedObjectClass "." FieldName

ObjectClassFieldValue ::= OpenTypeFieldVal | FixedTypeFieldVal

OpenTypeFieldVal ::= Type ":" Value

FixedTypeFieldVal ::= BuiltinValue | ReferencedValue

XMLObjectClassFieldValue ::=
XMLOpenTypeFieldVal |
XMLFixedTypeFieldVal

XMLOpenTypeFieldVal ::= XMLTypedValue

XMLFixedTypeFieldVal ::= XMLBuiltinValue

InformationFromObjects ::= ValueFromObject | ValueSetFromObjects |
TypeFromObject | ObjectFromObject | ObjectSetFromObjects

ReferencedObjects ::=
DefinedObject | ParameterizedObject |
DefinedObjectSet | ParameterizedObjectSet

ValueFromObject ::= ReferencedObjects "." FieldName

ValueSetFromObjects ::= ReferencedObjects "." FieldName

TypeFromObject ::= ReferencedObjects "." FieldName

ObjectFromObject ::= ReferencedObjects "." FieldName

ObjectSetFromObjects ::= ReferencedObjects "." FieldName

InstanceOfType ::= `INSTANCE OF` DefinedObjectClass

InstanceOfValue ::= Value

XMLInstanceOfValue ::= XMLValue

ITU-T Recommendation X.682
International Standard 8824-3

Information Technology –
Abstract Syntax Notation One (ASN.1):
Constraint specification

INTERNATIONAL STANDARD 8824-3

ITU-T RECOMMENDATION X.682

Information Technology – Abstract Syntax Notation One (ASN.1): Constraint Specification

Summary

This Recommendation | International Standard provides the ASN.1 notation for the general case of constraint and exception specification by which the data values of a structured data type can be limited. The notation also provides for signalling if and when a constraint is violated.

Source

The ITU-T Recommendation X.682 was approved on the 13th of July 2002. The identical text is also published as ISO/IEC International Standard 8824-3.

CONTENTS

		Page
Introduction		iii
1	Scope	1
2	Normative references	1
	2.1 Identical Recommendations | International Standards	1
3	Definitions	1
	3.1 Specification of basic notation	1
	3.2 Information object specification	1
	3.3 Parameterization of ASN.1 specification	1
	3.4 Additional definitions	1
4	Abbreviations	2
5	Convention	2
6	Notation	2
	6.1 Constraint	2
7	ASN.1 lexical items	2
	7.1 Additional keyword items	2
8	General constraint specification	3
9	User-defined constraints	3
10	Table constraints, including component relation constraints	4
11	Contents Constraints	7
Annex A Constraining instance-of types		8
Annex B Summary of the notation		9

Introduction

Application designers require a notation to define a structured data type to convey their semantics. This is provided in ITU-T Rec. X.680 | ISO/IEC 8824-1 and ITU-T Rec. X.681 | ISO/IEC 8824-2. A notation is also required to further constrain the values that can appear. Examples of such constraints are restricting the range of some component(s), or using a specified information object set to constrain an "ObjectClassFieldType" component, or using the "AtNotation" to specify a relation between components.

This Recommendation | International Standard provides the notation for the general case of constraint specification.

NOTE 1 – For historical reasons the special case of a "subtype constraint" is specified in ITU-T Rec. X.680 | ISO/IEC 8824-1.

Constraint notation can appear (in round brackets) after any use of the syntactic construct "Type", and the purpose of this Recommendation | International Standard is to specify the general case of what goes in the round brackets.

NOTE 2 – Multiple constraints (each inside its own round brackets) can be applied to the same "Type", as the result of constraining a "Type" is itself formally a "Type" construct.

When a constraint is applied to the textually outermost use of a "Type" construct, it results in the creation of a new type which is a subtype of the original (parent) type.

A subtype of a parent type can itself be used in defining other subtypes of the same parent type in other uses of the constraint notation. Thus the subset of values constituting a subtype can be defined either by limiting the range of the parent type, or by specifying the subtype as a union of sets of values.

NOTE 3 – The "ValueSet" notation specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 15.7, provides a further means of specifying a subtype.

Constraints may also be used to produce a subtype of a parent type (as described above) when the notation is embedded within another type. However, some "component relation" constraints are textually included following a "Type" (within a set or sequence type definition), but are not used to restrict the set of possible values of the "Type" which they follow (the referencing component). Rather, they specify a relation between the value of the referencing component and the value of one or more other "Type"s in the same set or sequence type (called the referenced components).

Component relation constraints can be seen as subtyping the sequence type within which they are embedded, but not necessarily the referencing type.

A constraint on an "ObjectClassFieldType" component can be applied by restricting the type or values in the component by using an information object set. Such constraints are called table constraints, since they are specified in terms of the "associated table" of the object set. The component relation constraints defined in this Recommendation | International Standard are a special case of table constraints.

Finally, a "Type" may be subtyped by specifying the set of values in the subtype by human-readable text. Such a constraint is called a user-defined constraint. For example, a user-defined constraint can be specified to constrain a **BIT STRING** to the set of values produced by the encryption of a value of a specified ASN.1 type.

It is the purpose of this Recommendation | International Standard to provide the notation to be used for specifying table constraints (including component relation constraints), and user-defined constraints.

NOTE 4 – In general, full support for the specification of constraints in a flexible way (particularly component relation constraints, subtyping constraints, and user-defined constraints with a formally defined body) would require notation with a power comparable to that of programming languages. Such power can only be sensibly provided by the establishment of links from the ASN.1 notation into some other defined computer language. This version of this Recommendation | International Standard does not provide such links, and hence supports only a small number of constraining mechanisms.

While the embedding of notation defining constraints (subtypes and relationships) will frequently be the most convenient form of specification (particularly for the simple subtyping of primitive components of structures), separate (external) specification will sometimes be preferred, particularly where the constraints are being imposed by a separate group from that which defined the basic protocol.

NOTE 5 – The parameterization defined in ITU-T Rec. X.683 | ISO/IEC 8824-4 is specifically designed to enable a piece of ASN.1 specification (and in particular, a constraint) to be parameterized, allowing the actual constraint to be imposed by some other group that provides actual parameters for the parameterized construct.

The notations for constraint specification supported here are:

- user-defined constraints (see clause 9);
- table constraints, including component relation constraints between two components which are carrying values related to an information object, defined using the notation of ITU-T Rec. X.681 | ISO/IEC 8824-2 (see clause 10 of this Recommendation | International Standard);

– contents constraints (see clause 11).

The application of table constraints to the "InstanceOfType" construct of ITU-T Rec. X.681 | ISO/IEC 8824-2, Annex C, is specified in Annex A of this Recommendation | International Standard.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

Information Technology – Abstract Syntax Notation One (ASN.1): Constraint Specification

1 Scope

This Recommendation | International Standard is part of Abstract Syntax Notation One (ASN.1) and provides notation for specifying user-defined constraints, table constraints, and contents constraints.

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 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.

2.1 Identical Recommendations | International Standards

- ITU-T Recommendation X.680 (2002) | ISO/IEC 8824-1:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.*
- ITU-T Recommendation X.681 (2002) | ISO/IEC 8824-2:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.*
- ITU-T Recommendation X.683 (2002) | ISO/IEC 8824-4:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications.*

3 Definitions

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

3.1 Specification of basic notation

This Recommendation | International Standard uses the terms defined in ITU-T Rec. X.680 | ISO/IEC 8824-1.

3.2 Information object specification

This Recommendation | International Standard uses the terms defined in ITU-T Rec. X.681 | ISO/IEC 8824-2.

3.3 Parameterization of ASN.1 specification

This Recommendation | International Standard uses the following term defined in ITU-T Rec. X.683 | ISO/IEC 8824-4:

- parameterized type.

3.4 Additional definitions

3.4.1 component relation constraint: A constraint on the values of a set type or sequence type which is textually associated with one of the component types (the referencing component) of the set type or sequence type, and which

specifies the relationship between the value of that component and the values of one or more other components (the referenced components).

3.4.2 constrained type: The innermost "Type" which contains the referencing component and all of the referenced components of some component relation constraint.

3.4.3 constraining set: The information object set referenced in some component relation constraint.

3.4.4 constraining table: The associated table (see ITU-T Rec. X.681 | ISO/IEC 8824-2, clause 13) corresponding to a constraining set.

3.4.5 referenced component: A component of a set type or sequence type identified in a component relation constraint.

3.4.6 referencing component: A component of a set type or sequence type which has an associated component relation constraint.

3.4.7 selected rows: Those rows of a constraining table which contain, in the appropriate columns, the values of all of the referenced components.

3.4.8 table constraint: A constraint applied to an object class field type which demands that its values conform to the contents of the appropriate column of some table.

3.4.9 user-defined constraint: A constraint which requires a more complicated statement than can be accommodated by the other forms of constraint, and which must therefore involve specification by some means outside of ASN.1.

4 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviation applies:

ASN.1 Abstract Syntax Notation One

5 Convention

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

6 Notation

This clause summarizes the notation defined in this Recommendation | International Standard.

6.1 Constraint

The following notation which can be used as an alternative for "ConstraintSpec" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 45.6) is defined in this Recommendation | International Standard:

– GeneralConstraint (see 8.1).

7 ASN.1 lexical items

In addition to the lexical items specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 11, this Recommendation | International Standard makes use of the lexical items specified in the following subclauses. The general rules applicable to these lexical items are as defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.1. These new lexical items make use of the ASN.1 character set, as specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 10.

7.1 Additional keywords

The names **`CONSTRAINED`**, **`CONTAINING`**, **`ENCODED`** and **`BY`** are listed in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.27 as reserved words.

8 General constraint specification

8.1 The notation for a "GeneralConstraint" is as follows:

```
GeneralConstraint ::=
    UserDefinedConstraint     |
    TableConstraint           |
    ContentsConstraint
```

8.2 The various possibilities for specification of the constraint are defined as follows:

a) "UserDefinedConstraint", in clause 9;

b) "TableConstraint", in clause 10;

c) "ContentsConstraint", in clause 11.

9 User-defined constraints

NOTE 1 – This form of constraint specification can be regarded as a special form of ASN.1 comment, since it is not fully machine-processable. However, it would be possible for an automatic tool to use the presence of a particular user-defined constraint to invoke user-supplied constraint checking.

NOTE 2 – Protocol designers should be aware that since the definition of a constraint in this way is not fully machine-processable, a specification which employs this capability may be less easy to handle with automatic tools.

9.1 A user-defined constraint is specified by the syntax:

```
UserDefinedConstraint ::=
    CONSTRAINED BY "{" UserDefinedConstraintParameter "," * "}"
```

9.2 It is recommended that the actual constraint be referenced by a comment anywhere inside the braces ("**{**" and "**}**"). This comment should clearly state what constraint is imposed by the "UserDefinedConstraint".

NOTE – If there are any "UserDefinedConstraintParameter"s within the braces (see 9.3), the comments may precede, follow, or be interspersed among them, at the definer's convenience.

9.3 The actual constraint to be applied may depend on some parameters. For each such parameter, a "UserDefinedConstraintParameter" shall be included in the "UserDefinedConstraint". Each "UserDefinedConstraintParameter" shall be any "Value", "ValueSet", "Object", "ObjectSet", "Type" or "DefinedObjectClass" which is defined inline or is a reference name.

NOTE – The reference name may be a dummy parameter if the "UserDefinedConstraint" is used within a "ParameterizedAssignment".

```
UserDefinedConstraintParameter ::=
    Governor ":" Value        |
    Governor ":" ValueSet     |
    Governor ":" Object       |
    Governor ":" ObjectSet    |
    Type                      |
    DefinedObjectClass
```

The notation "Governor" is defined in ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.3. When the first or second alternatives are used, the "Governor" shall be a "Type". When the third or fourth alternatives are used, the "Governor" shall be a "DefinedObjectClass".

9.4 Example

If an application designer wishes to specify that certain components are to be bit strings carrying an encryption of the value of some ASN.1 type (different for each component), then (using the parameterization of ITU-T Rec. X.683 | ISO/IEC 8824-4) the parameterized **ENCRYPTED** type can be defined as follows:

```
ENCRYPTED {ToBeEnciphered} ::= BIT STRING
      (CONSTRAINED BY
          {-- must be the result of the encipherment of some BER-encoded
           -- value of -- ToBeEnciphered}
      ! Error : securityViolation)
```

```
Error ::= ENUMERATED {securityViolation}
```

and a use of the **ENCRYPTED** parameterized subtype of **BIT STRING** (which is what the **ENCRYPTED** type is) becomes simply:

```
ENCRYPTED{SecurityParameters}
```

or, equivalently, at the whim of the designer:

```
BIT STRING (ENCRYPTED{SecurityParameters})
```

The occurrence of a **securityViolation** is handled according to local security policy.

10 Table constraints, including component relation constraints

NOTE 1 – Information object classes, information objects, information object sets, and the object class field type are defined in ITU-T Rec. X.681 | ISO/IEC 8824-2. An understanding of these concepts is assumed in this clause.

NOTE 2 – This clause describes the application of the table constraint using an information object set that is identified within the main notation defining the parent type, in other words, defined and identified by the protocol designer. This does not satisfy the requirement for the actual information object set which is to be used as the constraint in particular abstract syntaxes to vary from syntax to syntax. ITU-T Rec. X.683 | ISO/IEC 8824-4 provides notation which, among other things, enables the information object set used in this constraint to be a parameter whose value is supplied at a later date by varying groups.

Example

For the purpose of illustrating the text of this clause, the following example will be used. An **ErrorReturn** type carries an **errorCategory** and one or more **errorCodes** with corresponding **errorInfo** from that category. This is supported by an **ERROR-CLASS** information object class with a specific set of objects defined in the information object set **ErrorSet** that is used to constrain the fields of **ErrorReturn**.

We have:

```
ERROR-CLASS ::= CLASS
      {
      &category  PrintableString (SIZE(1)),
      &code INTEGER,
      &Type
      }
      WITH SYNTAX {&category &code &Type}

ErrorSet ERROR-CLASS ::=
      {
      {"A" 1 INTEGER} |
      {"A" 2 REAL} |
      {"B" 1 CHARACTER STRING} |
      {"B" 2 GeneralString}
      }

ErrorReturn ::= SEQUENCE
      {
        errorCategory        ERROR-CLASS.&category ({ErrorSet}) OPTIONAL,
        errors               SEQUENCE OF SEQUENCE
        {
          errorCode  ERROR-CLASS.&code ({ErrorSet}{@errorCategory}),
          errorInfo  ERROR-CLASS.&Type ({ErrorSet}{@errorCategory,@.errorCode})
        } OPTIONAL
      }
```

The associated table of **ErrorSet** can be depicted as follows:

&category	&code	&Type
"A"	1	INTEGER
"A"	2	REAL
"B"	1	CHARACTER STRING
"B"	2	GeneralString

10.1 A table constraint can only be applied to types "ObjectClassFieldType" or an "InstanceOfType". The former case is defined in the remainder of this clause, the latter in Annex A.

10.2 An "ObjectClassFieldType" identifies an information object class, and one of the permissible "FieldName"s of that class. The table constraint identifies the set of information objects whose associated table (as defined in ITU-T Rec. X.681 | ISO/IEC 8824-2, clause 13) determines the set of constrained values.

10.3 The "TableConstraint" notation is:

TableConstraint ::=
SimpleTableConstraint |
ComponentRelationConstraint

SimpleTableConstraint ::= ObjectSet

10.4 The "ObjectSet" in the "SimpleTableConstraint" is governed by the class which appears in the "ObjectClassFieldType" being constrained.

10.5 The semantics of the "SimpleTableConstraint" are specified using the associated table of the constraining information object set.

10.6 The "FieldName" of the type being constrained is used to select the applicable column of the associated table, and the following rules then apply:

a) for a type field, the component is constrained to be any value of any one of the types in any of the rows of that column;

b) for a value field, the component is constrained to be any one of the values in any of the rows of that column;

c) for a value set field, the component is constrained to be any one of the values in the value set in any one of the rows of that column.

NOTE – If, for some given object set, the above algorithms deliver no permissible value, then the constraint is always violated if that component is present in a value of a containing type.

Example

In the example in clause 10, case b) applies to the component **`errorCategory`**:

```
errorCategory    ERROR-CLASS.&category ({ErrorSet}) OPTIONAL,
```

with the associated table of **`ErrorSet`** restricting its possible values to **`"A"`** and **`"B"`**.

10.7 A component relation constraint is applied using the associated table of an information object set and the following production:

ComponentRelationConstraint ::=
`"{"` DefinedObjectSet `"}"` `"{"` AtNotation `","` + `"}"`

AtNotation ::=
`"@"` ComponentIdList |
`"@."` Level ComponentIdList

Level ::= `"."` Level | empty

ComponentIdList ::= identifier `"."` +

10.8 Each "identifier" in the "ComponentIdList" identifies a component whose parent is a set, sequence or choice type, and shall be the last "identifier" if the component it identifies is not a set, sequence or choice type.

10.9 In the case where the parent is a set or sequence type the "identifier" shall be one of the "identifier"s of the "NamedType" in the "ComponentTypeLists" of that parent. In the case where the parent is a choice type the "identifier" shall be one of the "identifier"s of a "NamedType" alternative in the "AlternativeTypeLists" of that choice type.

10.10 The "AtNotation" provides a pointer to other components of the ASN.1 structure in which it appears. The parent structure for the first "identifier" in the "ComponentIdList" is determined as follows:

a) if the first alternative of "AtNotation" is selected (there is no `"."` following the `"@"`), then the parent structure is the outermost textually enclosing set type, sequence type or choice type.

b) if the second alternative is selected (there is a `"."` following the `"@"`), then the parent structure is obtained by moving upwards from the innermost textually enclosing set type or sequence type by a number of levels (set, set-of, sequence, sequence-of, choice) equal to the number of additional `"."` following `"@."`.

The number of additional "**.**" shall not exceed the number of constructions (set, set-of, sequence, sequence-of, choice) containing the innermost set or sequence type where the "AtNotation" occurs.

NOTE – The "AtNotation" is only permitted when it is textually within a set type or sequence type, and references some other field which is textually within the same set or sequence type, though possibly at a different level of nesting in constructions involving combination of sequence, sequence-of, set, set-of and choice types.

Example

In the following example "**@...**" illustrates case b) above:

```
ErrorMessage     ::=   SEQUENCE    {
            severity          ERROR.&severity({Errors}),
            parameters  SEQUENCE OF SEQUENCE{
                  errorId     ERROR.&id({Errors}),
                  data        SEQUENCE OF SEQUENCE{
                        value ERROR.&Type({Errors}{@severity,@...errorId}),
                        text  VisibleString}}}
```

10.11 The component where this notation is used is the referencing component, and the components identified by the "AtNotation"s are the referenced components.

10.12 The "ObjectSet" (see 10.3) or the "DefinedObjectSet" (see 10.7) is the constraining set, and the associated table derived from it (as specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, clause 13) is the constraining table.

10.13 The component relation constraint can only be applied to an ASN.1 type which is textually within an enclosing "Type" (the constrained type) which textually contains all the referenced components. The constrained type is defined to be the innermost "Type" which satisfies the above condition.

Example

In the example in clause 10, the constrained type is **ErrorReturn**.

NOTE – In some respects it is possible to regard the application of this constraint as using the values of the referenced components to identify a row in the constraining table, and then using the value of the appropriate column to constrain the referencing component. With this view, the referenced components themselves could not be regarded as constrained.

However, the approach taken below is slightly different. It regards the constraint as operating over all possible values of the constrained type (which, as explained above, is not that of the referencing component), and selecting some of those values as satisfying the constraint. This approach makes it possible to discuss questions about values of the constrained type that do not contain values of either the referencing component or of one or more of the referenced components (because they were optional, or in choices), and values of the constrained type in which one of the referenced components has a value which does not correspond to any row in the constraining table.

10.14 The referencing and all the referenced components are required to be "ObjectClassFieldType"s referencing the same class. The constraining set is required to be an information object set of this class. The referenced components are required to be value fields or value set fields constrained by the same object set as the referencing component.

Example

In the example in clause 10, the "ObjectClassFieldType"s are all of class **ERROR-CLASS**, as is the constraining set, which is **ErrorSet**.

10.15 The following paragraphs determine the set of values of the constrained type that satisfy this constraint.

10.16 If the referencing component is absent in a value of the constrained type, that value always satisfies the constraint.

Example

In the example in clause 10, if the component **errors** is missing, then the constraints on **errors** are satisfied.

10.17 If any referenced component is absent in a value of the constrained type, that value does not satisfy the constraint unless the referencing component is also absent, in which case the constraint is always satisfied.

10.18 If all referenced components are present and the referencing component is present, then the constraint is not satisfied unless there exists in the constraining table one or more selected rows such that, for each selected row:

a) every referenced component which is a value field has a value that is the value of the corresponding column of the selected row;

b) every referenced component which is a value set field has a value that is one of the values in the value set of the corresponding column of the selected row.

10.19 The constraint is then satisfied if and only if the referencing component satisfies a simple table constraint (as defined above) obtained by applying a table containing only the selected rows to the referencing component.

Example

In the example in clause 10, the components **errorCategory**, **errorCode**, and **errorInfo** have to correspond to one of the rows of the associated table of **ErrorSet**.

10.20 If an "ObjectClassFieldType" is constrained by means of one or more "TableConstraint"s, and the "FieldName" denotes a type field, a variable-type value field, or a variable-type value set field, then in each instance of communication, the number of selected rows shall be exactly one if one of the referenced components is an identifier field, otherwise at least one shall be selected.

Example

In the example in clause 10, if there had been a further object **{"B" 2 PrintableString}**, then there could be more than one selected row.

11 Contents Constraints

11.1 A contents constraint is specified by the syntax:

ContentsConstraint ::=
CONTAINING Type |
ENCODED BY Value |
CONTAINING Type ENCODED BY Value

11.2 "Value" shall be a value of type object identifier.

11.3 The "ContentsConstraint" shall only be applied to octet string types and to bit string types defined without a "NamedBitList". Such constrained types shall not have further constraints applied to them, either directly or through the use of "typereference" names.

11.4 The first production of "ContentsConstraint" specifies that the abstract value of the octet string or bit string is the encoding of an (any) abstract value of "Type" that is produced by the encoding rules that are applied to the octet string or bit string. The following restrictions apply:

a) If this constraint is applied to an octet string, it is a specification error if any encoding of an abstract value of "Type" is not a multiple of eight bits.

b) If the octet string or bit string has a length constraint, the abstract values of "Type" are constrained to be those whose encoding can be contained within the constrained octet string or bit string. It is a specification error if there are no such abstract values.

11.5 The second production of "ContentsConstraint" specifies that the procedures identified by the object identifier value "Value" shall be used to produce and to interpret the contents of the bit string or octet string. If the bit string or octet string is already constrained, it is a specification error if these procedures do not produce encodings that satisfy the constraint.

11.6 The third production of "ContentsConstraint" specifies that the abstract value of the octet string or bit string is the encoding of an (any) abstract value of "Type" that is produced by the encoding rules identified by the object identifier value "Value". The following restrictions apply:

a) If this constraint is applied to an octet string, it is a specification error if any encoding of an abstract value of "Type" is not a multiple of eight bits.

b) If the octet string or bit string has a length constraint, the abstract values of "Type" are constrained to be those whose encoding can be contained within the constrained octet string or bit string. It is a specification error if there are no such abstract values.

Annex A

Constraining instance-of types

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

A.1 This annex specifies the application of constraints to "InstanceOfType" as defined in ITU-T Rec. X.681 | ISO/IEC 8824-2, Annex C.

A.2 The only constraint that can be applied to such a type is the simple table constraint, as specified in clause 10. The equivalent sequence type of the "InstanceOfType", when constrained in this way is:

```
SEQUENCE
      {
      type-id   <DefinedObjectClass>.&id({  <DefinedObjectSet>  }),
      value [0] <DefinedObjectClass>.&Type({  <DefinedObjectSet>  }{@.type-id})
      }
```

where "<DefinedObjectClass>" is replaced by the particular "DefinedObjectClass" used in the "InstanceOfType" notation and "<DefinedObjectSet>" by the particular "DefinedObjectSet" used in the simple table constraint.

A.3 Where multiple constraints are applied to the instance-of type, each one produces a constraint of the above form, so that there are multiple constraints applied to each element of the equivalent sequence type.

A.4 Example

An example, building on the example in ITU-T Rec. X.681 | ISO/IEC 8824-2, C.10, is as follows.

The type:

```
INSTANCE OF MHS-BODY-CLASS ({PossibleBodyTypes})
```

has an equivalent sequence type of:

```
[UNIVERSAL 8] IMPLICIT SEQUENCE
      {
      type-id    MHS-BODY-CLASS.&id ({PossibleBodyTypes}),
      value [0] MHS-BODY-CLASS.&Type ({PossibleBodyTypes}{@.type-id})
      }
```

Here the **type-id** component of the sequence is constrained to take the value of the **&id** field of one of the **PossibleBodyTypes**, while the **value** component is constrained to be any value of the **&Type** field of that same information object.

In this case, the **PossibleBodyTypes** would likely be a parameter of the specification (see ITU-T Rec. X.683 | ISO/IEC 8824-4, clause 10 and A.8) which might not be resolved until a Protocol Implementation Conformance Statement (PICS) is produced, making the above constraints variable constraints as defined in ITU-T Rec. X.683 | ISO/IEC 8824-4, 10.3.

Annex B

Summary of the notation

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

The following lexical items are defined in 7.1:

CONSTRAINED
CONTAINING
ENCODED
BY

The following lexical items are defined in ITU-T Rec. X.680 | ISO/IEC 8824-1 and used in this Recommendation | International Standard:

modulereference
number
"::="
"{"
"}"
","
"."

The following productions are defined in ITU-T Rec. X.680 | ISO/IEC 8824-1 and are used in this Recommendation | International Standard:

Type
Value
ValueSet

The following productions are defined in ITU-T Rec. X.681 | ISO/IEC 8824-2 and are used in this Recommendation | International Standard:

DefinedObjectClass
DefinedObjectSet
Object
ObjectSet

The following production is defined in ITU-T Rec. X.683 | ISO/IEC 8824-4 and is used in this Recommendation | International Standard:

Governor

The following productions are defined in this Recommendation | International Standard:

GeneralConstraint ::= UserDefinedConstraint | TableConstraint | ContentsConstraint

UserDefinedConstraint ::= CONSTRAINED BY "{" UserDefinedConstraintParameter "," * "}"

UserDefinedConstraintParameter ::=
Governor ":" Value |
Governor ":" ValueSet |
Governor ":" Object |
Governor ":" ObjectSet |
Type |
DefinedObjectClass

TableConstraint ::= SimpleTableConstraint | ComponentRelationConstraint

SimpleTableConstraint ::= ObjectSet

ComponentRelationConstraint ::= "{" DefinedObjectSet "}" "{" AtNotation "," + "}"

AtNotation ::= "@" ComponentIdList | "@." Level ComponentIdList

Level ::= " . " Level | empty

ComponentIdList ::= identifier " . " +

ContentsConstraint ::=
CONTAINING Type |
ENCODED BY Value |
CONTAINING Type ENCODED BY Value

ITU-T Recommendation X.683
International Standard 8824-4

Information Technology –
Abstract Syntax Notation One (ASN.1):
Parameterization of ASN.1 specifications

INTERNATIONAL STANDARD 8824-4

ITU-T RECOMMENDATION X.683

Information Technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications

Summary

This Recommendation | International Standard defines the provisions for parameterized reference names and parameterized assignments for data types which are useful for the designer when writing specifications where some aspects are left undefined at certain stages of the development to be filled in at a later stage to produce a complete definition of an abstract syntax.

Source

The ITU-T Recommendation X.683 was approved on the 13th of July 2002. The identical text is also published as ISO/IEC International Standard 8824-4.

CONTENTS

Page

Introduction iii

1 Scope 1

2 Normative references 1

2.1 Identical Recommendations | International Standards 1

3 Definitions 1

3.1 Specification of basic notation 1

3.2 Information object specification 1

3.3 Constraint specification 1

3.4 Additional definitions 1

4 Abbreviations 2

5 Convention 2

6 Notation 2

6.1 Assignments 2

6.2 Parameterized definitions 2

6.3 Symbols 3

7 ASN.1 lexical items 3

8 Parameterized assignments 3

9 Referencing parameterized definitions 5

10 Abstract syntax parameters 8

Annex A Examples 9

A.1 Example of the use of a parameterized type definition 9

A.2 Example of use of parameterized definitions together with an information object class 9

A.3 Example of parameterized type definition that is finite 10

A.4 Example of a parameterized value definition 11

A.5 Example of a parameterized value set definition 11

A.6 Example of a parameterized class definition 11

A.7 Example of a parameterized object set definition 12

A.8 Example of a parameterized object set definition 12

Annex B Summary of the notation 13

Introduction

Application designers need to write specifications in which certain aspects are left undefined. Those aspects will later be defined by one or more other groups (each in its own way), to produce a fully defined specification for use in the definition of an abstract syntax (one for each group).

In some cases, aspects of the specification (for example, bounds) may be left undefined even at the time of abstract syntax definition, being completed by the specification of International Standardized Profiles or functional profiles from some other body.

NOTE 1 – It is a requirement imposed by this Recommendation | International Standard that any aspect that is not solely concerned with the application of constraints has to be completed prior to the definition of an abstract syntax.

In the extreme case, some aspects of the specification may be left for the implementor to complete, and would then be specified as part of the Protocol Implementation Conformance Statement.

While the provisions of ITU-T Rec. X.681 | ISO/IEC 8824-2 and ITU-T Rec. X.682 | ISO/IEC 8824-3 provide a framework for the later completion of parts of a specification, they do not of themselves solve the above requirements.

Additionally, a single designer sometimes requires to define many types, or many information object classes, or many information object sets, or many information objects, or many values, which have the same outer level structure, but differ in the types, or information object classes, or information object sets, or information objects, or values, that are used at an inner level. Instead of writing out the outer level structure for every such occurrence, it is useful to be able to write it out once, with parts left to be defined later, then to refer to it and provide the additional information.

All these requirements are met by the provision for parameterized reference names and parameterized assignments by this Recommendation | International Standard.

The syntactic form of a parameterized reference name is the same as that of the corresponding normal reference name, but the following additional considerations apply:

- When it is assigned in a parameterized assignment statement, it is followed by a list of dummy reference names in braces, each possibly accompanied by a governor; these reference names have a scope which is the right-hand side of the assignment statement, and the parameter list itself.

  NOTE 2 – This is what causes it to be recognized as a parameterized reference name.

- When it is exported or imported, it is followed by a pair of empty braces to distinguish it as a parameterized reference name.
- When it is used in any construct, it is followed by a list of syntactic constructions, one for each dummy reference name, that provide an assignment to the dummy reference name for the purposes of that use only.

Dummy reference names have the same syntactic form as the corresponding normal reference name, and can be used anywhere on the right-hand side of the assignment statement that the corresponding normal reference name could be used. All such usages are required to be consistent.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

Information Technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications

1 Scope

This Recommendation | International Standard is part of Abstract Syntax Notation One (ASN.1) and defines notation for parameterization of ASN.1 specifications.

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 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.

2.1 Identical Recommendations | International Standards

- ITU-T Recommendation X.680 (2002) | ISO/IEC 8824-1:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.*
- ITU-T Recommendation X.681 (2002) | ISO/IEC 8824-2:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.*
- ITU-T Recommendation X.682 (2002) | ISO/IEC 8824-3:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*

3 Definitions

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

3.1 Specification of basic notation

This Recommendation | International Standard uses the terms defined in ITU-T Rec. X.680 | ISO/IEC 8824-1.

3.2 Information object specification

This Recommendation | International Standard uses the terms defined in ITU-T Rec. X.681 | ISO/IEC 8824-2.

3.3 Constraint specification

This Recommendation | International Standard uses the terms defined in ITU-T Rec. X.682 | ISO/IEC 8824-3.

3.4 Additional definitions

3.4.1 **normal reference name**: A reference name defined, without parameters, by means of an "Assignment" other than a "ParameterizedAssignment". Such a name references a complete definition and is not supplied with actual parameters when used.

3.4.2 **parameterized reference name**: A reference name defined using a parameterized assignment, which references an incomplete definition and which, therefore, must be supplied with actual parameters when used.

3.4.3 parameterized type: A type defined using a parameterized type assignment and thus whose components are incomplete definitions which must be supplied with actual parameters when the type is used.

3.4.4 parameterized value: A value defined using a parameterized value assignment and thus whose value is incompletely specified and must be supplied with actual parameters when used.

3.4.5 parameterized value set: A value set defined using a parameterized value set assignment and thus whose values are incompletely specified and must be supplied with actual parameters when used.

3.4.6 parameterized object class: An information object class defined using a parameterized object class assignment and thus whose field specifications are incompletely specified and must be supplied with actual parameters when used.

3.4.7 parameterized object: An information object defined using a parameterized object assignment and thus whose components are incompletely specified and must be supplied with actual parameters when used.

3.4.8 parameterized object set: An information object set defined using a parameterized object set assignment and thus whose objects are incompletely specified and must be supplied with actual parameters when used.

3.4.9 variable constraint: A constraint employed in specifying a parameterized abstract syntax, and which depends on some parameter of the abstract syntax.

4 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviation applies:

ASN.1 Abstract Syntax Notation One

5 Convention

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

6 Notation

This clause summarizes the notation defined in this Recommendation | International Standard.

6.1 Assignments

The following notation which can be used as an alternative for "Assignment" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 12) is defined in this Recommendation | International Standard:

– ParameterizedAssignment (see 8.1).

6.2 Parameterized definitions

6.2.1 The following notation which can be used as an alternative for "DefinedType" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 13.1) is defined in this Recommendation | International Standard:

– ParameterizedType (see 9.2).

6.2.2 The following notation which can be used as an alternative for "DefinedValue" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 13.1) is defined in this Recommendation | International Standard:

– ParameterizedValue (see 9.2).

6.2.3 The following notation which can be used as an alternative for "DefinedType" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 13.1) is defined in this Recommendation | International Standard:

– ParameterizedValueSetType (see 9.2).

6.2.4 The following notation which can be used as an alternative for "ObjectClass" (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.2) is defined in this Recommendation | International Standard:

– ParameterizedObjectClass (see 9.2).

6.2.5 The following notation which can be used as an alternative for "Object" (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 11.3) is defined in this Recommendation | International Standard:

– ParameterizedObject (see 9.2).

6.2.6 The following notation which can be used as an alternative for "ObjectSet" (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 12.3) is defined in this Recommendation | International Standard:

– ParameterizedObjectSet (see 9.2).

6.3 Symbols

The following notation which can be used as an alternative for "Symbol" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 12.1) is defined in this Recommendation | International Standard:

– ParameterizedReference (see 9.1).

7 ASN.1 lexical items

This Recommendation | International Standard makes use of the lexical items specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 11.

8 Parameterized assignments

8.1 There are parameterized assignment statements corresponding to each of the assignment statements specified in ITU-T Rec. X.680 | ISO/IEC 8824-1 and ITU-T Rec. X.681 | ISO/IEC 8824-2. The "ParameterizedAssignment" construct is:

```
ParameterizedAssignment ::=
        ParameterizedTypeAssignment             |
        ParameterizedValueAssignment            |
        ParameterizedValueSetTypeAssignment     |
        ParameterizedObjectClassAssignment      |
        ParameterizedObjectAssignment           |
        ParameterizedObjectSetAssignment
```

8.2 Each "Parameterized<X>Assignment" has the same syntax as "<X>Assignment" except that following the initial lexical item there is a "ParameterList". The initial item thereby becomes a parameterized reference name (see 3.4.2):

NOTE – ITU-T Rec. X.680 | ISO/IEC 8824-1 imposes the requirement that all reference names assigned within a module, whether parameterized or not, must be distinct:

```
ParameterizedTypeAssignment ::=
        typereference
        ParameterList
        "::="
        Type

ParameterizedValueAssignment ::=
        valuereference
        ParameterList
        Type
        "::="
        Value

ParameterizedValueSetTypeAssignment ::=
        typereference
        ParameterList
        Type
        "::="
        ValueSet

ParameterizedObjectClassAssignment ::=
        objectclassreference
        ParameterList
```

"::="
ObjectClass

ParameterizedObjectAssignment ::=
objectreference
ParameterList
DefinedObjectClass
"::="
Object

ParameterizedObjectSetAssignment ::=
objectsetreference
ParameterList
DefinedObjectClass
"::="
ObjectSet

8.3 A "ParameterList" is a list of "Parameter"s between braces:

ParameterList ::= "{" Parameter "," + "}"

Each "Parameter" consists of a "DummyReference" and possibly a "ParamGovernor":

Parameter ::= ParamGovernor ":" DummyReference | DummyReference

ParamGovernor ::= Governor | DummyGovernor

Governor ::= Type | DefinedObjectClass

DummyGovernor ::= DummyReference

DummyReference ::= Reference

A "DummyReference" in "Parameter" may stand for:

a) a "Type" or "DefinedObjectClass", in which case there shall be no "ParamGovernor";

b) a "Value" or "ValueSet", in which case the "ParamGovernor" shall be present, and in case "ParamGovernor" is a "Governor" it shall be a "Type", and in case "ParamGovernor" is a "DummyGovernor" the actual parameter for the "ParamGovernor" shall be a "Type";

c) an "Object" or "ObjectSet", in which case the "ParamGovernor" shall be present, and in case "ParamGovernor" is a "Governor" it shall be a "DefinedObjectClass", and in case "ParamGovernor" is a "DummyGovernor" the actual parameter for the "ParamGovernor" shall be a "DefinedObjectClass".

A "DummyGovernor" shall be a "DummyReference" that has no "Governor".

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 "**::=**". The "DummyReference" hides any other "Reference" with the same name in that scope.

NOTE – This sub-clause does not apply to "identifier"s defined in "NamedNumberList"s, "Enumeration"s and "NamedBitList"s, since they are not "Reference"s. The "DummyReference" does not hide these "identifier"s (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 18.11 and 19.10).

8.5 The usage of a "DummyReference" within its scope shall be consistent with its syntactic form, and, where applicable, governor, and all usages of the same "DummyReference" shall be consistent with one another.

NOTE – Where the syntactic form of a dummy reference name is ambiguous (for example, between whether it is an "objectclassreference" or "typereference"), the ambiguity can normally be resolved on the first use of the dummy reference name on the right-hand side of the assignment statement. Thereafter, the nature of the dummy reference name is known. The nature of the dummy reference is, however, not determined solely by the right hand side of the assignment statement when it is in turn used only as an actual parameter in a parameterized reference; in this case, the nature of the dummy reference must be determined by examining the definition of this parameterized reference. Users of the notation are warned that such a practice can make ASN.1 specifications less clear, and it is suggested that adequate comments are provided to explain this for human readers.

Example

Consider the following parameterized object class assignment:

```
PARAMETERIZED-OBJECT-CLASS { TypeParam, INTEGER:valueParam, INTEGER:ValueSetParam }
::=
          CLASS {
          &valueField1          TypeParam,
```

```
&valueField2            INTEGER DEFAULT valueParam,
&valueField3            INTEGER (ValueSetParam),
&ValueSetField    INTEGER DEFAULT { ValueSetParam }
}
```

For the purpose of determining proper usage of the "DummyReference"s in the scope of the "Parameterized Assignment", and for that purpose only, the "DummyReference"s can be regarded to be defined as follows:

```
TypeParam ::= UnspecifiedType

valueParam INTEGER ::= unspecifiedIntegerValue

ValueSetParam INTEGER ::= { UnspecifiedIntegerValueSet }
```

where:

a) **TypeParam** is a "DummyReference" which stands for a "Type". Therefore **TypeParam** can be used wherever a "typereference" can be used, e.g. as a "Type" for the fixed-type value field **valueField1**.

b) **valueParam** is a "DummyReference" which stands for a value of an integer type. Therefore **valueParam** can be used wherever a "valuereference" of an integer value can be used, e.g. as a default value for the fixed-type value field **valueField2**.

c) **ValueSetParam** is a "DummyReference" which stands for a value set of an integer type. Therefore **ValueSetParam** can be used wherever a "typereference" of an integer value can be used, e.g. as a "Type" in the "ContainedSubtype" notation for **valueField3** and **ValueSetField**.

8.6 Each "DummyReference" shall be employed at least once within its scope.

NOTE – If the "DummyReference" did not so appear, then the corresponding "ActualParameter" would have no effect on the definition, and would simply be "discarded", while to the user it might seem that some specification was taking place.

"ParameterizedValueAssignment"s, "ParameterizedValueSetTypeAssignment"s, "ParameterizedObjectAssignment"s and "ParameterizedObjectSetAssignment"s that contain either a direct or indirect reference to themselves are illegal.

8.7 In the definition of a "ParameterizedType", "ParameterizedValueSet", or "ParameterizedObjectClass, a "DummyReference" shall not be passed as a tagged type (as an actual parameter) to a recursive reference to that "ParameterizedType", "ParameterizedValueSet", or "ParameterizedObjectClass" (see A.3).

8.8 In the definition of a "ParameterizedType", "ParameterizedValueSet", or "ParameterizedObjectClass, a circular reference to the item being defined shall not be made, unless such reference is directly or indirectly marked **OPTIONAL** or, in the case of "ParameterizedType" and "ParameterizedValueSet", made through a reference to a choice type, at least one of whose alternatives is non-circular in definition.

8.9 The governor of a "DummyReference" shall not include a reference to another "DummyReference" if that other "DummyReference" also has a governor.

8.10 In a parameterized assignment the right side of the "**::=**" shall not consist solely of a "DummyReference".

8.11 The governor of a "DummyReference" shall not require knowledge of either the "DummyReference" or of the parameterized reference name being defined.

8.12 When a value or value set is supplied to a parameterized type as an actual parameter, the type of the actual parameter is required to be compatible with the governor of the corresponding dummy parameter. (See ITU-T Rec. X.680 | ISO/IEC 8824-1, B.6.2 and B.6.3 for details.)

8.13 In defining a parameterized type with a value or a value set dummy parameter, the type used to govern that dummy parameter shall be a type, all of whose values are valid for use in all places to the right of the assignment where the dummy parameter is used. (See ITU-T Rec. X.680 | ISO/IEC 8824-1, B.6.5 for details.)

9 Referencing parameterized definitions

9.1 Within a "SymbolList" (in "Exports" or "Imports") a parameterized definition shall be referenced by a "ParameterizedReference":

ParameterizedReference ::= Reference | Reference "{" "}"

where "Reference" is the first lexical item in the "ParameterizedAssignment", as specified in 8.2 above.

NOTE – The first alternative of "ParameterizedReference" is provided solely as an aid to human understanding. Both alternatives have the same meaning.

9.2 Other than in "Exports" or "Imports", a parameterized definition shall be referenced by a "Parameterized<X>" construct, which can be used as an alternative for the corresponding "<X>":

ParameterizedType ::=
SimpleDefinedType
ActualParameterList

SimpleDefinedType ::=
ExternalTypeReference |
typereference

ParameterizedValue ::=
SimpleDefinedValue
ActualParameterList

SimpleDefinedValue ::=
ExternalValueReference |
valuereference

ParameterizedValueSetType ::=
SimpleDefinedType
ActualParameterList

ParameterizedObjectClass ::=
DefinedObjectClass
ActualParameterList

ParameterizedObjectSet ::=
DefinedObjectSet
ActualParameterList

ParameterizedObject ::=
DefinedObject
ActualParameterList

9.3 The reference name in the "Defined<X>" shall be a reference name to which an assignment is made in a "ParameterizedAssignment".

9.4 The restrictions on the "Defined<X>" alternative to be used, which are specified in ITU-T Rec. X.680 | ISO/IEC 8824-1 and ITU-T Rec. X.681 | ISO/IEC 8824-2 as normal reference names, apply equally to the corresponding parameterized reference names.

NOTE – In essence, the restrictions are as follows: each "Defined<X>" has two alternatives, "<x>reference" and "External<x>Reference". The former is used within the module of definition or if the definition has been imported and there is no name conflict; the latter is used where there is no imports listed (deprecated), or if there is a conflict between the imported name and a local definition (also deprecated) or between imports.

9.5 The "ActualParameterList" is:

ActualParameterList ::=
"{" ActualParameter "," + "}"

ActualParameter ::=
Type |
Value |
ValueSet |
DefinedObjectClass |
Object |
ObjectSet

9.6 There shall be exactly one "ActualParameter" for each "Parameter" in the corresponding "ParameterizedAssignment" and they shall appear in the same order. The particular choice of "ActualParameter", and the governor (if any) shall be determined by examination of the syntactic form of the "Parameter" and the environment in which it occurs in the "ParameterizedAssignment". The form of the "ActualParameter" shall be the form required to replace the "DummyReference" everywhere in its scope (see 8.4).

Example

The parameterized object class definition of the previous example (see 8.5) can be referenced, for instance, as follows:

```
MY-OBJECT-CLASS ::= PARAMETERIZED-OBJECT-CLASS { BIT STRING, 123, {4 | 5 | 6} }
```

9.7 The actual parameter takes the place of the dummy reference name in determining the actual type, value, value set, object class, object, or object set that is being referenced by this instance of use of the parameterized reference name.

9.8 The meaning of any references which appear in the "ActualParameter", and the tag default applicable to any tags which so appear, are determined according to the tagging environment of the "ActualParameter" rather than that of the corresponding "DummyReference".

NOTE – Thus, parameterization, like referencing, selection types, and **COMPONENTS OF**, among others, is not exactly textual substitution.

Example

Consider the following modules:

```
M1 DEFINITIONS AUTOMATIC TAGS ::= BEGIN
            EXPORTS T1;

      T1 ::= SET {
            f1    INTEGER,
            f2    BOOLEAN
      }

END

M2 DEFINITIONS EXPLICIT TAGS ::= BEGIN
            IMPORTS T1 FROM  M1;

      T3 ::= T2{T1}

      T2{X} ::= SEQUENCE {
            a     INTEGER,
            b     X
      }

END
```

Application of 9.8 implies that the tag for the component **f1** of **T3** (i.e. @T3.b.f1) will be implicitly tagged because the tagging environment of the dummy parameter **X**, namely explicit tagging, does not affect the tagging of the components of the actual parameter **T1**.

Consider the module **M3**:

```
M3 DEFINITIONS AUTOMATIC TAGS ::= BEGIN
            IMPORTS T1 FROM  M1;

      T5 ::= T4{T1}

      T4{Y} ::= SEQUENCE {
            a     INTEGER,
            b     Y
      }

END
```

Application of ITU-T Rec. X.680 | ISO/IEC 8824-1, 30.6, implies that the tag for the component **b** of **T5** (i.e. @T5.b) will be explicitly tagged because the dummy parameter **Y** is always explicitly tagged, hence **T5** is equivalent to:

```
T5 ::= SEQUENCE {
            a     [0] IMPLICIT INTEGER,
            b     [1] EXPLICIT SET {
            f1    [0] INTEGER,
            f2    [1] BOOLEAN
            }
      }
```

while **T3** is equivalent to:

```
T3 ::= SEQUENCE {
            a     INTEGER,
            b     SET {
            f1    [0] IMPLICIT INTEGER,
            f2    [1] IMPLICIT BOOLEAN
            }
      }
```

10 Abstract syntax parameters

10.1 ITU-T Rec. X.681 | ISO/IEC 8824-2, Annex B provides the **ABSTRACT-SYNTAX** information object class and recommends its use to define abstract syntaxes, using as an example an abstract syntax defined as the set of values of a single ASN.1 type which was not parameterized at the outer level.

10.2 Where the ASN.1 type used to define the abstract syntax *is* parameterized, some parameters may be supplied as actual parameters when the abstract syntax is defined, while others may be left as parameters of the abstract syntax itself.

Example

If a parameterized type has been defined called **YYY-PDU** with two dummy references (the first an object set of some defined object class, and the second an integer value for a bound, say), then:

```
yyy-Abstract-Syntax { INTEGER:bound } ABSTRACT-SYNTAX ::=
            { YYY-PDU { {ValidObjects} , bound } IDENTIFIED BY {yyy 5} }
```

defines a parameterized abstract syntax in which the object set has been resolved, but **bound** remains as a parameter of the abstract syntax.

An abstract syntax parameter shall be used:

a) directly or indirectly in the context of a constraint;

b) directly or indirectly as actual parameters that eventually are used in the context of a constraint.

NOTE – See the example in A.2, and the example in ITU-T Rec. X.680 | ISO/IEC 8824-1, F.5.

10.3 A constraint whose value set depends on one or more parameters of the abstract syntax is a variable constraint. Such constraints are determined after the definition of the abstract syntax (perhaps by International Standardized Profiles or in Protocol Implementation Conformance Statements).

NOTE – If somewhere in the chain of definitions involved in the specification of the constraint values a parameter of the abstract syntax appears, the constraint is a variable constraint. It is a variable constraint even if the value set of the resulting constraint is independent of the actual value of the parameter of the abstract syntax.

Example

The value of **(((1..3) EXCEPT a) UNION (1 .. 3))** is always 1..3 no matter what the value of **a** is, nonetheless it is still a variable constraint if **a** is a parameter of the abstract syntax.

10.4 Formally, a variable constraint does not constrain the set of values in the abstract syntax.

NOTE – It is strongly recommended that constraints that are expected to remain as variable constraints in an abstract syntax have an exception specification using the notation provided by ITU-T Rec. X.680 | ISO/IEC 8824-1, 49.4.

Annex A

Examples

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

A.1 Example of the use of a parameterized type definition

Suppose that a protocol designer frequently needs to carry an authenticator with one or more of the fields of the protocol. This will be carried as a **BIT STRING**, alongside the field. Without parameterization, **Authenticator** would need to be defined as a **BIT STRING**, then **authenticator** would need to be added wherever it was to appear, with text to identify what it applied to. Alternatively, the designer could adopt the discipline of turning any field that has an authenticator into a **SEQUENCE** of that field and **authenticator**. The parameterization mechanism provides a convenient short-hand for doing this task.

First we define the parameterized type **SIGNED{}**:

```
SIGNED { ToBeSigned } ::= SEQUENCE
      {
                  authenticated-data     ToBeSigned,
                  authenticator                BIT STRING
      }
```

then, in the body of the protocol, the notation (for example)

```
SIGNED { OrderInformation }
```

is a type notation standing for:

```
SEQUENCE
      {
            authenticated-data     OrderInformation,
            authenticator    BIT STRING
      }
```

Suppose further that for some fields, the sender is to have the option of adding the authenticator or not. This could be achieved by making the **BIT STRING** optional, but a more elegant solution (less bits on the line) would be to define another parameterized type:

```
OPTIONALLY-SIGNED {ToBeSigned} ::= CHOICE
      {
            unsigned-data    [0]   ToBeSigned,
            signed-data      [1]   SIGNED { ToBeSigned }
      }
```

NOTE – The tagging in the **CHOICE** is not necessary if the writer ensures that none of the uses of the parameterized type produce an actual argument which is a **BIT STRING** (the type of **SIGNED**), but is useful in preventing errors in other parts of the specification.

A.2 Example of use of parameterized definitions together with an information object class

Use information object classes to collect all the parameters for an abstract syntax. In that way the number of parameters for an abstract syntax can be reduced to one which is an instance of the collection class. The "InformationFromObject" production can be used to extract information from the parameter object.

Example

```
-- An instance of this class contains all the parameters for the abstract
-- syntax, Message-PDU.

MESSAGE-PARAMETERS ::= CLASS {
                  &maximum-priority-level      INTEGER,
                  &maximum-message-buffer-size INTEGER,
                  &maximum-reference-buffer-sizeINTEGER
      }
      WITH SYNTAX {
                  THE MAXIMUM PRIORITY LEVEL IS&maximum-priority-level
                  THE MAXIMUM MESSAGE BUFFER SIZE IS&maximum-message-buffer-size
                  THE MAXIMUM REFERENCE BUFFER SIZE IS&maximum-reference-buffer-size
      }
```

```
        -- The "ValueFromObject" production is used to extract values
        -- from the abstract syntax parameter, "param". The values can be
        -- used only in constraints. In addition the parameter is passed
        -- through to another parameterized type.

Message-PDU { MESSAGE-PARAMETERS : param } ::= SEQUENCE {
      priority-level   INTEGER (0..param.&maximum-priority-level),
      message          BMPString (SIZE (0..param.&maximum-message-buffer-size)),
      reference        Reference { param }
      }

Reference { MESSAGE-PARAMETERS : param } ::=
            SEQUENCE OF IA5String (SIZE (0..param.&maximum-reference-buffer-size))
      -- Definition of a parameterized abstract syntax information object.
      -- The abstract syntax parameter is used only in constraints.

message-Abstract-Syntax { MESSAGE-PARAMETERS : param }
      ABSTRACT-SYNTAX ::=
      {
            Message-PDU { param }
            IDENTIFIED BY { joint-iso-ccitt asn1(1) examples(123) 0 }
      }
```

The class **MESSAGE-PARAMETERS** and the parameterized abstract syntax object, **message-Abstract-Syntax**, are used as follows:

```
-- This instance of MESSAGE-PARAMETERS defines parameter values
      -- for the abstract syntax.

my-message-parameters MESSAGE-PARAMETERS ::= {
            THE MAXIMUM PRIORITY LEVEL IS 10
            THE MAXIMUM MESSAGE BUFFER SIZE IS  2000
            THE MAXIMUM REFERENCE BUFFER SIZE IS 100
      }
-- The abstract syntax can now be defined with all variable constraints specified.

my-message-Abstract-Syntax ABSTRACT-SYNTAX ::=
            message-Abstract-Syntax { my-message-parameters }
```

A.3 Example of parameterized type definition that is finite

When specifying a parameterized type which represents a generic list, specify the type so that the resulting ASN.1 notation is finite. For example, we may specify:

```
List1 { ElementTypeParam } ::= SEQUENCE {
            elem  ElementTypeParam,
            next  List1 { ElementTypeParam } OPTIONAL
      }
```

which is finite, for when it is used:

```
IntegerList1 ::= List1 { INTEGER }
```

the resulting ASN.1 notation is as you would normally define it:

```
IntegerList1 ::= SEQUENCE {
            elem  INTEGER,
            next  IntegerList1 OPTIONAL
      }
```

Contrast this to the following:

```
List2 { ElementTypeParam } ::= SEQUENCE {
            elem  ElementTypeParam,
            next  List2 { [0] ElementTypeParam } OPTIONAL
      }

IntegerList2 ::= List2 { INTEGER }
```

where the resulting ASN.1 notation is infinite:

```
IntegerList2 ::= SEQUENCE {
      elem  INTEGER,
      next  SEQUENCE {
            elem [0] INTEGER,
            next SEQUENCE {
                  elem       [0][0] INTEGER,
                  next             SEQUENCE {
                        elem [0][0][0]     INTEGER,
                        next             SEQUENCE {
                    -- and so on
                        } OPTIONAL
                  } OPTIONAL
            } OPTIONAL
      } OPTIONAL
            }
```

A.4 Example of a parameterized value definition

If a parameterized string value is defined as follows:

```
genericBirthdayGreeting { IA5String : name } IA5String ::= { "Happy birthday, ",
name, "!!" }
```

then the following two string values are the same:

```
greeting1 IA5String ::= genericBirthdayGreeting { "John" }
      greeting2 IA5String ::= "Happy birthday, John!!"
```

A.5 Example of a parameterized value set definition

If two parameterized value sets are defined as follows:

```
QuestList1 {IA5String : extraQuest} IA5String ::= { "Jack" | "John" | extraQuest }
      QuestList2 {IA5String : ExtraQuests} IA5String ::= { "Jack" | "John" |
      ExtraQuests }
```

then the following value sets denote the same value set:

```
SetOfQuests1 IA5String ::= { QuestList1 { "Jill" } }
      SetOfQuests2 IA5String ::= { QuestList2 { {"Jill"} } }
      SetOfQuests3 IA5String ::= { "Jack" | "John" | "Jill" }
```

and the following value sets denote the same value set:

```
SetOfQuests4 IA5String ::= { QuestList2 { {"Jill" | "Mary"} } }
      SetOfQuests5 IA5String ::= { "Jack" | "John" | "Jill" | "Mary"}
```

Notice that a value set is *always* specified within braces, even when it is a parameterized value set reference. By omitting the braces from a reference to an "identifier" that was created in a value set assignment or from a reference to a "ParameterizedValueSetType" the notation is that of a "Type", not a value set.

A.6 Example of a parameterized class definition

The following parameterized class can be used to define error classes which contain error codes of different types. Note that the **`ErrorCodeType`** parameter is used only as a "DummyGovernor" for the **`ValidErrorCodes`** parameter:

```
GENERIC-ERROR { ErrorCodeType, ErrorCodeType : ValidErrorCodes } ::= CLASS {
            &errorCode      ValidErrorCodes
      }
      WITH SYNTAX {
            CODE &errorCode
      }
```

The parameterized class definition can be used as follows to define different classes which share some characteristics like the same defined syntax:

```
ERROR-1 ::= GENERIC-ERROR { INTEGER, { 1 | 2 | 3 } }

ERROR-2 ::= GENERIC-ERROR { ErrorCodeString, { StringErrorCodes } }

ERROR-3 ::= GENERIC-ERROR { EnumeratedErrorCode, { fatal | error } }

ErrorCodeString ::= IA5String (SIZE (4))

StringErrorCodes ErrorCodeString ::= { "E001" | "E002" | "E003" }

EnumeratedErrorCode ::= ENUMERATED { fatal, error, warning }
```

The defined classes can then be used as follows:

```
My-Errors ERROR-2 ::= { { CODE "E001" } | { CODE "E002" } }

fatalError ERROR-3 ::= { CODE fatal }
```

A.7 Example of a parameterized object set definition

The parameterized object set definition **AllTypes** forms an object set which contains a basic set of objects, **BaseTypes**, and a set of additional objects which are supplied as a parameter, **AdditionalTypes**:

```
AllTypes { TYPE-IDENTIFIER : AdditionalTypes } TYPE-IDENTIFIER ::= { BaseTypes |
AdditionalTypes }

BaseTypes TYPE-IDENTIFIER ::= {
          { BasicType-1 IDENTIFIED BY basic-type-obj-id-value-1 } |
          { BasicType-2 IDENTIFIED BY basic-type-obj-id-value-2 } |
          { BasicType-3 IDENTIFIED BY basic-type-obj-id-value-3 }
     }
```

The parameterized object set definition, **AllTypes**, can be used as follows:

```
My-All-Types TYPE-IDENTIFIER ::= { AllTypes { {
          { My-Type-1 IDENTIFIED BY my-obj-id-value-1 } |
          { My-Type-2 IDENTIFIED BY my-obj-id-value-2 } |
          { My-Type-3 IDENTIFIED BY my-obj-id-value-3 }
     } } }
```

A.8 Example of a parameterized object set definition

The type defined in ITU-T Rec. X.682 | ISO/IEC 8824-3, A.4, can be used in a parameterized abstract syntax definition as follows:

```
-- PossibleBodyTypes is a parameter for an abstract syntax.

message-abstract-syntax { MHS-BODY-CLASS : PossibleBodyTypes } ABSTRACT-SYNTAX ::= {
          INSTANCE OF MHS-BODY-CLASS ({PossibleBodyTypes})
          IDENTIFIED BY { joint-iso-itu asn1(1) examples(1) 123 }
     }
-- This object set lists all the possible pairs of values and type-ids
     -- for the instance-of type. The object set is used as an actual parameter
     -- for the parameterized abstract syntax definition.

My-Body-Types MHS-BODY-CLASS ::= {
          { My-First-Type    IDENTIFIED BY my-first-obj-id } |
          { My-Second-Type   IDENTIFIED BY my-second-obj-id }
     }

my-message-abstract-syntax ABSTRACT-SYNTAX ::=
          message-abstract-syntax { { My-Body-Types } }
```

Annex B

Summary of the notation

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

The following lexical items are defined in ITU-T Rec. X.680 | ISO/IEC 8824-1 and used in this Recommendation | International Standard:

typereference
valuereference
"::="
"{"
"}"
","

The following lexical items are defined in ITU-T Rec. X.681 | ISO/IEC 8824-2 and used in this Recommendation | International Standard:

objectclassreference
objectreference
objectsetreference

The following productions are defined in ITU-T Rec. X.680 | ISO/IEC 8824-1 and used in this Recommendation | International Standard:

DefinedType
DefinedValue
Reference
Type
Value
ValueSet

The following productions are defined in ITU-T Rec. X.681 | ISO/IEC 8824-2 and used in this Recommendation | International Standard:

DefinedObjectClass
DefinedObject
DefinedObjectSet
ObjectClass
Object
ObjectSet

The following productions are defined in this Recommendation | International Standard:

```
ParameterizedAssignment ::=
        ParameterizedTypeAssignment             |
        ParameterizedValueAssignment            |
        ParameterizedValueSetTypeAssignment     |
        ParameterizedObjectClassAssignment      |
        ParameterizedObjectAssignment           |
        ParameterizedObjectSetAssignment

ParameterizedTypeAssignment ::=
        typereference ParameterList "::=" Type

ParameterizedValueAssignment ::=
        valuereference ParameterList Type "::=" Value

ParameterizedValueSetTypeAssignment ::=
        typereference ParameterList Type "::=" ValueSet

ParameterizedObjectClassAssignment ::=
        objectclassreference ParameterList "::=" ObjectClass
```

ParameterizedObjectAssignment ::=

objectreference ParameterList DefinedObjectClass ":::=" Object

ParameterizedObjectSetAssignment ::=
objectsetreference ParameterList DefinedObjectClass ":::=" ObjectSet

ParameterList ::= "{" Parameter "," + "}"

Parameter ::= ParamGovernor ":" DummyReference | DummyReference

ParamGovernor ::= Governor | DummyGovernor

Governor ::= Type | DefinedObjectClass

DummyGovernor ::= DummyReference

DummyReference ::= Reference

ParameterizedReference ::=
Reference | Reference "{" "}"

SimpleDefinedType ::= ExternalTypeReference | typereference

SimpleDefinedValue ::= ExternalValueReference | valuereference

ParameterizedType ::= SimpleDefinedType ActualParameterList

ParameterizedValue ::= SimpleDefinedValue ActualParameterList

ParameterizeValueSetType ::= SimpleDefinedType ActualParameterList

ParameterizedObjectClass ::= DefinedObjectClass ActualParameterList

ParameterizedObjectSet ::= DefinedObjectSet ActualParameterList

ParameterizedObject ::= DefinedObject ActualParameterList

ActualParameterList ::= "{" ActualParameter "," + "}"

ActualParameter ::= Type | Value | ValueSet | DefinedObjectClass | Object | ObjectSet

ITU-T Recommendation X.690
International Standard 8825-1

Information Technology –
ASN.1 encoding rules:
Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)

INTERNATIONAL STANDARD 8825-1

ITU-T RECOMMENDATION X.690

Information Technology – ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)

Summary

This Recommendation | International Standard defines a set of Basic Encoding Rules (BER) that may be applied to values of types defined using the ASN.1 notation. Application of these encoding rules produces a transfer syntax for such values. It is implicit in the specification of these encoding rules that they are also used for decoding. This Recommendation | International Standard defines also a set of Distinguished Encoding Rules (DER) and a set of Canonical Encoding Rules (CER) both of which provide constraints on the Basic Encoding Rules (BER). The key difference between them is that DER uses the definite length form of encoding while CER uses the indefinite length form. DER is more suitable for the small encoded values, while CER is more suitable for the large ones. It is implicit in the specification of these encoding rules that they are also used for decoding.

Source

The ITU-T Recommendation X.690 was approved on the 13th of July 2002. The identical text is also published as ISO/IEC International Standard 8825-1.

CONTENTS

Page

Introduction iv

1 Scope 1

2 Normative references 1
- 2.1 Identical Recommendations | International Standards 1
- 2.2 Additional references 1

3 Definitions 2

4 Abbreviations 3

5 Notation 3

6 Convention 3

7 Conformance 3

8 Basic encoding rules 4
- 8.1 General rules for encoding 4
  - 8.1.1 Structure of an encoding 4
  - 8.1.2 Identifier octets 4
  - 8.1.3 Length octets 6
  - 8.1.4 Contents octets 7
  - 8.1.5 End-of-contents octets 7
- 8.2 Encoding of a boolean value 7
- 8.3 Encoding of an integer value 8
- 8.4 Encoding of an enumerated value 8
- 8.5 Encoding of a real value 8
- 8.6 Encoding of a bitstring value 10
- 8.7 Encoding of an octetstring value 11
- 8.8 Encoding of a null value 12
- 8.9 Encoding of a sequence value 12
- 8.10 Encoding of a sequence-of value 13
- 8.11 Encoding of a set value 13
- 8.12 Encoding of a set-of value 13
- 8.13 Encoding of a choice value 13
- 8.14 Encoding of a tagged value 13
- 8.15 Encoding of an open type 14
- 8.16 Encoding of an instance-of value 14
- 8.17 Encoding of a value of the embedded-pdv type 15
- 8.18 Encoding of a value of the external type 15
- 8.19 Encoding of an object identifier value 16
- 8.20 Encoding of a relative object identifier value 17
- 8.21 Encoding for values of the restricted character string types 17
- 8.22 Encoding for values of the unrestricted character string type 20

9 Canonical encoding rules 20
- 9.1 Length forms 20
- 9.2 String encoding forms 20
- 9.3 Set components 20

10 Distinguished encoding rules 21
- 10.1 Length forms 21
- 10.2 String encoding forms 21
- 10.3 Set components 21

11 Restrictions on BER employed by both CER and DER 22
- 11.1 Boolean values 22
- 11.2 Unused bits 22

11.3 Real values 22
11.4 GeneralString values 23
11.5 Set and sequence components with default value 23
11.6 Set-of components 23
11.7 GeneralizedTime 23
11.8 UTCTime 24
11.8.4 Examples of valid representations 24
11.8.5 Examples of invalid representations 24
12 Use of BER, CER and DER in transfer syntax definition 24
Annex A Example of encodings 26
A.1 ASN.1 description of the record structure 26
A.2 ASN.1 description of a record value 26
A.3 Representation of this record value 26
Annex B Assignment of object identifier values 29
Annex C Illustration of real value encoding 30

Introduction

ITU-T Rec. X.680 | ISO/IEC 8824-1, ITU-T Rec. X.681 | ISO/IEC 8824-2, ITU-T Rec. X.682 | ISO/IEC 8824-3, ITU-T Rec. X.683 | ISO/IEC 8824-4 (Abstract Syntax Notation One or ASN.1) together specify a notation for the definition of abstract syntaxes, enabling application standards to define the types of information they need to transfer. It also specifies a notation for the specification of values of a defined type.

This Recommendation | International Standard defines encoding rules that may be applied to values of types defined using the ASN.1 notation. Application of these encoding rules produces a transfer syntax for such values. It is implicit in the specification of these encoding rules that they are also to be used for decoding.

There may be more than one set of encoding rules that can be applied to values of types that are defined using the ASN.1 notation. This Recommendation | International Standard defines three sets of encoding rules, called *basic encoding rules*, *canonical encoding rules* and *distinguished encoding rules*. Whereas the basic encoding rules give the sender of an encoding various choices as to how data values may be encoded, the canonical and distinguished encoding rules select just one encoding from those allowed by the basic encoding rules, eliminating all of the sender's options. The canonical and distinguished encoding rules differ from each other in the set of restrictions that they place on the basic encoding rules.

The distinguished encoding rules is more suitable than the canonical encoding rules if the encoded value is small enough to fit into the available memory and there is a need to rapidly skip over some nested values. The canonical encoding rules is more suitable than the distinguished encoding rules if there is a need to encode values that are so large that they cannot readily fit into the available memory or it is necessary to encode and transmit a part of a value before the entire value is available. The basic encoding rules is more suitable than the canonical or distinguished encoding rules if the encoding contains a set value or set-of value and there is no need for the restrictions that the canonical and distinguished encoding rules impose. This is due to the memory and CPU overhead that the latter encoding rules exact in order to guarantee that set values and set-of values have just one possible encoding.

Annex A gives an example of the application of the basic encoding rules. It does not form an integral part of this Recommendation | International Standard.

Annex B summarizes the assignment of object identifier values made in this Recommendation | International Standard. It does not form an integral part of this Recommendation | International Standard.

Annex C gives examples of applying the basic encoding rules for encoding reals. It does not form an integral part of this Recommendation | International Standard.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

Information Technology – ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)

1 Scope

This Recommendation | International Standard specifies a set of basic encoding rules that may be used to derive the specification of a transfer syntax for values of types defined using the notation specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, ITU-T Rec. X.681 | ISO/IEC 8824-2, ITU-T Rec. X.682 | ISO/IEC 8824-3, and ITU-T Rec. X.683 | ISO/IEC 8824-4, collectively referred to as Abstract Syntax Notation One or ASN.1. These basic encoding rules are also to be applied for decoding such a transfer syntax in order to identify the data values being transferred. It also specifies a set of canonical and distinguished encoding rules that restrict the encoding of values to just one of the alternatives provided by the basic encoding rules.

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 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.

2.1 Identical Recommendations | International Standards

– ITU-T Recommendation X.680 (2002) | ISO/IEC 8824-1:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.*

– ITU-T Recommendation X.681 (2002) | ISO/IEC 8824-2:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.*

– ITU-T Recommendation X.682 (2002) | ISO/IEC 8824-3:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*

– ITU-T Recommendation X.683 (2002) | ISO/IEC 8824-4:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications.*

2.2 Additional references

– ISO *International Register of Coded Character Sets to be used with Escape Sequences.*

– ISO/IEC 2022:1994, *Information technology – Character code structure and extension techniques.*

ISO 2375:1985, *Data processing – Procedure for registration of escape sequences.*

– ISO 6093:1985, *Information processing – Representation of numerical values in character strings for information interchange.*

– ISO/IEC 6429:1992, *Information technology – Control functions for coded character sets.*

– ISO/IEC 10646-1:2000, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane.*

3 Definitions

For the purposes of this Recommendation | International Standard, the definitions of ITU-T Rec. X.200 | ISO/IEC 7498-1 and ITU-T Rec. X.680 | ISO/IEC 8824-1 and the following definitions apply.

3.1 **canonical encoding**: A complete encoding of an abstract value obtained by the application of encoding rules that have no implementation-dependent options. Such rules result in the definition of a 1-1 mapping between unambiguous and unique encodings and values in the abstract syntax.

3.2 **constructed encoding**: A data value encoding in which the contents octets are the complete encoding of one or more data values.

3.3 **contents octets**: That part of a data value encoding which represents a particular value, to distinguish it from other values of the same type.

3.4 **data value**: Information specified as the value of a type; the type and the value are defined using ASN.1.

3.5 **dynamic conformance**: A statement of the requirement for an implementation to adhere to the prescribed behavior in an instance of communication.

3.6 **encoding (of a data value)**: The complete sequence of octets used to represent the data value.

3.7 **end-of-contents octets**: Part of a data value encoding, occurring at its end, which is used to determine the end of the encoding.

NOTE – Not all encodings require end-of-contents octets.

3.8 **identifier octets**: Part of a data value encoding which is used to identify the type of the value.

NOTE – Some ITU-T Recommendations use the term "data element" for this sequence of octets, but the term is not used in this Recommendation | International Standard, as other Recommendations | International Standards use it to mean "data value".

3.9 **length octets**: Part of a data value encoding following the identifier octets which is used to determine the end of the encoding.

3.10 **primitive encoding**: A data value encoding in which the contents octets directly represent the value.

3.11 **receiver**: An implementation decoding the octets produced by a sender, in order to identify the data value which was encoded.

3.12 **sender**: An implementation encoding a data value for transfer.

3.13 **static conformance**: A statement of the requirement for support by an implementation of a valid set of features from among the defined features.

3.14 **trailing 0 bit**: A 0 in the last position of a bitstring value.

NOTE – The 0 in a bitstring value consisting of a single 0 bit is a trailing 0 bit. Its removal produces an empty bitstring.

4 Abbreviations

For the purposes of this Recommendatrion | International Standard, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules of ASN.1
CER	Canonical Encoding Rules of ASN.1
DER	Distinguished Encoding Rules of ASN.1
ULA	Upper Layer Architecture

5 Notation

This Recommendation | International Standard references the notation defined by ITU-T Rec. X.680 | ISO/IEC 8824-1.

6 Convention

6.1 This Recommendation | International Standard specifies 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 purposes of this Recommendation | International Standard only, 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 purpose of this Recommendation | International Standard, two octet strings can be compared. One octet string is equal to another if they are of the same length and are the same at each octet position. An octet string, S_1, is greater than another, S_2, if and only if either:

a) S_1 and S_2 have identical octets in every position up to and including the final octet in S_2, but S_1 is longer; or

b) S_1 and S_2 have different octets in one or more positions, and in the first such position, the octet in S_1 is greater than that in S_2, considering the octets as unsigned binary numbers whose bit n has weight 2^{n-1}.

7 Conformance

7.1 Dynamic conformance is specified by clauses 8 to 12 inclusive.

7.2 Static conformance is specified by those standards which specify the application of one or more of these encoding rules.

7.3 Alternative encodings are permitted by the basic encoding rules as a sender's option. Receivers who claim conformance to the basic encoding rules shall support all alternatives.

NOTE – Examples of such alternative encodings appear in 8.1.3.2 b) and Table 3.

7.4 No alternative encodings are permitted by the Canonical Encoding Rules or Distinguished Encoding Rules.

8 Basic encoding rules

8.1 General rules for encoding

8.1.1 Structure of an encoding

8.1.1.1 The encoding of a data value shall consist of four components which shall appear in the following order:

a) identifier octets (see 8.1.2);

b) length octets (see 8.1.3);

c) contents octets (see 8.1.4);

d) end-of-contents octets (see 8.1.5).

8.1.1.2 The end-of-contents octets shall not be present unless the value of the length octets requires them to be present (see 8.1.3).

8.1.1.3 Figure 1 illustrates the structure of an encoding (primitive or constructed). Figure 2 illustrates an alternative constructed encoding.

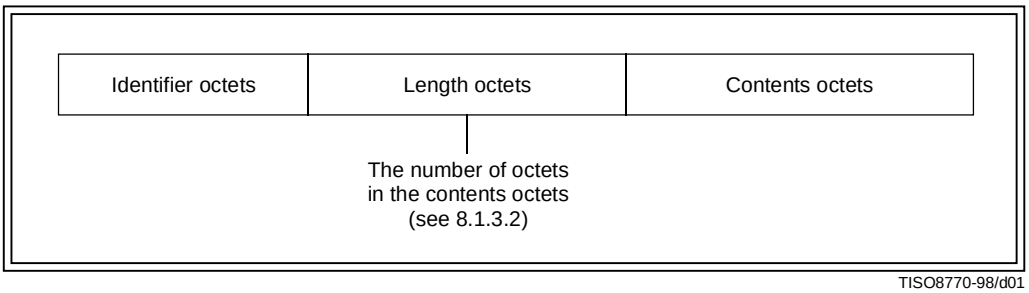


Figure 1 – Structure of an encoding

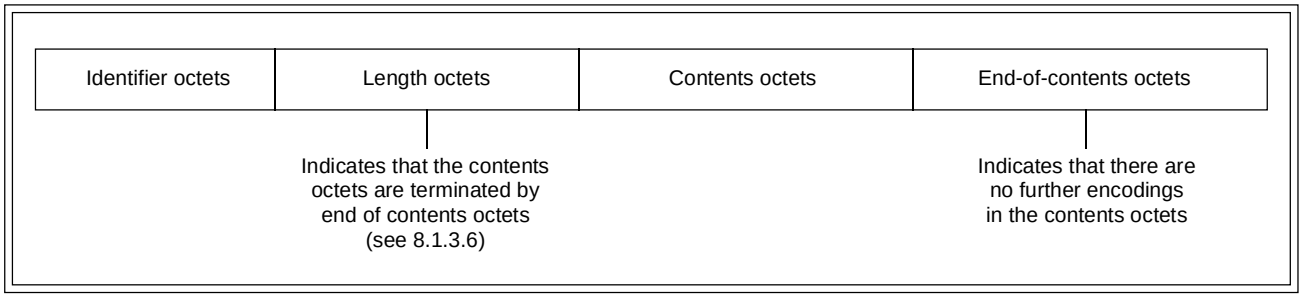


Figure 2 – An alternative constructed encoding

8.1.1.4 Encodings specified in this Recommendation | International Standard are not affected by either the ASN.1 subtype notation or the ASN.1 type extensibility notation.

NOTE – This means that all constraint notation is ignored when determining encodings, and all extensibility markers in **CHOICE**, **SEQUENCE** and **SET** are ignored, with the extensions treated as if they were in the extension root of the type.

8.1.2 Identifier octets

8.1.2.1 The identifier octets shall encode the ASN.1 tag (class and number) of the type of the data value.

8.1.2.2 For tags with a number ranging from zero to 30 (inclusive), the identifier octets shall comprise a single octet encoded as follows:

a) bits 8 and 7 shall be encoded to represent the class of the tag as specified in Table 1;

b) bit 6 shall be a zero or a one according to the rules of 8.1.2.5;

c) bits 5 to 1 shall encode the number of the tag as a binary integer with bit 5 as the most significant bit.

Table 1 – Encoding of class of tag

Class	Bit 8	Bit 7
Universal	0	0
Application	0	1
Context-specific	1	0
Private	1	1

8.1.2.3 Figure 3 illustrates the form of an identifier octet for a type with a tag whose number is in the range zero to 30 (inclusive).

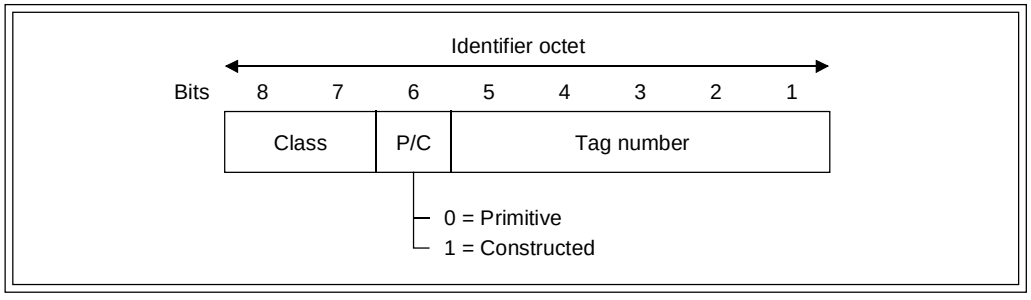


Figure 3 – Identifier octet (low tag number)

8.1.2.4 For tags with a number greater than or equal to 31, the identifier shall comprise a leading octet followed by one or more subsequent octets.

8.1.2.4.1 The leading octet shall be encoded as follows:

a) bits 8 and 7 shall be encoded to represent the class of the tag as listed in Table 1;

b) bit 6 shall be a zero or a one according to the rules of 8.1.2.5;

c) bits 5 to 1 shall be encoded as 11111_2.

8.1.2.4.2 The subsequent octets shall encode the number of the tag as follows:

a) bit 8 of each octet shall be set to one unless it is the last octet of the identifier octets;

b) bits 7 to 1 of the first subsequent octet, followed by bits 7 to 1 of the second subsequent octet, followed in turn by bits 7 to 1 of each further octet, up to and including the last subsequent octet in the identifier octets shall be the encoding of an unsigned binary integer equal to the tag number, with bit 7 of the first subsequent octet as the most significant bit;

c) bits 7 to 1 of the first subsequent octet shall not all be zero.

8.1.2.4.3 Figure 4 illustrates the form of the identifier octets for a type with a tag whose number is greater than 30.

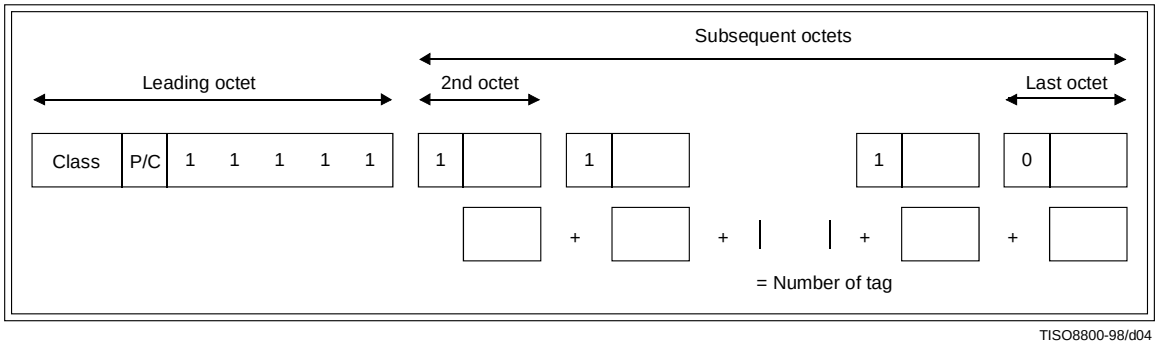


Figure 4 – Identifier octets (high tag number)

8.1.2.5 Bit 6 shall be set to zero if the encoding is primitive, and shall be set to one if the encoding is constructed.

NOTE – Subsequent subclauses specify whether the encoding is primitive or constructed for each type.

8.1.2.6 ITU-T Rec. X.680 | ISO/IEC 8824-1 specifies that the tag of a type defined using the **`CHOICE`** keyword takes the value of the tag of the type from which the chosen data value is taken.

8.1.2.7 ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.2 and 14.4 specifies, that the tag of a type defined using "ObjectClassFieldType" is indeterminate if it is a type field, a variable-type value field, or a variable-type value set field. This type is subsequently defined to be an ASN.1 type, and the complete encoding is then identical to that of a value of the assigned type (including the identifier octets).

8.1.3 Length octets

8.1.3.1 Two forms of length octets are specified. These are:

a) the definite form (see 8.1.3.3); and

b) the indefinite form (see 8.1.3.6).

8.1.3.2 A sender shall:

a) use the definite form (see 8.1.3.3) if the encoding is primitive;

b) use either the definite form (see 8.1.3.3) or the indefinite form (see 8.1.3.6), a sender's option, if the encoding is constructed and all immediately available;

c) use the indefinite form (see 8.1.3.6) if the encoding is constructed and is not all immediately available.

8.1.3.3 For the definite form, the length octets shall consist of one or more octets, and shall represent the number of octets in the contents octets using either the short form (see 8.1.3.4) or the long form (see 8.1.3.5) as a sender's option.

NOTE – The short form can only be used if the number of octets in the contents octets is less than or equal to 127.

8.1.3.4 In the short form, the length octets shall consist of a single octet in which bit 8 is zero and bits 7 to 1 encode the number of octets in the contents octets (which may be zero), as an unsigned binary integer with bit 7 as the most significant bit.

Example

L = 38 can be encoded as 00100110_2

8.1.3.5 In the long form, the length octets shall consist of an initial octet and one or more subsequent octets. The initial octet shall be encoded as follows:

a) bit 8 shall be one;

b) bits 7 to 1 shall encode the number of subsequent octets in the length octets, as an unsigned binary integer with bit 7 as the most significant bit;

c) the value 11111111_2 shall not be used.

NOTE 1 – This restriction is introduced for possible future extension.

Bits 8 to 1 of the first subsequent octet, followed by bits 8 to 1 of the second subsequent octet, followed in turn by bits 8 to 1 of each further octet up to and including the last subsequent octet, shall be the encoding of an unsigned binary integer equal to the number of octets in the contents octets, with bit 8 of the first subsequent octet as the most significant bit.

Example

L = 201 can be encoded as:

10000001_2

11001001_2

NOTE 2 – In the long form, it is a sender's option whether to use more length octets than the minimum necessary.

8.1.3.6 For the indefinite form, the length octets indicate that the contents octets are terminated by end-of-contents octets (see 8.1.5), and shall consist of a single octet.

8.1.3.6.1 The single octet shall have bit 8 set to one, and bits 7 to 1 set to zero.

8.1.3.6.2 If this form of length is used, then end-of-contents octets (see 8.1.5) shall be present in the encoding following the contents octets.

8.1.4 Contents octets

The contents octets shall consist of zero, one or more octets, and shall encode the data value as specified in subsequent clauses.

NOTE – The contents octets depend on the type of the data value; subsequent clauses follow the same sequence as the definition of types in ASN.1.

8.1.5 End-of-contents octets

The end-of-contents octets shall be present if the length is encoded as specified in 8.1.3.6, otherwise they shall not be present.

The end-of-contents octets shall consist of two zero octets.

NOTE – The end-of-contents octets can be considered as the encoding of a value whose tag is universal class, whose form is primitive, whose number of the tag is zero, and whose contents are absent, thus:

End-of-contents	Length	Contents
00_{16}	00_{16}	Absent

8.2 Encoding of a boolean value

8.2.1 The encoding of a boolean value shall be primitive. The contents octets shall consist of a single octet.

8.2.2 If the boolean value is:

FALSE

the octet shall be zero.

If the boolean value is

TRUE

the octet shall have any non-zero value, as a sender's option.

Example

If of type **BOOLEAN**, the value **TRUE** can be encoded as:

Boolean	Length	Contents
01_{16}	01_{16}	FF_{16}

8.3 Encoding of an integer value

8.3.1 The encoding of an integer value shall be primitive. The contents octets shall consist of one or more octets.

8.3.2 If the contents octets of an integer value encoding consist of more than one octet, then the bits of the first octet and bit 8 of the second octet:

a) shall not all be ones; and

b) shall not all be zero.

NOTE – These rules ensure that an integer value is always encoded in the smallest possible number of octets.

8.3.3 The contents octets shall be a two's complement binary number equal to the integer value, and consisting of bits 8 to 1 of the first octet, followed by bits 8 to 1 of the second octet, followed by bits 8 to 1 of each octet in turn up to and including the last octet of the contents octets.

NOTE – The value of a two's complement binary number is derived by numbering the bits in the contents octets, starting with bit 1 of the last octet as bit zero and ending the numbering with bit 8 of the first octet. Each bit is assigned a numerical value of 2^N, where N is its position in the above numbering sequence. The value of the two's complement binary number is obtained by summing the numerical values assigned to each bit for those bits which are set to one, excluding bit 8 of the first octet, and then reducing this value by the numerical value assigned to bit 8 of the first octet if that bit is set to one.

8.4 Encoding of an enumerated value

The encoding of an enumerated value shall be that of the integer value with which it is associated.

NOTE – It is primitive.

8.5 Encoding of a real value

8.5.1 The encoding of a real value shall be primitive.

8.5.2 If the real value is the value zero, there shall be no contents octets in the encoding.

8.5.3 For a non-zero real value, if the base of the abstract value is 10, then the base of the encoded value shall be 10, and if the base of the abstract value is 2 the base of the encoded value shall be 2, 8 or 16 as a sender's option.

8.5.4 If the real value is non-zero, then the base used for the encoding shall be B' as specified in 8.5.3. If B' is 2, 8 or 16, a binary encoding, specified in 8.5.6, shall be used. If B' is 10, a character encoding, specified in 8.5.7, shall be used.

8.5.5 Bit 8 of the first contents octet shall be set as follows:

a) if bit 8 = 1, then the binary encoding specified in 8.5.6 applies;

b) if bit 8 = 0 and bit 7 = 0, then the decimal encoding specified in 8.5.7 applies;

c) if bit 8 = 0 and bit 7 = 1, then a "SpecialRealValue" (see ITU-T Rec. X.680 | ISO/IEC 8824-1) is encoded as specified in 8.5.8.

8.5.6 When binary encoding is used (bit 8 = 1), then if the mantissa M is non-zero, it shall be represented by a sign S, a non-negative integer value N and a binary scaling factor F, such that:

$$M = S \times N \times 2^F$$

$$0 \le F < 4$$

$$S = +1 \text{ or } -1$$

NOTE – The binary scaling factor F is required under certain circumstances in order to align the implied point of the mantissa to the position required by the encoding rules of this subclause. This alignment cannot always be achieved by modification of the exponent E. If the base B' used for encoding is 8 or 16, the implied point can only be moved in steps of 3 or 4 bits, respectively,

by changing the component E. Therefore, values of the binary scaling factor F other than zero may be required in order to move the implied point to the required position.

8.5.6.1 Bit 7 of the first contents octets shall be 1 if S is –1 and 0 otherwise.

8.5.6.2 Bits 6 to 5 of the first contents octets shall encode the value of the base B' as follows:

Bits 6 to 5	*Base*
00	base 2
01	base 8
10	base 16
11	Reserved for further editions of this Recommendation | International Standard.

8.5.6.3 Bits 4 to 3 of the first contents octet shall encode the value of the binary scaling factor F as an unsigned binary integer.

8.5.6.4 Bits 2 to 1 of the first contents octet shall encode the format of the exponent as follows:

a) if bits 2 to 1 are 00, then the second contents octet encodes the value of the exponent as a two's complement binary number;

b) if bits 2 to 1 are 01, then the second and third contents octets encode the value of the exponent as a two's complement binary number;

c) if bits 2 to 1 are 10, then the second, third and fourth contents octets encode the value of the exponent as a two's complement binary number;

d) if bits 2 to 1 are 11, then the second contents octet encodes the number of octets, X say, (as an unsigned binary number) used to encode the value of the exponent, and the third up to the (X plus 3)th (inclusive) contents octets encode the value of the exponent as a two's complement binary number; the value of X shall be at least one; the first nine bits of the transmitted exponent shall not be all zeros or all ones.

8.5.6.5 The remaining contents octets encode the value of the integer N (see 8.5.6) as an unsigned binary number.

NOTE 1 – For non-canonical BER there is no requirement for floating point normalization of the mantissa. This allows an implementor to transmit octets containing the mantissa without performing shift functions on the mantissa in memory. In the Canonical Encoding Rules and the Distinguished Encoding Rules normalization is specified and the mantissa (unless it is 0) needs to be repeatedly shifted until the least significant bit is a 1.

NOTE 2 – This representation of real numbers is very different from the formats normally used in floating point hardware, but has been designed to be easily converted to and from such formats (see Annex C).

8.5.7 When decimal encoding is used (bits 8 to 7 = 00), all the contents octets following the first contents octet form a field, as the term is used in ISO 6093, of a length chosen by the sender, and encoded according to ISO 6093. The choice of ISO 6093 number representation is specified by bits 6 to 1 of the first contents octet as follows:

Bits 6 to 1	*Number representation*
00 0001	ISO 6093 NR1 form
00 0010	ISO 6093 NR2 form
00 0011	ISO 6093 NR3 form

The remaining values of bits 6 to 1 are reserved for further edition of this Recommendation | International Standard.

There shall be no use of scaling factors specified in accompanying documentation (see ISO 6093).

NOTE 1 – The recommendations in ISO 6093 concerning the use of at least one digit to the left of the decimal mark are also recommended in this Recommendation | International Standard, but are not mandatory.

NOTE 2 – Use of the normalized form (see ISO 6093) is a sender's option, and has no significance.

8.5.8 When "SpecialRealValues" are to be encoded (bits 8 to 7 = 01), there shall be only one contents octet, with values as follows:

01000000	Value is **`PLUS-INFINITY`**
01000001	Value is **`MINUS-INFINITY`**

All other values having bits 8 and 7 equal to 0 and 1 respectively are reserved for addenda to this Recommendation | International Standard.

8.6 Encoding of a bitstring value

8.6.1 The encoding of a bitstring value shall be either primitive or constructed at the option of the sender.

NOTE – Where it is necessary to transfer part of a bit string before the entire bitstring is available, the constructed encoding is used.

8.6.2 The contents octets for the primitive encoding shall contain an initial octet followed by zero, one or more subsequent octets.

8.6.2.1 The bits in the bitstring value, commencing with the leading bit and proceeding to the trailing bit, shall be placed in bits 8 to 1 of the first subsequent octet, followed by bits 8 to 1 of the second subsequent octet, followed by bits 8 to 1 of each octet in turn, followed by as many bits as are needed of the final subsequent octet, commencing with bit 8.

NOTE – The terms "leading bit" and "trailing bit" are defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 21.2.

8.6.2.2 The initial octet shall encode, as an unsigned binary integer with bit 1 as the least significant bit, the number of unused bits in the final subsequent octet. The number shall be in the range zero to seven.

8.6.2.3 If the bitstring is empty, there shall be no subsequent octets, and the initial octet shall be zero.

8.6.2.4 Where ITU-T Rec. X.680 | ISO/IEC 8824-1, 21.7 applies, a BER encoder/decoder can add or remove trailing 0 bits from the value.

NOTE – If a bitstring value has no 1 bits, then an encoder (as a sender's option) may encode the value with a length of 1 and with an initial octet set to 0 or may encode it as a bit string with one or more 0 bits following the initial octet.

8.6.3 The contents octets for the constructed encoding shall consist of zero, one, or more nested encodings.

NOTE – Each such encoding includes identifier, length, and contents octets, and may include end-of-contents octets if it is constructed.

8.6.4 To encode a bitstring value in this way, it is segmented. Each segment shall consist of a series of consecutive bits of the value, and with the possible exception of the last, shall contain a number of bits which is a multiple of eight. Each bit in the overall value shall be in precisely one segment, but there shall be no significance placed on the segment boundaries.

NOTE – A segment may be of size zero, i.e. contain no bits.

8.6.4.1 Each encoding in the contents octets shall represent a segment of the overall bitstring, the encoding arising from a recursive application of this subclause. In this recursive application, each segment is treated as if it were a bitstring value. The encodings of the segments shall appear in the contents octets in the order in which their bits appear in the overall value.

NOTE 1 – As a consequence of this recursion, each encoding in the contents octets may itself be primitive or constructed. However, such encodings will usually be primitive.

NOTE 2 – In particular, the tags in the contents octets are always universal class, number 3.

8.6.4.2 Example

If of type **`BIT STRING`**, the value **`'0A3B5F291CD'H`** can be encoded as shown below. In this example, the bit string is represented as a primitive:

BitString	Length	Contents
03_{16}	07_{16}	$040A3B5F291CD0_{16}$

The value shown above can also be encoded as shown below. In this example, the bit string is represented as a constructor:

BitString	Length	Contents		
23_{16}	80_{16}			
		BitString	Length	Contents
		03_{16}	03_{16}	$000A3B_{16}$
		03_{16}	05_{16}	$045F291CD0_{16}$
EOC	Length			
00_{16}	00_{16}			

8.7 Encoding of an octetstring value

8.7.1 The encoding of an octetstring value shall be either primitive or constructed at the option of the sender.

NOTE – Where it is necessary to transfer part of an octet string before the entire octetstring is available, the constructed encoding is used.

8.7.2 The primitive encoding contains zero, one or more contents octets equal in value to the octets in the data value, in the order they appear in the data value, and with the most significant bit of an octet of the data value aligned with the most significant bit of an octet of the contents octets.

8.7.3 The contents octets for the constructed encoding shall consist of zero, one, or more encodings.

NOTE – Each such encoding includes identifier, length, and contents octets, and may include end-of-contents octets if it is constructed.

8.7.3.1 To encode an octetstring value in this way, it is segmented. Each segment shall consist of a series of consecutive octets of the value. There shall be no significance placed on the segment boundaries.

NOTE – A segment may be of size zero, i.e. contain no octets.

8.7.3.2 Each encoding in the contents octets shall represent a segment of the overall octetstring, the encoding arising from a recursive application of this subclause. In this recursive application, each segment is treated as if it were a octetstring value. The encodings of the segments shall appear in the contents octets in the order in which their octets appear in the overall value.

NOTE 1 – As a consequence of this recursion, each encoding in the contents octets may itself be primitive or constructed. However, such encodings will usually be primitive.

NOTE 2 – In particular, the tags in the contents octets are always universal class, number 4.

8.8 Encoding of a null value

8.8.1 The encoding of a null value shall be primitive.

8.8.2 The contents octets shall not contain any octets.

NOTE – The length octet is zero.

Example

If of type **NULL**, the **NULL** can be encoded as:

Null	*Length*
05_{16}	00_{16}

8.9 Encoding of a sequence value

8.9.1 The encoding of a sequence value shall be constructed.

8.9.2 The contents octets shall consist of the complete encoding of one data value from each of the types listed in the ASN.1 definition of the sequence type, in the order of their appearance in the definition, unless the type was referenced with the keyword **OPTIONAL** or the keyword **DEFAULT**.

8.9.3 The encoding of a data value may, but need not, be present for a type which was referenced with the keyword **OPTIONAL** or the keyword **DEFAULT**. If present, it shall appear in the encoding at the point corresponding to the appearance of the type in the ASN.1 definition.

Example

If of type:

```
SEQUENCE {name  IA5String,  ok  BOOLEAN}
```

the value:

```
{name  "Smith",  ok  TRUE}
```

can be encoded as:

Sequence	Length	Contents		
30_{16}	$0A_{16}$			
		IA5String	Length	Contents
		16_{16}	05_{16}	"Smith"
		Boolean	Length	Contents

01_{16} 01_{16} FF_{16}

8.10 Encoding of a sequence-of value

8.10.1 The encoding of a sequence-of value shall be constructed.

8.10.2 The contents octets shall consist of zero, one or more complete encodings of data values from the type listed in the ASN.1 definition.

8.10.3 The order of the encodings of the data values shall be the same as the order of the data values in the sequence-of value to be encoded.

8.11 Encoding of a set value

8.11.1 The encoding of a set value shall be constructed.

8.11.2 The contents octets shall consist of the complete encoding of a data value from each of the types listed in the ASN.1 definition of the set type, in an order chosen by the sender, unless the type was referenced with the keyword **`OPTIONAL`** or the keyword **`DEFAULT`**.

8.11.3 The encoding of a data value may, but need not, be present for a type which was referenced with the keyword **`OPTIONAL`** or the keyword **`DEFAULT`**.

NOTE – The order of data values in a set value is not significant, and places no constraints on the order during transfer.

8.12 Encoding of a set-of value

8.12.1 The encoding of a set-of value shall be constructed.

8.12.2 The text of 8.10.2 applies.

8.12.3 The order of data values need not be preserved by the encoding and subsequent decoding.

8.13 Encoding of a choice value

The encoding of a choice value shall be the same as the encoding of a value of the chosen type.

NOTE 1 – The encoding may be primitive or constructed depending on the chosen type.

NOTE 2 – The tag used in the identifier octets is the tag of the chosen type, as specified in the ASN.1 definition of the choice type.

8.14 Encoding of a tagged value

8.14.1 The encoding of a tagged value shall be derived from the complete encoding of the corresponding data value of the type appearing in the "TaggedType" notation (called the base encoding) as specified in 8.14.2 and 8.14.3.

8.14.2 If implicit tagging (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 30.6) was not used in the definition of the type, the encoding shall be constructed and the contents octets shall be the complete base encoding.

8.14.3 If implicit tagging was used in the definition of the type, then:

a) the encoding shall be constructed if the base encoding is constructed, and shall be primitive otherwise; and

b) the contents octets shall be the same as the contents octets of the base encoding.

Example

With ASN.1 type definitions (in an explicit tagging environment) of:

```
Type1 ::= VisibleString

Type2 ::= [APPLICATION 3] IMPLICIT Type1

Type3 ::= [2] Type2

Type4 ::= [APPLICATION 7] IMPLICIT Type3

Type5 ::= [2] IMPLICIT Type2
```

a value of:

"Jones"

is encoded as follows:

For Type1:

VisibleString	Length	Contents
$1A_{16}$	05_{16}	$4A6F6E6573_{16}$

For Type2:

[Application 3]	Length	Contents
43_{16}	05_{16}	$4A6F6E6573_{16}$

For Type3:

[2]	Length	Contents		
$A2_{16}$	07_{16}			
		[APPLICATION 3]	Length	Contents
		43_{16}	05_{16}	$4A6F6E6573_{16}$

For Type4:

[Application 7]	Length	Contents		
67_{16}	07_{16}			
		[APPLICATION 3]	Length	Contents
		43_{16}	05_{16}	$4A6F6E6573_{16}$

For Type5:

[2]	Length	Contents
82_{16}	05_{16}	$4A6F6E6573_{16}$

8.15 Encoding of an open type

The value of an open type is also a value of some (other) ASN.1 type. The encoding of such a value shall be the complete encoding herein specified for the value considered as being of that other type.

8.16 Encoding of an instance-of value

8.16.1 The encoding of the instance-of type shall be the BER encoding of the following sequence type with the value as specified in 8.16.2:

```
[UNIVERSAL 8] IMPLICIT SEQUENCE
     {
          type-id    <DefinedObjectClass>.&id,
          value [0] EXPLICIT <DefinedObjectClass>.&Type
     }
```

where "<DefinedObjectClass>" is replaced by the particular "DefinedObjectClass" used in the "InstanceOfType" notation.

NOTE – When the value is a value of a single ASN.1 type and BER encoding is used for it, the encoding of this type is identical to an encoding of a corresponding value of the external type, where the **syntax** alternative is in use for representing the abstract value.

8.16.2 The value of the components of the sequence type in 8.16.1 shall be the same as the values of the corresponding components of the associated type in ITU-T Rec. X.681 | ISO/IEC 8824-2, C.7.

8.17 Encoding of a value of the embedded-pdv type

8.17.1 The encoding of a value of the embedded-pdv type shall be the BER encoding of the type as defined in **33.5** of ITU-T Rec. X.680 | ISO/IEC 8824-1.

8.17.2 The contents of the **data-value OCTET STRING** shall be the encoding of the abstract data value of the embedded-pdv type [see **33.3** a) in ITU-T Rec. X.680 | ISO/IEC 8824-1] using the identified transfer syntax, and the value of all other fields shall be the same as the values appearing in the abstract value.

8.18 Encoding of a value of the external type

8.18.1 The encoding of a value of the external type shall be the BER encoding of the following sequence type, assumed to be defined in an environment of **EXPLICIT TAGS**, with a value as specified in the subclauses below:

```
[UNIVERSAL 8] IMPLICIT SEQUENCE {
          direct-reference          OBJECT IDENTIFIER OPTIONAL,
          indirect-reference             INTEGER OPTIONAL,
          data-value-descriptor          ObjectDescriptor OPTIONAL,
          encoding                  CHOICE {
          single-ASN1-type               [0] ABSTRACT-SYNTAX.&Type,
          octet-aligned                  [1] IMPLICIT OCTET STRING,
          arbitrary                      [2] IMPLICIT BIT STRING } }
```

NOTE – This sequence type differs from that in ITU-T Rec. X.680 | ISO/IEC 8824-1 for historical reasons.

8.18.2 The value of the fields depends on the abstract value being transmitted, which is a value of the type specified in 33.5 of ITU-T Rec. X.680 | ISO/IEC 8824-1.

8.18.3 The **data-value-descriptor** above shall be present if and only if the **data-value-descriptor** is present in the abstract value, and shall have the same value.

8.18.4 Values of **direct-reference** and **indirect-reference** above shall be present or absent in accordance with Table 2 . Table 2 maps the external type alternatives of **identification** defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 33.5, to the external type components **direct-reference** and **indirect-reference** defined in 8.18.1.

Table 2 – Alternative encodings for "identification"

identification	direct-reference	indirect-reference
syntaxes	*** CANNOT OCCUR ***	*** CANNOT OCCUR ***
syntax	syntax	ABSENT
presentation-context-id	ABSENT	presentation-context-id
context-negotiation	transfer-syntax	presentation-context-id
transfer-syntax	*** CANNOT OCCUR ***	*** CANNOT OCCUR ***
fixed	*** CANNOT OCCUR ***	*** CANNOT OCCUR ***

8.18.5 The data value shall be encoded according to the transfer syntax identified by the encoding, and shall be placed in an alternative of the **encoding** choice as specified below.

8.18.6 If the data value is the value of a single ASN.1 data type, and if the encoding rules for this data value are one of those specified in this Recommendation | International Standard, then the sending implementation shall use any of the **encoding** choices:

– **single-ASN1-type**,

– **octet-aligned**,

– **arbitrary**

as an implementation option.

8.18.7 If the encoding of the data value, using the agreed or negotiated encoding, is an integral number of octets, then the sending implementation shall use any of the **encoding** choices:

– **octet-aligned**,

– **arbitrary**

as an implementation option.

NOTE – A data value which is a series of ASN.1 types, and for which the transfer syntax specifies simple concatenation of the octet strings produced by applying the ASN.1 Basic Encoding Rules to each ASN.1 type, falls into this category, not that of 8.18.6.

8.18.8 If the encoding of the data value, using the agreed or negotiated encoding, is not an integral number of octets, the **encoding** choice shall be:

– **arbitrary**.

8.18.9 If the **encoding** choice is chosen as **single-ASN1-type**, then the ASN.1 type shall replace the open type, with a value equal to the data value to be encoded.

NOTE – The range of values which might occur in the open type is determined by the registration of the object identifier value associated with the **direct-reference**, and/or the integer value associated with the **indirect-reference**.

8.18.10 If the **encoding** choice is chosen as **octet-aligned**, then the data value shall be encoded according to the agreed or negotiated transfer syntax, and the resulting octets shall form the value of the octetstring.

8.18.11 If the **encoding** choice is chosen as **arbitrary**, then the data value shall be encoded according to the agreed or negotiated transfer syntax, and the result shall form the value of the bitstring.

8.19 Encoding of an object identifier value

8.19.1 The encoding of an object identifier value shall be primitive.

8.19.2 The contents octets shall be an (ordered) list of encodings of subidentifiers (see 8.19.3 and 8.19.4) concatenated together.

Each subidentifier is represented as a series of (one or more) octets. Bit 8 of each octet indicates whether it is the last in the series: bit 8 of the last octet is zero; bit 8 of each preceding octet is one. Bits 7 to 1 of the octets in the series collectively encode the subidentifier. Conceptually, these groups of bits are concatenated to form an unsigned binary number whose most significant bit is bit 7 of the first octet and whose least significant bit is bit 1 of the last octet. The subidentifier shall be encoded in the fewest possible octets, that is, the leading octet of the subidentifier shall not have the value 80_{16}.

8.19.3 The number of subidentifiers (N) shall be one less than the number of object identifier components in the object identifier value being encoded.

8.19.4 The numerical value of the first subidentifier is derived from the values of the first *two* object identifier components in the object identifier value being encoded, using the formula:

$$(X*40) + Y$$

where X is the value of the first object identifier component and Y is the value of the second object identifier component.

NOTE – This packing of the first two object identifier components recognizes that only three values are allocated from the root node, and at most 39 subsequent values from nodes reached by X = 0 and X = 1.

8.19.5 The numerical value of the i*th* subidentifier, ($2 \le i \le N$) is that of the (i + 1)*th* object identifier component.

Example

An **OBJECT IDENTIFIER** value of:

```
{joint-iso-itu-t 100 3}
```

which is the same as:

```
{2 100 3}
```

has a first subidentifier of 180 and a second subidentifier of 3. The resulting encoding is:

OBJECT IDENTIFIER	Length	Contents
06_{16}	03_{16}	813403_{16}

8.20 Encoding of a relative object identifier value

NOTE – The encoding of the object identifier components in a relative object identifier is the same as the encoding of components (after the second) in an object identifier.

8.20.1 The encoding of a relative object identifier value shall be primitive.

8.20.2 The contents octets shall be an (ordered) list of encodings of sub-identifiers (see 8.20.3 and 8.20.4) concatenated together. Each sub-identifier is represented as a series of (one or more) octets. Bit 8 of each octet indicates whether it is the last in the series: bit 8 of the last octet is zero; bit 8 of each preceding octet is one. Bits 7-1 of the octets in the series collectively encode the sub-identifier. Conceptually, these groups of bits are concatenated to form an unsigned binary number whose most significant bit is bit 7 of the first octet and whose least significant bit is bit 1 of the

last octet. The sub-identifier shall be encoded in the fewest possible octets, that is, the leading octet of the sub-identifier shall not have the value 80_{16}.

8.20.3 The number of sub-identifiers (N) shall be equal to the number of object identifier arcs in the relative object identifier value being encoded.

8.20.4 The numerical value of the ith sub-identifier ($1 \leq i \leq N$) is that of the ith object identifier arc in the relative object identifier value being encoded.

8.20.5 **Example** – A relative object identifier value of:

```
{8571 3 2}
```

has sub-identifiers of 8571, 3, and 2. The resulting encoding is:

RELATIVE OBJECT IDENTIFIER	Length	Contents
$0D_{16}$	04_{16}	$C27B0302_{16}$

8.21 Encoding for values of the restricted character string types

8.21.1 The data value consists of a string of characters from the character set specified in the ASN.1 type definition.

8.21.2 Each data value shall be encoded independently of other data values of the same type.

8.21.3 Each character string type shall be encoded as if it had been declared:

```
[UNIVERSAL x] IMPLICIT OCTET STRING
```

where x is the number of the universal class tag assigned to the character string type in ITU-T Rec. X.680 | ISO/IEC 8824-1. The value of the octet string is specified in 8.21.4 and 8.21.5.

8.21.4 Where a character string type is specified in ITU-T Rec. X.680 | ISO/IEC 8824-1 by direct reference to an enumerating table (**`NumericString`** and **`PrintableString`**), the value of the octet string shall be that specified in 8.21.5 for a **`VisibleString`** type with the same character string value.

8.21.5 For restricted character strings apart from **`UniversalString`** and **`BMPString`**, the octet string shall contain the octets specified in ISO/IEC 2022 for encodings in an 8-bit environment, using the escape sequence and character codings registered in accordance with ISO 2375.

8.21.5.1 An escape sequence shall not be used unless it is one of those specified by one of the registration numbers used to define the character string type in ITU-T Rec. X.680 | ISO/IEC 8824-1.

8.21.5.2 At the start of each string, certain registration numbers shall be assumed to be designated as G0 and/or C0 and/or C1, and invoked (using the terminology of ISO/IEC 2022). These are specified for each type in Table 3, together with the assumed escape sequence they imply.

Table 3 – Use of escape sequences

Type	Assumed G0 (Registration number)	Assumed C0 & C1 (Registration number)	Assumed escape sequence(s) and locking shift (where applicable)	Explicit escape sequences allowed?
NumericString	6	None	ESC 2/8 4/2 LS0	No
PrintableString	6	None	ESC 2/8 4/2 LS0	No
TeletexString (T61String)	102	106 (C0) 107 (C1)	ESC 2/8 7/5 LS0 ESC 2/1 4/5 ESC 2/2 4/8	Yes
VideotexString	2	1 (C0) 73 (C1)	ESC 2/8 7/5 LS0 ESC 2/1 4/0 ESC 2/2 4/1	Yes
VisibleString (ISO646String)	6	None	ESC 2/8 4/2 LS0	No
IA5String	6	1 (C0)	ESC 2/8 4/2 LS0 ESC 2/1 4/0	No
GraphicString	6	None	ESC 2/8 4/2 LS0	Yes
GeneralString	6	1 (C0)	ESC 2/8 4/2 LS0 ESC 2/1 4/0	Yes
NOTE – Many of the commonly used characters (for example, A-Z) appear in a number of character repertoires with individual registration numbers and escape sequences. Where ASN.1 types allow escape sequences, a number of encodings may be possible for a particular character string (see also 7.3).				

8.21.5.3 Certain character string types shall not contain explicit escape sequences in their encodings; in all other cases, any escape sequence allowed by 8.21.5.1 can appear at any time, including at the start of the encoding. Table 3 lists the types for which explicit escape sequences are allowed.

8.21.5.4 Announcers shall not be used unless explicitly permitted by the user of ASN.1.

NOTE – The choice of ASN.1 type provides a limited form of announcer functionality. Specific application protocols may choose to carry announcers in other protocol elements, or to specify in detail the manner of use of announcers.

Example

With the ASN.1 type definition:

```
Name ::= VisibleString
```

a value:

```
"Jones"
```

can be encoded (primitive form) as:

VisibleString	Length	Contents
$1A_{16}$	05_{16}	$4A6F6E6573_{16}$

or (constructor form, definite length) as:

VisibleString	Length	Contents		
$3A_{16}$	09_{16}			
		OctetString	Length	Contents
		04_{16}	03_{16}	$4A6F6E_{16}$
		OctetString	Length	Contents
		04_{16}	02_{16}	6573_{16}

or (constructor form, indefinite length) as:

VisibleString	Length	Contents		
$3A_{16}$	80_{16}			
		OctetString	Length	Contents
		04_{16}	03_{16}	$4A6F6E_{16}$
		OctetString	Length	Contents

04_{16}	02_{16}	6573_{16}
EOC	Length	
00_{16}	00_{16}	

8.21.6 The above example illustrates three of the (many) possible forms available as a sender's option. Receivers are required to handle all permitted forms (see 7.3).

8.21.7 For the **`UniversalString`** type, the octet string shall contain the octets specified in ISO/IEC 10646-1, using the 4-octet canonical form (see 13.2 of ISO/IEC 10646-1). Signatures shall not be used. Control functions may be used provided they satisfy the restrictions imposed by 8.21.9.

8.21.8 For the **`BMPString`** type, the octet string shall contain the octets specified in ISO/IEC 10646-1, using the 2-octet BMP form (see 13.1 of ISO/IEC 10646-1). Signatures shall not be used. Control functions may be used provided they satisfy the restrictions imposed by 8.21.9.

8.21.9 The C0 and C1 control functions of ISO/IEC 6429 may be used with the following exceptions.

NOTE 1 – The effect of this subclause is to allow the useful control functions such as LF, CR, TAB, etc., while forbidding the use of escapes to other character sets.

NOTE 2 – The C0 and C1 control functions are each encoded in two octets for BMPString and four for UniversalString.

a) Announcer escape sequences defined in ISO/IEC 2022 shall not be used.

   NOTE 3 – The assumed character coding environment is ISO/IEC 10646-1.

b) Designating or identifying escape sequences defined in ISO/IEC 2022 shall not be used, including the identifying escape sequences permitted by ISO/IEC 10646-1, 17.2 and 17.4.

   NOTE 4 – ASN.1 allows the use of the PermittedAlphabet subtype notation to select the set of allowed characters. PermittedAlphabet is also used to select the level of implementation of ISO/IEC 10646-1. **`BMPString`** is always used for the two-octet form and **`UniversalString`** for the four-octet form.

c) Invoking escape sequence or control sequences of ISO/IEC 2022 shall not be used, such as SHIFT IN (SI), SHIFT OUT (SO), or LOCKING SHIFT FOR G3 (SS3)

d) The coding shall conform to ISO/IEC 10646-1 and remain in that code set.

e) Control sequences for identifying subsets of graphic characters according to ISO/IEC 10646-1, 16.3, shall not be used.

   NOTE 5 – ASN.1 applications use subtyping to indicate subsets of the graphic characters of ISO/IEC 10646-1 and to select the ISO/IEC 10646-1 cells that correspond to the control characters of ISO/IEC 6429.

f) The escape sequences of ISO/IEC 10646-1, 16.5, shall not be used to switch to ISO/IEC 2022 codes.

8.21.10 For the **`UTF8String`** type, the octet string shall contain the octets specified in ISO/IEC 10646-1, Annex D. Announcers and escape sequences shall not be used, and each character shall be encoded in the smallest number of octets available for that character.

8.22 Encoding for values of the unrestricted character string type

8.22.1 The encoding of a value of the unrestricted character string type shall be the BER encoding of the type as defined in 40.5 of ITU-T Rec. X.680 | ISO/IEC 8824-1.

8.22.2 The contents of the **`string-value OCTET STRING`** shall be the encoding of the abstract character string value of the unrestricted character string type [see 40.3 a) of ITU-T Rec. X.680 | ISO/IEC 8824-1] using the identified character transfer syntax, and the value of all other fields shall be the same as the values appearing in the abstract value.

8.23 The following "useful types" shall be encoded as if they had been replaced by their definitions given in clauses 42-44 of ITU-T Rec. X.680 | ISO/IEC 8824-1:

- generalized time;
- universal time;
- object descriptor.

9 Canonical encoding rules

The encoding of a data values employed by the canonical encoding rules is the basic encoding described in clause 8, together with the following restrictions and those also listed in clause 11.

9.1 Length forms

If the encoding is constructed, it shall employ the indefinite length form. If the encoding is primitive, it shall include the fewest length octets necessary. [Contrast with 8.1.3.2 b).]

9.2 String encoding forms

Bitstring, octetstring, and restricted character string values shall be encoded with a primitive encoding if they would require no more than 1000 contents octets, and as a constructed encoding otherwise. The string fragments contained in the constructed encoding shall be encoded with a primitive encoding. The encoding of each fragment, except possibly the last, shall have 1000 contents octets. (Contrast with 8.21.6.)

9.3 Set components

The encodings of the component values of a set value shall appear in an order determined by their tags as specified in 8.6 of ITU-T Rec. X.680 | ISO/IEC 8824-1. Additionally, for the purposes of determining the order in which components are encoded when one or more component is an untagged choice type, each untagged choice type is ordered as though it has a tag equal to that of the smallest tag in that choice type or any untagged choice types nested within.

Example

In the following which assumes a tagging environment of **IMPLICIT TAGS**:

```
A ::= SET
      {
      a     [3] INTEGER,
      b     [1] CHOICE
      {
      c     [2] INTEGER,
      d     [4] INTEGER
      },
      e     CHOICE
      {
      f     CHOICE
      {
      g     [5] INTEGER,
      h     [6] INTEGER
      },
      i     CHOICE
      {
      j     [0] INTEGER
      }
      }
      }
```

the order in which the components of the set are encoded will always be e, b, a, since the tag **[0]** sorts lowest, then **[1]**, then **[3]**.

10 Distinguished encoding rules

The encoding of a data values employed by the distinguished encoding rules is the basic encoding described in clause 8, together with the following restrictions and those also listed in clause 11.

10.1 Length forms

The definite form of length encoding shall be used, encoded in the minimum number of octets. [Contrast with 8.1.3.2 b).]

10.2 String encoding forms

For bitstring, octetstring and restricted character string types, the constructed form of encoding shall not be used. (Contrast with 8.21.6.)

10.3 Set components

The encodings of the component values of a set value shall appear in an order determined by their tags as specified in 8.6 of ITU-T Rec. X.680 | ISO/IEC 8824-1.

NOTE – Where a component of the set is an untagged choice type, the location of that component in the ordering will depend on the tag of the choice component being encoded.

11 Restrictions on BER employed by both CER and DER

References in clause 8 and its subclauses to "shall be the BER encoding" shall be interpreted as "shall be the CER or DER encoding, as appropriate". (See 8.16.1, 8.17.1, 8.18.1 and 8.22.1.)

11.1 Boolean values

If the encoding represents the boolean value **TRUE**, its single contents octet shall have all eight bits set to one. (Contrast with 8.2.2.)

11.2 Unused bits

11.2.1 Each unused bit in the final octet of the encoding of a bit string value shall be set to zero.

11.2.2 Where ITU-T Rec. X.680 | ISO/IEC 8824-1, 21.7, applies, the bitstring shall have all trailing 0 bits removed before it is encoded.

NOTE 1 – In the case where a size constraint has been applied, the abstract value delivered by a decoder to the application will be one of those satisfying the size constraint and differing from the transmitted value only in the number of trailing 0 bits.

NOTE 2 – If a bitstring value has no 1 bits, then an encoder shall encode the value with a length of 1 and an initial octet set to 0.

11.3 Real values

11.3.1 If the encoding represents a real value whose base B is 2, then binary encoding employing base 2 shall be used. Before encoding, the mantissa M and exponent E are chosen so that M is either 0 or is odd.

NOTE – This is necessary because the same real value can be regarded as both {M, 2, E} and {M', 2, E'} with $M \neq M'$ if, for some non-zero integer n:

$$M' = M \times 2^{-n}$$

$$E' = E + n$$

In encoding the value, the binary scaling factor F shall be zero, and M and E shall each be represented in the fewest octets necessary.

11.3.2 If the encoding represents a real value whose base B is 10, then decimal encoding shall be used. In forming the encoding, the following applies:

11.3.2.1 The ISO 6093 NR3 form shall be used (see 8.5.7).

11.3.2.2 SPACE shall not be used within the encoding.

11.3.2.3 If the real value is negative, then it shall begin with a MINUS SIGN (–), otherwise, it shall begin with a digit.

11.3.2.4 Neither the first nor the last digit of the mantissa may be a 0.

11.3.2.5 The last digit in the mantissa shall be immediately followed by FULL STOP (.), followed by the exponent-mark "E".

11.3.2.6 If the exponent has the value 0, it shall be written "+0", otherwise the exponent's first digit shall not be zero, and PLUS SIGN shall not be used.

11.4 GeneralString values

The encoding of values of the **GeneralString** type (and its subtypes) shall generate escape sequences to designate and invoke a new register entry only when the register entry for the character is not currently designated as the G0, G1, G2, G3, C0, or C1 set. All designations and invocations shall be into the smallest numbered G or C set for which there is an escape sequence defined in the entry of the International Register of Coded Character Sets to be used with Escape Sequences.

NOTE 1 – For the purposes of the above clause, G0 is the smallest numbered G set, followed by G1, G2, and G3 in order. C0 is the smallest numbered C set, followed by C1.

NOTE 2 – Each character in a character string value is associated with a particular entry in the International Register of Coded Character Sets.

11.5 Set and sequence components with default value

The encoding of a set value or sequence value shall not include an encoding for any component value which is equal to its default value.

11.6 Set-of components

The encodings of the component values of a set-of value shall appear in ascending order, the encodings being compared as octet strings with the shorter components being padded at their trailing end with 0-octets.

NOTE – The padding octets are for comparison purposes only and do not appear in the encodings.

11.7 GeneralizedTime

11.7.1 The encoding shall terminate with a "Z", as described in the ITU-T Rec. X.680 | ISO/IEC 8824-1 clause on **`GeneralizedTime`**.

11.7.2 The seconds element shall always be present.

11.7.3 The fractional-seconds elements, if present, shall omit all trailing zeros; if the elements correspond to 0, they shall be wholly omitted, and the decimal point element also shall be omitted.

Example

A seconds element of "26.000" shall be represented as "26"; a seconds element of "26.5200" shall be represented as "26.52".

11.7.4 The decimal point element, if present, shall be the point option "**`.`**".

11.7.5 Midnight (GMT) shall be represented in the form:

"YYYYMMDD000000Z"

where "YYYYMMDD" represents the day following the midnight in question.

EXAMPLE

Examples of valid representations

`"19920521000000Z"`

`"19920622123421Z"`

`"19920722132100.3Z"`

Examples of invalid representations

`"19920520240000Z"` (midnight represented incorrectly)

`"19920622123421.0Z"` (spurious trailing zeros)

`"19920722132100.30Z"` (spurious trailing zeros)

11.8 UTCTime

11.8.1 The encoding shall terminate with "Z", as described in the ITU-T X.680 | ISO/IEC 8824-1 clause on UTCTime.

11.8.2 The seconds element shall always be present.

11.8.3 Midnight (GMT) shall be represented in the form:

"YYMMDD000000Z"

where "YYMMDD" represents the day following the midnight in question.

11.8.4 Examples of valid representations

`"920521000000Z"`

`"920622123421Z"`

`"920722132100Z"`

11.8.5 Examples of invalid representations

`"920520240000Z"` (midnight represented incorrectly)

`"9207221321Z"` (seconds of "00" omitted)

12 Use of BER, CER and DER in transfer syntax definition

12.1 The encoding rules specified in this Recommendation | International Standard can be referenced and applied whenever there is a need to specify an unambiguous, undivided and self-delimiting octet string representation for all of the values of a single ASN.1 type.

NOTE – All such octet strings are unambiguous within the scope of the single ASN.1 type. They would not necessarily be unambiguous if mixed with encodings of a different ASN.1 type.

12.2 The following object identifier and object descriptor values are assigned to identify and describe the basic encoding rules specified in this Recommendation | International Standard:

{joint-iso-itu-t asn1 (1) basic-encoding (1)}

and

"Basic Encoding of a single ASN.1 type".

12.3 The following object identifier and object descriptor values are assigned to identify and describe the canonical encoding rules specified in this Recommendation | International Standard:

{joint-iso-itu-t asn1(1) ber-derived(2) canonical-encoding(0)}

and

"Canonical encoding of a single ASN.1 type".

12.4 The following object identifier and object descriptor values are assigned to identify and describe the distinguished encoding rules specified in this Recommendation | International Standard:

{joint-iso-itu-t asn1(1) ber-derived(2) distinguished-encoding(1)}

and

"Distinguished encoding of a single ASN.1 type".

12.5 Where an unambiguous specification defines an abstract syntax as a set of abstract values, each of which is a value of some specifically named ASN.1 type, usually (but not necessarily) a choice type, then one of the object identifier values specified in 12.2, 12.3 or 12.4 may be used with the abstract syntax name to identify the basic encoding rules, canonical encoding rules or distinguished encoding rules, respectively, to the specifically named ASN.1 type used in defining the abstract syntax.

12.6 The names specified in 12.2, 12.3 and 12.4 shall not be used with an abstract syntax name to identify a transfer syntax unless the conditions of 12.5 for the definition of the abstract syntax are met.

Annex A

Example of encodings

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

This annex illustrates the basic encoding rules specified in this Recommendation | International Standard by showing the representation in octets of a (hypothetical) personnel record which is defined using ASN.1.

A.1 ASN.1 description of the record structure

The structure of the hypothetical personnel record is formally described below using ASN.1 specified in ITU-T Rec. X.680 | ISO/IEC 8824-1 for defining types.

```
PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET {
      Name              Name,
      title             [0] VisibleString,
      number            EmployeeNumber,
      dateOfHire        [1] Date,
      nameOfSpouse      [2] Name,
      children          [3] IMPLICIT
            SEQUENCE OF ChildInformation  DEFAULT  {} }

ChildInformation ::= SET
      { name            Name,
        dateOfBirth     [0] Date}

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
      { givenName       VisibleString,
        initial         VisibleString,
        familyName      VisibleString}

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER

Date ::= [APPLICATION 3] IMPLICIT VisibleString  -- YYYYMMDD
```

A.2 ASN.1 description of a record value

The value of John Smith's personnel record is formally described below using ASN.1.

```
{ name {givenName "John",initial "P",familyName "Smith"},
      title             "Director",
      number            51,
      dateOfHire        "19710917",
      nameOfSpouse      {givenName "Mary",initial "T",familyName "Smith"},
      children
            {
                  {name {givenName "Ralph",initial "T",familyName "Smith"},
                        dateOfBirth "19571111"
                  },
                  {name {givenName "Susan",initial "B",familyName "Jones"},
                        dateOfBirth "19590717"
                  }
            }
      }
```

A.3 Representation of this record value

The representation in octets of the record value given above (after applying the basic encoding rules defined in this Recommendation | International Standard) is shown below. The values of identifiers, lengths, and the contents of integers are shown in hexadecimal, two hexadecimal digits per octet. The values of the contents of character strings are shown as text, one character per octet.

Personnel

Record	Length	Contents
60	8185	

Name	Length	Contents						
61	10							
		VisibleString	Length	Contents				
		1A	04	"John"				
		VisibleString	Length	Contents				
		1A	01	"P"				
		VisibleString	Length	Contents				
		1A	05	"Smith"				
Title	Length	Contents						
A0	0A							
		VisibleString	Length	Contents				
		1A	08	"Director"				
Employee Number	Length	Contents						
42	01	33						
Date of Hire	Length	Contents						
A1	0A							
		Date	Length	Contents				
		43	08	"19710917"				
Name of Spouse	Length	Contents						
A2	12							
		Name	Length	Contents				
		61	10					
				VisibleString	Length	Contents		
				1A	04	"Mary"		
				VisibleString	Length	Contents		
				1A	01	"T"		
				VisibleString	Length	Contents		
				1A	05	"Smith"		
[3]	Length	Contents						
A3	42							
		Set	Length	Contents				
		31	1F					
				Name	Length	Contents		
				61	11			
					VisibleString	Length	Contents	
					1A	05	"Ralph"	
					VisibleString	Length	Contents	
					1A	01	"T"	
					VisibleString	Length	Contents	

					1A	05	"Smith"
			Date of Birth	Length	Contents		
			A0	0A			
					Date	Length	Contents
					43	08	"19571111"
Set	Length	Contents					
31	1F						
		Name	Length	Contents			
		61	11				
					VisibleString	Length	Contents
					1A	05	"Susan"
					VisibleString	Length	Contents
					1A	01	"B"
					VisibleString	Length	Contents
					1A	05	"Jones"
		Date of Birth	Length	Contents			
		A0	0A				
					Date	Length	Contents
					43	08	"19590717"

Annex B

Assignment of object identifier values

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

The following values are assigned in this Recommendation | International Standard:

Subclause **Object Identifier Value**

12.2 **{joint-iso-itu-t asn1 (1) basic-encoding (1)}**

Object Descriptor Value

"Basic Encoding of a single ASN.1 type"

Subclause **Object Identifier Value**

12.3 **{joint-iso-itu-t asn1(1) ber-derived(2) canonical-encoding(0)}**

Object Descriptor Value

"Canonical encoding of a single ASN.1 type"

Subclause **Object Identifier Value**

12.4 **{joint-iso-itu-t asn1(1) ber-derived(2) distinguished-encoding(1)}**

Object Descriptor Value

"Distinguished encoding of a single ASN.1 type"

Annex C

Illustration of real value encoding

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

C.1 A sender will normally examine his own hardware floating point representation to determine the (value-independent) algorithms to be used to transfer values between this floating-point representation and the length and contents octets of the encoding of an ASN.1 real value. This annex illustrates the steps which could be taken in such a process by using the (artificial) hardware floating point representation of the mantissa shown in Figure C.1.

It is assumed that the exponent can easily be obtained from the floating point hardware as an integer value E.

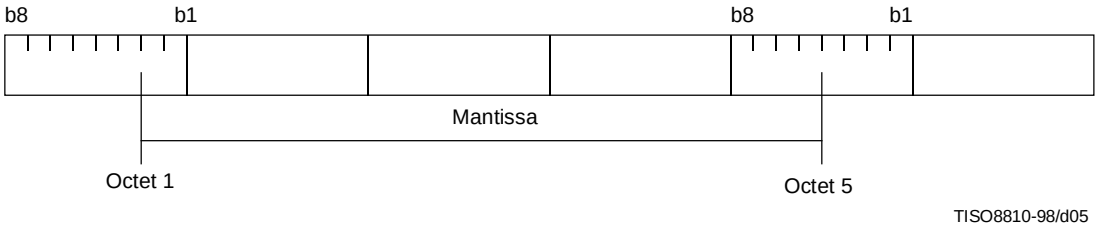


Figure C.1 – Floating point representation

C.2 The contents octets which need to be generated for sending a non-zero value using binary encoding (as specified in the body of this Recommendation | International Standard) are:

1 S bb ff ee Octets for E Octets for N

where S (the mantissa sign) is dependent on the value to be converted, bb is a fixed value (say 10) to represent the base (in this case let us assume base 16), ff is the fixed F value calculated as described in C.3, and ee is a fixed length of exponent value calculated as described in C.4. (This annex does not treat the case where E needs to exceed three octets.)

C.3 The algorithm will transmit octets 1 to 5 of the hardware representation as the value of N, after forcing bits 8 to 3 of octet 1 and bits 4 to 1 of octet 5 to zero. The implied decimal point is assumed to be positioned between bits 2 and 1 of octet 1 in the hardware representation which delivers the value of E. Its implied position can be shifted to the nearest point after the end of octet 5 by reducing the value of E before transmission. In our example system we can shift by four bits for every exponent decrement (because we are assuming base 16), so a decrement of 9 will position the implied point between bits 6 and 5 of octet 6. Thus the value of M is N multiplied by 2^3 to position the point correctly in M. (The implied position in N, the octets transferred, is after bit 1 of octet 5.) Thus we have the crucial parameters:

F = 3 (so ff is 11)

exponent decrement = 9

C.4 The length needed for the exponent is now calculated by working out the maximum number of octets needed to represent the values:

E_{min} – excess – exponent decrement

E_{max} – excess – exponent decrement

where E_{min} and E_{max} are minimum and maximum integer values of the exponent representation, excess is any value which needs subtracting to produce the true exponent value, and the exponent decrement is as calculated in C.3. Let us assume this gives a length of 3 octets. Then ee is 10. Let us also assume excess is zero.

C.5 The transmission algorithm is now:

a) Transmit the basic encoding rules identifier octets field with a tag for ASN.1 type real.

b) Test for zero, and if so, transmit an ASN.1 basic encoding rules length field with value of zero (no contents octets), and end the algorithm.

c) Test and remember the mantissa sign, and negate the mantissa if negative.

d) Transmit an ASN.1 basic encoding rules length field with value of 9, then:

   – 11101110, if negative; or

– 10101110, if positive.

e) Produce and transmit the 3 octet exponent with value:

$$E - 9$$

f) Zero bits 8 to 3 of octet 1 and bits 4 to 1 of octet 5, then transmit the 5 octet mantissa.

C.6 The receiving algorithm has to be prepared to handle any ASN.1 basic encoding, but here the floating point unit can be directly used. We proceed as follows:

a) Check octet 1 of the contents; if it is 1 x 101110 we have a transmission compatible with ours, and can simply reverse the sending algorithm.

b) Otherwise, for character encoding, invoke standard character decimal to floating point conversion software, and deal with a "SpecialRealValue" according to the application semantics (perhaps setting the largest and smallest number the hardware floating point can handle).

c) For a binary transmission, put N into the floating point unit, losing octets at the least significant end if necessary, multiply by 2^F, and by B^E, then negate if necessary. Implementors may find optimization possible in special cases, but may find (apart from the optimization relating to transmissions from a compatible machine) that testing for them loses more than they gain.

C.7 The above algorithms are illustrative only. Implementors will, of course, determine their own best strategies.

ITU-T Recommendation X.691
International Standard 8825-2

Information Technology –
ASN.1 encoding rules:
Specification of Packed Encoding Rules (PER)

INTERNATIONAL STANDARD 8825-2

ITU-T RECOMMENDATION X.691

Information Technology – ASN.1 Encoding Rules – Specification of Packed Encoding Rules (PER)

Summary

This Recommendation | International Standard describes a set of encoding rules that can be applied to values of all ASN.1 types to achieve a much more compact representation than that achieved by the Basic Encoding Rules and its derivatives (described in ITU-T Rec. X.690 | ISO/IEC 8825-1).

Source

The ITU-T Recommendation X.691 was approved on the 13th of July 2002. The identical text is also published as ISO/IEC International Standard 8825-2.

CONTENTS

		Page
Introduction		iv
1	Scope	5
2	Normative references	5
	2.1 Identical Recommendations | International	5
	2.2 Paired Recommendations | International Standards equivalent in technical content	5
	2.3 Additional references	5
3	Definitions	6
	3.1 Specification of Basic Notation	6
	3.2 Information Object Specification	6
	3.3 Constraint Specification	6
	3.4 Parameterization of ASN.1 Specification	6
	3.5 Basic Encoding Rules	6
	3.6 Additional definitions	6
4	Abbreviations	9
5	Notation	9
6	Convention	9
7	Encoding rules defined in this Recommendation | International Standard	9
8	Conformance	10
9	The approach to encoding used for PER	11
	9.1 Use of the type notation	11
	9.2 Use of tags to provide a canonical order	11
	9.3 PER-visible constraints	11
	9.4 Type and value model used for encoding	13
	9.5 Structure of an encoding	13
	9.6 Types to be encoded	14
10	Encoding procedures	14
	10.1 Production of the complete encoding	14
	10.2 Open type fields	15
	10.3 Encoding as a non-negative-binary-integer	15
	10.4 Encoding as a 2's-complement-binary-integer	16
	10.5 Encoding of a constrained whole number	16
	10.6 Encoding of a normally small non-negative whole number	17
	10.7 Encoding of a semi-constrained whole number	17
	10.8 Encoding of an unconstrained whole number	18
	10.9 General rules for encoding a length determinant	18
11	Encoding the boolean type	21
12	Encoding the integer type	21
13	Encoding the enumerated type	22
14	Encoding the real type	22
15	Encoding the bitstring type	22
16	Encoding the octetstring type	23
17	Encoding the null type	24
18	Encoding the sequence type	24
19	Encoding the sequence-of type	25
20	Encoding the set type	25
21	Encoding the set-of type	26

22 Encoding the choice type 26
23 Encoding the object identifier type 27
24 Encoding the relative object identifier type 27
25 Encoding the embedded-pdv type 27
26 Encoding of a value of the external type 27
27 Encoding the restricted character string types 29
28 Encoding the unrestricted character string type 30
29 Object identifiers for transfer syntaxes 31
Annex A Example of encodings 32
A.1 Record that does not use subtype constraints 32
A.1.1 ASN.1 description of the record structure 32
A.1.2 ASN.1 description of a record value 32
A.1.3 ALIGNED PER representation of this record value 32
A.1.4 UNALIGNED PER representation of this record value 34
A.2 Record that uses subtype constraints 35
A.2.1 ASN.1 description of the record structure 35
A.2.2 ASN.1 description of a record value 35
A.2.3 ALIGNED PER representation of this record value 35
A.2.4 UNALIGNED PER representation of this record value 37
A.3 Record that uses extension markers 38
A.3.1 ASN.1 description of the record structure 38
A.3.2 ASN.1 description of a record value 38
A.3.3 ALIGNED PER representation of this record value 38
A.3.4 UNALIGNED PER representation of this record value 40
A.4 Record that uses extension addition groups 41
A.4.1 ASN.1 description of the record structure 41
A.4.2 ASN.1 description of a record value 42
A.4.3 ALIGNED PER representation of this record value 42
A.4.4 UNALIGNED PER representation of this record value 43
Annex B Observations on combining PER-visible constraints 44
Annex C Support for the PER algorithms 49
Annex D Support for the ASN.1 rules of extensibility 50
Annex E Tutorial annex on concatenation of PER encodings 51
Annex F Assignment of object identifier values 52

Introduction

The publications ITU-T Rec. X.680 | ISO/IEC 8824-1, ITU-T Rec. X.681 | ISO/IEC 8824-2, ITU-T Rec. X.682 | ISO/IEC 8824-3, ITU-T Rec. X.683 | ISO/IEC 8824-4 together describe Abstract Syntax Notation One (ASN.1), a notation for the definition of messages to be exchanged between peer applications.

This Recommendation | International Standard defines encoding rules that may be applied to values of types defined using the notation specified in ITU-T Rec. X.680 | ISO/IEC 8824-1. Application of these encoding rules produces a transfer syntax for such values. It is implicit in the specification of these encoding rules that they are also to be used for decoding.

There are more than one set of encoding rules that can be applied to values of ASN.1 types. This Recommendation | International Standard defines a set of Packed Encoding Rules (PER), so called because they achieve a much more compact representation than that achieved by the Basic Encoding Rules (BER) and its derivatives described in ITU-T Rec. X.690 | ISO/IEC 8825-1 which is referenced for some parts of the specification of these Packed Encoding Rules.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

Information Technology – ASN.1 Encoding Rules – Specification of Packed Encoding Rules (PER)

1 Scope

This Recommendation | International Standard specifies a set of Packed Encoding Rules that may be used to derive a transfer syntax for values of types defined in ITU-T Rec. X.680 | ISO/IEC 8824-1. These Packed Encoding Rules are also to be applied for decoding such a transfer syntax in order to identify the data values being transferred.

The encoding rules specified in this Recommendation | International Standard:

- are used at the time of communication;
- are intended for use in circumstances where minimizing the size of the representation of values is the major concern in the choice of encoding rules;
- allow the extension of an abstract syntax by addition of extra values, preserving the encodings of the existing values, for all forms of extension described in ITU-T Rec. X.680 | ISO/IEC 8824-1.

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 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.

2.1 Identical Recommendations | International

- ITU-T Recommendation X.680 (2002) | ISO/IEC 8824-1:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*.
- ITU-T Recommendation X.681 (2002) | ISO/IEC 8824-2:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification*.
- ITU-T Recommendation X.682 (2002) | ISO/IEC 8824-3:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*
- ITU-T Recommendation X.683 (2002) | ISO/IEC 8824-4:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications*.
- ITU-T Recommendation X.690 (2002) | ISO/IEC 8825-1:2002, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).*

2.2 Paired Recommendations | International Standards equivalent in technical content

2.3 Additional references

- ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange*.
- ISO/IEC 2022:1994, *Information technology – Character code structure and extension techniques*.

- ISO 2375:1985, *Data processing – Procedure for registration of escape sequences.*
- ISO 6093:1985, *Information processing – Representation of numerical values in character strings for information interchange.*
- *ISO International Register of Coded Character Sets to be Used with Escape Sequences.*
- ISO/IEC 10646-1:2000, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane.*

3 Definitions

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

3.1 Specification of Basic Notation

For the purposes of this Recommendation | International Standard, all the definitions in ITU-T Rec. X.680 | ISO/IEC 8824-1 apply.

3.2 Information Object Specification

For the purposes of this Recommendation | International Standard, all the definitions in ITU-T Rec. X.681 | ISO/IEC 8824-2 apply.

3.3 Constraint Specification

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.682 | ISO/IEC 8824-3:

a) component relation constraint;

b) table constraint.

3.4 Parameterization of ASN.1 Specification

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.683 | ISO/IEC 8824-4:

- variable constraint.

3.5 Basic Encoding Rules

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.690 | ISO/IEC 8825-1:

a) dynamic conformance;

b) static conformance;

c) data value;

d) encoding (of a data value);

e) sender;

f) receiver.

3.6 Additional definitions

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

3.6.1 2's-complement-binary-integer encoding: The encoding of a whole number into a bit-field (octet-aligned in the ALIGNED variant) of a specified length, or into the minimum number of octets that will accommodate that whole number encoded as a 2's-complement-integer, which provides representations for whole numbers that are equal to, greater than, or less than zero, as specified in 10.4.

NOTE 1 – The value of a two's complement binary number is derived by numbering the bits in the contents octets, starting with bit 1 of the last octet as bit zero and ending the numbering with bit 8 of the first octet. Each bit is assigned a numerical value of 2^N, where N is its position in the above numbering sequence. The value of the two's complement binary number is obtained by summing the numerical values assigned to each bit for those bits which are set to one, excluding bit 8 of the first octet, and then reducing this value by the numerical value assigned to bit 8 of the first octet if that bit is set to one.

NOTE 2 – *Whole number* is a synonym for the mathematical term *integer*. It is used here to avoid confusion with the ASN.1 type *integer*.

3.6.2 abstract syntax value: A value of an abstract syntax (defined as the set of values of a single ASN.1 type), which is to be encoded by PER, or which is to be generated by PER decoding.

NOTE – The single ASN.1 type associated with an abstract syntax is formally identified by an object of class "**ABSTRACT-SYNTAX**".

3.6.3 bit-field: The product of some part of the encoding mechanism that consists of an ordered set of bits that are not necessarily a multiple of eight.

NOTE – If the use of this term is followed by "octet-aligned in the ALIGNED variant", this means that the bit-field is required to begin on an octet boundary in the complete encoding for the aligned variant of PER.

3.6.4 canonical encoding: A complete encoding of an abstract syntax value obtained by the application of encoding rules that have no implementation-dependent options; such rules result in the definition of a 1-1 mapping between unambiguous and unique bitstrings in the transfer syntax and values in the abstract syntax.

3.6.5 composite type: A set, sequence, set-of, sequence-of, choice, embedded-pdv, external or unrestricted character string type.

3.6.6 composite value: The value of a composite type.

3.6.7 constrained whole number: A whole number which is constrained by PER-visible constraints to lie within a range from "lb" to "ub" with the value "lb" less than or equal to "ub", and the values of "lb" and "ub" as permitted values.

NOTE – Constrained whole numbers occur in the encoding which identifies the chosen alternative of a choice type, the length of character, octet and bit string types whose length has been restricted by PER-visible constraints to a maximum length, the count of the number of components in a sequence-of or set-of type that has been restricted by PER-visible constraints to a maximum number of components, the value of an integer type that has been constrained by PER-visible constraints to lie within finite minimum and maximum values, and the value that denotes an enumeration in an enumerated type.

3.6.8 effective size constraint (for a constrained string type): A single finite size constraint that could be applied to a built-in string type and whose effect would be to permit all and only those lengths that can be present in the constrained string type.

NOTE 1 – For example, the following has an effective size constraint:

```
A ::= IA5String (SIZE(1..4) | SIZE(10..15))
```

since it can be rewritten with a single size constraint that applies to all values:

```
A ::= IA5String (SIZE(1..4 | 10..15))
```

whereas the following has no effective size constraint since the string can be arbitrarily long if it does not contain any characters other than 'a', 'b' and 'c':

```
B ::= IA5String (SIZE(1..4) | FROM("abc"))
```

NOTE 2 – The effective size constraint is used only to determine the encoding of lengths.

3.6.9 effective permitted-alphabet constraint (for a constrained restricted character string type): A single permitted-alphabet constraint that could be applied to a built-in known-multiplier character string type and whose effect would be to permit all and only those characters that can be present in at least one character position of any one of the values in the constrained restricted character string type.

NOTE 1 – For example, in:

```
Ax ::= IA5String (FROM("AB") | FROM("CD"))
Bx ::= IA5String (SIZE(1..4) | FROM("abc"))
```

Ax has an effective permitted-alphabet constraint of **"ABCD"**. **Bx** has an effective permitted-alphabet constraint that consists of the entire **IA5String** alphabet since there is no smaller permitted-alphabet constraint that applies to all values of **Bx**.

NOTE 2 – The effective permitted-alphabet constraint is used only to determine the encoding of characters.

3.6.10 enumeration index: The non-negative whole number associated with an "EnumerationItem" in an enumerated type. The enumeration indices are determined by sorting the "EnumerationItem"s into ascending order by their enumeration value, then by assigning an enumeration index starting with zero for the first "EnumerationItem", one for the second, and so on up to the last "EnumerationItem" in the sorted list.

NOTE – "EnumerationItem"s in the "RootEnumeration" are sorted separately from those in the "AdditionalEnumeration".

3.6.11 extensible for PER encoding: A property of a type which requires that PER identifies an encoding of a value as that of a root value or as that of an extension addition.

NOTE – Root values are normally encoded more efficiently than extension additions.

3.6.12 field-list: An ordered set of bit-fields that is produced as a result of applying these encoding rules to components of an abstract value.

3.6.13 indefinite-length: An encoding whose length is greater than 64K-1 or whose maximum length cannot be determined from the ASN.1 notation.

3.6.14 fixed-length type: A type such that the value of the outermost length determinant in an encoding of this type can be determined (using the mechanisms specified in this Recommendation | International Standard) from the type notation (after the application of PER-visible constraints only) and is the same for all possible values of the type.

3.6.15 fixed value: A value such that it can be determined (using the mechanisms specified in this Recommendation | International Standard) that this is the only permitted value (after the application of PER-visible constraints only) of the type governing it.

3.6.16 known-multiplier character string type: A restricted character string type where the number of octets in the encoding is a known fixed multiple of the number of characters in the character string for all permitted character string values. The known-multiplier character string types are **`IA5String`**, **`PrintableString`**, **`VisibleString`**, **`NumericString`**, **`UniversalString`** and **`BMPString`**.

3.6.17 length determinant: A count (of bits, octets, characters, or components) determining the length of part or all of a PER encoding.

3.6.18 normally small non-negative whole number: A part of an encoding which represents values of an unbounded non-negative integer, but where small values are more likely to occur than large ones.

3.6.19 normally small length: A length encoding which represents values of an unbounded length, but where small lengths are more likely to occur than large ones.

3.6.20 non-negative-binary-integer encoding: The encoding of a constrained or semi-constrained whole number into either a bit-field of a specified length, or into a bit-field (octet-aligned in the ALIGNED variant) of a specified length, or into the minimum number of octets that will accommodate that whole number encoded as a non-negative-binary-integer which provides representations for whole numbers greater than or equal to zero, as specified in 10.3.

NOTE – The value of a two's complement binary number is derived by numbering the bits in the contents octets, starting with bit 1 of the last octet as bit zero and ending the numbering with bit 8 of the first octet. Each bit is assigned a numerical value of 2^N, where N is its position in the above numbering sequence. The value of the two's complement binary number is obtained by summing the numerical values assigned to each bit for those bits which are set to one.

3.6.21 outermost type: An ASN.1 type whose encoding is included in a non-ASN.1 carrier or as the value of other ASN.1 constructs (see 10.1.1).

NOTE – PER encodings of an outermost type are always an integral multiple of eight bits.

3.6.22 PER-visible constraint: An instance of use of the ASN.1 constraint notation which affects the PER encoding of a value.

3.6.23 relay-safe encoding: A complete encoding of an abstract syntax value which can be decoded (including any embedded encodings) without knowledge of the environment in which the encoding was performed.

3.6.24 semi-constrained whole number: A whole number which is constrained by PER-visible constraints to exceed or equal some value "lb" with the value "lb" as a permitted value, and which is not a constrained whole number.

NOTE – Semi-constrained whole numbers occur in the encoding of the length of unconstrained (and in some cases constrained) character, octet and bit string types, the count of the number of components in unconstrained (and in some cases constrained) sequence-of and set-of types, and the value of an integer type that has been constrained to exceed some minimum value.

3.6.25 simple type: A type that is not a composite type.

3.6.26 textually dependent: A term used to identify the case where if some reference name is used in evaluating an element set, the value of the element set is considered to be dependent on that reference name, regardless of whether the actual set arithmetic being performed is such that the final value of the element set is independent of the actual element set value assigned to the reference name.

NOTE – For example, the following definition of **Foo** is textually dependent on **Bar** even though **Bar** has no effect on **Foo**s set of values (thus, according to 9.3.5 the constraint on **Foo** is not PER-visible since **Bar** is constrained by a table constraint and **Foo** is textually dependent on **Bar**).

```
MY-CLASS ::= CLASS { &name PrintableString, &age INTEGER } WITH SYNTAX{&name , &age}
MyObjectSet MY-CLASS ::= { {"Jack", 7} | {"Jill", 5} }
Bar ::= MY-CLASS.&age ({MyObjectSet})
Foo ::= INTEGER (Bar | 1..100)
```

3.6.27 **unconstrained whole number**: A whole number which is not constrained by PER-visible constraints.

NOTE – Unconstrained whole numbers occur only in the encoding of a value of the integer type.

4 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules of ASN.1
CER	Canonical Encoding Rules of ASN.1
DER	Distinguished Encoding Rules of ASN.1
PER	Packed Encoding Rules of ASN.1
16K	16384
32K	32768
48K	49152
64K	65536

5 Notation

This Recommendation | International Standard references the notation defined by ITU-T Rec. X.680 | ISO/IEC 8824-1.

6 Convention

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 purposes 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 the "least significant bit".

6.3 The term "octet" is frequently used in this Recommendation | International Standard to stand for "eight bits". The use of this term in place of "eight bits" does not carry any implications of alignment. Where alignment is intended, it is explicitly stated in this Recommendation | International Standard.

7 Encoding rules defined in this Recommendation | International Standard

7.1 This Recommendation | International Standard specifies four encoding rules (together with their associated object identifiers) which can be used to encode and decode the values of an abstract syntax defined as the values of a single (known) ASN.1 type. This clause describes their applicability and properties.

7.2 Without knowledge of the type of the value encoded, it is not possible to determine the structure of the encoding (under any of the PER encoding rule algorithms). In particular, the end of the encoding cannot be determined from the encoding itself without knowledge of the type being encoded.

7.3 PER encodings are always relay-safe provided the abstract values of the types **EXTERNAL**, **EMBEDDED PDV** and **CHARACTER STRING** are constrained to prevent the carriage of OSI presentation context identifiers.

7.4 The most general encoding rule algorithm specified in this Recommendation | International Standard is BASIC-PER, which does not in general produce a canonical encoding.

7.5 A second encoding rule algorithm specified in this Recommendation | International Standard is CANONICAL-PER, which produces encodings that are canonical. This is defined as a restriction of implementation-dependent choices in the BASIC-PER encoding.

NOTE 1 – CANONICAL-PER produces canonical encodings that have applications when authenticators need to be applied to abstract values.

NOTE 2 – Any implementation conforming to CANONICAL-PER for encoding is conformant to BASIC-PER for encoding. Any implementation conforming to BASIC-PER for decoding is conformant to CANONICAL-PER for decoding. Thus, encodings made according to CANONICAL-PER are encodings that are permitted by BASIC-PER.

7.6 If a type encoded with BASIC-PER or CANONICAL-PER contains **EMBEDDED PDV**, **CHARACTER STRING** or **EXTERNAL** types, then the outer encoding ceases to be relay-safe unless the transfer syntax used for all the **EMBEDDED PDV**, **CHARACTER STRING** and **EXTERNAL** types is relay safe. If a type encoded with CANONICAL-PER contains **EMBEDDED PDV**, **EXTERNAL** or **CHARACTER STRING** types, then the outer encoding ceases to be canonical unless the transfer syntax used for all the **EMBEDDED PDV**, **EXTERNAL** and **CHARACTER STRING** types is canonical.

NOTE – The character transfer syntaxes supporting all character abstract syntaxes of the form {iso standard 10646 level-1(1) ....} are canonical. Those supporting {iso standard 10646 level-2(2) ....} and {iso standard 10646 level-3(3) ....} are not always canonical. All the above character transfer syntaxes are relay-safe.

7.7 Both BASIC-PER and CANONICAL-PER come in two variants, the ALIGNED variant, and the UNALIGNED variant. In the ALIGNED variant, padding bits are inserted from time to time to restore octet alignment. In the UNALIGNED variant, no padding bits are ever inserted.

7.8 There are no interworking possibilities between the ALIGNED variant and the UNALIGNED variant.

7.9 PER encodings are self-delimiting only with knowledge of the type of the encoded value. Encodings are always a multiple of eight bits. When carried in an **EXTERNAL** type they shall be carried in the **OCTET STRING** choice alternative, unless the **EXTERNAL** type itself is encoded in PER, in which case the value may be encoded as a single ASN.1 type (i.e., an open type). When carried in OSI presentation protocol, the "full encoding" (as defined in ITU-T Rec. X.226 | ISO/IEC 8823-1) with the **OCTET STRING** choice alternative shall be used.

7.10 The rules of this Recommendation | International Standard apply to both algorithms and to both variants unless otherwise stated.

7.11 Annex C is informative, and gives recommendations on which combinations of PER to implement in order to maximize the chances of interworking.

8 Conformance

8.1 Dynamic conformance is specified by clause 9 onwards.

8.2 Static conformance is specified by those standards which specify the application of these Packed Encoding Rules.

NOTE – Annex C provides guidance on static conformance in relation to support for the two variants of the two encoding rule algorithms. This guidance is designed to ensure interworking, while recognizing the benefits to some applications of encodings that are neither relay-safe nor canonical.

8.3 The rules in this Recommendation | International Standard are specified in terms of an encoding procedure. Implementations are not required to mirror the procedure specified, provided the bit string produced as the complete encoding of an abstract syntax value is identical to one of those specified in this Recommendation | International Standard for the applicable transfer syntax.

8.4 Implementations performing decoding are required to produce the abstract syntax value corresponding to any received bit string which could be produced by a sender conforming to the encoding rules identified in the transfer syntax associated with the material being decoded.

NOTE 1 – In general there are no alternative encodings defined for the BASIC-PER explicitly stated in this Recommendation | International Standard. The BASIC-PER becomes canonical by specifying relay-safe operation and by restricting some of the encoding options of other ISO/IEC Standards that are referenced. CANONICAL-PER provides an alternative to both the Distinguished

Encoding Rules and Canonical Encoding Rules (see ITU-T Rec. X.690 | ISO/IEC 8825-1) where a canonical and relay-safe encoding is required.

NOTE 2 – When CANONICAL-PER is used to provide a canonical encoding, it is recommended that any resulting encrypted hash value that is derived from it should have associated with it an algorithm identifier that identifies CANONICAL-PER as the transformation from the abstract syntax value to an initial bitstring (which is then hashed).

9 The approach to encoding used for PER

9.1 Use of the type notation

9.1.1 These encoding rules make specific use of the ASN.1 type notation as specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, and can only be applied to encode the values of a single ASN.1 type specified using that notation.

9.1.2 In particular, but not exclusively, they are dependent on the following information being retained in the ASN.1 type and value model underlying the use of the notation:

a) the nesting of choice types within choice types;

b) the tags placed on the components in a set type, and on the alternatives in a choice type, and the values given to an enumeration;

c) whether a set or sequence type component is optional or not;

d) whether a set or sequence type component has a **DEFAULT** value or not;

e) the restricted range of values of a type which arise through the application of PER-visible constraints (only);

f) whether a component is an open type;

g) whether a type is extensible for PER encoding.

9.2 Use of tags to provide a canonical order

This Recommendation | International Standard requires components of a set type and a choice type to be canonically ordered independent of the textual ordering of the components. The canonical order is determined by sorting the outermost tag of each component, as specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 8.6.

9.3 PER-visible constraints

NOTE – The fact that some ASN.1 constraints may not be PER-visible for the purposes of encoding and decoding does not in any way affect the use of such constraints in the handling of errors detected during decoding, nor does it imply that values violating such constraints are allowed to be transmitted by a conforming sender. However, this Recommendation | International Standard makes no use of such constraints in the specification of encodings.

9.3.1 Constraints that are expressed in human-readable text or in ASN.1 comment are not PER-visible.

9.3.2 Variable constraints are not PER-visible (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 10.3 and 10.4).

9.3.3 Table constraints are not PER-visible (see ITU-T Rec. X.682 | ISO/IEC 8824-3).

9.3.4 Component relation constraints (see ITU-T Rec. X.682 | ISO/IEC 8824-3, 10.7) are not PER-visible.

9.3.5 Constraints whose evaluation is textually dependent on a table constraint or a component relation constraint are not PER-visible (see ITU-T Rec. X.682 | ISO/IEC 8824-3).

9.3.6 Constraints on restricted character string types which are not (see ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 37) known-multiplier character string types are not PER-visible (see 3.6.16).

9.3.7 Pattern constraints are not PER-visible.

9.3.8 Subject to the above, all size constraints are PER-visible.

9.3.9 The effective size constraint for a constrained type is a single size constraint such that a size is permitted if and only if there is some value of the constrained type that has that (permitted) size.

9.3.10 Permitted-alphabet constraints on known-multiplier character string types which are not extensible after application of ITU-T Rec. X.680 | ISO/IEC 8824-1, 48.3 to 48.5, are PER-visible. Permitted-alphabet constraints which are extensible are not PER-visible.

9.3.11 The effective permitted-alphabet constraint for a constrained type is a single permitted-alphabet constraint which allows a character if and only if there is some value of the constrained type that contains that character. If all characters of the type being constrained can be present in some value of the constrained type, then the effective permitted-alphabet constraint is the set of characters defined for the unconstrained type.

9.3.12 Constraints applied to real types are not PER-visible.

9.3.13 An inner type constraint applied to an unrestricted character string or embbeded-pdv type is PER-visible only when it is used to restrict the value of the **`syntaxes`** component to a single value, or when it is used to restrict **`identification`** to the **`fixed`** alternative (see clauses 25 and 28).

9.3.14 Constraints on the useful types are not PER-visible.

9.3.15 Single value subtype constraints applied to a character string type are not PER-visible.

9.3.16 Subject to the above, all other constraints are PER-visible if and only if they are applied to an integer type or to a known-multiplier character string type.

9.3.17 In general the constraint on a type will consist of individual constraints combined using some or all of set arithmetic, contained subtype constraints, and serial application of constraints. The following clauses specify the effect if some of the component parts of the total constraint are PER-visible and some are not.

NOTE – See Annex B for further discussion on the effect of combining constraints that individually are PER-visible or not PER-visible.

9.3.18 If a constraint consists of a serial application of constraints, the constraints which are not PER-visible, if any, do not affect PER encodings, but cause the extensibility (and extension additions) present in any earlier constraints to be removed as specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 46.8.

NOTE 1 – If the final constraint in a serial application is not PER-visible, then the type is not extensible for PER-encodings, and is encoded without an extension bit.

NOTE 2 – For example:

```
A ::= IA5String(SIZE(1..4))(FROM("ABCD",...))
```

has an effective permitted-alphabet constraint that consists of the entire **`IA5String`** alphabet since the extensible permitted-alphabet constraint is not PER-visible. It has nevertheless an effective size constraint which is "**`SIZE(1..4)`**".

Similarly,

```
B ::= IA5String(A)
```

has the same effective size constraint and the same effective permitted-alphabet constraint.

9.3.19 If a constraint that is PER-visible is part of an **`INTERSECTION`** construction, then the resulting constraint is PER-visible, and consists of the **`INTERSECTION`** of all PER-visible parts (with the non-PER-visible parts ignored). If a constraint which is not PER-visible is part of a **`UNION`** construction, then the resulting constraint is not PER-visible. If a constraint has an **`EXCEPT`** clause, the **`EXCEPT`** and the following value set is completely ignored, whether the value set following the **`EXCEPT`** is PER-visible or not.

NOTE – For example:

```
A ::= IA5String (SIZE(1..4) INTERSECTION FROM("ABCD",...))
```

has an effective size constraint of 1..4 but the alphabet constraint is not visible because it is extensible.

9.3.20 A type is also extensible for PER encodings (whether subsequently constrained or not) if any of the following occurs:

a) it is derived from an **`ENUMERATED`** type (by subtyping, type referencing, or tagging) and there is an extension marker in the "Enumerations" production; or

b) it is derived from a **`SEQUENCE`** type (by subtyping, type referencing, or tagging) and there is an extension marker in the "ComponentTypeLists" or in the "SequenceType" productions; or

c) it is derived from a **`SET`** type (by subtyping, type referencing, or tagging) and there is an extension marker in the "ComponentTypeLists" or in the "SetType" productions; or

d) it is derived from a **`CHOICE`** type (by subtyping, type referencing, or tagging) and there is an extension marker in the "AlternativeTypeLists" production.

9.4 Type and value model used for encoding

9.4.1 An ASN.1 type is either a simple type or is a type built using other types. The notation permits the use of type references and tagging of types. For the purpose of these encoding rules, the use of type references and tagging have no effect on the encoding and are invisible in the model, except as stated in 9.2. The notation also permits the application of constraints and of error specifications. PER-visible constraints are present in the model as a restriction of the values of a type. Other constraints and error specifications do not affect encoding and are invisible in the PER type and value model.

9.4.2 A value to be encoded can be considered as either a simple value or as a composite value built using the structuring mechanisms from components which are either simple or composite values, paralleling the structure of the ASN.1 type definition.

9.4.3 When a constraint includes a value as an extension addition that is present in the root, that value is always encoded as a value in the root, not as a value which is an extension addition.

EXAMPLE

```
INTEGER (0..10, ..., 5)
-- The value 5 encodes as a root value, not as an extension addition.
```

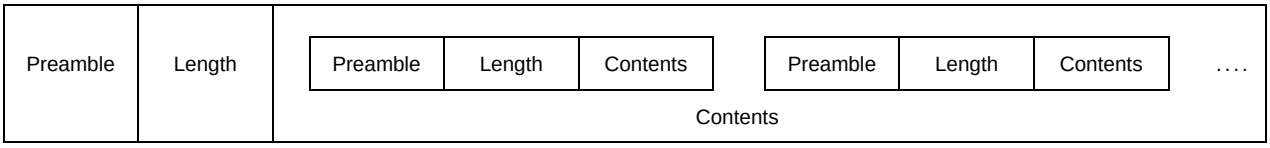
9.5 Structure of an encoding

9.5.1 These encoding rules specify:

a) the encoding of a simple value into a field-list; and

b) the encoding of a composite value into a field-list, using the field-lists generated by application of these encoding rules to the components of the composite value; and

c) the transformation of the field-list of the outermost value into the complete encoding of the abstract syntax value (see 10.1).



NOTE – The preamble, length, and contents are all "fields" which, concatenated together, form a "field-list". The field-list of a composite type other than the choice type may consist of the fields of several values concatenated together. Either the preamble, length and/or contents of any value may be missing.

Figure 1 – Encoding of a composite value into a field-list

9.5.2 The encoding of a component of a data value either:

a) consists of three parts, as shown in Figure 1, which appear in the following order:

1) a preamble (see clauses 18, 20 and 22);

2) a length determinant (see 10.9);

3) contents; or

b) (where the contents are large) consists of an arbitrary number of parts, as shown in Figure 2, of which the first is a preamble (see clauses 18, 20 and 22) and the following parts are pairs of bit-fields (octet-aligned in the ALIGNED variant), the first being a length determinant for a fragment of the contents, and the second that fragment of the contents; the last pair of fields is identified by the length determinant part, as specified in 10.9.

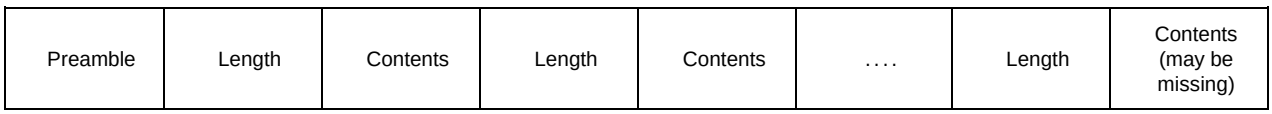


Figure 2 – Encoding of a long data value

9.5.3 Each of the parts mentioned in 9.5.2 generates either:

a) a null field (nothing); or

b) a bit-field (unaligned); or

c) a bit-field (octet-aligned in the ALIGNED variant); or

d) a field-list which may contain either bit-fields (unaligned), bit-fields (octet-aligned in the ALIGNED variant), or both.

9.6 Types to be encoded

9.6.1 The following clauses specify the encoding of the following types into a field-list: boolean, integer, enumerated, real, bitstring, octetstring, null, sequence, sequence-of, set, set-of, choice, open, object identifier, relative object identifier, embedded-pdv, external, restricted character string and unrestricted character string types.

9.6.2 The selection type shall be encoded as an encoding of the selected type.

9.6.3 Encoding of tagged types is not included in this Recommendation | International Standard as, except as stated in 9.2, tagging is not visible in the type and value model used for these encoding rules. Tagged types are thus encoded according to the encoding of the type which has been tagged.

9.6.4 The following "useful types" shall be encoded as if they had been replaced by their definitions given in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 41:

– generalized time;

– universal time;

– object descriptor.

Constraints on the useful types are not PER-visible. The restrictions imposed on the encoding of the generalized time and universal time types by ITU-T Rec. X.690 | ISO/IEC 8825-1, 11.7 and 11.8 shall apply here.

9.6.5 A type defined using a value set assignment shall be encoded as if the type had been defined using the production specified in ITU-T Rec. X.690 | ISO/IEC 8825-1, 15.8.

10 Encoding procedures

10.1 Production of the complete encoding

10.1.1 If an ASN.1 type is encoded using any of the encoding rules identified by the object identifiers listed in clause 29.2 (or by direct textual reference to this Recommendation | International Standard), and the encoding is included in:

a) an ASN.1 bitstring or an ASN.1 octetstring (with or without a contents constraint); or

b) an ASN.1 open type; or

c) any part of an ASN.1 external or embedded pdv type; or

d) any carrier protocol that is not defined using ASN.1

then that ASN.1 type is defined as an outermost type for this application, and clause 10.1.2 shall apply to all encodings of its values.

NOTE 1 – This means that all complete PER encodings (for all variants) that are used in this way are always an integral multiple of eight bits.

NOTE 2 – It is possible using the Encoding Control Notation (see ITU-T Recommendation X.692 | ISO/IEC 8825-3) to specify a variant of PER encodings in which the encoding is not padded to an octet boundary as specified in 10.1.2. Many tools support this option.

NOTE 3 – It is recognized that a carrier protocol not defined using ASN.1 need not explicitly carry the additional zero bits for padding (specified in 10.1.2), but can imply their presence.

10.1.2 The field-list produced as a result of applying this Recommendation | International Standard to an abstract value of an outermost type shall be used to produce the complete encoding of that abstract syntax value as follows: each field in the field-list shall be taken in turn and concatenated to the end of the bit string which is to form the complete encoding of the abstract syntax value preceded by additional zero bits for padding as specified below.

10.1.3 In the UNALIGNED variant of these encoding rules, all fields shall be concatenated without padding. If the result of encoding the outermost value is an empty bit string, the bit string shall be replaced with a single octet with all bits set to 0. If it is a non-empty bit string and it is not a multiple of eight bits, (zero to seven) zero bits shall be appended to it to produce a multiple of eight bits.

10.1.4 In the ALIGNED variant of these encoding rules, any bit-fields in the field-list shall be concatenated without padding, and any octet-aligned bit-fields shall be concatenated after (zero to seven) zero bits have been concatenated to make the length of the encoding produced so far a multiple of eight bits. If the result of encoding the outermost value is an empty bit string, the bit string shall be replaced with a single octet with all bits set to 0. If it is a non-empty bit string and it is not a multiple of eight bits, (zero to seven) zero bits shall be appended to it to produce a multiple of eight bits.

NOTE – The encoding of the outermost value is the empty bit string if, for example, the abstract syntax value is of the null type or of an integer type constrained to a single value.

10.1.5 The resulting bit string is the complete encoding of the abstract syntax value of an outermost type.

10.2 Open type fields

10.2.1 In order to encode an open type field, the value of the actual type occupying the field shall be encoded to a field-list which shall then be converted to a complete encoding of an abstract syntax value as specified in 10.1 to produce an octet string of length "n" (say).

10.2.2 The field-list for the value in which the open type is to be embedded shall then have added to it (as specified in 10.9) an unconstrained length of "n" (in units of octets) and an associated bit-field (octet-aligned in the ALIGNED variant) containing the bits produced in 10.2.1.

NOTE – Where the number of octets in the open type encoding is large, the fragmentation procedures of 10.9 will be used, and the encoding of the open type will be broken without regard to the position of the fragment boundary in the encoding of the type occupying the open type field.

10.3 Encoding as a non-negative-binary-integer

NOTE – (Tutorial) This subclause gives precision to the term "non-negative-binary-integer encoding", putting the integer into a field which is a fixed number of bits, a field which is a fixed number of octets, or a field that is the minimum number of octets needed to hold it.

10.3.1 Subsequent subclauses refer to the generation of a non-negative-binary-integer encoding of a non-negative whole number into a field which is either a bit-field of specified length, a single octet, a double octet, or the minimum number of octets for the value. This subclause (10.3) specifies the precise encoding to be applied when such references are made.

10.3.2 The leading bit of the field is defined as the leading bit of the bit-field, or as the most significant bit of the first octet in the field, and the trailing bit of the field is defined as the trailing bit of the bit-field or as the least significant bit of the last octet in the field.

10.3.3 For the following definition only, the bits shall be numbered zero for the trailing bit of the field, one for the next bit, and so on up to the leading bit of the field.

10.3.4 In a non-negative-binary-integer encoding, the value of the whole number represented by the encoding shall be the sum of the values specified by each bit. A bit which is set to "0" has zero value. A bit with number "n" which is set to "1" has the value 2^n.

10.3.5 The encoding which sums (as defined above) to the value being encoded is an encoding of that value.

NOTE – Where the size of the encoded field is fixed (a bit-field of specified length, a single octet, or a double octet), then there is a unique encoding which sums to the value being encoded.

10.3.6 A minimum octet non-negative-binary-integer encoding of the whole number (which does not predetermine the number of octets to be used for the encoding) has a field which is a multiple of eight bits and also satisfies the condition that the leading eight bits of the field shall not all be zero unless the field is precisely eight bits long.

NOTE – This is a necessary and sufficient condition to produce a unique encoding.

10.4 Encoding as a 2's-complement-binary-integer

NOTE – (Tutorial) This subclause gives precision to the term "2's-complement-binary-integer encoding", putting a signed integer into a field that is the minimum number of octets needed to hold it. These procedures are referenced in later encoding specifications.

10.4.1 Subsequent subclauses refer to the generation of a 2's-complement-binary-integer encoding of a whole number (which may be negative, zero, or positive) into the minimum number of octets for the value. This subclause (10.4) specifies the precise encoding to be applied when such references are made.

10.4.2 The leading bit of the field is defined as the most significant bit of the first octet, and the trailing bit of the field is defined as the least significant bit of the last octet.

10.4.3 For the following definition only, the bits shall be numbered zero for the trailing bit of the field, one for the next bit, and so on up to the leading bit of the field.

10.4.4 In a 2's-complement-binary-integer encoding, the value of the whole number represented by the encoding shall be the sum of the values specified by each bit. A bit which is set to "0" has zero value. A bit with number "n" which is set to "1" has the value 2^n unless it is the leading bit, in which case it has the (negative) value -2^n.

10.4.5 Any encoding which sums (as defined above) to the value being encoded is an encoding of that value.

10.4.6 A minimum octet 2's-complement-binary-integer encoding of the whole number has a field-width that is a multiple of eight bits and also satisfies the condition that the leading nine bits of the field shall not all be zero and shall not all be ones.

NOTE – This is a necessary and sufficient condition to produce a unique encoding.

10.5 Encoding of a constrained whole number

NOTE – (Tutorial) This subclause is referenced by other clauses, and itself references earlier clauses for the production of a non-negative-binary-integer or a 2's-complement-binary-integer encoding. For the UNALIGNED variant the value is always encoded in the minimum number of bits necessary to represent the range (defined in 10.5.3). The rest of this Note addresses the ALIGNED variant. Where the range is less than or equal to 255, the value encodes into a bit-field of the minimum size for the range. Where the range is exactly 256, the value encodes into a single octet octet-aligned bit-field. Where the range is 257 to 64K, the value encodes into a two octet octet-aligned bit-field. Where the range is greater than 64K, the range is ignored and the value encodes into an octet-aligned bit-field which is the minimum number of octets for the value. In this latter case, later procedures (see 10.9) also encode a length field (usually a single octet) to indicate the length of the encoding. For the other cases, the length of the encoding is independent of the value being encoded, and is not explicitly encoded.

10.5.1 This subclause (10.5) specifies a mapping from a constrained whole number into either a bit-field (unaligned) or a bit-field (octet-aligned in the ALIGNED variant), and is invoked by later clauses in this Recommendation | International Standard.

10.5.2 The procedures of this subclause are invoked only if a constrained whole number to be encoded is available, and the values of the lower bound, "lb", and the upper bound, "ub", have been determined from the type notation (after the application of PER-visible constraints).

NOTE – A lower bound cannot be determined if **MIN** evaluates to an infinite number, nor can an upper bound be determined if **MAX** evaluates to an infinite number. For example, no upper or lower bound can be determined for **INTEGER(MIN..MAX)**.

10.5.3 Let "range" be defined as the integer value ("ub" – "lb" + 1), and let the value to be encoded be "n".

10.5.4 If "range" has the value 1, then the result of the encoding shall be an empty bit-field (no bits).

10.5.5 There are five other cases (leading to different encodings) to consider, where one applies to the UNALIGNED variant and four to the ALIGNED variant.

10.5.6 In the case of the UNALIGNED variant the value ("n" – "lb") shall be encoded as a non-negative- binary-integer in a bit-field as specified in 10.3 with the minimum number of bits necessary to represent the range.

NOTE – If "range" satisfies the inequality $2^m <$ "range" $\leq 2^{m+1}$, then the number of bits = m + 1.

10.5.7 In the case of the ALIGNED variant the encoding depends on whether:

a) "range" is less than or equal to 255 (the bit-field case);

b) "range" is exactly 256 (the one-octet case);

c) "range" is greater than 256 and less than or equal to 64K (the two-octet case);

d) "range" is greater than 64K (the indefinite length case).

10.5.7.1 (The bit-field case.) If "range" is less than or equal to 255, then invocation of this subclause requires the generation of a bit-field with a number of bits as specified in the table below, and containing the value ("n" – "lb") as a non-negative-binary-integer encoding in a bit-field as specified in 10.3.

"Range"	Bit-field size (in bits)
2	1
3, 4	2
5, 6, 7, 8	3
9 to 16	4
17 to 32	5
33 to 64	6
65 to 128	7
129 to 255	8

10.5.7.2 (The one-octet case.) If the range has a value of 256, then the value ("n" – "lb") shall be encoded in a one-octet bit-field (octet-aligned in the ALIGNED variant) as a non-negative-binary-integer as specified in 10.3.

10.5.7.3 (The two-octet case.) If the "range" has a value greater than or equal to 257 and less than or equal to 64K, then the value ("n" – "lb") shall be encoded in a two-octet bit-field (octet-aligned in the ALIGNED variant) as a non-negative-binary-integer encoding as specified in 10.3.

10.5.7.4 (The indefinite length case.) Otherwise, the value ("n" – "lb") shall be encoded as a non-negative-binary-integer in a bit-field (octet-aligned in the ALIGNED variant) with the minimum number of octets as specified in 10.3, and the number of octets "len" used in the encoding is used by other clauses that reference this subclause to specify an encoding of the length.

10.6 Encoding of a normally small non-negative whole number

NOTE – (Tutorial) This procedure is used when encoding a non-negative whole number that is expected to be small, but whose size is potentially unlimited due to the presence of an extension marker. An example is a choice index.

10.6.1 If the non-negative whole number, "n", is less than or equal to 63, then a single-bit bit-field shall be appended to the field-list with the bit set to 0, and "n" shall be encoded as a non-negative-binary-integer into a 6-bit bit-field.

10.6.2 If "n" is greater than or equal to 64, a single-bit bit-field with the bit set to 1 shall be appended to the field-list. The value "n" shall then be encoded as a semi-constrained whole number with "lb" equal to 0 and the procedures of 10.9 shall be invoked to add it to the field-list preceded by a length determinant.

10.7 Encoding of a semi-constrained whole number

NOTE – (Tutorial) This procedure is used when a lower bound can be identified but not an upper bound. The encoding procedure places the offset from the lower bound into the minimum number of octets as a non-negative-binary-integer, and requires an explicit length encoding (typically a single octet) as specified in later procedures.

10.7.1 This subclause specifies a mapping from a semi-constrained whole number into a bit-field (octet-aligned in the ALIGNED variant), and is invoked by later clauses in this Recommendation | International Standard.

10.7.2 The procedures of this subclause (10.7) are invoked only if a semi-constrained whole number ("n" say) to be encoded is available, and the value of "lb" has been determined from the type notation (after the application of PER-visible constraints).

NOTE – A lower bound cannot be determined if **MIN** evaluates to an infinite number. For example, no lower bound can be determined for **INTEGER(MIN..MAX)**.

10.7.3 The procedures of this subclause always produce the indefinite length case.

10.7.4 (The indefinite length case.) The value ("n" – "lb") shall be encoded as a non-negative-binary-integer in a bit-field (octet-aligned in the ALIGNED variant) with the minimum number of octets as specified in 10.3, and the number of octets "len" used in the encoding is used by other clauses that reference this subclause to specify an encoding of the length.

10.8 Encoding of an unconstrained whole number

NOTE – (Tutorial) This case only arises in the encoding of the value of an integer type with no lower bound. The procedure encodes the value as a 2's-complement-binary-integer into the minimum number of octets required to accommodate the encoding, and requires an explicit length encoding (typically a single octet) as specified in later procedures.

10.8.1 This subclause (10.8) specifies a mapping from an unconstrained whole number ("n" say) into a bit-field (octet-aligned in the ALIGNED variant), and is invoked by later clauses in this Recommendation | International Standard.

10.8.2 The procedures of this subclause always produce the indefinite length case.

10.8.3 (The indefinite length case.) The value "n" shall be encoded as a 2's-complement-binary-integer in a bit-field (octet-aligned in the ALIGNED variant) with the minimum number of octets as specified in 10.4, and the number of octets "len" used in the encoding is used by other clauses that reference this subclause to specify an encoding of the length.

10.9 General rules for encoding a length determinant

NOTE 1 – (Tutorial) The procedures of this subclause are invoked when an explicit length field is needed for some part of the encoding regardless of whether the length count is bounded above (by PER-visible constraints) or not. The part of the encoding to which the length applies may be a bit string (with the length count in bits), an octet string (with the length count in octets), a known-multiplier character string (with the length count in characters), or a list of fields (with the length count in components of a sequence-of or set-of).

NOTE 2 – (Tutorial) In the case of the ALIGNED variant if the length count is bounded above by an upper bound that is less than 64K, then the constrained whole number encoding is used for the length. For sufficiently small ranges the result is a bit-field, otherwise the unconstrained length ("n" say) is encoded into an octet-aligned bit-field in one of three ways (in order of increasing size):

a) ("n" less than 128) a single octet containing "n" with bit 8 set to zero;

b) ("n" less than 16K) two octets containing "n" with bit 8 of the first octet set to 1 and bit 7 set to zero;

c) (large "n") a single octet containing a count "m" with bit 8 set to 1 and bit 7 set to 1. The count "m" is one to four, and the length indicates that a fragment of the material follows (a multiple "m" of 16K items). For all values of "m", the fragment is then followed by another length encoding for the remainder of the material.

NOTE 3 – (Tutorial) In the UNALIGNED variant, if the length count is bounded above by an upper bound that is less than 64K, then the constrained whole number encoding is used to encode the length in the minimum number of bits necessary to represent the range. Otherwise, the unconstrained length ("n" say) is encoded into a bit-field in the manner described above in Note 2.

10.9.1 This subclause is not invoked if, in accordance with the specification of later clauses, the value of the length determinant, "n", is fixed by the type definition (constrained by PER-visible constraints) to a value less than 64K.

10.9.2 This subclause is invoked for addition to the field-list of a field, or list of fields, preceded by a length determinant "n" which determines either:

a) the length in octets of an associated field (units are octets); or

b) the length in bits of an associated field (units are bits); or

c) the number of component encodings in an associated list of fields (units are components of a set-of or sequence-of); or

d) the number of characters in the value of an associated known-multiplier character string type (units are characters).

10.9.3 (ALIGNED variant) The procedures for the ALIGNED variant are specified in 10.9.3.1 to 10.9.3.8.4. (The procedures for the UNALIGNED variant are specified in 10.9.4.)

10.9.3.1 As a result of the analysis of the type definition (specified in later clauses) the length determinant (a whole number "n") will have been determined to be either:

a) a normally small length with a lower bound "lb" equal to one; or

b) a constrained whole number with a lower bound "lb" (greater than or equal to zero), and an upper bound "ub" less than 64K; or

c) a semi-constrained whole number with a lower bound "lb" (greater than or equal to zero), or a constrained whole number with a lower bound "lb" (greater than or equal to zero) and an upper bound "ub" greater than or equal to 64K.

10.9.3.2 The subclauses invoking the procedures of this subclause will have determined a value for "lb", the lower bound of the length (this is zero if the length is unconstrained), and for "ub", the upper bound of the length. "ub" is unset if there is no upper bound determinable from PER-visible constraints.

10.9.3.3 Where the length determinant is a constrained whole number with "ub" less than 64K, then the field-list shall have appended to it the encoding of the constrained whole number for the length determinant as specified in 10.5. If "n" is non-zero, this shall be followed by the associated field or list of fields, completing these procedures. If "n" is zero there shall be no further addition to the field-list, completing these procedures.

NOTE 1 – For example:

```
A ::= IA5String (SIZE (3..6))           -- Length is encoded in a 2-bit bit-field.
B ::= IA5String (SIZE (40000..40254))   -- Length is encoded in an 8-bit bit-field.
C ::= IA5String (SIZE (0..32000))       -- Length is encoded in a 2-octet
                                        -- bit-field (octet-aligned in the ALIGNED variant).
D ::= IA5String (SIZE (64000))          -- Length is not encoded.
```

NOTE 2 – The effect of making no addition in the case of "n" equals zero is that padding to an octet boundary does not occur when these procedures are invoked to add an octet-aligned-bit-field of zero length, unless required by 10.5.

10.9.3.4 Where the length determinant is a normally small length and "n" is less than or equal to 64, a single-bit bit-field shall be appended to the field-list with the bit set to 0, and the value "n-1" shall be encoded as a non-negative-binary-integer into a 6-bit bit-field. This shall be followed by the associated field, completing these procedures. If "n" is greater than 64, a single-bit bit-field shall be appended to the field-list with the bit set to 1, followed by the encoding of "n" as an unconstrained length determinant followed by the associated field, according to the procedures of 10.9.3.5 to 10.9.3.8.4.

NOTE – Normally small lengths are only used to indicate the length of the bitmap that prefixes the extension addition values of a set or sequence type.

10.9.3.5 Otherwise (unconstrained length, or large "ub"), "n" is encoded and appended to the field-list followed by the associated fields as specified below.

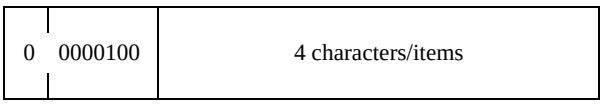
NOTE – The lower bound, "lb", does not affect the length encodings specified in 10.9.3.6 to 10.9.3.8.4.

10.9.3.6 If "n" is less than or equal to 127, then "n" shall be encoded as a non-negative-binary-integer (using the procedures of 10.3) into bits 7 (most significant) to 1 (least significant) of a single octet and bit 8 shall be set to zero. This shall be appended to the field-list as a bit-field (octet-aligned in the ALIGNED variant) followed by the associated field or list of fields, completing these procedures.

NOTE – For example, if in the following a value of **A** is 4 characters long, and that of **B** is 4 items long:

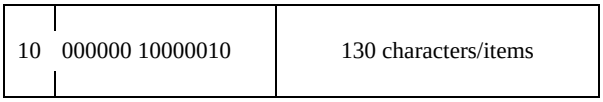
```
A ::= IA5String
B ::= SEQUENCE (SIZE (4..123456)) OF INTEGER
```

both values are encoded with the length octet occupying one octet, and with the most significant set to 0 to indicate that the length is less than or equal to 127:



10.9.3.7 If "n" is greater than 127 and less than 16K, then "n" shall be encoded as a non-negative-binary-integer (using the procedures of 10.3) into bit 6 of octet one (most significant) to bit 1 of octet two (least significant) of a two-octet bit-field (octet-aligned in the ALIGNED variant) with bit 8 of the first octet set to 1 and bit 7 of the first octet set to zero. This shall be appended to the field-list followed by the associated field or list of fields, completing these procedures.

NOTE – If in the example of 10.9.3.6 a value of **A** is 130 characters long, and a value of **B** is 130 items long, both values are encoded with the length component occupying 2 octets, and with the two most significant bits (bits 8 and 7) of the octet set to 10 to indicate that the length is greater than 127 but less than 16K.



Length Value

10.9.3.8 If "n" is greater than or equal to 16K, then there shall be appended to the field-list a single octet in a bit-field (octet-aligned in the ALIGNED variant) with bit 8 set to 1 and bit 7 set to 1, and bits 6 to 1 encoding the value 1, 2, 3 or 4 as a non-negative-binary-integer (using the procedures of 10.8). This single octet shall be followed by part of the associated field or list of fields, as specified below.

NOTE – The value of bits 6 to 1 is restricted to 1-4 (instead of the theoretical limits of 0-63) so as to limit the number of items that an implementation has to have knowledge of to a more manageable number (64K instead of 1024K).

10.9.3.8.1 The value of bits 6 to 1 (1 to 4) shall be multiplied by 16K giving a count ("m" say). The choice of the integer in bits 6 to 1 shall be the maximum allowed value such that the associated field or list of fields contains more than or exactly "m" octets, bits, components or characters, as appropriate.

NOTE 1 – The unfragmented form handles lengths up to 16K. The fragmentation therefore provides for lengths up to 64K with a granularity of 16K.

NOTE 2 – If in the example of 10.9.3.6 a value of "B" is 144K + 1 (i.e., 64K + 64K + 16K + 1) items long, the value is fragmented, with the two most significant bits (bits 8 and 7) of the first three fragments set to 11 to indicate that one to four blocks each of 16K items follow, and that another length component will follow the last block of each fragment:

11 000100	64K items	11 000100	64K items	11 000001	16K items	0 0000001	1 item
Length	Value	Length	Value	Length	Value	Length	Value

10.9.3.8.2 That part of the contents specified by "m" shall then be appended to the field-list as either:

a) a single bit-field (octet-aligned in the ALIGNED variant) of "m" octets containing the first "m" octets of the associated field, for units which are octets; or

b) a single bit-field (octet-aligned in the ALIGNED variant) of "m" bits containing the first "m" bits of the associated field, for units which are bits; or

c) the list of fields encoding the first "m" components in the associated list of fields, for units which are components of a set-of or sequence-of types; or

d) a single bit-field (octet-aligned in the ALIGNED variant) of "m" characters containing the first "m" characters of the associated field, for units which are characters.

10.9.3.8.3 The procedures of 10.9 shall then be reapplied to add the remaining part of the associated field or list of fields to the field-list with a length which is a semi-constrained whole number equal to ("n" – "m") with a lower bound of zero.

NOTE – If the last fragment that contains part of the encoded value has a length that is an exact multiple of 16K, it is followed by a final fragment that consists only of a single octet length component set to 0.

10.9.3.8.4 The addition of only a part of the associated field(s) to the field-list with reapplication of these procedures is called *the fragmentation procedure*.

10.9.4 (UNALIGNED variant) The procedures for the UNALIGNED variant are specified in 10.9.4.1 to 10.9.4.2 (the procedures for the ALIGNED variant are specified in 10.9.3).

10.9.4.1 If the length determinant "n" to be encoded is a constrained whole number with "ub" less than 64K, then ("n"–"lb") shall be encoded as a non-negative-binary-integer (as specified in 10.3) using the minimum number of bits necessary to encode the "range" ("ub" – "lb" + 1), unless "range" is 1, in which case there shall be no length encoding. If "n" is non-zero this shall be followed by an associated field or list of fields, completing these procedures. If "n" is zero there shall be no further addition to the field-list, completing these procedures.

NOTE – If "range" satisfies the inequality $2^m <$ "range" $\leq 2^{m+1}$, then the number of bits in the length determinant is m + 1.

10.9.4.2 If the length determinant "n" to be encoded is a normally small length, or a constrained whole number with "ub" greater than or equal to 64K, or is a semi-constrained whole number, then "n" shall be encoded as specified in 10.9.3.4 to 10.9.3.8.4.

NOTE – Thus, if "ub" is greater than or equal to 64K, the encoding of the length determinant is the same as it would be if the length were unconstrained.

11 Encoding the boolean type

11.1 A value of the boolean type shall be encoded as a bit-field consisting of a single bit.

11.2 The bit shall be set to 1 for **`TRUE`** and 0 for **`FALSE`**.

11.3 The bit-field shall be appended to the field-list with no length determinant.

12 Encoding the integer type

NOTE 1 – (Tutorial ALIGNED variant) Ranges which allow the encoding of all values into one octet or less go into a minimum-sized bit-field with no length count. Ranges which allow encoding of all values into two octets go into two octets in an octet-aligned bit-field with no length count. Otherwise, the value is encoded into the minimum number of octets (using non-negative-binary-integer or 2's-complement-binary-integer encoding as appropriate) and a length determinant is added. In this case, if the integer value can be encoded in less than 127 octets (as an offset from any lower bound that might be determined), and there is no finite upper and lower bound, there is a one-octet length determinant, else the length is encoded in the fewest number of bits needed. Other cases are not of any practical interest, but are specified for completeness.

NOTE 2 – (Tutorial UNALIGNED variant) Constrained integers are encoded in the fewest number of bits necessary to represent the range regardless of its size. Unconstrained integers are encoded as in Note 1.

12.1 If an extension marker is present in the constraint specification of the integer type, then a single bit shall be added to the field-list in a bit-field of length one. The bit shall be set to 1 if the value to be encoded is not within the range of the extension root, and zero otherwise. In the former case, the value shall be added to the field-list as an unconstrained integer value, as specified in 12.2.4 to 12.2.6, completing this procedure. In the latter case, the value shall be encoded as if the extension marker is not present.

12.2 If an extension marker is not present in the constraint specification of the integer type, then the following applies.

12.2.1 If PER-visible constraints restrict the integer value to a single value, then there shall be no addition to the field-list, completing these procedures.

12.2.2 If PER-visible constraints restrict the integer value to be a constrained whole number, then it shall be converted to a field according to the procedures of 10.5 (encoding of a constrained whole number), and the procedures of 12.2.5 to 12.2.6 shall then be applied.

12.2.3 If PER-visible constraints restrict the integer value to be a semi-constrained whole number, then it shall be converted to a field according to the procedures of 10.7 (encoding of a semi-constrained whole number), and the procedures of 12.2.6 shall then be applied.

12.2.4 If PER-visible constraints do not restrict the integer to be either a constrained or a semi-constrained whole number, then it shall be converted to a field according to the procedures of 10.8 (encoding of an unconstrained whole number), and the procedures of 12.2.6 shall then be applied.

12.2.5 If the procedures invoked to encode the integer value into a field did not produce the indefinite length case (see 10.5.7.4 and 10.8.2), then that field shall be appended to the field-list completing these procedures.

12.2.6 Otherwise, (the indefinite length case) the procedures of 10.9 shall be invoked to append the field to the field-list preceded by one of the following:

a) A constrained length determinant "len" (as determined by 10.5.7.4) if PER-visible constraints restrict the type with finite upper and lower bounds and, if the type is extensible, the value lies within the range of the extension root. The lower bound "lb" used in the length determinant shall be 1, and the upper bound "ub" shall be the count of the number of octets required to hold the range of the integer value.

   NOTE – The encoding of the value "**`foo INTEGER (256..1234567) ::= 256`**" would thus be encoded as 00xxxxxx00000000, where each 'x' represents a zero pad bit that may or may not be present depending on where within the octet the length occurs (e.g. the encoding is 00 xxxxxx 00000000 if the length starts on an octet boundary, and 00 00000000 if it starts with the two least signigicant bits (bits 2 and 1) of an octet).

b) An unconstrained length determinant equal to "len" (as determined by 10.7 and 10.8) if PER-visible constraints do not restrict the type with finite upper and lower bounds, or if the type is extensible and the value does not lie within the range of the extension root.

13 Encoding the enumerated type

NOTE – (Tutorial) An enumerated type without an extension marker is encoded as if it were a constrained integer whose subtype constraint does not contain an extension marker. This means that an enumerated type will almost always in practice be encoded as a bit-field in the smallest number of bits needed to express every enumeration. In the presence of an extension marker, it is encoded as a normally small non-negative whole number if the value is not in the extension root.

13.1 The enumerations in the enumeration root shall be sorted into ascending order by their enumeration value, and shall then be assigned an enumeration index starting with zero for the first enumeration, one for the second, and so on up to the last enumeration in the sorted list. The extension additions (which are always defined in ascending order) shall be assigned an enumeration index starting with zero for the first enumeration, one for the second, and so on up to the last enumeration in the extension additions.

NOTE – ITU-T Rec. X.680 | ISO/IEC 8824-1 requires that each successive extension addition shall have a greater enumeration value than the last.

13.2 If the extension marker is absent in the definition of the enumerated type, then the enumeration index shall be encoded. Its encoding shall be as though it were a value of a constrained integer type for which there is no extension marker present, where the lower bound is 0 and the upper bound is the largest enumeration index associated with the type, completing this procedure.

13.3 If the extension marker is present, then a single bit shall be added to the field-list in a bit-field of length one. The bit shall be set to 1 if the value to be encoded is not within the extension root, and zero otherwise. In the former case, the enumeration additions shall be sorted according to 13.1 and the value shall be added to the field-list as a normally small non-negative whole number whose value is the enumeration index of the additional enumeration and with "lb" set to 0, completing this procedure. In the latter case, the value shall be encoded as if the extension marker is not present, as specified in 13.2.

NOTE – There are no PER-visible constraints that can be applied to an enumerated type that are visible to these encoding rules.

14 Encoding the real type

NOTE – (Tutorial) A real uses the contents octets of CER/DER preceded by a length determinant that will in practice be a single octet.

14.1 If the base of the abstract value is 10, then the base of the encoded value shall be 10, and if the base of the abstract value is 2 the base of the encoded value shall be 2.

14.2 The encoding of **REAL** specified for CER and DER in ITU-T Rec. X.690 | ISO/IEC 8825-1 shall be applied to give a bit-field (octet-aligned in the ALIGNED variant) which is the contents octets of the CER/DER encoding. The contents octets of this encoding consists of "n" (say) octets and is placed in a bit-field (octet-aligned in the ALIGNED variant) of "n" octets. The procedures of 10.9 shall be invoked to append this bit-field (octet-aligned in the ALIGNED variant) of "n" octets to the field-list, preceded by an unconstrained length determinant equal to "n".

15 Encoding the bitstring type

NOTE – (Tutorial) Bitstrings constrained to a fixed length less than or equal to 16 bits do not cause octet alignment. Larger bitstrings are octet-aligned in the ALIGNED variant. If the length is fixed by constraints and the upper bound is less than 64K, there is no explicit length encoding, otherwise a length encoding is included which can take any of the forms specified earlier for length encodings, including fragmentation for large bit strings.

15.1 PER-visible constraints can only constrain the length of the bitstring.

15.2 Where there are no PER-visible constraints and ITU-T Rec. X.680 | ISO/IEC 8824-1, 21.7 applies, the value shall be encoded with no trailing 0 bits (note that this means that a value with no 1 bits is always encoded as an empty bit string).

15.3 Where there is a PER-visible constraint and ITU-T Rec. X.680 | ISO/IEC 8824-1, 21.7 applies (i.e. the bitstring type is defined with a "NamedBitList"), the value shall be encoded with trailing 0 bits added or removed as necessary to ensure that the size of the transmitted value is the smallest size capable of carrying this value and satisfies the effective size constraint.

15.4 Let the maximum number of bits in the bitstring (as determined by PER-visible constraints on the length) be "ub" and the minimum number of bits be "lb". If there is no finite maximum we say that "ub" is unset. If there is no constraint on the minimum, then "lb" has the value zero. Let the length of the actual bit string value to be encoded be "n" bits.

15.5 When a bitstring value is placed in a bit-field as specified in 15.6 to 15.11, the leading bit of the bitstring value shall be placed in the leading bit of the bit-field, and the trailing bit of the bitstring value shall be placed in the trailing bit of the bit-field.

15.6 If an extension marker is present in the size constraint specification of the bitstring type, a single bit shall be added to the field-list in a bit-field of length one. The bit shall be set to 1 if the length of this encoding is not within the range of the extension root, and zero otherwise. In the former case, 15.11 shall be invoked to add the length as a semi-constrained whole number to the field-list, followed by the bitstring value. In the latter case the length and value shall be encoded as if the extension marker is not present.

15.7 If an extension marker is not present in the constraint specification of the bitstring type, then 15.8 to 15.11 apply.

15.8 If the bitstring is constrained to be of zero length ("ub" equals zero), then it shall not be encoded (no additions to the field-list), completing the procedures of this clause.

15.9 If all values of the bitstring are constrained to be of the same length ("ub" equals "lb") and that length is less than or equal to sixteen bits, then the bitstring shall be placed in a bit-field of the constrained length "ub" which shall be appended to the field-list with no length determinant, completing the procedures of this clause.

15.10 If all values of the bitstring are constrained to be of the same length ("ub" equals "lb") and that length is greater than sixteen bits but less than 64K bits, then the bitstring shall be placed in a bit-field (octet-aligned in the ALIGNED variant) of length "ub" (which is not necessarily a multiple of eight bits) and shall be appended to the field-list with no length determinant, completing the procedures of this clause.

15.11 If 15.8-15.10 do not apply, the bitstring shall be placed in a bit-field (octet-aligned in the ALIGNED variant) of length "n" bits and the procedures of 10.9 shall be invoked to add this bit-field (octet-aligned in the ALIGNED variant) of "n" bits to the field-list, preceded by a length determinant equal to "n" bits as a constrained whole number if "ub" is set and is less than 64K or as a semi-constrained whole number if "ub" is unset. "lb" is as determined above.

NOTE – Fragmentation applies for unconstrained or large "ub" after 16K, 32K, 48K or 64K bits.

16 Encoding the octetstring type

NOTE – Octet strings of fixed length less than or equal to two octets are not octet-aligned. All other octet strings are octet-aligned in the ALIGNED variant. Fixed length octet strings encode with no length octets if they are shorter than 64K. For unconstrained octet strings the length is explicitly encoded (with fragmentation if necessary).

16.1 PER-visible constraints can only constrain the length of the octetstring.

16.2 Let the maximum number of octets in the octetstring (as determined by PER-visible constraints on the length) be "ub" and the minimum number of octets be "lb". If there is no finite maximum we say that "ub" is unset. If there is no constraint on the minimum then "lb" has the value zero. Let the length of the actual octetstring value to be encoded be "n" octets.

16.3 If there is a PER-visible size constraint and an extension marker is present in it, a single bit shall be added to the field-list in a bit-field of length one. The bit shall be set to 1 if the length of this encoding is not within the range of the extension root, and zero otherwise. In the former case 16.8 shall be invoked to add the length as a semi-constrained whole number to the field-list, followed by the octetstring value. In the latter case the length and value shall be encoded as if the extension marker is not present.

16.4 If an extension marker is not present in the constraint specification of the octetstring type, then 16.5 to 16.8 apply.

16.5 If the octetstring is constrained to be of zero length ("ub" equals zero), then it shall not be encoded (no additions to the field-list), completing the procedures of this clause.

16.6 If all values of the octetstring are constrained to be of the same length ("ub" equals "lb") and that length is less than or equal to two octets, the octetstring shall be placed in a bit-field with a number of bits equal to the constrained length "ub" multiplied by eight which shall be appended to the field-list with no length determinant, completing the procedures of this clause.

16.7 If all values of the octetstring are constrained to be of the same length ("ub" equals "lb") and that length is greater than two octets but less than 64K, then the octetstring shall be placed in a bit-field (octet-aligned in the ALIGNED variant) with the constrained length "ub" octets which shall be appended to the field-list with no length determinant, completing the procedures of this clause.

16.8 If 16.5 to 16.7 do not apply, the octetstring shall be placed in a bit-field (octet-aligned in the ALIGNED variant) of length "n" octets and the procedures of 10.9 shall be invoked to add this bit-field (octet-aligned in the ALIGNED variant) of "n" octets to the field-list, preceded by a length determinant equal to "n" octets as a constrained whole number if "ub" is set, and as a semi-constrained whole number if "ub" is unset. "lb" is as determined above.

NOTE – The fragmentation procedures may apply after 16K, 32K, 48K, or 64K octets.

17 Encoding the null type

NOTE – (Tutorial) The null type is essentially a place holder, with practical meaning only in the case of a choice or an optional set or sequence component. Identification of the null in a choice, or its presence as an optional element, is performed in these encoding rules without the need to have octets representing the null. Null values therefore never contribute to the octets of an encoding.

There shall be no addition to the field-list for a null value.

18 Encoding the sequence type

NOTE – (Tutorial) A sequence type begins with a preamble which is a bit-map. If the sequence type has no extension marker, then the bit-map merely records the presence or absence of default and optional components in the type, encoded as a fixed length bit-field. If the sequence type does have an extension marker, then the bit-map is preceded by a single bit that says whether values of extension additions are actually present in the encoding. The preamble is encoded without any length determinant provided it is less than 64K bits long, otherwise a length determinant is encoded to obtain fragmentation. The preamble is followed by the fields that encode each of the components, taken in turn. If there are extension additions, then immediately before the first one is encoded there is the encoding (as a normally small length) of a count of the number of extension additions in the type being encoded, followed by a bit-map equal in length to this count which records the presence or absence of values of each extension addition. This is followed by the encodings of the extension additions as if each one was the value of an open type field.

18.1 If the sequence type has an extension marker, then a single bit shall first be added to the field-list in a bit-field of length one. The bit shall be one if values of extension additions are present in this encoding, and zero otherwise. (This bit is called the "extension bit" in the following text.) If there is no extension marker, there shall be no extension bit added.

18.2 If the sequence type has "n" components in the extension root that are marked **`OPTIONAL`** or **`DEFAULT`**, then a single bit-field with "n" bits shall be produced for addition to the field-list. The bits of the bit-field shall, taken in order, encode the presence or absence of an encoding of each optional or default component in the sequence type. A bit value of 1 shall encode the presence of the encoding of the component, and a bit value of 0 shall encode the absence of the encoding of the component. The leading bit in the preamble shall encode the presence or absence of the first optional or default component, and the trailing bit shall encode the presence or absence of the last optional or default component.

18.3 If "n" is less than 64K, the bit-field shall be appended to the field-list. If "n" is greater than or equal to 64K, then the procedures of 10.9 shall be invoked to add this bit-field of "n" bits to the field-list, preceded by a length determinant equal to "n" bits as a constrained whole number with "ub" and "lb" both set to "n".

NOTE – In this case, "ub" and "lb" will be ignored by the length procedures. These procedures are invoked here in order to provide fragmentation of a large preamble. The situation is expected to arise only rarely.

18.4 The preamble shall be followed by the field-lists of each of the components of the sequence value which are present, taken in turn.

18.5 For CANONICAL-PER, encodings of components marked **`DEFAULT`** shall always be absent if the value to be encoded is the default value. For BASIC-PER, encodings of components marked **`DEFAULT`** shall always be absent if the value to be encoded is the default value of a simple type (see 3.6.25), otherwise it is a sender's option whether or not to encode it.

18.6 This completes the encoding if the extension bit is absent or is zero. If the extension bit is present and set to one, then the following procedures apply.

18.7 Let the number of extension additions in the type being encoded be "n", then a bit-field with "n" bits shall be produced for addition to the field-list. The bits of the bit-field shall, taken in order, encode the presence or absence of an encoding of each extension addition in the type being encoded. A bit value of 1 shall encode the presence of the encoding of the extension addition, and a bit value of 0 shall encode the absence of the encoding of the extension addition. The leading bit in the bit-field shall encode the presence or absence of the first extension addition, and the trailing bit shall encode the presence or absence of the last extension addition.

NOTE – If conformance is claimed to a particular version of a specification, then the value "n" is always equal to the number of extension additions in that version.

18.8 The procedures of 10.9 shall be invoked to add this bit-field of "n" bits to the field-list, preceded by a length determinant equal to "n" as a normally small length.

NOTE – "n" cannot be zero, as this procedure is only invoked if there is at least one extension addition being encoded.

18.9 This shall be followed by field-lists containing the encodings of each extension addition that is present, taken in turn. Each extension addition that is a "ComponentType" (i.e., not an "ExtensionAdditionGroup") shall be encoded as if it were the value of an open type field as specified in 10.2.1. Each extension addition that is an "ExtensionAdditionGroup" shall be encoded as a sequence type as specified in 18.2 to 18.6, which is then encoded as if it were the value of an open type field as specified in 10.2.1. If all components values of the "ExtensionAdditionGroup" are missing then, the "ExtensionAdditionGroup" shall be encoded as a missing extension addition (i.e., the corresponding bit in the bit-field described in 18.7 shall be set to 0).

NOTE 1 – If an "ExtensionAdditionGroup" contains components marked **OPTIONAL** or **DEFAULT**, then the "ExtensionAdditionGroup" is prefixed with a bit-map that indicates the presence/absence of values for each component marked **OPTIONAL** or **DEFAULT**.

NOTE 2 – "RootComponentTypeList" components that are defined after the extension marker pair are encoded as if they were defined immediately before the extension marker pair.

19 Encoding the sequence-of type

19.1 PER-visible constraints can constrain the number of components of the sequence-of type.

19.2 Let the maximum number of components in the sequence-of (as determined by PER-visible constraints) be "ub" components and the minimum number of components be "lb". If there is no finite maximum or "ub" is greater than or equal to 64K we say that "ub" is unset. If there is no constraint on the minimum, then "lb" has the value zero. Let the number of components in the actual sequence-of value to be encoded be "n" components.

19.3 The encoding of each component of the sequence-of will generate a number of fields to be appended to the field-list for the sequence-of type.

19.4 If there is a PER-visible constraint and an extension marker is present in it, a single bit shall be added to the field-list in a bit-field of length one. The bit shall be set to 1 if the number of components in this encoding is not within the range of the extension root, and zero otherwise. In the former case 10.9 shall be invoked to add the length determinant as a semi-constrained whole number to the field-list, followed by the component values. In the latter case the length and value shall be encoded as if the extension marker is not present.

19.5 If the number of components is fixed ("ub" equals "lb") and "ub" is less than 64K, then there shall be no length determinant for the sequence-of, and the fields of each component shall be appended in turn to the field-list of the sequence-of.

19.6 Otherwise, the procedures of 10.9 shall be invoked to add the list of fields generated by the "n" components to the field-list, preceded by a length determinant equal to "n" components as a constrained whole number if "ub" is set, and as a semi-constrained whole number if "ub" is unset. "lb" is as determined above.

NOTE 1 – The fragmentation procedures may apply after 16K, 32K, 48K, or 64K components.

NOTE 2 – The break-points for fragmentation are between fields. The number of bits prior to a break-point are not necessarily a multiple of eight.

20 Encoding the set type

The set type shall have the elements in its "RootComponentTypeList" sorted into the canonical order specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 8.6 and additionally for the purposes of determining the order in which components are encoded when one or more component is an untagged choice type, each untagged choice type is ordered as though it has a tag equal to that of the smallest tag in the "RootAlternativeTypeList" of that choice type or any untagged choice types nested within. The set elements that occur in the "RootComponentTypeList" shall then be encoded as if it had been declared a sequence type. The set elements that occur in the "ExtensionAdditionList" shall be encoded as though they were components of a sequence type as specified in 18.9 (i.e., they are encoded in the order in which they are defined).

EXAMPLE – In the following which assumes a tagging environment of **IMPLICIT TAGS**:

```
A ::= SET
{
      a     [3] INTEGER,
      b     [1] CHOICE
      {
```

```
                c     [2] INTEGER,
                d     [4] INTEGER
          },
          e     CHOICE
          {
                f     CHOICE
                {
                      g     [5] INTEGER,
                      h     [6] INTEGER
                },
                i           CHOICE
                {
                      j     [0] INTEGER
                }
          }
}
```

the order in which the components of the set are encoded will always be **e**, **b**, **a**, since the tag **[0]** sorts lowest, then **[1]**, then **[3]**.

21 Encoding the set-of type

21.1 For CANONICAL-PER the encoding of the component values of the set-of type shall appear in ascending order, the component encodings being compared as bit strings padded at their trailing ends with as many as seven 0 bits to an octet boundary, and with 0-octets added to the shorter one if necessary to make the length equal to that of the longer one.

NOTE – Any pad bits or pad octets added for the sort do not appear in the actual encoding.

21.2 For BASIC-PER the set-of shall be encoded as if it had been declared a sequence-of type.

22 Encoding the choice type

NOTE – (Tutorial) A choice type is encoded by encoding an index specifying the chosen alternative. This is encoded as for a constrained integer (unless the extension marker is present in the choice type, in which case it is a normally small non-negative whole number) and would therefore typically occupy a fixed length bit-field of the minimum number of bits needed to encode the index. (Although it could in principle be arbitrarily large.) This is followed by the encoding of the chosen alternative, with alternatives that are extension additions encoded as if they were the value of an open type field. Where the choice has only one alternative, there is no encoding for the index.

22.1 Encoding of choice types are not affected by PER-visible constraints.

22.2 Each component of a choice has an index associated with it which has the value zero for the first alternative in the root of the choice (taking the alternatives in the canonical order specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 8.6), one for the second, and so on up to the last component in the extension root of the choice. An index value is similarly assigned to each "NamedType" within the "ExtensionAdditionAlternativesList", starting with 0 just as with the components of the extension root. Let "n" be the value of the largest index in the root.

NOTE – ITU-T Rec. X.680 | ISO/IEC 8824-1, 28.4, requires that each successive extension addition shall have a greater tag value than the last added to the "ExtensionAdditionAlternativesList".

22.3 For the purposes of canonical ordering of choice alternatives that contain an untagged choice, each untagged choice type shall be ordered as though it has a tag equal to that of the smallest tag in the extension root of either that choice type or any untagged choice types nested within.

22.4 If the choice has only one alternative in the extension root, there shall be no encoding for the index if that alternative is chosen.

22.5 If the choice type has an extension marker, then a single bit shall first be added to the field-list in a bit-field of length one. The bit shall be 1 if a value of an extension addition is present in the encoding, and zero otherwise. (This bit is called the "extension bit" in the following text.) If there is no extension marker, there shall be no extension bit added.

22.6 If the extension bit is absent, then the choice index of the chosen alternative shall be encoded into a field according to the procedures of clause 12 as if it were a value of an integer type (with no extension marker in its subtype constraint) constrained to the range 0 to "n", and that field shall be appended to the field-list. This shall then be followed by the fields of the chosen alternative, completing the procedures of this clause.

22.7 If the extension bit is present and the chosen alternative lies within the extension root, the choice index of the chosen alternative shall be encoded as if the extension marker is absent, according to the procedure of clause 12, completing the procedures of this clause.

22.8 If the extension bit is present and the chosen alternative does not lie within the extension root, the choice index of the chosen alternative shall be encoded as a normally small non-negative whole number with "lb" set to 0 and that field shall be appended to the field-list. This shall then be followed by a field-list containing the encoding of the chosen alternative encoded as if it were the value of an open type field as specified in 10.2, completing the procedures of this clause.

NOTE – Version brackets in the definition of choice extension additions have no effect on how "ExtensionAdditionAlternatives" are encoded.

23 Encoding the object identifier type

NOTE – (Tutorial) An object identifier type encoding uses the contents octets of BER preceded by a length determinant that will in practice be a single octet.

The encoding specified for BER shall be applied to give a bit-field (octet-aligned in the ALIGNED variant) which is the contents octets of the BER encoding. The contents octets of this BER encoding consists of "n" (say) octets and is placed in a bit-field (octet-aligned in the ALIGNED variant) of "n" octets. The procedures of 10.9 shall be invoked to append this bit-field (octet-aligned in the ALIGNED variant) to the field-list, preceded by a length determinant equal to "n" as a semi-constrained whole number octet count.

24 Encoding the relative object identifier type

NOTE – (Tutorial) A relative object identifier type encoding uses the contents octets of BER preceded by a length determinant that will in practice be a single octet. The following text is identical to that of clause 23.

The encoding specified for BER shall be applied to give a bit-field (octet-aligned in the ALIGNED variant) which is the contents octets of the BER encoding. The contents octets of this BER encoding consists of "n" (say) octets and is placed in a bit-field (octet-aligned in the ALIGNED variant) of "n" octets. The procedures of 10.9 shall be invoked to append this bit-field (octet-aligned in the ALIGNED variant) to the field-list, preceded by a length determinant equal to "n" as a semi-constrained whole number octet count.

25 Encoding the embedded-pdv type

25.1 There are two ways in which an embedded-pdv type can be encoded:

a) the **`syntaxes`** alternative of the embedded-pdv type is constrained with a PER-visible inner type constraint to a single value or **`identification`** is constrained with a PER-visible inner type constraint to the **`fixed`** alternative, in which case only the **`data-value`** shall be encoded; this is called the "predefined" case;

b) an inner type constraint is not employed to constrain the **`syntaxes`** alternative to a single value, nor to constrain **`identification`** to the **`fixed`** alternative, in which case both the **`identification`** and **`data-value`** shall be encoded; this is called the "general" case.

25.2 In the "predefined" case, the encoding of the value of the embedded-pdv type shall be the PER-encoding of a value of the **`OCTET STRING`** type. The value of the **`OCTET STRING`** shall be the octets which form the complete encoding of the single data value referenced in ITU-T Rec. X.680 | ISO/IEC 8824-1, 33.3.a.

25.3 In the "general" case, the encoding of a value of the embedded-pdv type shall be the PER encoding of the type defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 33.5 with the **`data-value-descriptor`** element removed (that is, there shall be no **`OPTIONAL`** bit-map at the head of the encoding of the **`SEQUENCE`**). The value of the **`data-value`** component of type **`OCTET STRING`** shall be the octets which form the complete encoding of the single data value referenced in ITU-T Rec. X.680 | ISO/IEC 8824-1, 33.3.a.

26 Encoding of a value of the external type

26.1 The encoding of a value of the external type shall be the PER encoding of the following sequence type, assumed to be defined in an environment of **`EXPLICIT TAGS`**, with a value as specified in the subclauses below:

```
[UNIVERSAL 8] IMPLICIT SEQUENCE {
          direct-reference          OBJECT IDENTIFIER OPTIONAL,
          indirect-reference        INTEGER OPTIONAL,
          data-value-descriptor     ObjectDescriptor OPTIONAL,
          encoding                  CHOICE {
               single-ASN1-type     [0] ABSTRACT-SYNTAX.&Type,
               octet-aligned        [1] IMPLICIT OCTET STRING,
               arbitrary            [2] IMPLICIT BIT STRING } }
```

NOTE – This sequence type differs from that in ITU-T Rec. X.680 | ISO/IEC 8824-1 for historical reasons.

26.2 The value of the components depends on the abstract value being transmitted, which is a value of the type specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 33.5.

26.3 The **data-value-descriptor** above shall be present if and only if the **data-value-descriptor** is present in the abstract value, and shall have the same value.

26.4 Values of **direct-reference** and **indirect-reference** above shall be present or absent in accordance with Table 1. Table 1 maps the external type alternatives of **identification** defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 33.5 to the external type components **direct-reference** and **indirect-reference** defined in 26.1.

Table 1 – Alternative encodings for "identification"

identification	direct-reference	indirect-reference
syntaxes	*** CANNOT OCCUR ***	*** CANNOT OCCUR ***
syntax	syntax	ABSENT
presentation-context-id	ABSENT	presentation-context-id
context-negotiation	transfer-syntax	presentation-context-id
transfer-syntax	*** CANNOT OCCUR ***	*** CANNOT OCCUR ***
fixed	*** CANNOT OCCUR ***	*** CANNOT OCCUR ***

26.5 The data value shall be encoded according to the transfer syntax identified by the encoding, and shall be placed in an alternative of the **encoding** choice as specified below.

26.6 If the data value is the value of a single ASN.1 data type (see the note on 26.7), and if the encoding rules for this data value are those specified in this Recommendation | International Standard, then the sending implementation shall use the **single-ASN1-type** alternative.

26.7 Otherwise, if the encoding of the data value, using the agreed or negotiated encoding, is an integral number of octets, then the sending implementation shall encode as **octet-aligned**.

NOTE – A data value which is a series of ASN.1 types, and for which the transfer syntax specifies simple concatenation of the octet strings produced by applying the ASN.1 Basic Encoding Rules to each ASN.1 type, falls into this category, not that of 26.6.

26.8 Otherwise, if the encoding of the data value, using the agreed or negotiated encoding, is not an integral number of octets, the **encoding** choice shall be **arbitrary**.

26.9 If the **encoding** choice is chosen as **single-ASN1-type**, then the ASN.1 type shall be encoded as specified in 10.2 with a value equal to the data value to be encoded.

NOTE – The range of values which might occur in the open type is determined by the registration of the object identifier value associated with the **direct-reference**, and/or the integer value associated with the **indirect-reference**.

26.10 If the **encoding** choice is **octet-aligned**, then the data value shall be encoded according to the agreed or negotiated transfer syntax, and the resulting octets shall form the value of the octetstring.

26.11 If the **encoding** choice is **arbitrary**, then the data value shall be encoded according to the agreed or negotiated transfer syntax, and the result shall form the value of the bitstring.

27 Encoding the restricted character string types

NOTE 1 – (Tutorial ALIGNED variant) Character strings of fixed length less than or equal to two octets are not octet-aligned. Character strings of variable length that are constrained to have a maximum length of less than two octets are not octet-aligned. All other character strings are octet-aligned in the ALIGNED variant. Fixed length character strings encode with no length octets if they are shorter than 64K characters. For unconstrained character strings or constrained character strings longer than 64K–1, the length is explicitly encoded (with fragmentation if necessary). Each **`NumericString`**, **`PrintableString`**, **`VisibleString`** (**`ISO646String`**), **`IA5String`**, **`BMPString`** and **`UniversalString`** character is encoded into the number of bits that is the smallest power of two that can accommodate all characters allowed by the effective permitted-alphabet constraint.

NOTE 2 – (Tutorial UNALIGNED variant) Character strings are not octet-aligned. If there is only one possible length value there is no length encoding if they are shorter than 64K characters. For unconstrained character strings or constrained character strings longer than 64K–1, the length is explicitly encoded (with fragmentation if necessary). Each **`NumericString`**, **`PrintableString`**, **`VisibleString`** (**`ISO646String`**), **`IA5String`**, **`BMPString`** and **`UniversalString`** character is encoded into the number of bits that is the smallest that can accommodate all characters allowed by the effective permitted-alphabet constraint.

NOTE 3 – (Tutorial on size of each encoded character) Encoding of each character depends on the effective permitted-alphabet constraint (see 9.3.11), which defines the alphabet in use for the type. Suppose this alphabet consists of a set of characters ALPHA (say). For each of the known-multiplier character string types (see 3.6.16), there is an integer value associated with each character, obtained by reference to some code table associated with the restricted character string type. The set of values BETA (say) corresponding to the set of characters ALPHA is used to determine the encoding to be used, as follows: the number of bits for the encoding of each character is determined solely by the number of elements, N, in the set BETA (or ALPHA). For the UNALIGNED variant is the smallest number of bits that can encode the value N – 1 as a non-negative binary integer. For the ALIGNED variant this is the smallest number of bits that is a power of two and that can encode the value N – 1. Suppose the selected number of bits is B. Then if every value in the set BETA can be encoded (with no transformation) in B bits, then the value in set BETA is used to represent the corresponding characters in the set ALPHA. Otherwise, the values in set BETA are taken in ascending order and replaced by values 0, 1, 2, and so on up to N – 1, and it is these values that are used to represent the corresponding character. In summary: minimum bits (taken to the next power of two for the ALIGNED variant) are always used. Preference is then given to using the value normally associated with the character, but if any of these values cannot be encoded in the minimum number of bits a compaction is applied.

27.1 The following restricted character string types are known-multiplier character string types: **`NumericString`**, **`PrintableString`**, **`VisibleString`** (**`ISO646String`**), **`IA5String`**, **`BMPString`**, and **`UniversalString`**. Effective permitted-alphabet constraints are PER-visible only for these types.

27.2 The effective size constraint notation may determine an upper bound "aub" for the length of the abstract character string. Otherwise, "aub" is unset.

27.3 The effective size constraint notation may determine a non-zero lower bound "alb" for the length of the abstract character string. Otherwise, "alb" is zero.

NOTE – PER-visible constraints only apply to known-multiplier character string types. For other restricted character string types "aub" will be unset and "alb" will be zero.

27.4 If the type is extensible for PER encodings (see 9.3.16), then a bit-field consisting of a single bit shall be added to the field-list. The single bit shall be set to zero if the value is within the range of the extension root, and to one otherwise. If the value is outside the range of the extension root, then the following encoding shall be as if there was no effective size constraint, and shall have an effective permitted-alphabet constraint that consists of the set of characters of the unconstrained type.

NOTE – Only the known-multiplier character string types can be extensible for PER encodings. Extensibility markers on other character string types do not affect the PER encoding.

27.5 This subclause applies to known-multiplier character strings. Encoding of the other restricted character string types is specified in 27.6.

27.5.1 The effective permitted alphabet is defined to be that alphabet permitted by the permitted-alphabet constraint, or the entire alphabet of the built-in type if there is no PermittedAlphabet constraint.

27.5.2 Let N be the number of characters in the effective permitted alphabet. Let B be the smallest integer such that 2 to the power B is greater than or equal to N. Let B2 be the smallest power of 2 that is greater than or equal to B. Then in the ALIGNED variant, each character shall encode into B2 bits, and in the UNALIGNED variant into B bits. Let the number of bits identified by this rule be "b".

27.5.3 A numerical value "v" is associated with each character by reference to ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 39 as follows. For **`UniversalString`**, the value is that used to determine the canonical order in ITU-T Rec. X.680 | ISO/IEC 8824-1, 39.3 (the value is in the range 0 to $2^{32} - 1$). For **`BMPString`**, the value is that used to determine the canonical order in ITU-T Rec. X.680 | ISO/IEC 8824-1, 39.3 (the value is in the range 0 to $2^{16} - 1$). For **`NumericString`** and **`PrintableString`** and **`VisibleString`** and **`IA5String`** the value is that defined for the ISO/IEC 646 encoding of the

corresponding character. (For **IA5String** the range is 0 to 127, for **VisibleString** it is 32 to 126, for **NumericString** it is 32 to 57, and for **PrintableString** it is 32 to 122. For **IA5String** and **VisibleString** all values in the range are present, but for **NumericString** and **PrintableString** not all values in the range are in use.)

27.5.4 Let the smallest value in the range for the set of characters in the permitted alphabet be "lb" and the largest value be "ub". Then the encoding of a character into "b" bits is the non-negative-binary-integer encoding of the value "v" identified as follows:

a) if "ub" is less than or equal to $2^b - 1$, then "v" is the value specified in above; otherwise

b) the characters are placed in the canonical order defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 39. The first is assigned the value zero and the next in canonical order is assigned a value that is one greater than the value assigned to the previous character in the canonical order. These are the values "v".

NOTE – Item a) above can never apply to a constrained or unconstrained **NumericString** character, which always encodes into four bits or less using b).

27.5.5 The encoding of the entire character string shall be obtained by encoding each character (using an appropriate value "v") as a non-negative-binary-integer into "b" bits which shall be concatenated to form a bit-field that is a multiple of "b" bits.

27.5.6 If "aub" equals "alb" and is less than 64K, then the bit-field shall be added to the field-list as a field (octet-aligned in the ALIGNED variant) if "aub" times "b" is greater than 16, but shall otherwise be added as a bit-field that is not octet-aligned. This completes the procedures of this subclause.

27.5.7 If "aub" does not equal "alb" or is greater than or equal to 64K, then 10.9 shall be invoked to add a length determinant with "n" as a count of the characters in the character string with a lower bound for the length determinant of "alb" and an upper bound of "aub". The bit-field shall then be added as a field (octet-aligned in the ALIGNED variant) if "aub" times "b" is greater than or equal to 16, but shall otherwise be added as a bit-field that is not octet-aligned. This completes the procedures of this subclause.

NOTE – Both 27.5.6 and 27.5.7 specify no alignment if "aub" times "b" is less than 16, and alignment if the product is greater than 16. For a value exactly equal to 16, 27.5.6 specifies no alignment and 27.5.7 specifies alignment.

27.6 This subclause applies to character strings that are not known-multiplier character strings. In this case, constraints are never PER-visible, and the type can never be extensible for PER encoding.

27.6.1 For BASIC-PER, reference below to "base encoding" means the contents octets of a BER encoding. For CANONICAL-PER it means the contents octets of the encoding specified for the CER and DER in ITU-T Rec. X.690 | ISO/IEC 8825-1.

27.6.2 The "base encoding" shall be applied to the character string to give a field of "n" octets.

27.6.3 Subclause 10.9 shall be invoked to add an unconstrained length determinant with "n" as a count in octets and the field of "n" octets shall be added as a bit-field (octet-aligned in the ALIGNED variant), completing the procedures of this subclause.

28 Encoding the unrestricted character string type

28.1 There are two ways in which an unrestricted character string type can be encoded:

a) the **syntaxes** alternative of the unrestricted character string type is constrained with a PER-visible inner type constraint to a single value or **identification** is constrained with a PER-visible inner type constraint to the **fixed** alternative, in which case only the **string-value** shall be encoded; this is called the "predefined" case;

b) an inner type constraint is not employed to constrain the **syntaxes** alternative to a single value, nor to constrain **identification** to the **fixed** alternative, in which case both the **identification** and **string-value** shall be encoded; this is called the "general" case.

28.2 For the "predefined" case, the encoding of the value of the **CHARACTER STRING** type shall be the PER-encoding of a value of the **OCTET STRING** type. The value of the **OCTET STRING** shall be the octets which form the complete encoding of the character string value referenced in ITU-T Rec. X.680 | ISO/IEC 8824-1, 40.3 a.

28.3 In the "general" case, the encoding of a value of the unrestricted character string type shall be the PER encoding of the type defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 40.5 with the **data-value-descriptor** component removed (that

is, there shall be no **OPTIONAL** bit-map at the head of the encoding of the **SEQUENCE**). The value of the **string-value** component of type **OCTET STRING** shall be the octets which form the complete encoding of the character string value referenced in ITU-T Rec. X.680 | ISO/IEC 8824-1, 40.3 a.

29 Object identifiers for transfer syntaxes

29.1 The encoding rules specified in this Recommendation | International Standard can be referenced and applied whenever there is a need to specify an unambiguous bit string representation for all of the values of a single ASN.1 type.

29.2 The following object identifiers and object descriptor values are assigned to identify and describe the encoding rules specified in this Recommendation | International Standard:

For BASIC-PER, ALIGNED variant:

```
{joint-iso-itu-t asn1 (1) packed-encoding (3) basic (0) aligned (0)}
"Packed encoding of a single ASN.1 type (basic aligned)"
```

For BASIC-PER, UNALIGNED variant:

```
{joint-iso-itu-t asn1 (1) packed-encoding (3) basic (0) unaligned (1)}
"Packed encoding of a single ASN.1 type (basic unaligned)"
```

For CANONICAL-PER, ALIGNED variant:

```
{joint-iso-itu-t asn1 (1) packed-encoding (3) canonical (1) aligned (0)}
"Packed encoding of a single ASN.1 type (canonical aligned)"
```

For CANONICAL-PER, UNALIGNED variant:

```
{joint-iso-itu-t asn1 (1) packed-encoding (3) canonical (1) unaligned (1)}
"Packed encoding of a single ASN.1 type (canonical unaligned)"
```

29.3 Where an application standard defines an abstract syntax as a set of abstract values, each of which is a value of some specifically named ASN.1 type defined using the ASN.1 notation, then the object identifier values specified in 29.2 may be used with the abstract syntax name to identify those transfer syntaxes which result from the application of the encoding rules specified in this Recommendation | International Standard to the specifically named ASN.1 type used in defining the abstract syntax.

29.4 The names specified in 29.2 shall not be used with an abstract syntax name to identify a transfer syntax if the conditions of 29.3 for the definition of the abstract syntax are not met.

Annex A

Example of encodings

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

This annex illustrates the use of the Packed Encoding Rules specified in this Recommendation | International Standard by showing representations in octets of a (hypothetical) personnel record which is defined using ASN.1.

A.1 Record that does not use subtype constraints

A.1.1 ASN.1 description of the record structure

The structure of the hypothetical personnel record is formally described below using ASN.1 specified in ITU-T Rec. X.680 | ISO/IEC 8824-1 for defining types. This is identical to the example defined in ITU-T Rec. X.690 | ISO/IEC 8825-1, Annex A.

```
PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET {
      name                  Name,
      title                 [0] VisibleString,
      number                EmployeeNumber,
      dateOfHire            [1] Date,
      nameOfSpouse          [2] Name,
      children              [3] IMPLICIT
            SEQUENCE OF ChildInformation DEFAULT {} }

ChildInformation ::= SET
      { name                Name,
        dateOfBirth         [0] Date}

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
      { givenName           VisibleString,
        initial             VisibleString,
        familyName          VisibleString}

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER

Date ::= [APPLICATION 3] IMPLICIT VisibleString  -- YYYYMMDD
```

A.1.2 ASN.1 description of a record value

The value of John Smith's personnel record is formally described below using ASN.1.

```
{ name {givenName "John",initial "P",familyName "Smith"},
       title                "Director",
       number               51,
       dateOfHire           "19710917",
       nameOfSpouse         {givenName "Mary",initial "T",familyName "Smith"},
       children
           {{name {givenName "Ralph",initial "T",familyName "Smith"},
                   dateOfBirth "19571111"},
           {name {givenName "Susan",initial "B",familyName "Jones"},
                   dateOfBirth "19590717"}}}
```

A.1.3 ALIGNED PER representation of this record value

The representation of the record value given above (after applying the ALIGNED variant of the Packed Encoding Rules defined in this Recommendation | International Standard) is shown below. The encoding is shown in hexadecimal, followed by a commented description of the encoding shown in binary.

The length of this encoding is 94 octets. For comparison, the same PersonnelRecord value encoded using the UNALIGNED variant of PER is 84 octets, BER with the definite length form is at least 136 octets, and BER with the indefinite length form is at least 161 octets.

A.1.3.1 Hexadecimal view

```
80044A6F  686E0150  05536D69  74680133  08446972  6563746F  72083139  37313039
3137044D  61727901  5405536D  69746802  0552616C  70680154  05536D69  74680831
39353731  31313105  53757361  6E014205  4A6F6E65  73083139  35393037  3137
```

A.1.3.2 Binary view

So as to make it easier to read the binary view of the data, blank lines are used to group fields that logically belong together (typically length/value pairs); a newline is used to delineate fields; space is used to delineate characters within a character string; and an 'x' represents a zero pad bit that is used from time to time to align fields on an octet boundary.

1xxxxxxx	Bitmap bit = 1 indicates "children" is present
00000100 01001010 01101111 01101000 01101110	Length of name.givenName = 4 name.givenName = "John"
00000001 01010000	Length of name.initial = 1 name.initial = "P"
00000101 01010011 01101101 01101001 01110100 01101000	Length of name.familyName = 5 name.familyName = "Smith"
00000001 00110011	Length of (employee) number = 1 (employee) number = 51
00001000 01000100 01101001 01110010 01100101 01100011 01110100 01101111 01110010	Length of title = 8 title = "Director"
00001000 00110001 00111001 00110111 00110001 00110000 00111001 00110001 00110111	Length of dateOfHire = 8 dateOfHire = "19710917"
00000100 01001101 01100001 01110010 01111001	Length of nameOfSpouse.givenName = 4 nameOfSpouse.givenName = "Mary"
00000001 01010100	Length of nameOfSpouse.initial = 1 nameOfSpouse.initial = "T"
00000101 01010011 01101101 01101001 01110100 01101000	Length of nameOfSpouse.familyName = 5 nameOfSpouse.familyName = "Smith"
00000010	Number of children
00000101 01010010 01100001 01101100 01110000 01101000	Length of children[0].givenName = 5 children[0].givenName = "Ralph"
00000001 01010100	Length of children[0].initial = 1 children[0].initial = "T"
00000101 01010011 01101101 01101001 01110100 01101000	Length of children[0].familyName = 5 children[0].familyName = "Smith"
00001000 00110001 00111001 00110101 00110111 00110001 00110001 00110001 00110001	Length of children[0].dateOfBirth = 8 children[0].dateOfBirth = "19571111"
00000101 01010011 01110101 01110011 01100001 01101110	Length of children[1].givenName = 5 children[1].givenName = "Susan"
00000001 01000010	Length of children[1].initial = 1 children[1].initial = "B"
00000101 01001010 01101111 01101110 01100101 01110011	Length of children[1].familyName = 5 children[1].familyName = "Jones"
00001000 00110001 00111001 00110101 00111001 00110000 00110111 00110001 00110111	Length of children[1].dateOfBirth = 8 children[1].dateOfBirth = "19590717"

A.1.4 UNALIGNED PER representation of this record value

The representation of the record value given above (after applying the UNALIGNED variant of the Packed Encoding Rules defined in this Recommendation | International Standard) is shown below. The encoding is shown in hexadecimal, followed by a commented description of the encoding shown in binary. Note that pad bits do not occur in the UNALIGNED variant, and characters are encoded in the fewest number of bits possible.

The length of this encoding is 84 octets. For comparison, the same PersonnelRecord value encoded using the ALIGNED variant of PER is 94 octets, BER with the definite length form is at least 136 octets, and BER with the indefinite length form is at least 161 octets.

A.1.4.1 Hexadecimal view

```
824ADFA3  700D005A  7B74F4D0  02661113  4F2CB8FA  6FE410C5  CB762C1C  B16E0937
0F2F2035  0169EDD3  D340102D  2C3B3868  01A80B4F  6E9E9A02  18B96ADD  8B162C41
69F5E787  700C2059  5BF765E6  10C5CB57  2C1BB16E
```

A.1.4.2 Binary view

So as to make it easier to read the binary view of the data, blank lines are used to group fields that logically belong together (typically length/value pairs); a newline is used to delineate fields; space is used to delineate characters within a character string; a period (.) is used to mark octet boundaries; and an 'x' represents a zero-bit used to pad the final octet to an octet boundary.

```
1                                                                   Bitmap bit = 1 indicates "children" is present

0000010.0                                                           Length of name.givenName = 4
1001010 .1101111  1.101000  11.01110                               name.givenName = "John"

000.00001                                                           Length of name.initial = 1
101.0000                                                            name.initial = "P"

0000.0101                                                           Length of name.familyName = 5
1010.011  11011.01  110100.1  1110100  .1101000                   name.familyName = "Smith"

0.0000001                                                           Length of (employee) number = 1
0.0110011                                                           (employee) number = 51

0.0001000                                                           Length of title = 8
1.000100  11.01001  111.0010  1100.101  11000.11  111010.0  1101111 .1110010   title = "Director"

0.0001000                                                           Length of dateOfHire = 8
0.110001  01.11001  011.0111  0110.001  01100.00  011100.1  0110001  .0110111  dateOfHire = "19710917"

0.0000100                                                           Length of nameOfSpouse.givenName = 4
1.001101  11.00001  111.0010  1111.001                            nameOfSpouse.givenName = "Mary"

00000.001                                                           Length of nameOfSpouse.initial = 1
10101.00                                                            nameOfSpouse.initial = "T"

000001.01                                                           Length of nameOfSpouse.familyName = 5
101001.1  1101101  .1101001  1.110100  11.01000                    nameOfSpouse.familyName = "Smith"

000.00010                                                           Number of children

000.00101                                                           Length of children[0].givenName = 5
101.0010  1100.001  11011.00  111000.0  1101000                   children[0].givenName = "Ralph"

.00000001                                                           Length of children[0].initial = 1
.1010100                                                            children[0].initial = "T"

0.0000101                                                           Length of children[0].familyName = 5
1.010011  11.01101  110.1001  1110.100  11010.00                  children[0].familyName = "Smith"

000010.00                                                           Length of children[0].dateOfBirth = 8
011000.1  0111001  .0110101  0.110111  01.10001  011.0001  0110.001  01100.01  children[0].dateOfBirth = "19571111"
```

000001.01	Length of children[1].givenName = 5
101001.1 1110101 .1110011 1.100001 11.01110	children[1].givenName = "Susan"
000.00001	Length of children[1].initial = 1
100.0010	children[1].initial = "B"
0000.0101	Length of children[1].familyName = 5
1001.100 11011.11 110111.0 1100101 .1110011	children[1].familyName = "Jones"
0.0001000	Length of children[1].dateOfBirth = 8
0.110001 01.11001 011.0101 0111.001 01100.00 011011.1 0110001 .0110111x	children[1].dateOfBirth = "19590717"

A.2 Record that uses subtype constraints

This example is the same as that shown in clause A.1, except that it makes use of the subtype notation to impose constraints on some items.

A.2.1 ASN.1 description of the record structure

The structure of the hypothetical personnel record is formally described below using ASN.1 specified in ITU-T Rec. X.680 | ISO/IEC 8824-1 for defining types.

```
PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET {
      name                  Name,
      title                 [0] VisibleString,
      number                EmployeeNumber,
      dateOfHire            [1] Date,
      nameOfSpouse          [2] Name,
      children              [3] IMPLICIT
            SEQUENCE OF ChildInformation  DEFAULT  {} }

ChildInformation ::= SET
      { name                Name,
        dateOfBirth         [0] Date}

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
      { givenName           NameString,
        initial             NameString (SIZE(1)),
        familyName          NameString}

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER

Date ::= [APPLICATION 3] IMPLICIT VisibleString
                            (FROM("0".."9") ^ SIZE(8)) -- YYYYMMDD

NameString ::= VisibleString (FROM("a".."z" | "A".."Z" | "-.") ^ SIZE(1..64))
```

A.2.2 ASN.1 description of a record value

The value of John Smith's personnel record is formally described below using ASN.1.

```
{ name {givenName "John",initial "P",familyName "Smith"},
       title                "Director",
       number               51,
       dateOfHire           "19710917",
       nameOfSpouse   {givenName "Mary",initial "T",familyName "Smith"},
       children
           {{name {givenName "Ralph",initial "T",familyName "Smith"},
                   dateOfBirth "19571111"},
            {name {givenName "Susan",initial "B",familyName "Jones"},
                   dateOfBirth "19590717"}}}
```

A.2.3 ALIGNED PER representation of this record value

The representation of the record value given above (after applying the ALIGNED variant of the Packed Encoding Rules defined in this Recommendation | International Standard) is shown below. The encoding is shown in hexadecimal, followed by a commented description of the encoding shown in binary. In the binary view an 'x' is used to represent pad bits that are encoded as zero-bits; they are used to align the fields from time to time.

The length of this encoding is 74 octets. For comparison, the same PersonnelRecord value encoded using the UNALIGNED variant of PER is 61 octets, BER with the definite length form is at least 136 octets, and BER with the indefinite length form is at least 161 octets.

A.2.3.1 Hexadecimal view

```
864A6F68  6E501053  6D697468  01330844  69726563  746F7219  7109170C  4D617279
5410536D  69746802  1052616C  70685410  536D6974  68195711  11105375  73616E42
104A6F6E  65731959  0717
```

A.2.3.2 Binary view

So as to make it easier to read the binary view of the data, blank lines are used to group fields that logically belong together (typically length/value pairs); a newline is used to delineate fields; space is used to delineate characters within a character string; and an 'x' represents a zero pad bit that is used from time to time to align fields on an octet boundary.

1	Bitmap bit = 1 indicates "children" is present
000011x	Length of name.givenName = 4
01001010 01101111 01101000 01101110	name.givenName = "John"
01010000	name.initial = "P"
000100xx	Length of name.familyName = 5
01010011 01101101 01101001 01110100 01101000	name.familyName = "Smith"
00000001	Length of (employee) number = 1
00110011	(employee) number = 51
00001000	Length of title = 8
01000100 01101001 01110010 01100101 01100011 01110100 01101111 01110010	title = "Director"
0001 1001 0111 0001 0000 1001 0001 0111	dateOfHire = "19710917"
000011xx	Length of nameOfSpouse.givenName = 4
01001101 01100001 01110010 01111001	nameOfSpouse.givenName = "Mary"
01010100	nameOfSpouse.initial = "T"
000100xx	Length of nameOfSpouse.familyName = 5
01010011 01101101 01101001 01110100 01101000	nameOfSpouse.familyName = "Smith"
00000010	Number of children
000100xx	Length of children[0].givenName = 5
01010010 01100001 01101100 01110000 01101000	children[0].givenName = "Ralph"
01010100	children[0].initial = "T"
000100xx	Length of children[0].familyName = 5
01010011 01101101 01101001 01110100 01101000	children[0].familyName = "Smith"
0001 1001 0101 0111 0001 0001 0001 0001	children[0].dateOfBirth = "19571111"
000100xx	Length of children[1].givenName = 5
01010011 01110101 01110011 01100001 01101110	children[1].givenName = "Susan"
01000010	children[1].initial = "B"
000100xx	Length of children[1].familyName = 5
01001010 01101111 01101110 01100101 01110011	children[1].familyName = "Jones"
0001 1001 0101 1001 0000 0111 0001 0111	children[1].dateOfBirth = "19590717"

A.2.4 UNALIGNED PER representation of this record value

The representation of the record value given above (after applying the UNALIGNED variant of the Packed Encoding Rules defined in this Recommendation | International Standard) is shown below. The encoding is shown in hexadecimal, followed by a commented description of the encoding shown in binary. Note that pad bits do not occur in the UNALIGNED variant, and characters are encoded in the fewest number of bits possible.

The length of this encoding is 61 octets. For comparison, the same PersonnelRecord value encoded using the ALIGNED variant of PER is 74 octets, BER with the definite length form is at least 136 octets, and BER with the indefinite length form is at least 161 octets.

A.2.4.1 Hexadecimal view

```
865D51D2  888A5125  F1809984  44D3CB2E  3E9BF90C  B8848B86  7396E8A8  8A5125F1
81089B93  D71AA229  4497C632  AE222222  985CE521  885D54C1  70CAC838  B8
```

A.2.4.2 Binary view

So as to make it easier to read the binary view of the data, blank lines are used to group fields that logically belong together (typically length/value pairs); a newline is used to delineate fields; space is used to delineate characters within a character string; a period (.) is used to mark octet boundaries; and an 'x' represents a zero-bit used to pad the final octet to an octet boundary:

```
1                                                                   Bitmap bit = 1 indicates "children" is present

000011                                                              Length of name.givenName = 4
0.01011  101.010  10001.1  101001                                   name.givenName = "John"

0.10001                                                             name.initial = "P"

000.100                                                             Length of name.familyName = 5
01010.0  101000  1.00100  101.111  10001.1                         name.familyName = "Smith"

0000000.1                                                           Length of (employee) number = 1
0011001.1                                                           (employee) number = 51

0000100.0                                                           Length of title = 8
1000100  .1101001 1.110010  11.00101 110.0011 1110.100 11011.11 111001.0   title = "Director"

0001  100.1  0111  000.1  0000  100.1  0001  011.1                  dateOfHire = "19710917"

000011                                                              Length of nameOfSpouse.givenName = 4
0.01110  011.100  10110.1  110100                                   nameOfSpouse.givenName = "Mary"

0.10101                                                             nameOfSpouse.initial = "T"

000.100                                                             Length of nameOfSpouse.familyName = 5
01010.0  101000  1.00100  101.111  10001.1                         nameOfSpouse.familyName = "Smith"

0000001.0                                                           Number of children

000100                                                              Length of children[0].givenName = 5
0.10011  011.100  10011.1  101011  1.00011                        children[0].givenName = "Ralph"

010.101                                                             children[0].initial = "T"

00010.0                                                             Length of children[0].familyName = 5
010100  1.01000  100.100  10111.1  100011                         children[0].familyName = "Smith"
0.001  1001  0.101  0111  0.001  0001  0.001  0001                 children[0].dateOfBirth = "19571111"

0.00100                                                             Length of children[1].givenName = 5
010.100  11000.0  101110  0.11100  101.001                         children[1].givenName = "Susan"

00001.1                                                             children[1].initial = "B"

000100                                                              Length of children[1].familyName = 5
0.01011  101.010  10100.1  100000  1.01110                        children[1].familyName = "Jones"
```

000.1 1001 010.1 1001 000.0 0111 000.1 0111xxx children[1].dateOfBirth = "19590717"

A.3 Record that uses extension markers

A.3.1 ASN.1 description of the record structure

The structure of the hypothetical personnel record is formally described below using ASN.1 specified in ITU-T Rec. X.680 | ISO/IEC 8824-1 for defining types:

```
PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET {
      name                  Name,
      title                 [0] VisibleString,
      number                EmployeeNumber,
      dateOfHire            [1] Date,
      nameOfSpouse          [2] Name,
      children              [3] IMPLICIT
            SEQUENCE (SIZE(2, ...)) OF ChildInformation  OPTIONAL,
      ...
}

ChildInformation ::= SET
      { name                Name,
        dateOfBirth         [0] Date,
        ...,
        sex                 [1] IMPLICIT ENUMERATED {male(1), female(2),
                                    unknown(3)} OPTIONAL
      }

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
      { givenName      NameString,
        initial        NameString (SIZE(1)),
        familyName     NameString,
        ...
      }

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER (0..9999, ...)

Date ::= [APPLICATION 3] IMPLICIT VisibleString
                       (FROM("0".."9") ^ SIZE(8, ..., 9..20)) -- YYYYMMDD

NameString ::= VisibleString
                       (FROM("a".."z" | "A".."Z" | "-.") ^ SIZE(1..64, ...))
```

A.3.2 ASN.1 description of a record value

The value of John Smith's personnel record is formally described below using ASN.1:

```
{ name {givenName "John",initial "P",familyName "Smith"},
        title               "Director",
        number              51,
        dateOfHire          "19710917",
        nameOfSpouse        {givenName "Mary",initial "T",familyName "Smith"},
        children
            {{name {givenName "Ralph",initial "T",familyName "Smith"},
                       dateOfBirth "19571111"},
             {name {givenName "Susan",initial "B",familyName "Jones"},
                       dateOfBirth "19590717", sex female}}}
```

A.3.3 ALIGNED PER representation of this record value

The representation of the record value given above (after applying the ALIGNED variant of the Packed Encoding Rules defined in this Recommendation | International Standard) is shown below. The encoding is shown in hexadecimal, followed by a commented description of the encoding shown in binary. In the binary view an 'x' is used to represent pad bits that are encoded as zero-bits; they are used to align the fields from time to time.

The length of this encoding is 83 octets. For comparison, the same PersonnelRecord value encoded using the UNALIGNED variant of PER is 65 octets, BER with the definite length form is at least 139 octets, and BER with the indefinite length form is at least 164 octets.

A.3.3.1 Hexadecimal view

```
40C04A6F  686E5008  536D6974  68000033  08446972  6563746F  72001971  0917034D
61727954  08536D69  74680100  52616C70  68540853  6D697468  00195711  11820053
7573616E  42084A6F  6E657300  19590717  010140
```

A.3.3.2 Binary view

So as to make it easier to read the binary view of the data, blank lines are used to group fields that logically belong together (typically length/value pairs); a newline is used to delineate fields; space is used to delineate characters within a character string; and an 'x' represents a zero pad bit that is used from time to time to align fields on an octet boundary:

```
0                                                                                   No extension values present in PersonnelRecord
1                                                                                   Bitmap bit = 1 indicates "children" is present

0                                                                                   No extension values present in "name"

0                                                                                   Length is within range of extension root
0000    11xxxxxx                                                                    Length of name.givenName = 4
01001010 01101111 01101000 01101110                                                 name.givenName = "John"

01010000                                                                            name.initial = "P"

0                                                                                   Length is within range of extension root
000100x                                                                             Length of name.familyName = 5
01010011 01101101 01101001 01110100 01101000                                        name.familyName = "Smith"

0xxxxxxx                                                                            Value is within range of extension root
00000000 00110011                                                                   (employee) number = 51

00001000                                                                            Length of title = 8
01000100 01101001 01110010 01100101 01100011 01110100 01101111 01110010            title = "Director"

0xxxxxxx                                                                            Length is within range of extension root
0001  1001  0111  0001  0000  1001  0001  0111                                      dateOfHire = "19710917"

0                                                                                   No extension values present in nameOfSpouse

0                                                                                   Length is within range of extension root
000011                                                                              Length of nameOfSpouse.givenName = 4
01001101 01100001 01110010 01111001                                                 nameOfSpouse.givenName = "Mary"

01010100                                                                            nameOfSpouse.initial = "T"

0                                                                                   Length is within range of extension root
000100x                                                                             Length of nameOfSpouse.familyName = 5
01010011 01101101 01101001 01110100 01101000                                        nameOfSpouse.familyName = "Smith"

0                                                                                   Number of "children" is within the range of the
                                                                                    extension root

0                                                                                   No extension values present in children[0]

0                                                                                   No extension values present in children[0].name

0                                                                                   Length is within range of extension root
000100xx xxxx                                                                       Length of children[0].givenName = 5
01010010 01100001 01101100 01110000 01101000                                        children[0].givenName = "Ralph"

01010100                                                                            children[0].initial = "T"

0                                                                                   Length is within range of extension root
000100x                                                                             Length of children[0].familyName = 5
01010011 01101101 01101001 01110100 01101000                                        children[0].familyName = "Smith"

0xxxxxxx                                                                            Length is within range of extension root
0001  1001  0101  0111  0001  0001  0001  0001                                      children[0].dateOfBirth = "19571111"
```

1	Extension value(s) present in children[1]
0	No extension values present in children[1].name
0 00010 0xxxxxxx 01010011 01110101 01110011 01100001 01101110	Length is within range of extension root Length of children[1].givenName = 5 children[1].givenName = "Susan"
01000010	children[1].initial = "B"
0 000100x 01001010 01101111 01101110 01100101 01110011	Length is within range of extension root Length of children[1].familyName = 5 children[1].familyName = "Jones"
0xxxxxxx 0001 1001 0101 1001 0000 0111 0001 0111	Length is within range of extension root children[1].dateOfBirth = "19590717"
0000000	Length of extension addition bitmap for children[1] = 1
1	Indicate extension value for "sex" is present
00000001 01xxxxxx	Length of the complete encoding of "sex" Complete encoding of "sex" = female

A.3.4 UNALIGNED PER representation of this record value

The representation of the record value given above (after applying the UNALIGNED variant of the Packed Encoding Rules defined in this Recommendation | International Standard) is shown below. The encoding is shown in hexadecimal, followed by a commented description of the encoding shown in binary. Note that pad bits do not occur in the UNALIGNED variant, and characters are encoded in the fewest number of bits possible.

The length of this encoding is 65 octets. For comparison, the same PersonnelRecord value encoded using the ALIGNED variant of PER is 83 octets, BER with the definite length form is at least 139 octets, and BER with the indefinite length form is at least 164 octets.

A.3.4.1 Hexadecimal view

```
40CBAA3A  5108A512  5F180330  889A7965  C7D37F20  CB8848B8  19CE5BA2  A114A24B
E3011372  7AE35422  94497C61  95711118  22985CE5  21842EAA  60B832B2  0E2E0202
80
```

A.3.4.2 Binary view

So as to make it easier to read the binary view of the data, blank lines are used to group fields that logically belong together (typically length/value pairs); a newline is used to delineate fields; space is used to delineate characters within a character string; a period (.) is used to mark octet boundaries; and an 'x' represents a zero-bit used to pad the final octet to an octet boundary:

0 1	No extension values present in PersonnelRecord Bitmap bit = 1 indicates "children" is present
0	No extension values present in "name"
0 0000.11 001011 .101010 10.0011 1010.01	Length is within range of extension root Length of name.givenName = 4 name.givenName = "John"
010001	name.initial = "P"
.0 000100 0.10100 101.000 10010.0 101111 1.00011	Length is within range of extension root Length of name.familyName = 5 name.familyName = "Smith"
0 00.00000011.0011	Value is within range of extension root (employee) number = 51

0000.1000 1000.100 11010.01 111001.0 1100101 1100011 1.110100 11.01111 111.0010	Length of title = 8 title = "Director"
0 000.1 1001 011.1 0001 000.0 1001 000.1 0111	Length is within range of extension root dateOfHire = "19710917"
0	No extension values present in nameOfSpouse
0 0.00011 001.110 01110.0 101101 1.10100	Length is within range of extension root Length of nameOfSpouse.givenName = 4 nameOfSpouse.givenName = "Mary"
010.101	nameOfSpouse.initial = "T"
0 0001.00 010100 .101000 10.0100 1011.11 100011	Length is within range of extension root Length of nameOfSpouse.familyName = 5 nameOfSpouse.familyName = "Smith"
.0	Number of "children" is within the range of the extension root
0	No extension values present in children[0]
0	No extension values present in children[0].name
0 0001.00 010011 .011100 10.0111 1010.11 100011	Length is within range of extension root Length of children[0].givenName = 5 children[0].givenName = "Ralph"
.010101	children[0].initial = "T"
0 0.00100 010.100 10100.0 100100 1.01111 100.011	Length is within range of extension root Length of children[0].familyName = 5 children[0].familyName = "Smith"
0 0001 .1001 0101 .0111 0001 .0001 0001 .0001	Length is within range of extension root children[0].dateOfBirth = "19571111"
1	Extension value(s) present in children[1]
0	No extension values present in children[1].name
0 0.00100 010.100 11000.0 101110 0.11100 101.001	Length is within range of extension root Length of children[1].givenName = 5 children[1].givenName = "Susan"
00001.1	children[1].initial = "B"
0 000100 .001011 10.1010 1010.01 100000 .101110	Length is within range of extension root Length of children[1].familyName = 5 children[1].familyName = "Jones"
0 0.001 1001 0.101 1001 0.000 0111 0.001 0111	Length is within range of extension root children[1].dateOfBirth = "19590717"
0.000000	Length of extension addition bitmap for children[1] = 1
1	Indicate extension value for "sex" is present
0.0000001 0.1xxxxxx x	Length of the complete encoding of "sex" Complete encoding of "sex" = female Pad bit to create complete encoding of PersonnelRecord

A.4 Record that uses extension addition groups

A.4.1 ASN.1 description of the record structure

The structure of the hypothetical customer record is formally described below using ASN.1 specified in ITU-T Rec. X.680 | ISO/IEC 8824-1 for defining types. **`AUTOMATIC TAGS`** is assumed:

```
Ax ::= SEQUENCE {
      a     INTEGER (250..253),
      b     BOOLEAN,
      c     CHOICE {
            d     INTEGER,
            ...,
            [[
               e  BOOLEAN,
               f  IA5String
            ]],
            ...
      },
      ...,
      [[
         g  NumericString (SIZE(3)),
         h  BOOLEAN OPTIONAL
      ]],
      ...,
      i     BMPString OPTIONAL,
      j     PrintableString OPTIONAL
}
```

A.4.2 ASN.1 description of a record value

The value of **Ax** is formally described below using ASN.1:

```
{ a  253,  b  TRUE,  c  e : TRUE,  g  "123",   h  TRUE }
```

A.4.3 ALIGNED PER representation of this record value

The representation of the value given above (after applying the ALIGNED variant of the Packed Encoding Rules defined in this Recommendation | International Standard) is shown below. The encoding is shown in hexadecimal, followed by a commented description of the encoding shown in binary. In the binary view an 'x' is used to represent pad bits that are encoded as zero-bits; they are used to align the fields from time to time.

The length of this encoding is 8 octets. For comparison, the same value encoded using the UNALIGNED variant of PER is 8 octets, BER with the definite length form is at least 22 octets, and BER with the indefinite length form is at least 26 octets.

A.4.3.1 Hexadecimal view

```
9E000180 010291A4
```

A.4.3.2 Binary view

So as to make it easier to read the binary view of the data, blank lines are used to group fields that logically belong together (typically length/value pairs); a newline is used to delineate fields; space is used to delineate characters within a character string; and an 'x' represents a zero pad bit that is used from time to time to align fields on an octet boundary:

1	Extension addition values present in Ax
00	Bitmap bits = 0 indicates optional fields (i & j) absent
11	a = 253
1	b = TRUE
1	c's choice value is an extension addition value
0000000 xx	Choice index selects c.e
00000001	Length of c.e
1xxxxxxx	c.e = TRUE
0000000	Number of extension additions defined in Ax = 1
1	First extension addition is present
00000010	Length of extension addition encoding = 2
1	Bitmap = 1 indicates 'h' is present
0010 0011 0100	g = "123"

1xx	h = TRUE

A.4.4 UNALIGNED PER representation of this record value

The representation of the record value given above (after applying the UNALIGNED variant of the Packed Encoding Rules defined in this Recommendation | International Standard) is shown below. The encoding is shown in hexadecimal, followed by a commented description of the encoding shown in binary. Note that pad bits do not occur in the UNALIGNED variant, except possibly at the end of the encoding of the outermost value – and thus implicitly at the end of the value carried by an open type.

The length of this encoding is 8 octets. For comparison, the same value encoded using the ALIGNED variant of PER is 8 octets, BER with the definite length form is at least 22 octets, and BER with the indefinite length form is at least 26 octets.

A.4.4.1 Hexadecimal view

```
9E000600 040A4690
```

A.4.4.2 Binary view

So as to make it easier to read the binary view of the data, blank lines are used to group fields that logically belong together (typically length/value pairs); a newline is used to delineate fields; space is used to delineate characters within a character string; a period (.) is used to mark octet boundaries; and an 'x' represents a zero-bit used to pad the final octet to an octet boundary:

1	Extension addition values present in Ax
00	Bitmap bits = 0 indicates optional fields (i & j) absent
11	a = 253
1	b = TRUE
1	c's choice value is an extension addition value
0.000000	Choice index selects c.e
00.000001	Length of c.e
1x.xxxxxx	c.e = TRUE
00.00000	Number of extension additions defined in Ax = 1
1	First extension addition is present
00.000010	Length of extension addition encoding = 2
1	Bitmap = 1 indicates 'h' is present
0.010 0011 0.100	g = "123"
1xxxx	h = TRUE

Annex B

Combining PER-visible and non-PER-visible constraints

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

B.1 General

B.1.1 The correct determination of PER extensibility is critical to the interworking of implementations. It is also important that different implementations make the same determination of the values that are to be encoded by PER as root values and of the values that are to be encoded as extension additions for an extensible type.

B.1.2 Things written by users are usually simple, and the PER encoding is intuitive, but for complicated constructions, the interactions between PER-visibility, PER-extensibility, and set arithmetic needs further discussion, and is the content of this clause.

B.1.3 Because some constraints are defined to be not PER-visible (see 9.3), a type may be defined to be extensible by the rules of ITU-T Rec. X.680 | ISO/IEC 8824-1 but to be considered not extensible (with relaxed constraints that would cover all possible extensions) for PER encoding.

B.1.4 Where a type is considered extensible in both cases, the set of root values for PER encoding is not always the same as the set of values that would be considered to be root values by the definitions in ITU-T Rec. X.680 | ISO/IEC 8824-1.

B.1.5 In most of the cases that occur in actual specifications, the two determinations are easy and straightforward.

B.1.6 However, ASN.1 provides considerable power and generality in the application of complex constraints resulting from set arithmetic and/or the serial application of simple or complex constraints.

B.1.7 User specifications are unlikely to define ASN.1 constructs involving the complexities discussed in this Annex, but implementers of tools need to know what code to produce if such constraints are, in fact, applied.

B.1.8 The rules for very complex constraints (perhaps involving type reference names) are not always intuitive, but have been designed to simplify tool implementation and the complexity of the ASN.1 specification.

B.1.9 For **SEQUENCE**, **SET**, **CHOICE**, and **ENUMERATED**, a type is always extensible if it contains the extension marker (the ellipsis "**...**"), even if constrained (see 9.3.20). A value is a root value if and only if the value does not include any elements (or alternatives for **CHOICE** and enumerations for **ENUMERATED**) after the ellipsis. A non-extensible **SEQUENCE**, **SET**, **CHOICE** or **ENUMERATED** can be a parent type to which an extensible constraint is applied, resulting in an extensible sequence, set, choice, or enumerated type. However, constraints on these types are never PER-visible and the resulting types encode without the extensibility bit in PER. These types are not discussed further in this annex, which is concerned solely with extensibility arising from the use of extensible constraints on integer and restricted known-multiplier character string types. (Constraints on other types do not affect PER encodings, except for size constraints on octet string and bit string types, which are similar to size constraints on character string types, and are not considered further here.)

B.1.10 The normative text specifies the precise rules, but this tutorial annex is intended to assist tool vendors in understanding the rules.

B.1.11 For simplicity of exposition, the set of values in a non-extensible type or constraint are described below as root values, although this term strictly only applies to extensible types or constraints.

B.1.12 ITU-T Rec. X.680 | ISO/IEC 8824-1, G.4, provides tutorial information on the combination of constraints when all constraints are PER-visible as specified by that Recommendation | International Standard, and should be read in conjunction with this Annex. When constraints are involved that are not PER-visible, or where constraints are applied to character string types, then the rules require further additions. These additional rules are covered in B.2.

B.2 Extensibility and visibility of constraints in PER

B.2.1 General

B.2.1.1 In BER, encodings of values are the same for root values and extension additions, so extensibility has no impact on the encoding. In PER, abstract values are generally encoded in an efficient manner if they are in the (usually, but not necessarily, finite) set of root values, and less efficiently if they are extension additions.

B.2.1.2 However, for many PER encodings, there are values in the extension additions of a type (as determined by ITU-T Rec. X.680 | ISO/IEC 8824-1) that are encoded by PER as if they were root values, not as extension additions. The precise identification of these values is performed by noting that some constraints are "not PER-visible".

B.2.1.3 The concept of PER-visibility was introduced into this Recommendation | International Standard in order to ease the task of encoders in trying to determine whether a value to be encoded is in the root of an extensible type or not. Constraints that may be difficult for encoders to handle in an efficient manner are defined to be "not visible" for PER encoding (have no effect on it).

B.2.1.4 With one exception, the visibility of a simple constraint depends only on the type being constrained, and/or on aspects of the constraint that are not related to extensibility. For example, does the constraint depend textually on a table constraint, or is it a variable constraint (a constraint which is textually dependent on a parameter of the abstract syntax)?

B.2.1.5 If a constraint is a variable constraint, or is textually dependent on a table constraint, it is never PER-visible, no matter what type it is applied to.

B.2.1.6 Additionally, constraints are never PER-visible unless they are applied to an integer or to a known-multiplier restricted character string type (or are size constraints on a bitstring or octetstring).

B.2.1.7 The exception is a permitted-alphabet constraint on a known-multiplier restricted character string type. This is PER-visible if and only if it is not extensible.

B.2.1.8 It is also important to note that single value subtype constraints on character string types are not PER-visible.

B.2.1.9 In PER, constraints on character string types have two independent dimensions - constraints on the size of the string, and constraints on the permitted-alphabet. The first affects the presence and form of a length field in the encoding, and the second affects the number of bits used to encode each character. In simple use, it is clear that a constraint specifies one or other of these. Thus:

```
A1 ::= VisibleString (SIZE (20))
-- A size constraint

A2 ::= VisibleString (FROM ("A".."F"))
-- A permitted-alphabet constraint

A3 ::= VisibleString (SIZE (2))(FROM ("A".."F"))
-- Both a size and a permitted-alphabet constraint
```

B.2.1.10 But consider:

```
B ::= VisibleString (SIZE (20) INTERSECTION FROM ("A".."F")
                     UNION
                     SIZE (3) INTERSECTION FROM ("F".."K") )
```

B.2.1.11 To specify the encoding of types with complex constraints of this sort, PER introduces the concepts of an *effective* size constraint, and an *effective* permitted-alphabet constraint. These are constraints that, taken together, will allow all the abstract values in the root of the actual constraint, but usually some additional abstract values. In the example above the effective size constraint is **20**, and the effective permitted-alphabet constraint is **FROM("A".."K")**.

B.2.1.12 In order to handle extensibility, this Recommendation | International Standard introduces the further concept that either or both of an effective size and an effective permitted-alphabet constraint can be extensible (the latter would not be PER-visible, and would be ignored when determining encodings), and it is necessary to consider the effect of (non) PER-visibility of extensible permitted-alphabet constraints on the effective constraints on a type.

B.2.1.13 The following clauses address the main issues: the effect of PER-visibility, and the calculation of effective constraints for serial application of constraints, and for set arithmetic.

B.2.2 PER-visibility of constraints

B.2.2.1 B.2.2.10 describes when a complete (complex) constraint is PER-visible and when it is not. First, however, we consider simply the serial application of constraints, each of which is (as a whole) PER-visible or not PER-visible.

B.2.2.2 The rule is very simple: If a complete constraint in serial application of constraints is not PER-visible, then for the purposes of PER encodings, that constraint is simply completely ignored.

NOTE – When non-visible constraints are removed for the purposes of defining PER encodings, this does not imply that applications can now legally transmit additional abstract values. The original constraints still apply to the values that can be transmitted, although encoders would normally use only PER-visible constraints to perform checks and issue diagnostics.

B.2.2.3 It is important to realize that the removal of non-visible constraints can have quite dramatic effects in complex cases, and it is always important to consider extensibility (and what are root values) after removal of the serially applied constraints that are not PER-visible. (If none of the serially applied constraints is PER-visible, then the type is unconstrained - and not extensible - for the purposes of PER encodings.)

B.2.2.4 A type which is extensible according to ITU-T Rec. X.680 | ISO/IEC 8824-1 could be inextensible for PER.

B.2.2.5 Even when the effects are not so dramatic, values which are extension additions according to ITU-T Rec. X.680 | ISO/IEC 8824-1 may be part of the root values when some constraints are removed, and hence would encode in PER as root values and not as extension additions.

NOTE – This means that the PER encodings are more verbose than is theoretically possible, but still have a unique encoding for all abstract values in the type being encoded.

B.2.2.6 Three main types of factor affect the visibility of a complex constraint which is being serially applied.

B.2.2.7 The first factor to consider is whether the constraint is a variable constraint (depends textually on a parameter of the abstract syntax), or depends textually on a table constraint. In such cases, the entire constraint that is being serially applied is not PER-visible, and is discarded.

B.2.2.8 The second factor to consider applies only to constraints on character string types. Single value subtype constraints on such types are not PER-visible, but their presence does not necessarily make the entire constraint that is serially applied non-visible if set arithmetic is present within the constraint.

B.2.2.9 The rules for determining PER-visibility in this case are specified in 9.3.19, and are summarized here. Let "V" denote PER-visible, and "I" denote non-visible (invisible).

B.2.2.10 Because **UNION** and **INTERSECTION** are both commutative, the rule for the result is given only for the V first case. Where all components are V, then the normal rules of ITU-T Rec. X.680 | ISO/IEC 8824-1 apply, and these are not discussed further here. The cases where all components are I always give I, and are again not listed.) The rules are:

```
V UNION I => I
V INTERSECTION I => V
-- The resulting V is just the V part of the intersection
V EXCEPT I => V
-- The resulting V is just the V without the set difference
I EXCEPT V => I
V, ..., I => I
I, ..., V => I
```

B.2.2.11 There is one important consequence of eliminating single value subtype constraints (and **EXCEPT** clauses) in this way. It means that all the "atomic" constraints that can be applied to a character string type are either purely a size constraint, or purely a permitted-alphabet constraint. The total constraint is made up (only) of (arbitrarily complicated) intersections, unions, and extension additions using such "atomic" units.

B.2.2.12 This significantly simplifies the calculation of what PER calls "effective constraints" on character string types.

B.2.2.13 The third main factor is whether a permitted-alphabet constraint is extensible. Such constraints are not PER-visible either, but their treatment is different from that listed above, as their presence does not affect the visibility of any size constraints that might be present. This area is discussed in B.2.3.

B.2.3 Effective constraints

B.2.3.1 Every constraint on a known-multiplier character string type evaluates to a pair of effective constraints: an effective permitted-alphabet constraint and an effective size constraint. Either or both of these may be extensible, or may be null (no effective constraint).

B.2.3.2 In serial application, only the last constraint can have a member of the pair that is extensible, because of the rules in ITU-T Rec. X.680 | ISO/IEC 8824-1.

B.2.3.3 The definition of an effective size and an effective permitted-alphabet constraint is given in 3.6.8 and 3.6.9 and is not repeated here, but the definition is actually applied to the type with "invisible" constraints removed, as specified in B.2.2.9 and B.2.2.10.

B.2.3.4 As with the removal of constraints that are not PER-visible, replacing an actual constraint by serial application of an effective size constraint and an effective permitted-alphabet constraint adds new abstract values for the purpose of PER encodings (any value with a size in the effective size constraint and using only the effective permitted-alphabet is now included). However, such values will never be transmitted by a conforming application and the effect is again simply to make the PER encoding less efficient than it theoretically could be.

B.2.3.5 EXAMPLE

```
A ::= VisibleString ( SIZE(10) INTERSECTION FROM("A")
                      UNION
                      SIZE(20) INTERSECTION FROM("B") )
```

has only two values, so a one-bit encoding is theoretically possible, but PER encodings use the effective constraints and can encode all one million (approximately) values in:

```
B ::= VisibleString ( SIZE (10 UNION 20)
                      INTERSECTION
                      FROM ("AB") )
```

B.2.3.6 The effective constraints on the union of two sets of values is always the union of the effective constraints on each set of values, but in the general case (if all constraints were PER-visible), this simple rule would not hold for intersection.

B.2.3.7 It is here, however, that the removal of single-value subtype constraints and of **EXCEPT** clauses is important. When all "atomic" constraints are either purely a size constraint or purely a permitted-alphabet constraint (possibly extended), then effective constraints can be calculated for arbitrary set arithmetic (with no **EXCEPT** clauses) in a simple fashion.

B.2.3.8 Let {S, A} represent the set of all values permitted by a size constraint S serially applied with a permitted-alphabet constraint A. (Again, note that union and intersection are commutative.) Then we have:

```
{S1, A1} INTERSECTION {S2, A2} => {S1 INTERSECTION S2,
                                   A1 INTERSECTION A2}
{S1, A1} UNION {S2, A2} => {S1 UNION S2,
                            A1 UNION A2}
{S1, A1}, ... => {S1, ... }
```

B.2.3.9 The last case needs some explanation. An extensible permitted-alphabet constraint has no effect on encoding, as PER does not support a different number of bits for characters needed for root values and for those needed for extension addition values. Thus if an (effective) permitted-alphabet constraint is extensible, it is no longer a constraint - all characters have to be capable of representation. The effect of the "**...**" in the last case is to make both the permitted-alphabet and the size extensible, but only the extensible size remains as a constraint. This is expressed in the normative text by saying that extensible effective permitted-alphabet constraints are not PER-visible.

B.3 Examples

This clause contains a number of examples that provide further illustration.

```
A ::= INTEGER (MIN .. MAX, ..., 1..10)
-- A is extensible, but the root is unconstrained and the
-- extensibility bit is always set to zero

A1 ::= INTEGER (1..32, ..., 33..128)
-- A1 is extensible, and contains values 1 to 128 with 1 to 32 in the
-- root and 33 to 128 as extension additions

A2 ::= INTEGER (1..32, ..., 33..128) (1..128)
-- This is illegal, as 128 is not in the root of the parent
-- (see ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 46)

A3 ::= INTEGER ( (1..32, ..., 33..128) ^ (1..128) )
-- This is legal. A3 is extensible, and contains 1 to 32 in the root
-- and 33 to 128 as extensions

A4 ::= INTEGER (1..32) (MIN .. 63)
-- MIN is 1, and 63 is illegal

A5 ::= INTEGER ( (1..32) ^ (MIN..63) )
-- This is legal. MIN is minus infinity and A3 contains 1 to 32
```

```
A6 ::= INTEGER ( (1..64, ... , B) ^ (1..32) )
-- A6 always contains (only) the values 1..32, no matter what values
-- B contains, but is nonetheless formally extensible and PER will
-- encode all values in 5 bits, with an extensibility bit (always) set
-- to zero

A7 ::= INTEGER (1..32, ... , B) (1..256)
-- A7 is illegal, as the parent for (1..256) can never contain more
-- than 1 to 32 no matter what B contains

A8 ::= IA5String (SIZE(3..4) | SIZE(9..10))
-- A8 has an effective size constraint of SIZE(3..4|9..10)
-- PER will encode as if it were SIZE(3..10), using three bits
-- to encode the length field

A9 ::= IA5String (FROM ("AB") ^ SIZE(1..2) |
                  FROM ("DE") ^ SIZE(3) |
                  FROM ("AXE") ^ (SIZE(1..5) )
-- A9 has an effective size constraint of SIZE(1..5), and PER will
-- encode the length in three bits.  It has an effective alphabet
-- constraint of FROM("ABDEX") and PER will encode each character
-- using three bits

A10 ::= IA5String (SIZE(1..4) | SIZE(5..10) ^
                   FROM("ABCD") | SIZE(6..10))
-- A10 has an effective size constraint of SIZE(1..10), but
-- the permitted alphabet consists of the entire IA5String alphabet

A11 ::= IA5String (SIZE(1..10) | FROM("A".."D"))
-- No size constraint, alphabet is the entire IA5String alphabet

A12 ::= IA5String (SIZE(1..10) ^ FROM("A".."D"), ...)
-- A12 has an extensible effective size constraint of SIZE(1..10,...)
-- and the alphabet is the entire IA5String alphabet

A13 ::= IA5String (SIZE(1..10, ...) ^ FROM("A".."D"))
-- A13 has an extensible effective size constraint of SIZE(1..10,...)
-- and an effective alphabet constraint of FROM("A".."D")

A14 ::= IA5String (SIZE(1..10, ...,29)) (FROM("A".."D"))
-- An effective size constraint of SIZE(1..10), not extensible,
-- because of the serial application of the FROM.  Effective
-- alphabet constraint is FROM("A".."D")

A15 ::= IA5String (SIZE(1..10, ...) | FROM("A".."D"), ...)
-- An extensible effective size constraint, but from MIN to MAX, with
-- all values in the root, encoding with an extensibility bit always
-- set to zero.  The alphabet is the entire IA5String alphabet

A16 ::= IA5String (FROM("A".."D") ^ SIZE(1..10), ...)
-- The effective size constraint is SIZE(1..10,...), extensible
-- The alphabet is the entire IA5String alphabet

A17 ::= IA5String (FROM("A".."D"), ...) (SIZE(1..10))
-- An effective alphabet constraint of FROM("A".."D"), not extensible,
-- because of the serial application of the SIZE.  Effective size
-- constraint is SIZE(1..10)
```

Annex C

Support for the PER algorithms

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

An application standard, or an International Standardized Profile, may specify which of the Packed Encoding Rules are to be supported, and the corresponding transfer syntaxes to be offered or accepted in negotiation.

Where it has requirements for the use of relay safe and/or canonical encodings within **EMBEDDED PDV**s (or **EXTERNAL**s) or **CHARACTER STRING**s, this should be clearly stated.

The following text provides guidelines that can be used in the production of normative text.

C.1 A canonical encoding is intended for use when security features are being applied to the encoding. Use of CANONICAL-PER can involve significant additional CPU utilization cost when the value to be encoded includes a set-of type, and is generally not recommended for protocols unless security features are required.

C.2 Where an abstract syntax value contains embedded material that is encoded using a transfer or abstract syntax different from that associated with the abstract syntax value, it is strongly recommended that the embedded material be encoded in a relay-safe manner. A canonical encoding rule will be required if security features are important. In this context, particular attention should be placed on the level of ISO/IEC 10646-1 which is to be used for the type BMPString or UniversalString, as only ISO/IEC 10646-1 implementation level 1 is guaranteed to be canonical.

C.3 It is strongly recommended that all implementations supporting decoding of any PER ALIGNED variant transfer syntax support decoding of the BASIC-PER ALIGNED variant (and hence of the CANONICAL-PER ALIGNED variant). Similarly for the UNALIGNED variant.

C.4 It is recommended, in the interests of interworking, that all implementations of PER support both the ALIGNED and the UNALIGNED variant (the added implementation complexity is small). Which is offered in an instance of communication (either or both) is a local management matter, and which is accepted if both are offered is also a local management matter. If only one is offered, it should be accepted.

C.5 Acceptance of these recommendations is particularly important for the vendors of general-purpose tools. Where an implementation is specific to some particular application, support of a single PER transfer syntax (perhaps specified by that application designer) may be fully acceptable.

Annex D

Support for the ASN.1 rules of extensibility

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

D.1 These Packed Encoding Rules are dependent on the total definition of the type to which they are applied. In general, if any changes other than those of a purely syntactic nature are made to the type definition, then the encoding for all values using that part of the specification will be affected. In particular, addition of further optional components to a sequence, converting a component to a **CHOICE** of that component and some other type, or relaxing or tightening constraints on some component are all likely to change the encoding of values of the type.

D.2 Nonetheless, these encoding rules have been designed to ensure that the requirements on encoding rules specified in the ASN.1 model of type extension (see ITU-T Rec. X.680 | ISO/IEC 8824-1) are satisfied.

D.3 Where a type is not part of an extension sequence (no extension marker present), then the text earlier in this annex applies: PER provides no support for extensibility of that type. Where a sequence or set type has an extension marker, but no extension additions, then there is a one-bit overhead (which may become one octet due to padding in the ALIGNED variants), compared with the same type without an extension marker. Where additions are present in the type and are actually transmitted in an instance of communication, there is a further overhead of about one octet, plus an additional length field for each extension addition that is transmitted, compared with the same type with the extension marker removed.

D.4 It is important to note that both the addition and removal of an extension marker changes the bits on the line, and will in general require a version number change for the protocol.

D.5 There are no changes to the encoding from the inclusion of an extension marker in an information object set, or from the addition or removal of exception specifications, but these may of course represent changes in the required behavior of an implementation and could still require a version number change for the protocol.

Annex E

Tutorial annex on concatenation of PER encodings

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

E.1 PER encodings are self-delimiting given knowledge of the encoding rules and the type of the encoding. The complete encodings for the ALIGNED and UNALIGNED variant are always a multiple of 8 bits.

E.2 For the purposes of carrying PER encodings in the OSI presentation layer protocol, encodings of the ALIGNED and UNALIGNED variants can be concatenated in the octet string option.

Annex F

Assignment of object identifier values

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

The following object identifier and object descriptor values are assigned in this Recommendation | International Standard:

For BASIC-PER, ALIGNED variant:

```
{joint-iso-itu-t asn1 (1) packed-encoding (3) basic (0) aligned (0)}
"Packed encoding of a single ASN.1 type (basic aligned)"
```

For BASIC-PER, UNALIGNED variant:

```
{joint-iso-itu-t asn1 (1) packed-encoding (3) basic (0) unaligned (1)}
"Packed encoding of a single ASN.1 type (basic unaligned)"
```

For CANONICAL-PER, ALIGNED variant:

```
{joint-iso-itu-t asn1 (1) packed-encoding (3) canonical (1) aligned (0)}
"Packed encoding of a single ASN.1 type (canonical aligned)"
```

For CANONICAL-PER, UNALIGNED variant:

```
{joint-iso-itu-t asn1 (1) packed-encoding (3) canonical (1) unaligned (1)}
"Packed encoding of a single ASN.1 type (canonical unaligned)"
```

ITU-T Recommendation X.692
International Standard 8825-3

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

INTERNATIONAL STANDARD 8825-3

ITU-T RECOMMENDATION X.692

Information Technology – ASN.1 Encoding Rules – Specification of Encoding Control Notation (ECN)

Summary

This Recommendation | International Standard 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).

Source

ITU-T Recommendation X.692 was prepared by ITU-T Study Group 17 (2001-2004) and approved on 8 March 2002. An identical text is also published as ISO/IEC 8825-3.

CONTENTS

Page

Introduction viii
1 Scope 1
2 Normative references 1
2.1 Identical Recommendations | International Standards 1
2.2 Additional references 2
3 Definitions 2
3.1 ASN.1 definitions 2
3.2 ECN-specific definitions 2
4 Abbreviations 5
5 Definition of ECN syntax 5
6 Encoding conventions and notation 5
7 The ECN character set 6
8 ECN lexical items 6
8.1 Encoding object references 6
8.2 Encoding object set references 6
8.3 Encoding class references 7
8.4 Reserved word items 7
8.5 Reserved encoding class name items 7
8.6 Non-ECN item 7
9 ECN Concepts 8
9.1 Encoding Control Notation (ECN) specifications 8
9.2 Encoding classes 8
9.3 Encoding structures 9
9.4 Encoding objects 9
9.5 Encoding object sets 9
9.6 Defining new encoding classes 10
9.7 Defining encoding objects 11
9.8 Differential encoding-decoding 12
9.9 Encoders options in encodings 12
9.10 Properties of encoding objects 12
9.11 Parameterization 12
9.12 Governors 13
9.13 General aspects of encodings 13
9.14 Identification of information elements 14
9.15 Reference fields and determinants 14
9.16 Replacement classes and structures 14
9.17 Mapping abstract values onto fields of encoding structures 15
9.18 Transforms and transform composites 16
9.19 Contents of Encoding Definition Modules 17
9.20 Contents of the Encoding Link Module 17
9.21 Defining encodings for primitive encoding classes 17
9.22 Application of encodings 19
9.23 Combined encoding object set 20
9.24 Application point 20
9.25 Conditional encodings 20
9.26 Changes to ASN.1 Recommendations | International Standards 21
10 Identifying encoding classes, encoding objects, and encoding object sets 21
11 Encoding ASN.1 types 24

11.1 General 24
11.2 Built-in encoding classes used for implicitly generated encoding structures 24
11.3 Simplification and expansion of ASN.1 notation for encoding purposes 25
11.4 The implicitly generated encoding structure 27
12 The Encoding Link Module (ELM) 28
12.1 Structure of the ELM 28
12.2 Encoding types 28
13 Application of encodings 29
13.1 General 29
13.2 The combined encoding object set and its application 29
14 The Encoding Definition Module (EDM) 31
15 The renames clause 33
15.1 Explicitly generated and exported structures 33
15.2 Name changes 34
15.3 Specifying the region for name changes 35
16 Encoding class assignments 36
16.1 General 36
16.2 Encoding structure definition 38
16.3 Alternative encoding structure 41
16.4 Repetition encoding structure 41
16.5 Concatenation encoding structure 41
17 Encoding object assignments 42
17.1 General 42
17.2 Encoding with a defined syntax 43
17.3 Encoding with encoding object sets 44
17.4 Encoding using value mappings 44
17.5 Encoding an encoding structure 45
17.6 Differential encoding-decoding 47
17.7 Encoding options 47
17.8 Non-ECN definition of encoding objects 48
18 Encoding object set assignments 48
18.1 General 48
18.2 Built-in encoding object sets 49
19 Mapping values 50
19.1 General 50
19.2 Mapping by explicit values 51
19.3 Mapping by matching fields 52
19.4 Mapping by #TRANSFORM encoding objects 53
19.5 Mapping by abstract value ordering 53
19.6 Mapping by value distribution 55
19.7 Mapping integer values to bits 56
20 Defining encoding objects using defined syntax 57
21 Types used in defined syntax specification 58
21.1 The Unit type 58
21.2 The EncodingSpaceSize type 58
21.3 The EncodingSpaceDetermination type 59
21.4 The UnusedBitsDetermination type 59
21.5 The OptionalityDetermination type 60
21.6 The AlternativeDetermination type 61
21.7 The RepetitionSpaceDetermination type 61
21.8 The Justification type 62
21.9 The Padding type 63
21.10 The Pattern and Non-Null-Pattern types 63

21.11 The RangeCondition type 64
21.12 The SizeRangeCondition type 64
21.13 The ReversalSpecification type 64
21.14 The ResultSize type 65
21.15 The HandleValue type 65
22 Commonly used encoding property groups 66
22.1 Replacement specification 66
Encoding properties, syntax, and purpose 66
22.1.2 Specification restrictions 67
22.1.3 Encoder actions 68
22.1.4 Decoder actions 69
22.2 Pre-alignment and padding specification 69
Encoding properties, syntax, and purpose 69
22.2.2 Specification constraints 69
22.2.3 Encoder actions 69
22.2.4 Decoder actions 70
22.3 Start pointer specification 70
Encoding properties, syntax, and purpose 70
22.3.2 Specification constraints 70
22.3.3 Encoder actions 70
22.3.4 Decoder actions 71
22.4 Encoding space specification 71
Encoding properties, syntax, and purpose 71
22.4.2 Specification restrictions 72
22.4.3 Encoder actions 72
22.4.4 Decoder actions 73
22.5 Optionality determination 73
Encoding properties, syntax, and purpose 73
22.5.2 Specification restrictions 73
22.5.3 Encoder actions 74
22.5.4 Decoder actions 74
22.6 Alternative determination 75
Encoding properties, syntax, and purpose 75
22.6.2 Specification restrictions 75
22.6.3 Encoder actions 76
22.6.4 Decoder actions 76
22.7 Repetition space specification 76
Encoding properties, syntax, and purpose 76
22.7.2 Specification constraints 77
22.7.3 Encoder actions 78
22.7.4 Decoder actions 79
22.8 Value padding and justification 80
Encoding properties, syntax, and purpose 80
22.8.2 Specification restrictions 80
22.8.3 Encoder actions 81
22.8.4 Decoder actions 81
22.9 Identification handle specification 81
Encoding properties, syntax, and purpose 81
22.9.2 Specification constraints 82
22.9.3 Encoders actions 83
22.9.4 Decoders actions 83
22.10 Concatenation specification 83
Encoding properties, syntax, and purpose 83
22.10.2 Specification constraints 83
22.10.3 Encoder actions 84
22.10.4 Decoder actions 84
22.11 Contained type encoding specification 84
Encoding properties, syntax, and purpose 84
22.11.2 Encoder actions 85

22.11.3 Decoder actions 85
22.12 Bit reversal specification 85
Encoding properties, syntax, and purpose 85
22.12.2 Specification constraints 85
22.12.3 Encoder actions 85
22.12.4 Decoder actions 86
23 Defined syntax specification for bitfield and constructor classes 86
23.1 Defining encoding objects for classes in the alternatives category 86
23.1.1 The defined syntax 86
23.1.2 Purpose and restrictions 87
23.1.3 Encoder actions 87
23.1.4 Decoder actions 87
23.2 Defining encoding objects for classes in the bitstring category 87
23.2.1 The defined syntax 87
23.2.2 Model for the encoding of classes in the bitstring category 88
23.2.3 Purpose and restrictions 88
23.2.4 Encoder actions 89
23.2.5 Decoder actions 89
23.3 Defining encoding objects for classes in the boolean category 90
23.3.1 The defined syntax 90
23.3.2 Purpose and restrictions 91
23.3.3 Encoder actions 91
23.3.4 Decoder actions 92
23.4 Defining encoding objects for classes in the characterstring category 92
23.4.1 The defined syntax 92
23.4.2 Model for the encoding of classes in the characterstring category 93
23.4.3 Purpose and restrictions 93
23.4.4 Encoder actions 94
23.4.5 Decoder actions 94
23.5 Defining encoding objects for classes in the concatenation category 94
23.5.1 The defined syntax 94
23.5.2 Purpose and restrictions 96
23.5.3 Encoder actions 96
23.5.4 Decoder actions 96
23.6 Defining encoding objects for classes in the integer category 97
23.6.1 The defined syntax 97
23.6.2 Purpose and restrictions 97
23.6.3 Encoder actions 97
23.6.4 Decoder actions 97
23.7 Defining encoding objects for the #CONDITIONAL-INT class 97
23.7.1 The defined syntax 97
23.7.2 Purpose and restrictions 99
23.7.3 Encoder actions 99
23.7.4 Decoder actions 100
23.8 Defining encoding objects for classes in the null category 100
23.8.1 The defined syntax 100
23.8.2 Purpose and restrictions 102
23.8.3 Encoder actions 102
23.8.4 Decoder actions 102
23.9 Defining encoding objects for classes in the octetstring category 103
23.9.1 The defined syntax 103
23.9.2 Model for the encoding of classes in the octetstring category 103
23.9.3 Purpose and restrictions 104
23.9.4 Encoder actions 104
23.9.5 Decoder actions 105
23.10 Defining encoding objects for classes in the optionality category 105
23.10.1 The defined syntax 105
23.10.2 Purpose and restrictions 106
23.10.3 Encoder actions 106
23.10.4 Decoder actions 106

23.11 Defining encoding objects for classes in the pad category 106
23.11.1 The defined syntax 106
23.11.2 Purpose and restrictions 107
23.11.3 Encoder actions 107
23.11.4 Decoder actions 108
23.12 Defining encoding objects for classes in the repetition category 108
23.12.1 The defined syntax 108
23.12.2 Purpose and restrictions 108
23.12.3 Encoder actions 108
23.12.4 Decoder actions 108
23.13 Defining encoding objects for the #CONDITIONAL-REPETITION class 108
23.13.1 The defined syntax 108
23.13.2 Purpose and restrictions 110
23.13.3 Encoder actions 110
23.13.4 Decoder actions 111
23.14 Defining encoding objects for classes in the tag category 111
23.14.1 The defined syntax 111
23.14.2 Purpose and restrictions 112
23.14.3 Encoder actions 112
23.14.4 Decoder actions 113
23.15 Defining encoding objects for classes in the other categories 113
24 Defined syntax specification for the #TRANSFORM encoding class 113
24.1 Summary of encoding properties and defined syntax 113
24.2 Source and target of transforms 116
24.3 The int-to-int transform 116
24.4 The bool-to-bool transform 117
24.5 The bool-to-int transform 118
24.6 The int-to-bool transform 118
24.7 The int-to-chars transform 119
24.8 The int-to-bits transform 119
24.9 The bits-to-int transform 121
24.10 The char-to-bits transform 121
24.11 The bits-to-char transform 123
24.12 The bit-to-bits transform 124
24.13 The bits-to-bits transform 124
24.14 The chars-to-composite-char transform 125
24.15 The bits-to-composite-bits transform 125
24.16 The octets-to-composite-bits transform 125
24.17 The composite-char-to-chars transform 126
24.18 The composite-bits-to-bits transform 126
24.19 The composite-bits-to-octets transform 126
25 Complete encodings and the #OUTER class 126
Encoding properties, syntax, and purpose for the #OUTER class 126
25.2 Encoder actions for #OUTER 127
25.3 Decoder actions for #OUTER 128
Annex A Addendum to ITU-T Rec. X.680 | ISO/IEC 8824-1 129
A.1 Exports and imports clauses 129
A.2 Addition of "REFERENCE" 129
A.3 Notation for character string values 130
Annex B Addendum to ITU-T Rec. X.681 | ISO/IEC 8824-2 131
B.1 Definitions 131
B.2 Additional lexical items 131
B.3 Addition of "ENCODING-CLASS" 131
B.4 FieldSpec additions 131
B.5 Fixed-type ordered value list field spec 132
B.6 Fixed-class encoding object field spec 132

B.7 Variable-class encoding object field spec 132
B.8 Fixed-class encoding object set field spec 132
B.9 Fixed-class ordered encoding object list field spec 133
B.10 Encoding class field spec 133
B.11 Ordered value list notation 133
B.12 Ordered encoding object list notation 133
B.13 Primitive field names 134
B.14 Additional reserved words 134
B.15 Definition of encoding objects 134
B.16 Additions to "Setting" 134
B.17 Encoding class field type 135
Annex C Addendum to ITU-T Rec. X.683 | ISO/IEC 8824-4 136
Annex C Addendum to ITU-T Rec. X.683 | ISO/IEC 8824-4 136
C.1 Parameterized assignments 136
C.2 Parameterized encoding assignments 136
C.3 Referencing parameterized definitions 137
C.4 Actual parameter list 137
Annex D Examples 139
D.1 General examples 139
D.2 Specialization examples 146
D.3 Explicitly generated structure examples 154
D.4 A more-bit encoding example 158
D.5 Legacy protocol specified with tabular notation 160
Annex E Support for Huffman encodings 165
Annex F Additional information on the Encoding Control Notation (ECN) 167
Annex G Summary of the ECN notation 168

Introduction

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

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

a) The need to write ASN.1 types (and get the support of ASN.1 tools in implementations) for established ("legacy") protocols where the encoding is already determined and differs from all standardized encoding rules.

b) The need to produce encodings that are minor variations on standardized rules.

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

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

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

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

NOTE – 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 9.26).

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.

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.

2.1 Identical Recommendations | International Standards

- ITU-T Recommendation X.660 (1992) | ISO/IEC 9834-1:1993, *Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: General procedures (plus amendments).*
- ITU-T Recommendation X.680 (2002) | ISO/IEC 8824-1:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*
- ITU-T Recommendation X.681 (2002) | ISO/IEC 8824-2:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.*
- ITU-T Recommendation X.682 (2002) | ISO/IEC 8824-3:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*
- ITU-T Recommendation X.683 (2002) | ISO/IEC 8824-4:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications.*
- ITU-T Recommendation X.690 (2002) | ISO/IEC 8825-1:2002, *Information technology – ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER).*
- ITU-T Recommendation X.691 (2002) | ISO/IEC 8825-2:2002, *Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER).*

NOTE 1 – Notwithstanding the ISO publication date, the above specifications are normally referred to as "ASN.1:2002".

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

2.2 Additional references

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

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

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 ITU-T Rec. X.680 | ISO/IEC 8824-1, ITU-T Rec. X.681 | ISO/IEC 8824-2, ITU-T Rec. X.682 | ISO/IEC 8824-3, ITU-T Rec. X.683 | ISO/IEC 8824-4, ITU-T Rec. X.690 | ISO/IEC 8825-1 and ITU-T Rec. X.691 | ISO/IEC 8825-2.

3.2 ECN-specific definitions

3.2.1 alignment point: 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 3.2.30) 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 bounds condition or size range condition is satisfied.

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 3.2.42.)

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

3.2.25 generated encoding structure: 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 identification handle: Part of an encoding which serves to distinguish encodings of one encoding class from those of other encoding classes.

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

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

3.2.29 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) .

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

3.2.31 negative integer value: A value less than zero.

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

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

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

3.2.35 positive integer value: A value greater than zero.

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

3.2.37 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).

3.2.38 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) is permitted. Recursive definition of an encoding object or an encoding object set is forbidden by 17.1.4 and 18.1.3 respectively.

3.2.39 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) is permitted. Recursive instantiation of an encoding object or an encoding object set is forbidden by 17.1.4 and 18.1.3 respectively.

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

3.2.41 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).

3.2.42 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.

3.2.43 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.

3.2.44 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.

3.2.45 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.

3.2.46 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.

3.2.47 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.

3.2.48 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 3.2.49) 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.

3.2.49 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 classifcation (see 9.18.2)

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

4 Abbreviations

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 ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 5.

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

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

6 Encoding conventions and notation

6.1 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 document refers to the characters specified in ISO 10646-1, and full support for all possible ECN specifications can require the representation of all these characters.

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

7.3 Lexical items defined in clause 8 consist of a sequence of the characters listed in Table 1.

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

Table 1 – ECN characters

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

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

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

8 ECN lexical items

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

NOTE – Annex G lists all lexical items and all the productions used in this Recommendation | International Standard, identifying those that are defined in ITU-T Rec.X.680 | ISO/IEC 8824-1, ITU-T Rec. X.681 | ISO/IEC 8824-2 and ITU-T Rec. X.683 | ISO/IEC 8824-4.

8.1 Encoding object references

Name of item - encodingobjectreference

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

8.2 Encoding object set references

Name of item - encodingobjectsetreference

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

8.3 Encoding class references

Name of item - encodingclassreference

An "encodingclassreference" shall consist of the character "#" followed by the sequence of characters specified for a "typereference" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.2. It shall not be one of the character sequences listed in 8.5 except in an EDM imports list (see ITU-T Rec. X.680 | ISO/IEC 8824-1, **12.19**, as modified by A.1) or in an "ExternalEncodingClassReference" (see the note on 14.11).

8.4 Reserved word items

Names of reserved word items:

ALL	**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 ITU-T Rec. X.681 | ISO/IEC 8824-2, 7.9) used in the definition of encoding classes (within a "WITH SYNTAX" statement) in clause 23 are not reserved words (see also B.14).

8.5 Reserved encoding class name items

Names of reserved encoding class name items:

#ALTERNATIVES	**#GeneralizedTime**	**#PrintableString**
#BITS	**#GeneralString**	**#REAL**
#BIT-STRING	**#GraphicString**	**#RELATIVE-OID**
#BMPString	**#IA5String**	**#REPETITION**
#BOOL	**#INT**	**#SEQUENCE**
#BOOLEAN	**#INTEGER**	**#SEQUENCE-OF**
#CHARACTER-STRING	**#NUL**	**#SET**
#CHARS	**#NULL**	**#SET-OF**
#CHOICE	**#NumericString**	**#TAG**
#CONCATENATION	**#OBJECT-IDENTIFIER**	**#TeletexString**
#CONDITIONAL-INT	**#OCTETS**	**#TRANSFORM**
#CONDITIONAL-REPETITION	**#OCTET-STRING**	**#UniversalString**
#EMBEDDED-PDV	**#OPEN-TYPE**	**#UTCTime**
#ENCODINGS	**#OPTIONAL**	**#UTF8String**
#ENUMERATED	**#OUTER**	**#VideotexString**
#EXTERNAL	**#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 10646-1 character set, except that it shall not be the character sequence "NON-ECN-END" nor shall that character sequence appear within it.

9 ECN Concepts

This clause describes the main concepts underlying 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 ITU-T Rec. X.680 | ISO/IEC 8824-1, **12.12**, 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, and real 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 behavior, 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 ITU-T Rec. X.691 | ISO/IEC 8825-2, 9.3).

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

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

9.11 Parameterization

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

9.11.2 A primary use of parameterization is in the definition of an encoding object that needs the identification of a determinant to complete the definition of the encoding (see 9.13.2). (See D.1.11.3 for an example of a parameterized ECN definition.)

9.11.3 Another important use of parameterization is in the definition of an encoding structure that will be used to replace many different classes in an encoding (see also 9.16.5). For example, the mechanism used to handle optionality is often an immediately (mandatory) preceding "presence-bit" for each optional component. A parameterized structure can be defined consisting of a concatenation of a #BOOLEAN (used as a presence determinant) followed by an optional component defined as a dummy parameter (which will be instantiated with the 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 ITU-T Rec. X.683 | ISO/IEC 8824-4 as modified by C.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. The mechanism is similar in all these cases. Encodings for all values of any given "possible next class" encoding will have the same bit-pattern (their identification) at some place in their encoding (the handle), but the identification for different "possible next class" encodings will be different for each one.

All such encodings can be interpreted by a decoder as an encoding of any "possible next class", and the identification for the handle will determine which "possible next class" encoding is present. The concept is similar to that of using tags for such purposes in BER. Identification handles have names that are required to be unique within an ECN specification.

9.13.4 It is important here to note that ECN allows the definition of encodings in a very flexible way, but cannot guarantee that an encoding specification is correct - that is, that a decoder can successful recover the original abstract values from an encoding. For example, an ECN specifier could assign the same bit-pattern for boolean values true and false. This would be an error, and in this case a tool could fairly easily detect the error. Another error would be to claim that an encoding was self-delimiting (and required no length determinant), when in fact it was not. This error 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 seven forms of this statement, discussed in 9.7 and defined in clause 17).

"EncodingObjectSetAssignment" statements that define sets of encoding objects (see clause 17).

"EncodingClassAssignment" statements that define and name new encoding classes (see clause 15).

9.19.4 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 ITU-T Rec. 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.

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.

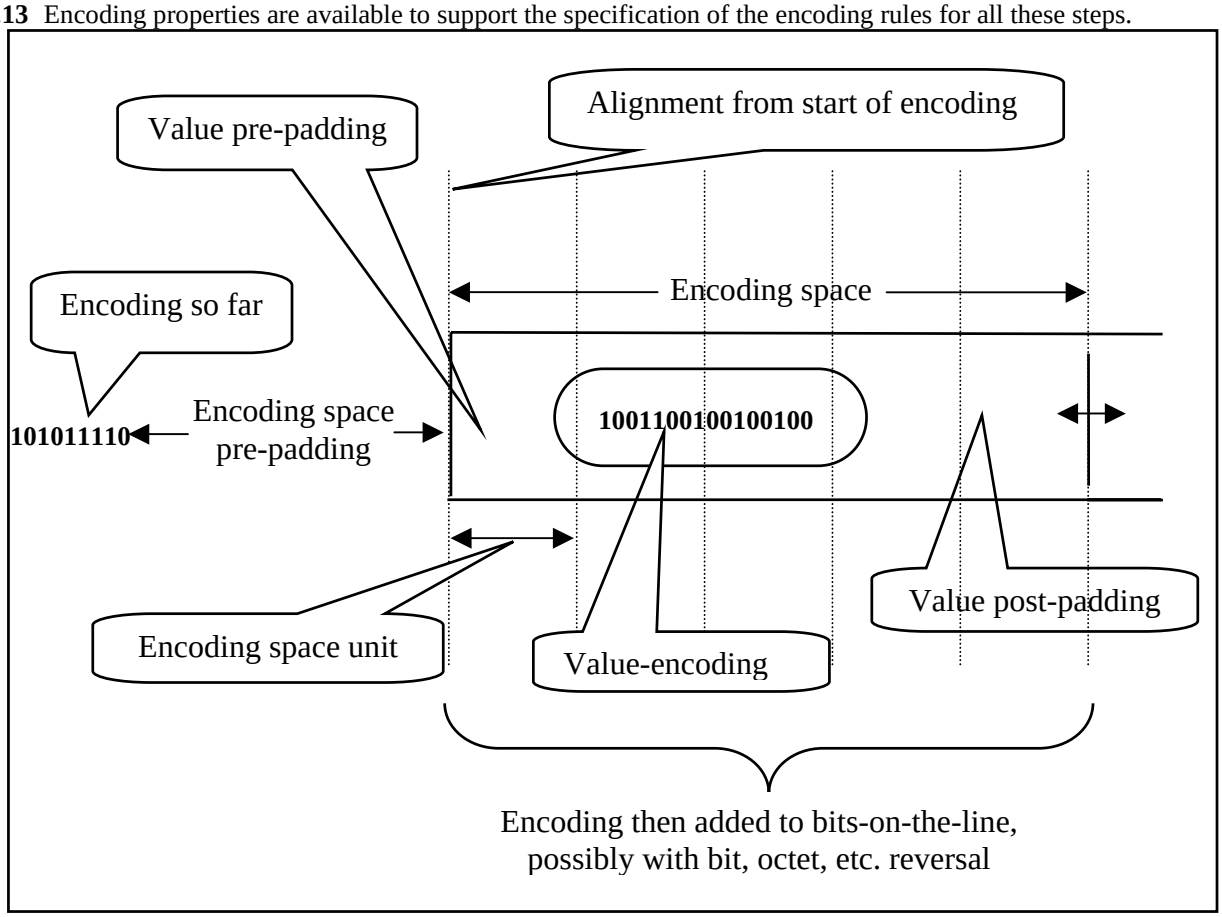


Figure 1 - Encoding space, value-encoding and padding concepts

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 23.13 respectively, or by non-ECN definition of an encoding object, and they can only be used in the definition of other encoding objects. They cannot appear in encoding object sets or be applied directly to encode fields of encoding structures (see 18.1.7).

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 Changes to ASN.1 Recommendations | International Standards

9.26.1 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.

9.26.2 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.

9.26.3 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.

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

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

a) defined implicitly in the ASN.1 module referenced by the "modulereference" (see 11.4.1); or

b) imported into another EDM referenced by the "modulereference" and exported from that module; or

c) generated in a renames clause of another EDM referenced by the "modulereference"; or

d) generated in this EDM in a renames clause, in which case the "modulereference" shall refer to this EDM.

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

10.6.3 The productions defined in 10.6 (except as specified in 10.6.2) shall be used if and only if the corresponding simple reference name has been imported from the module identified by the "modulereference", and either:

a) identical reference names have been imported from different modules, or have been generated in a renames clause in this EDM, or have been both imported and generated; or

b) the simple reference name is a "BuiltinEncodingClassReference" (see 10.5); or

c) both conditions hold.

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

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

10.8 The productions that allow all forms of identification are:

encoding classes	**"EncodingClass" (See clause 16.1.5)**
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 – Sub-clause 16.1.14 specifies additional classes that are used in the definition of user-defined encoding structures.

b) Transformations of the ASN.1 syntax (simplification and expansion) before the implicitly generated structure 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.

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**
GeneralizedTime	**#GeneralizedTime**	**#CHARS**
UTCTime	**#UTCTime**	**#CHARS**
BMPString	**#BMPString**	**#CHARS**
GeneralString	**#GeneralString**	**#CHARS**
GraphicString	**#GraphicString**	**#CHARS**
IA5String	**#IA5String**	**#CHARS**
NumericString	**#NumericString**	**#CHARS**
PrintableString	**#PrintableString**	**#CHARS**
TeletexString	**#TeletexString**	**#CHARS**
UniversalString	**#UniversalString**	**#CHARS**
UTF8String	**#UTF8String**	**#CHARS**
VideotexString	**#VideotexString**	**#CHARS**
VisibleString	**#VisibleString**	**#CHARS**
Textually present tag notation	**#TAG**	**#TAG**

Table 2 - Encoding classes for ASN.1 notation

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 ITU-T Rec. X.680 | ISO/IEC 8824-1 (including Annex F) with all published amendments and technical corrigenda; or

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

d) referenced in those clauses as "See 11.3.2 d" and fully-defined in ITU-T Rec. X.683 | ISO/IEC 8824-4 with all published amendments and technical corrigenda.

11.3.3 The ASN.1 syntactic constructs removed by the expansions and simplifications below are not referenced further in 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 ITU-T Rec. X.691 | ISO/IEC 8825-2, 9.3).

d) If there is a contents constraint with a "CONTAINING" construction, then the existence of the contents constraint, its contents type, and the presence or absence of an "ENCODED BY" clause become properties associated with the abstract values of such a constrained octetstring or bitstring type, and the constraint shall then be discarded. If there is a contents constraint with no "CONTAINING" construction, then it is not visible to ECN and shall be discarded.

NOTE – When specifying encodings for values with an associated contents constraint, a separate combined encoding object set can be supplied to encode the contents type. This can be specified to over-ride or not to over-ride any "ENCODED BY" that is present, as a designer's option (see 11.3 and 13.2).

e) All tagging which is not textually present in the ASN.1 notation shall be ignored in the mapping to encoding structures, but (in order to model BER encodings and PER procedures) the full tag-list of a type becomes a property of the field of the encoding structure to which the corresponding values are mapped.

f) Textually present tag notation has the class of the tag removed. (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 – 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 ITU-T Rec. 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 – 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 – 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 – 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 ITU-T Rec. X.691 | ISO/IEC 8825-2, 9.3) are mapped from the type definition to the corresponding field of the encoding structure.

f) The tag number of 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 ITU-T Rec. X.680 | ISO/IEC 8824-1, **40.5**, **33.5** and **34.5** 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 ITU-T Rec. X.680 | ISO/IEC 8824-1, **12.1**.

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

12.1.5 The "ELMModuleBody" is:

```
ELMModuleBody ::=
        Imports ?
        EncodingApplicationList

EncodingApplicationList ::=
        EncodingApplication
        EncodingApplicationList ?
```

12.1.6 The production "Imports" (and its semantics) is defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, **12.1**, **12.15**, and **12.16**, as modified by A.1 of 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 content constraint is that specified by the encoding object applied to the containing class in the octetstring or bitstring category, and can be any combined encoding object set, or can be the combined encoding object set that was applied to the containing class in the octetstring or bitstring category.

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

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

13 Application of encodings

13.1 General

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

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

13.1.3 The "CombinedEncodings" is:

CombinedEncodings ::=
WITH
PrimaryEncodings
CompletionClause ?

CompletionClause ::=
COMPLETED BY
SecondaryEncodings

PrimaryEncodings ::= EncodingObjectSet

SecondaryEncodings ::= EncodingObjectSet

13.1.4 "EncodingObjectSet" is defined in 18.1.1.

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

13.2 The combined encoding object set and its application

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

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

13.2.3 Otherwise,

a) all encoding objects in the "PrimaryEncodings" are placed in the combined encoding object set, then

b) every encoding object in the "SecondaryEncodings" is added to the combined encoding object set if (and only if) there is no encoding object already in the combined encoding object set that has the same encoding class (see 17.1.7 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 sub-clauses provide the rules for application to a single encoding structure.

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

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

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

a) The alternatives of a constructor that is in the alternatives category.

b) The field following a constructor that is in the repetition category.

c) The components of a constructor that is in the concatenation category.

d) A contained type (a type specified in a contents constraint).

e) The type chosen (in an instance of communication) for use with a class in the 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 component which is an encoding class (possibly preceded by one or more classes in the tag category), followed by an encoding class in the optionality category. The procedures and encodings for determining presence or absence are determined by the specification contained in an encoding object of the class in the optionality category. This encoding object may also require the replacement of the encoding class (together with all its preceding classes in the tag category) with a (parameterized) replacement structure before that class is encoded. The application point then passes to the first class in the tag category (if any), or to the component, or to its replacement.

e) An encoding class preceded by 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.

f) Any other built-in encoding class. This is completely encodable using the specification contained in an encoding object of that class.

13.2.10 Encoding proceeds as follows:

13.2.10.1 If the combined encoding object set contains an encoding object of the same class (see 17.1.7) 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 – This implies that the current combined encoding object set is applied to the type chosen (in an instance of communication) for use with a class in the opentype category (see 13.2.8e).

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

a) The contents constraint contains an "ENCODED BY", and the encoding object for this class either does not contain a specification of the encoding of the contained type, or specifies that it should not override an "ENCODED BY" (see 22.11). In this case the "ENCODED BY" specification shall be used for the contained type, and the application point passes to the contained type using this encoding specification.

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

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

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

13.2.10.7 If there is no encoding object in the combined encoding object set of the same class (see 17.1.7) 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 their components exhibit identification handles (of a given name) to resolve alternatives, or optionality, or order in a set-like concatenation. If in this case the encodings of the components do not exhibit the required identification handles, then the ECN specification is in error.

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

14 The Encoding Definition Module (EDM)

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

14.1 The "EDMDefinition" is:

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

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

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

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

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

14.5 The "EDMModuleBody" is:

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

EDMAssignmentList ::=
        EDMAssignment
        EDMAssignmentList ?


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

14.6 The productions "Exports" and "Imports" (and their semantics) are defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, **12.1**, as modified by A.1 of 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	**Sub-clause C.1**

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

15 The renames clause

15.1 Explicitly generated and exported structures

15.1.1 The production "RenamesAndExports" is:

```
RenamesAndExports ::=
        RENAMES
        ExplicitGenerationList ";"

ExplicitGenerationList ::=
        ExplicitGeneration
        ExplicitGenerationList ?

ExplicitGeneration ::=
        OptionalNameChanges
        FROM GlobalModuleReference

OptionalNameChanges ::=
        NameChanges | GENERATES
```

NOTE – An example of the use of the renames clause to produce explicitly generated encoding structures is given in D.3.7.

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

15.1.3 The "RenamesAndExports" is called a renames clause.

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

15.1.5 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|
#GeneralizedTime|
#UTCTime|
#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
#TRANSFORM	(primitive)	encoding procedure
#GeneralizedTime	#CHARS	
#UTCTime	#CHARS	
#BMPString	#CHARS	
#GeneralString	#CHARS	
#GraphicString	#CHARS	
#IA5String	#CHARS	
#NumericString	#CHARS	
#PrintableString	#CHARS	
#TeletexString	#CHARS	
#UniversalString	#CHARS	
#UTF8String	#CHARS	
#VideotexString	#CHARS	
#VisibleString	#CHARS	

16.2 Encoding structure definition

16.2.1 The "EncodingStructure" is:

```
EncodingStructure ::=
        TaggedStructure|
        UntaggedStructure

TaggedStructure ::=
        "["
        TagClass
        TagValue ?
        "]"
        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|
    #ENUMERATEDBounds?|
    #EXTERNAL|
    #INTEGERBounds?|
    #NULL|
    #OBJECT-IDENTIFIER|
    #OCTET-STRINGSize?|
    #OPEN-TYPE|
    #REAL|
    #RELATIVE-OID|
    #GeneralizedTime|
    #UTCTime|
    #BMPStringSize?|
    #GeneralStringSize?|
    #GraphicStringSize?|
    #IA5StringSize?|
    #NumericStringSize?|
    #PrintableStringSize?|
    #TeletexStringSize?|
    #UniversalStringSize?|
```

```
#UTF8StringSize?|
#VideotexStringSize?|
#VisibleStringSize?
```

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-IDENTIFIERObject identifier values
#OPEN-TYPEOpen type values
#REALReal values
#TAGTag 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 ITU-T Rec. X.680 | ISO/IEC 8824-1, **18.1**. It shall be non-negative when used in "Size". "ValueOrMin" and "ValueOrMax" specify lower and upper bounds respectively.

16.2.12 The "EncodingStructureDefn" is:

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

16.2.13 These encoding structures are defined in the following clauses:

AlternativesStructure 16.3
RepetitionStructure 16.4
ConcatenationStructure 16.5

16.3 Alternative encoding structure

16.3.1 The "AlternativesStructure" is:

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

AlternativesClass ::=
DefinedEncodingClass|
AlternativesClassReference

NamedFields ::= NamedField "," +

NamedField ::=
identifier
EncodingStructure

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

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

16.4 Repetition encoding structure

16.4.1 The "RepetitionStructure" is:

RepetitionStructure ::=
RepetitionClass
"{"
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 3.2.38) of an "encodingobjectreference", and there shall be no recursive instantiation (see 3.2.39) of an "encodingobjectreference".

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. The "encodingobjectreference" being defined exhibits an identification handle if and only if the "DefinedEncodingObject" exhibits that identification handle.

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 ITU-T Rec. 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.

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

Table 3: Categories and classes supported by a defined syntax

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.

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

17.3 Encoding with encoding object sets

17.3.1 The "EncodeWith" is:

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

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

17.4 Encoding using value mappings

17.4.1 The "EncodeByValueMapping" is:

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

ValueMappingEncodingObjects ::=
     EncodingObject|
     DefinedOrBuiltinEncodingObjectSet
```

17.4.2 The production "DefinedOrBuiltinEncodingClass" and its semantics are defined in 10.9.1. It shall be a 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 If the "EncodingObject" alternative of "ValueMappingEncodingObjects" is used, then the "encodingobjectreference" being defined exhibits an identification handle if and only if the "EncodingObject" exhibits that identification handle. If the "DefinedOrBuiltinEncodingObjectSet" alternative of "ValueMappingEncodingObjects" is used to define the encoding of the "DefinedOrBuiltinEncodingClass", then determination of whether the "encodingobjectreference" exhibits an identification handle is in accordance with 17.3.6.

17.5 Encoding an encoding structure

17.5.1 The "EncodeStructure" is:

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

StructureEncoding ::=
                STRUCTURED WITH
            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 The encoding object applied to the governing encoding constructor (whether from "STRUCTURED WITH" 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 any of the "identifier"s that are textually present in the construction obtained by de-referencing the governor. If the governing encoding constructor is a class in the repetition category, the actual parameter for the "REFERENCE" can be any identifier that is textually present in the definition of the "EncodingStructure" in the "RepetitionStructure" of the repetition. 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.

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

17.6 Differential encoding-decoding

17.6.1 The "DifferentialEncodeDecodeObject" is:

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

SpecForEncoding ::= EncodingObject

SpecForDecoders ::= EncodingObject
```

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

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

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

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

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

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

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

17.7 Encoding 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 17.7.5) using any of the "EncodingObject"s in the "EncodingOptionsList". These "EncodingObject"s shall all be encoding objects of the governing class.

NOTE – New implementations are strongly recommended to encode using the earliest "EncodingObject" in the ordered list that is capable of encoding the abstract value to be encoded (see 17.7.5). 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 components 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".

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.

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

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

NOTE 2 - It is possible to use an 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 ITU-T Rec. X.680 | ISO/IEC 8824-1, **12.1**, 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 ITU-T Rec. X.660 | ISO/IEC 9834 series.

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

18 Encoding object set assignments

18.1 General

18.1.1 The "EncodingObjectSetAssignment" is:

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

EncodingObjectSet ::=
    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 3.2.38) of an "encodingclassreference", and there shall be no recursive instantiation (see 3.2.39) of an "encodingclassreference".

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

NOTE – These Recommendations | International Standards were written before this ECN 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.

```
PER-BASIC-ALIGNED{joint-iso-itu-t(2) packed-encoding(3) basic(0) aligned(0)}
PER-BASIC-UNALIGNED      {joint-iso-itu-t(2) packed-encoding(3) basic(0) unaligned(1)}
PER-CANONICAL-ALIGNED{joint-iso-itu-t(2) 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)}
```

Table 4: Built-in encoding object set names and associated object identifiers

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

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

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

18.2.5.1 #INT is encoded as a PER-BASIC-UNALIGNED #INTEGER encoding, provided it is bounded. It is an ECN design error if the #INT does not have both a lower and an upper bound when this encoding object is applied to the #INT.

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

18.2.5.3 #CHARS, #OCTETS, and #BITS are encoded as PER-BASIC-UNALIGNED "UTF8String", #OCTET-STRING, and #BIT-STRING, respectively, provided they are a single size. It is an ECN design error if #CHARS, #OCTETS, or #BITS do not have an effective size constraint restricting them to a single size.

18.2.5.4 #CONCATENATION is encoded as a PER-BASIC-UNALIGNED encoding of a #SEQUENCE with no optional 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 to the type chosen for use with the #OPEN-TYPE class is always the same as the combined encoding object set applied to the #OPEN-TYPE class (see 13.2.10.5).

19 Mapping values

19.1 General

19.1.1 This clause specifies the syntax for mapping values (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 ITU-T 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:

MappingByExplicitValues19.2
MappingByMatchingFields19.3
MappingByTransformEncodingObjects19.4
MappingByAbstractValueOrdering19.5
MappingByValueDistribution19.6
MappingIntToBits19.7

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

19.2 Mapping by explicit values

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

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

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

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

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

Category of governing encoding class	ASN.1 value notation
bitstring	**"bstring" or "hstring"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, **11.10** and **11.12**)
boolean	**"BooleanValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, **17.3**)
characterstring	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, **37.8**)
integer	**"SignedNumber"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, **18.1**)
null	**"NullValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, **23.3**)
objectidentifier	**"DefinitiveIdentifier"** (see A.1)
octetstring	**"bstring" or "hstring"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, **11.10** and **11.12**)
real	**"RealValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, **20.6**)

Table 5: Categories of encoding classes and value notation used in mapping by explicit values

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 sub-clause 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, 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 in 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 ITU-T Rec. X.680 | ISO/IEC 8824-1, **18.1**.

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 ITU-T Rec. X.680 | ISO/IEC 8824-1, **18.1**, 11.10, and 11.12, 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 ITU-T Rec. X.690 | ISO/IEC 8825-1, **8.3.3**), "BitRangeValue2" shall represent an integer value ("B", say) greater than that represented by "BitRangeValue1" ("A", say), and the difference between the integer values corresponding to "BitRangeValue2" and "BitRangeValue1" ("B" - "A") shall equal the difference between the values of "IntRangeValue2" and "IntRangeValue1".

19.7.17 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 ITU-T Rec. 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 23.13.2.5).

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 by the encoding object for the optional component and by any possible alternative encoding that can follow if this optional component is absent, and the value of the handle shall be different for the encoding of the optional component and all possible alternative encodings that can follow. 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 this optional component, then it is absent. Otherwise, a 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 that 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.

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

21.7 The RepetitionSpaceDetermination type

21.7.1 The "RepetitionSpaceDetermination" type is:

```
RepetitionSpaceDetermination ::= ENUMERATED
      {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 by the element being repeated, and by all possible (taking account of optionality) following elements. The value of the identification handle for the element being repeated shall be different from that of all possible following elements.

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 10646-1 value as an 8-bit value.

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

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

21.10.8 The "any-of-length" alternative specifies a size for the pattern. 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}
```

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 enumeration value if and only if the following conditions are satisfied by the bounds on the encoding class in the integer category:

a) **unbounded-or-no-lower-bound**: either there are no bounds, or else there is only an upper bound but no lower bound.

b) **semi-bounded-with-negatives**: there is a lower bound that is less than zero, but no upper bound.

c) **bounded-with-negatives**: there is a lower bound that is less than zero, and an upper bound.

d) **semi-bounded-without-negatives**: there is a lower bound that is greater than or equal to zero, but no upper bound..

e) **bounded-without-negatives**: there is a lower bound that is greater than or equal to zero, and an upper bound

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

21.12 The SizeRangeCondition type

21.12.1 The "SizeRangeCondition" type is:

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

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

21.12.3 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.

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

a) **no-ub-with-zero-lb**: there is no upper bound on the size and the lower bound is zero.

b) **ub-with-zero-lb**: there is an upper bound on the size and the lower bound is zero.

c) **no-ub-with-non-zero-lb**: there is no upper bound on the size and the lower bound is non-zero.

d) **ub-with-non-zero-lb**: there is an upper bound on the size and the lower bound is non-zero.

e) **fixed-size**: the lower bound and the upper bound on the size are the same value.

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

21.13 The ReversalSpecification type

21.13.1 The "ReversalSpecification" type is:

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

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

21.13.3 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.

21.13.4 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).

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

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

a) no reversal of bits, or

b) reversal of the order of half-units (without changing the order of bits in each half unit), or

c) reversal of the order of bits in each half-unit but without reversing the order of the half-units, or

d) reversal of the order of the bits in each unit.

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

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

21.14 The ResultSize type

21.14.1 The "ResultSize" type is:

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

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

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

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

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

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

21.15 The HandleValue type

21.15.1 The "HandleValue" type is:

```
HandleValue ::= CHOICE {
      Bits        BIT  STRING,
      octets      OCTET STRING,
      number      INTEGER (0..MAX),
      tag         ENUMERATED {any}}
```

21.15.2 The "HandleValue" is used to specify the value of an identification handle that is exhibited by particular encoding objects.

21.15.3 Values of any identification handle that is exhibited by an encoding object are required to be the same for all abstract values which that encoding object encodes (see 22.9.2.2). The value of an identification handle can be used to identify the presence or absence of optional components, the choice of alternatives, or the end of a repetition. There are requirements in such circumstances that the handle values exhibited by the encoding of different alternatives or components be distinct (see 21.5.7, 21.6.6 and 21.7.10).

NOTE – Values of identification handles exhibited by a given encoding object can, in theory, be determined by encoding a trial value. However, to ease the implementation task, the ECN specifier is required to specify the value of the handle in all cases except where (for encodings of the tag class) the value of the identification handle 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.

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

21.15.5 The "tag:any" alternative specifies that the handle value is 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.

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 order of the original components.

22.1.2 Specification restrictions

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

22.1.2.2 The "WITH" replacement structures shall be parameterized encoding structures with a single encoding class parameter. When they are specified in the above defined syntax, 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 "HANDLE" is specified, then the component whose presence is being determined, together with all following optional and the next mandatory encoding (if any) shall all be produced by encoding objects whose specifications all exhibit an identification handle with the same name as "HANDLE". The next mandatory encoding may be a component of the concatenation containing the optional component, or may be an encoding following the concatenation. The value of the identification handle shall be different for all these components.

NOTE – It is a requirement that the bits that form an identification handle shall have the same value for all abstract values encoded by an encoding object exhibiting that identification handle (see 22.9.2.2).

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 match the value of the identification handle 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 "HANDLE" is specified, then all the alternatives of the encoding class in the alternatives category shall be encoded by encoding objects whose specification exhibits and defines an identification handle with the same name as "HANDLE", and with the same value of the identification handle. The value of the identification handle shall be different for all these alternatives.

NOTE – It is a requirement that an identification handle shall have the same value for all abstract values encoded by an encoding object exhibiting that identification handle (see 22.9.2.2).

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 value of the identification handle 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 '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 "HANDLE" is specified, then the repeated element, together with any element which (through the use of optionality) may follow the repeated element shall all be encoded by encoding objects whose specification exhibits an identification handle with the same name as "HANDLE". The value of the identification handle in the repeating element shall be different from that of any possible following element.

NOTE – It is a requirement that an identification handle shall have the same value for all abstract values encoded by an encoding object exhibiting that identification handle (see 22.9.2.2).

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 "REPETITIION-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 element (in parallel) as either a further repetition or as a following element, 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 OPTIONAL,
        &Handle-positions               INTEGER (0..MAX) OPTIONAL,
        &handle-value                   HandleValue 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]]
```

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

```
HandleValue ::= CHOICE {
      Bits        BIT  STRING,
      octets      OCTET STRING,
      number      INTEGER (0..MAX),
      tag         ENUMERATED {any}}  -- (see 21.15)
```

22.9.1.4 This specification is used to identify that an encoding object exhibits an identification handle within all its encodings (that is, for all possible abstract values that it encodes). The name of the identification handle is specified, and the bits that are associated with that identification handle. The value of the identification handle is specified by "HandleValue".

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 "HandleValue" 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 "HandleValue" 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 "HandleValue" 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 "HandleValue".

22.9.1.9 The "HandleValue" 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 "HandleValue". If, however, a value is specified by "HandleValue" 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 application of ECN, all identification handles with the same name shall specify the same set of bits for the location of the identification handle.

NOTE – There is no general requirement that the value of the identification handle (exhibited by different encoding objects) should be distinct, but distinct values are required when the identification handle is used to resolve optionality, alternative selection, or repetition termination (see 21.5.7, 21.6.6 and 21.7.10).

22.9.2.2 The ECN specifier shall ensure that any encoding object exhibiting an identification handle produces the same value of the identification handle for every abstract value that is encoded.

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 If an encoding object for a class in the repetition category exhibits an identification handle, then that identification handle shall also be exhibited (with the same value) by the encoding of the repeated element.

22.9.2.5 If an encoding object for a class in the alternatives category exhibits an identification handle, then that identification handle shall also be exhibited by (the encoding of) all alternatives, and the value of the identification handle shall be the same for all the alternatives.

NOTE – In this case that identification handle cannot be used for alternative determination in this alternative, and alternative determination has either to be done using a different identification handle or by some other means.

22.9.2.6 If an encoding object for a class in the concatenation category exhibits an identification handle, then the first (if any) encoded component (or, if it is tagged, the tag), taking account of optionality, shall exhibit that identification handle with the same value.

22.9.2.7 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 encoding has the value of the identification handle, and shall diagnose a specification or application error otherwise.

22.9.4 Decoders actions

22.9.4.1 There are no decoders actions directly resulting from the exhibition of an identification handle. Decoder actions only result from use of the identification handle to determine optionality, end of repetitions, or choice of alternatives.

22.10 Concatenation specification

22.10.1 Encoding 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 all components shall exhibit "HANDLE" with distinct values for the identification handle.

22.10.2.2 If "ALIGNMENT" is "aligned", then the pre-alignment specification assumes the default value unless set.

22.10.2.3 If a component has its own explicit pre-alignment, this is applied after any pre-alignment of the component resulting from the setting of "ALIGNMENT" in the encoding class of the concatenation category.

NOTE – The equivalent function is not provided for repetitions, as it can be achieved more simply by pre-alignment of the single component.

22.10.2.4 If "ORDER" is "tag", then every component shall start with an encoding class in the tag category. The tag number associated with this class is called the component-tag.

22.10.2.5 The component-tags of each alternative shall be distinct.

22.10.2.6 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 sub-clause 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 bits associated with "HANDLE".

22.10.4.4 Each component has a distinct value for the bits associated with "HANDLE" that enables the component to be identified. Decoding shall proceed until an abstract value for every component has been obtained, and 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

22.11.3.1 A decoder shall decode the contained type in accordance with the encoding applied by the encoder, as specified above.

22.12 Bit reversal specification

22.12.1 Encoding 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 21.13)
```

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 bitfield and constructor classes

This clause provides the full syntax for defining encoding objects of each encoding class in the different categories.

NOTE – Encoder and decoder actions are specified in the following clauses as conditional on 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 or component replacement specification (see 22.1)
      &#Replacement-structure                                    OPTIONAL,
      &replacement-structure-encoding-object  &#Replacement-structure   OPTIONAL,
      &#Head-end-structure                                       OPTIONAL,

      -- Pre-alignment and padding specification (see 22.2)
      &encoding-space-pre-alignment-unit Unit (ALL EXCEPT repetitions) DEFAULT bit,
      &encoding-space-pre-padding        Padding DEFAULT zero,
      &encoding-space-pre-pattern        Non-Null-Pattern (ALL EXCEPT different:any)
                                         DEFAULT bits:'0'B,

      -- Start pointer specification (see 22.3)
      &start-pointer                     REFERENCE OPTIONAL,
      &start-pointer-unit                Unit (ALL EXCEPT repetitions) DEFAULT bit,
      &Start-pointer-encoder-transforms  #TRANSFORM ORDERED OPTIONAL,

      -- 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 OPTIONAL,
      &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
      &handle-value                      HandleValue DEFAULT tag:any

} WITH SYNTAX {
      [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]]
    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]]

}
```

23.1.2 Purpose and restrictions

23.1.2.1 This syntax is used to define the start of the encoding space for an encoding class in the alternatives category, the determination of the alternative that has been encoded, and an optional declaration that all encodings exhibit a specified identification handle (with distinct identification handle values).

23.1.2.2 If "REPLACE STRUCTURE" is set, then no other encoding property groups shall be set.

23.1.2.3 Encodings of this class do not exhibit an identification handle unless "EXHIBITS HANDLE" is set (even if all components exhibit an identification handle, that may or may not be the same).

23.1.2.4 If "EXHIBITS HANDLE" is set, then encodings of all the alternatives of this class are required to exhibit the defined identification handle, and to have distinct values for that identification handle.

NOTE – This would normally require that every component had an "EXHIBITS HANDLE" set to the same value, unless a head-end insertion exhibited the identification handle (see 9.10.3).

23.1.3 Encoder actions

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

    -- Start pointer specification (see 22.3)
    &start-pointer                     REFERENCE OPTIONAL,
    &start-pointer-unit                Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms  #TRANSFORM ORDERED OPTIONAL,
```

```
        -- Bits value encoding
        &value-reversal                 BOOLEAN DEFAULT FALSE,
        &Transforms                     #TRANSFORM ORDERED OPTIONAL,
        &Bits-repetition-encodings      #CONDITIONAL-REPETITION ORDERED OPTIONAL,
        &bits-repetition-encoding       #CONDITIONAL-REPETITION OPTIONAL

        -- Identification handle specification (see 22.9)
        &exhibited-handle               PrintableString OPTIONAL,
        &Handle-positions               INTEGER (0..MAX) OPTIONAL,
        &handle-value                   HandleValue 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]]
        [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 all bits encodings exhibit a specified identification handle, 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.

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 3.2.41).

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", then all preceding encoding objects in that list shall be defined using "IF".

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 all encodings of values associated with this class shall exhibit the specified identification handle.

NOTE – This will in general require restrictions on the abstract values of the associated type or the addition of redundant bits in the transform into bits, or both.

23.2.3.11 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 OPTIONAL,
      &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
      &handle-value                      HandleValue 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]]
      [BIT-REVERSAL &bit-reversal]
}
```

23.3.2 Purpose and restrictions

23.3.2.1 This syntax is used to define the start of the encoding space for a class in the boolean category, the encoding of the abstract values of that class, their positioning within the encoding space, an optional declaration that all bits encodings exhibit a specified identification handle, and possible bit-reversal of the encoding space for the boolean.

23.3.2.2 If "REPLACE" is set, then no other encoding 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 3.2.41).

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 3.2.41).

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 OPTIONAL,
    &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
```

```
        &handle-value                    HandleValue 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]]

    }
```

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 #REPETITION-ENCODING that are used in the specification of this encoding object.

23.4.3 Purpose and restrictions

23.4.3.1 This syntax is used to define the start of the encoding space for a class in the characterstring category, the encoding of the abstract values associated with that class, an optional declaration that all chars encodings exhibit a specified identification handle.

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

23.4.3.3 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.

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 3.2.41).

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", then all preceding encoding objects in that list shall be defined using "IF".

23.4.3.9 If "EXHIBITS HANDLE" is set, then all encodings of values associated with this class shall exhibit the specified identification handle.

NOTE – This will in general require restrictions on the abstract values of the associated type, or the 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 OPTIONAL,
      &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
      &handle-value                      HandleValue 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]]
          [BIT-REVERSAL &bit-reversal]
     }
```

23.5.2 Purpose and restrictions

23.5.2.1 This syntax is used to define the start of the encoding space for a class in the concatenation category, the way in which the encodings of the components are to be combined, their positioning within the encoding space, an optional declaration that all encodings exhibit a specified identification handle, and possible bit-reversal of the encoding space.

23.5.2.2 If "REPLACE STRUCTURE" is set, then no other encoding parameter groups shall be set.

23.5.2.3 "ENCODING-SPACE SIZE" shall be either "variable-with-determinant" or "self-delimiting-values".

23.5.2.4 If "EXHIBITS HANDLE" is set then the encoding of all possible abstract values associated with this class shall exhibit the defined identification handle

NOTE – This would often be achieved by ensuring that the first component of the concatenation, or a head-end insert, exhibited the identification handle.

23.5.3 Encoder actions

23.5.3.1 For any encoding 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", then all preceding encoding objects in that list shall be defined using "IF".

23.6.3 Encoder actions

23.6.3.1 The encoder shall select and apply the first #CONDITIONAL-INT encoding object in "ENCODING(S)" whose conditions are satisfied. It is an ECN specification error if none of the conditional encodings have conditions that are satisfied.

NOTE – It would be unusual but not illegal if there were #CONDITIONAL-INT encoding objects present that could never be used because the conditions on use of earlier encoding objects would always be satisfied.

23.6.4 Decoder actions

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

23.7 Defining encoding objects for the #CONDITIONAL-INT class

23.7.1 The defined syntax

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

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

      -- Condition (see 21.11)
      &range-condition                        RangeCondition  OPTIONAL,

      -- Structure-only replacement specification (see 22.1)
      &#Replacement-structure                                         OPTIONAL,
      &replacement-structure-encoding-object  &#Replacement-structure OPTIONAL,

      -- Pre-alignment and padding specification (see 22.2)
      &encoding-space-pre-alignment-unit Unit (ALL EXCEPT repetitions) DEFAULT bit,
```

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

        -- Start pointer specification (see 22.3)
        &start-pointer                    REFERENCE OPTIONAL,
        &start-pointer-unit               Unit (ALL EXCEPT repetitions) DEFAULT bit,
        &Start-pointer-encoder-transforms #TRANSFORM ORDERED OPTIONAL,

        -- Encoding space specification (see 22.4)
        &encoding-space-size              EncodingSpaceSize
                                          DEFAULT self-delimiting-values,
        &encoding-space-unit              Unit (ALL EXCEPT repetitions)
                                          DEFAULT bit,
        &encoding-space-determination     EncodingSpaceDetermination
                                          DEFAULT 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 OPTIONAL,
        &Handle-positions                 INTEGER (0..MAX) OPTIONAL,
        &handle-value                     HandleValue DEFAULT tag:any,

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

    } WITH SYNTAX {
        [IF &range-condition] [ELSE]
        [REPLACE
            [STRUCTURE]
            WITH &#Replacement-structure
                [ENCODED BY &replacement-structure-encoding-object]]
        [ALIGNED TO
            [NEXT]
            [ANY]
                &encoding-space-pre-alignment-unit
                [PADDING &encoding-space-pre-padding
                [PATTERN &encoding-space-pre-pattern]]
        [START-POINTER   &start-pointer
            [MULTIPLE OF     &start-pointer-unit]
            [ENCODER-TRANSFORMS    &Start-pointer-encoder-transforms]]
        ENCODING-SPACE
            [SIZE &encoding-space-size
                [MULTIPLE OF &encoding-space-unit]]
            [DETERMINED BY &encoding-space-determination]
            [USING &encoding-space-reference
                [ENCODER-TRANSFORMS &Encoder-transforms]
                [DECODER-TRANSFORMS &Decoder-transforms]]
        [TRANSFORMS     &Transforms]
```

```
        [ENCODING       &encoding]
        [VALUE-PADDING
              [JUSTIFIED &value-justification]
              [PRE-PADDING &value-pre-padding
                    [PATTERN &value-pre-pattern]]
              [POST-PADDING &value-post-padding
                    [PATTERN &value-post-pattern]]
              [UNUSED BITS
                    [DETERMINED BY &unused-bits-determination]
                    [USING &unused-bits-reference
                          [ENCODER-TRANSFORMS &Unused-bits-encoder-transforms]
                          [DECODER-TRANSFORMS &Unused-bits-decoder-transforms]]]]
        [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
              [AS &handle-value]]
        [BIT-REVERSAL &bit-reversal]
}
```

23.7.2 Purpose and restrictions

23.7.2.1 This syntax is used to define a #CONDITIONAL-INT encoding object. The only use of such an encoding object is in the specification of an encoding object of a class in the integer category.

23.7.2.2 The syntax allows the specification of a single condition on the bounds of the integer for this encoding to be applied (use of "IF"). It also allows the specification that there is no condition. The use of "ELSE", or omission of both "IF" and "ELSE" specifies that there is no condition.

23.7.2.3 Using this syntax the ECN specifier can define the start of the encoding space for the encoding of a class in the integer category, the encoding of the abstract values associated with that class, their positioning within the encoding space, and possible bit-reversal of the encoding space.

23.7.2.4 At most one of "IF" and "ELSE" shall be present.

23.7.2.5 If "REPLACE" is set, then no other encoding 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" condition takes place on the bounds of the original value, and is not affected by these transforms.

23.7.2.7 The "INT-TO-INT" transform with the value "subtract:lower-bound" shall be included only if the "IF" condition restricts the application of this encoding to classes of the integer category with a lower bound, and (if present) shall be the first transform in the list.

23.7.2.8 The "ENCODING-SPACE SIZE" shall not be "fixed-to-max" unless the "IF" condition restricts the encoding to a class with both an upper and a lower bound.

23.7.2.9 "ENCODING-SPACE SIZE" shall not be set to "self-delimiting-values".

23.7.2.10 If "EXHIBITS HANDLE" is set, then the specifier asserts that the encoding of all values exhibits the identification handle.

NOTE – This will normally require use of "VALUE-PADDING" with justification from the left to allow the padding to exhibit the identification handle.

23.7.3 Encoder actions

23.7.3.1 The encoder shall detect an ECN specification or application error if any of the restrictions in 23.7.2 are violated.

23.7.3.2 For any encoding 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 ITU-T Rec. X.690 | ISO/IEC 8825-1, **8.3.2** and **8.3.3**. A reversal of these encodings is an encoding in which, following production of the "n" bits, the order of the "n" bits is reversed.

23.7.3.6 An encoder shall detect an ECN specification or an application error if a value is to be encoded into a number of bits which is insufficient, as specified in 23.7.3.4.

23.7.3.7 If the "ENCODING-SPACE SIZE" is a positive integer, then its size in bits is calculated as "SIZE" multiplied by "MULTIPLE OF" units. If "VALUE-PADDING" is not set, then this shall be the number of bits "n" that the integer shall encode into and there are no unused bits. If "VALUE-PADDING" is set, then the number of bits that the integer shall encode into is reduced by the integer value "m" specified for "JUSTIFIED", and there will be "m" unused bits.

23.7.3.8 If the "ENCODING-SPACE SIZE" is "fixed-to-max", then the encoder shall determine the minimum number of "MULTIPLE OF" units that has sufficient bits to encode any of the values of the class, and shall proceed (as specified above) as if "SIZE" were a positive integer set to that value.

23.7.3.9 If the "ENCODING-SPACE SIZE" is "variable-with-determinant", then the encoder shall determine the minimum number of "MULTIPLE OF" units ("s", say) that has sufficient bits to encode the actual abstract value being encoded, and shall proceed (as specified above) as if "SIZE" were a positive integer set to that value.

23.7.3.10 The encoder (as an encoder's option) may increase "s" (as determined in 23.7.3.9) in "MULTIPLE OF" units (subject to any restrictions that the range of values of any "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-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 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 OPTIONAL,
      &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
      &handle-value                      HandleValue 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]]
          [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.

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

      -- Start pointer specification (see 22.3)
      &start-pointer                     REFERENCE OPTIONAL,
      &start-pointer-unit                Unit (ALL EXCEPT repetitions) DEFAULT bit,
      &Start-pointer-encoder-transforms  #TRANSFORM ORDERED OPTIONAL,

      -- Octets value encoding
      &value-reversal                    BOOLEAN DEFAULT FALSE,
      &Transforms                        #TRANSFORM ORDERED OPTIONAL,
      &Octets-repetition-encodings       #CONDITIONAL-REPETITION ORDERED OPTIONAL,
      &octets-repetition-encoding        #CONDITIONAL-REPETITION OPTIONAL,

      -- Identification handle specification (see 22.9)
      &exhibited-handle                  PrintableString OPTIONAL,
      &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
      &handle-value                      HandleValue 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]]
      [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 all octetstring encodings exhibit a specified identification handle, 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.

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 3.2.41).

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", then all preceding encoding objects in that list shall be defined using "IF".

23.9.3.9 If "EXHIBITS HANDLE" is set, then all encodings of values of this class shall exhibit the specified identification handle.

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

23.9.3.10 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 optionality category

23.10.1 The defined syntax

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

```
#OPTIONAL ::= ENCODING-CLASS {

      -- Structure-only replacement specification (see 22.1)
      &#Replacement-structure                                    OPTIONAL,
      &replacement-structure-encoding-object  &#Replacement-structure   OPTIONAL,

      -- Pre-alignment and padding specification (see 22.2)
      &encoding-space-pre-alignment-unit Unit (ALL EXCEPT repetitions) DEFAULT bit,
      &encoding-space-pre-padding        Padding DEFAULT zero,
      &encoding-space-pre-pattern        Non-Null-Pattern (ALL EXCEPT different:any)
                                         DEFAULT bits:'0'B,

      -- Start pointer specification (see 22.3)
      &start-pointer                     REFERENCE OPTIONAL,
      &start-pointer-unit                Unit (ALL EXCEPT repetitions) DEFAULT bit,
      &Start-pointer-encoder-transforms  #TRANSFORM ORDERED OPTIONAL,

      -- Optionality determination (see 22.5)
      &optionality-determination         OptionalityDetermination
                                         DEFAULT 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]]
```

```
}
```

23.10.2 Purpose and restrictions

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

23.10.2.2 If "REPLACE STRUCTURE" is set, then no other encoding property groups shall be set.

23.10.3 Encoder actions

23.10.3.1 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 23.10.3.2).

b) Pre-alignment and padding.

c) Start pointer.

d) Optionality determination.

23.10.3.2 If "REPLACE STRUCTURE" is set then the entire 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.

23.10.4 Decoder actions

23.10.4.1 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.

23.11 Defining encoding objects for classes in the pad category

23.11.1 The defined syntax

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

```
#PAD ::= ENCODING-CLASS {

    -- Structure-only replacement specification (see 22.1)
    &#Replacement-structure                                    OPTIONAL,
    &replacement-structure-encoding-object  &#Replacement-structure   OPTIONAL,

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding        Padding DEFAULT zero,
    &encoding-space-pre-pattern        Non-Null-Pattern (ALL EXCEPT different:any)
                                       DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                     REFERENCE OPTIONAL,
    &start-pointer-unit                Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms  #TRANSFORM ORDERED OPTIONAL,

    -- Encoding space specification (see 22.4)
    &encoding-space-size               EncodingSpaceSize
                                       DEFAULT self-delimiting-values,
    &encoding-space-unit               Unit (ALL EXCEPT repetitions)
                                       DEFAULT bit,
    &encoding-space-determination      EncodingSpaceDetermination
                                       DEFAULT 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 OPTIONAL,
        &Handle-positions                INTEGER (0..MAX) OPTIONAL,
        &handle-value                    HandleValue 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]]
        [BIT-REVERSAL &bit-reversal]

    }
```

23.11.2 Purpose and restrictions

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

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

23.11.2.3 If "REPLACE STRUCTURE" is set, then no other encoding property group shall be set.

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.

b) Pre-alignment and padding.

c) Start pointer.

d) Encoding space.

e) Value encoding (see below).

f) Identification handle.

g) Bit reversal.

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

23.11.3.3 "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 23.11.3.4).

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

23.11.3.4 An encoder (as an encoder's option) may increase "s" (as determined in 23.11.3.3) in "MULTIPLE OF" units (subject to any restrictions that the range of values of any "field-to-be-set" or "field-to-be-used" imposes) if "ENCODING-SPACE SIZE" is set to "encoder-option-with-determinant".

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) Bit reversal.

d) Encoding space.

23.11.4.2 The decoder shall determine the size of the pad value encoding, and identify those bits in the encoding, but shall silently accept any value for those bits.

23.12 Defining encoding objects for classes in the repetition category

23.12.1 The defined syntax

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

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

      -- Repetition encoding
      &Repetition-encodings      #CONDITIONAL-REPETITION ORDERED OPTIONAL,
      &repetition-encoding       #CONDITIONAL-REPETITION OPTIONAL

} WITH SYNTAX {
      [REPETITION-ENCODINGS &Repetition-encodings]
      [REPETITION-ENCODING &repetition-encoding]

}
```

23.12.2 Purpose and restrictions

23.12.2.1 This syntax is used to define the encoding of a class in the repetition category by specifying one or more encodings of the #CONDITIONAL-REPETITION class.

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

NOTE – The sole purpose of allowing "REPETITION-ENCODING" as well as "REPETITION-ENCODINGS" is to provide a syntax that does not contain a double curly-bracket ("{{") in the common case of a single encoding object. Use of "REPETITION-ENCODINGS" when there is a single encoding object is deprecated but is allowed.

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

23.12.3 Encoder actions

23.12.3.1 The encoder shall select and apply the first #CONDITIONAL-REPETITION encoding object in "ENCODING(S)" whose conditions are satisfied. It is an ECN specification error if none of the conditional encodings have conditions that are satisfied.

NOTE – It would be unusual but not illegal if there were #CONDITIONAL-REPETITION encoding objects present that could never be used because the conditions on use of earlier encoding objects would always be satisfied.

23.12.4 Decoder actions

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

23.13 Defining encoding objects for the #CONDITIONAL-REPETITION class

23.13.1 The defined syntax

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

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

```
      -- Condition (see 21.12)
      &size-range-condition              SizeRangeCondition  OPTIONAL,

      -- Structure or component replacement specification (see 22.1)
      &#Replacement-structure                                        OPTIONAL,
      &replacement-structure-encoding-object  &#Replacement-structure  OPTIONAL,
      &#Head-end-structure                                           OPTIONAL,

      -- Pre-alignment and padding specification (see 22.2)
      &encoding-space-pre-alignment-unit Unit (ALL EXCEPT repetitions) DEFAULT bit,
      &encoding-space-pre-padding        Padding DEFAULT zero,
      &encoding-space-pre-pattern        Non-Null-Pattern (ALL EXCEPT different:any)
                                         DEFAULT bits:'0'B,

      -- Start pointer specification (see 22.3)
      &start-pointer                     REFERENCE OPTIONAL,
      &start-pointer-unit                Unit (ALL EXCEPT repetitions) DEFAULT bit,
      &Start-pointer-encoder-transforms  #TRANSFORM ORDERED OPTIONAL,

      -- Repetition space specification (see 22.7)
      &repetition-space-size             EncodingSpaceSize
                                         DEFAULT self-delimiting-values,
      &repetition-space-unit             Unit
                                         DEFAULT bit,
      &repetition-space-determination    RepetitionSpaceDetermination
                                         DEFAULT 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 '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 OPTIONAL,
      &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
      &handle-value                      HandleValue DEFAULT tag:any,

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

} WITH SYNTAX {
      [IF &size-range-condition] [ELSE]
      [REPLACE
            [STRUCTURE]
            [COMPONENT]
            [ALL COMPONENTS]
            WITH &Replacement-structure
            [ENCODED BY &replacement-structure-encoding-object
                  [INSERT AT HEAD &#Head-end-structure]]]
      [ALIGNED TO
            [NEXT]
            [ANY]
```

```
                &encoding-space-pre-alignment-unit
                [PADDING &encoding-space-pre-padding
                [PATTERN &encoding-space-pre-pattern]]
          [START-POINTER   &start-pointer
               [MULTIPLE OF     &start-pointer-unit]
               [ENCODER-TRANSFORMS    &Start-pointer-encoder-transforms]]
          REPETITION-SPACE
               [SIZE &repetition-space-size
                    [MULTIPLE OF &repetition-space-unit]]
               [DETERMINED BY &repetition-space-determination
                    [HANDLE &handle-id]]
               [USING &main-reference
                    [ENCODER-TRANSFORMS &Encoder-transforms]
                    [DECODER-TRANSFORMS &Decoder-transforms]]
               [PATTERN &termination-pattern]
          [ALIGNMENT  &repetition-alignment]
          [VALUE-PADDING
               [JUSTIFIED &value-justification]
               [PRE-PADDING &value-pre-padding
                    [PATTERN &value-pre-pattern]]
               [POST-PADDING &value-post-padding
                    [PATTERN &value-post-pattern]]
               [UNUSED BITS
                    [DETERMINED BY &unused-bits-determination]
                    [USING &unused-bits-reference
                         [ENCODER-TRANSFORMS &Unused-bits-encoder-transforms]
                         [DECODER-TRANSFORMS &Unused-bits-decoder-transforms]]]]
          [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
               [AS &handle-value]]
          [BIT-REVERSAL &bit-reversal]

     }
```

23.13.2 Purpose and restrictions

23.13.2.1 This syntax is used to define the encoding of a class in the repetition category subject to satisfaction of a condition based on the bounds of the repetition (use of "IF"). It also allows the specification that there is no condition. The use of "ELSE", or omission of both "IF" and "ELSE" specifies that there is no condition.

23.13.2.2 At most one of "IF" and "ELSE" shall be present.

23.13.2.3 If "REPLACE STRUCTURE" is set, then no other encoding property groups shall be set.

23.13.2.4 If "EXHIBITS HANDLE" is set, this asserts that all encodings of this class exhibit the specified identification handle (see also 22.9.2.4).

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

23.13.2.6 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.

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

23.13.3 Encoder actions

23.13.3.1 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 23.13.3.4).

f) Value padding and justification.

g) Identification handle.

h) Bit reversal.

23.13.3.2 If "ALIGNMENT" is set to "aligned", then the settings of pre-alignment and padding shall be used to pre-align each encoding of the component.

NOTE – This is performed before any pre-alignment specified by the component.

23.13.3.3 The complete encodings of the components (with any pre-alignment however specified) shall be concatenated to form the bits for the value of the repetition.

23.13.3.4 If the "REPETITION-SPACE SIZE" is "variable-with-determinant" or "encoder-option-with-determinant", then the size shall be the smallest multiple of "MULTIPLE OF" units ("s", say) that will contain the value of the repetition (but see 23.13.3.5).

23.13.3.5 An encoder (as an encoder's option) may increase "s" (as determined in 23.13.3.4) in "MULTIPLE OF" units (subject to any restrictions that the range of values of any "field-to-be-set" or "field-to-be-used" imposes) if "ENCODING-SPACE SIZE" is set to "encoder-option-with-determinant".

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

23.13.4 Decoder actions

23.13.4.1 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 23.13.4.2).

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

23.14 Defining encoding objects for classes in the tag category

23.14.1 The defined syntax

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

```
#TAG ::= ENCODING-CLASS {

    -- Structure-only replacement specification (see 22.1)
    &#Replacement-structure                                  OPTIONAL,
    &replacement-structure-encoding-object  &#Replacement-structure   OPTIONAL,

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding        Padding DEFAULT zero,
    &encoding-space-pre-pattern        Non-Null-Pattern (ALL EXCEPT different:any)
                                       DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                     REFERENCE OPTIONAL,
    &start-pointer-unit                Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms  #TRANSFORM ORDERED OPTIONAL,

    -- Encoding space specification (see 22.4)
    &encoding-space-size               EncodingSpaceSize
                                       DEFAULT self-delimiting-values,
    &encoding-space-unit               Unit (ALL EXCEPT repetitions)
                                       DEFAULT bit,
    &encoding-space-determination      EncodingSpaceDetermination
                                       DEFAULT 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 OPTIONAL,
        &Handle-positions                 INTEGER (0..MAX) OPTIONAL,
        &handle-value                     HandleValue 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]]
        [BIT-REVERSAL &bit-reversal]

    }
```

23.14.2 Purpose and restrictions

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

23.14.2.2 If "REPLACE STRUCTURE" is set, then no other specifications shall be set.

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

23.14.3 Encoder actions

23.14.3.1 For any encoding 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 23.14.3.3).

f) Value padding and justification.

g) Identification handle.

h) Bit reversal.

23.14.3.2 The encoder shall determine the minimum number of bits "n" needed to encode the tag number as the smallest value of "n" such that 2^n-1 is greater than or equal to the tag number. If "n" is zero, it shall be increased to 1.

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

23.14.3.4 An encoder shall detect an ECN specification error if a tag number is to be encoded into a number of bits which is insufficient, as specified above.

23.14.3.5 If "ENCODING-SPACE SIZE" is a positive integer, then its size in bits is calculated as "SIZE" multiplied by "MULTIPLE OF" units. If "VALUE-PADDING" is not set, then this shall be the number of bits "n" that the tag number shall encode into and there are no unused bits. If "VALUE-PADDING" is set, then the number of bits that the tag number shall encode into is reduced by the integer value "m" specified for "JUSTIFIED", and there will be "m" unused bits.

23.14.3.6 If "ENCODING-SPACE SIZE" is "variable-with-determinant" or "encoder-option-with-determinant", then the encoder shall determine the minimum number of "MULTIPLE OF" units that has sufficient bits to encode the tag number ("s", say), and shall proceed (as specified above) as if "SIZE" were a positive integer set to that value (but see 23.14.3.7).

23.14.3.7 An encoder (as an encoder's option) may increase "s" (as determined in 23.14.3.6) in "MULTIPLE OF" units (subject to any restrictions that the range of values of any "field-to-be-set" or "field-to-be-used" imposes) if "ENCODING-SPACE SIZE" is set to "encoder-option-with-determinant".

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) Encoding space.

d) Bit reversal.

e) Value padding and justification.

f) Value decoding.

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

23.15 Defining encoding objects for classes in the other categories

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

objectidentifier
opentype
real

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}
                       } 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 Sub-clauses 24.2.4.1 to 24.2.4.3 specify rules for using transforms in succession, and for the source and target classes of a list of transforms.

24.2.4.1 When encoding objects of the class #TRANSFORM are specified in an ordered list, the source of a following #TRANSFORM encoding object shall be the result of the preceding #TRANSFORM encoding object.

24.2.4.2 For the first and last of an ordered list of transforms used in the definition of encoding objects in clauses 22 and 23, text in those clauses specifies the source for the first transform and the required result for the last transform.

24.2.4.3 For the first and last of an ordered list of transforms used in the specification of value mapping by transforms in 19.4, text in that sub-clause specifies a source class and a target class, both of which will be of the bitstring, boolean, characterstring, integer or octetstring category (see 19.4.2). The required source for the first transform and the required result of the last transform (for each of these categories) are specified in 24.2.7.

24.2.5 Text in this clause specifies the source of a transform and the result of a transform as 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 proceseed 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 sub-clauses 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}
                        } OPTIONAL
```

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

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

24.3.3 Both the source and result of this transform are integer 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.

24.3.4 An int-to-int transform is defined by giving a value to "INT-TO-INT", permitting any given encoding object to specify precisely one arithmetic operation. General arithmetic can, however, be defined by the use of an ordered list of transforms (this is permitted wherever transforms involving integers are allowed).

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

24.3.6 The value "divide:n" is defined to produce an integer result which is the integer value that is closest to the mathematical result, but is no further from zero than that result. In programming terms, "divide:n" truncates towards zero, so a value of -1 with "divide:2" will give zero.

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

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

24.3.9 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.

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*

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 21.14)
```

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-chars transform shall be:

```
[INT-TO-BITS
     [AS &int-to-bits-encoded-as]
     [SIZE &int-to-bits-size]
     [MULTIPLE OF &int-to-bits-unit]]
```

24.8.3 The 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 21.14)
```

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 ITU-T Rec. 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 ITU-T Rec. X.690 | ISO/IEC 8825-1, **8.3.2** and **8.3.3**. The value of "AS" (or its default value if not set) determines the encoding to be assumed.

24.9.6 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 21.14)
```

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 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.

24.10.6 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 bistring. If the source is a single character composite, the result is a bitstring composite.

24.10.7 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.

24.10.8 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.

24.10.9 "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.

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

24.10.11 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 24.10.11.1 to 24.10.11.5.

24.10.11.1 "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".)

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

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

24.10.11.4 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.

24.10.11.5 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 3.2.41). A composite result is never self-delimiting.

24.10.12 If "AS" is "iso10646", the transform is specified in 24.10.12.1 to 24.10.12.5.

24.10.12.1 The character is first converted to an integer with the numerical value specified in ISO/IEC 10646-1.

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

24.10.12.2 If the character is from a character string that has an associated effective permitted alphabet constraint (see 24.10.8), then the integer has effective size constraints just sufficient to contain the numerical values of all characters in the effective permitted alphabet.

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

24.10.12.4 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.)

24.10.12.5 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.

24.10.13 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 24.10.13.1 to 24.10.13.4.

24.10.13.1 All characters in the effective permitted alphabet are placed in canonical order using their ISO/IEC 10646-1 value, lowest value first. The first in the list is then assigned the integer value zero, the next one, and so on.

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

24.10.13.3 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.

24.10.13.4 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 10646-1 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 3.2.41) 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 octetets-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 sise 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 other)
                                    DEFAULT bits:'0'B,
```

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

      -- Added bits action
      &added-bits                      ENUMERATED
                                       {hard-error, signal-application,
                                       silently-ignore, next-value}
                                       DEFAULT hard-error

} WITH SYNTAX {

      [ALIGNMENT &alignment-point]
      [PADDING
            [MULTIPLE OF &post-padding-unit]
            [POST-PADDING &post-padding
                  [PATTERN &post-padding-pattern]]]
      [BIT-REVERSAL &bit-reversal]
      [ADDED BITS DECODING         &added-bits]

}
```

25.1.2 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 ITU-T Rec. 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 ITU-T Rec. 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 **12.1**, as well as sub-clauses **12.12**, **12.15**, and **12.19** of ITU-T Rec. X.680 | ISO/IEC 8824-1 are modified as follows:*

12.1 **AssignedIdentifier ::= DefinitiveIdentifier**

Symbol ::=
Reference|
BuiltinEncodingClassReference|
ParameterizedReference

Reference ::=
encodingclassreference|
ExternalEncodingClassReference|
encodingobjectreference|
encodingobjectsetreference

NOTE 1 – The production "AssignedIdentifier" is changed because "valuereference"s can neither be defined nor imported into ELM or EDM modules.

NOTE 2 – "BuiltinEncodingClassReference" can only be used as a "Symbol" in an imports clause. The use of production "ExternalEncodingClassReference" in "Reference" is explained in 14.11.

12.12 When the "SymbolsExported" alternative of "Exports" is selected, then each "Symbol" in "SymbolsExported" shall satisfy one and only one of the following conditions:

a) it is defined in the module from which it is being exported; or

b) it appears exactly once in the "SymbolsImported" alternative of "Imports" in the module from which it is being exported.

12.15 When the "SymbolsImported" alternative of "Imports" is selected:

a) Each "Symbol" in "SymbolsFromModule" shall either

1) be defined in the body of the module denoted by the "GlobalModuleReference" in "SymbolsFromModule", or

2) be present precisely once in the imports clause of the module denoted by the "GlobalModuleReference" in "SymbolsFromModule".

NOTE – This does not prohibit the same symbol name defined in two different modules from being imported into another module. However, if the same "Symbol" name appears more than once in the imports clause of module "A", that "Symbol" name cannot be exported from "A" for import to another module "B".

b) All the "SymbolsFromModule" in the "SymbolsFromModuleList" shall include occurrences of "GlobalModuleReference" such that:

i) the "modulereference" in them are all different from each other (whether they are ASN.1, or EDM modules) and from the "modulereference" associated with the referencing module; and

ii) the "AssignedIdentifier", when non-empty, denotes object identifier values which are all different from each other and from the object identifier value (if any) associated with the referencing module.

A.2 Addition of "REFERENCE"

NOTE – This modification is introduced for the sole purpose of clause 23.

*The production "Type" in ITU-T Rec. X.680 | ISO/IEC 8824-1, **16.1**, is modified as follows:*

Type ::=
BuiltinType |
ReferencedType |
ConstrainedType|

REFERENCE

A.3 Notation for character string values

*The production "CharsDefn" of ITU-T Rec. X.680 | ISO/IEC 8824-1, **37.8**, 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 ITU-T Rec. X.680 | ISO/IEC 8824-1, **38.1**).

Annex B

Addendum to ITU-T Rec. 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 ITU-T Rec. X.681 | ISO/IEC 8824-2 are referenced in this Recommendation | International Standard.

B.1 Definitions

*The following definitions are added to ITU-T Rec. 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 ITU-T Rec. X.681 | ISO/IEC 8824-2, 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 ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.2.

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 ITU-T Rec. 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 number sign ("#") which itself is immediately followed by a sequence of characters as specified for an "encodingclassreference" in 8.3.

B.3 Addition of "ENCODING-CLASS"

NOTE – This modification is introduced for the sole purpose of clause 23.

*Replace the reserved word "CLASS" with "ENCODING-CLASS" in ITU-T Rec. X.681 | ISO/IEC 8824-2, **9.3**.*

B.4 FieldSpec additions

NOTE – This modification is introduced for the sole purpose of clause 23.

*ITU-T Rec. X.681 | ISO/IEC 8824-2, **9.4**, is modified as follows:*

FieldSpec ::=

FixedTypeValueFieldSpec|
FixedTypeValueSetFieldSpec|
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 ITU-T Rec. X.680 | ISO/IEC 8824-1, **25.3**), all of whose values shall be of "DefinedType".

B.6 Fixed-class encoding object field spec

NOTE – This modification is introduced for the sole purpose of clause 23.

A "FixedClassEncodingObjectFieldSpec" specifies that the field is a fixed-class encoding object field (see B.1 of 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

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.

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

*ITU-T Rec. X.681 | ISO/IEC 8824-2, **9.13**, is modified as follows*:

9.13 The construct "PrimitiveFieldName" is used to identify a field relative to the encoding class containing its specification:

PrimitiveFieldName ::=
valuefieldreference|
valuesetfieldreference|
orderedvaluelistfieldreference

B.14 Additional reserved words

*ITU-T Rec. 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	**MINUS-INFINITY**	**PLUS-INFINITY**
BER	**NON-ECN-BEGIN**	**TRUE**
CER	**NULL**	**UNION**
DER	**OUTER**	**USE**
ENCODE	**PER-BASIC-ALIGNED**	**USE-SET**
ENCODE-DECODE	**PER-BASIC-UNALIGNED**	
END	**PER-CANONICAL-UNALIGNED**	
FALSE	**PER-CANONICAL-UNALIGNED**	

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 ITU-T Rec. X.681 | ISO/IEC 8824-2, **10.12**.d, is removed.*

NOTE – This affects the defined syntax for defining encoding objects of some classes (see clauses 23 and 24). It means, for example, that, for a defined syntax such as:

[BOOL-TO-INT [AS &bool-to-int]]

the user is allowed to write:

BOOL-TO-INT

when defining an encoding object of this class. In such a case, the "DEFAULT" value associated with the parameter "&bool-to-int" (i.e., "false-zero") is used in the definition of the transform "BOOL-TO-INT".

B.16 Additions to "Setting"

*ITU-T Rec. X.681 | ISO/IEC 8824-2, **11.7**, is modified as follows:*

11.6 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 ITU-T Rec. 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 ITU-T Rec. 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 ITU-T Rec. X.681 | ISO/IEC 8824-2, **14.6**.

Annex C

Addendum to ITU-T Rec. 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 ITU-T Rec. X.683 | ISO/IEC 8824-4 are referenced in this Recommendation | International Standard.

C.1 Parameterized assignments

Clauses 8.1 and 8.3 of ITU-T Rec. X.683 | ISO/IEC 8824-4 are modified as follows:

8.1 There are parameterized assignment statements corresponding to each of the assignment statements specified in this Recommendation | International Standard. The "ParameterizedAssignment" construct is:

ParameterizedAssignment ::=
ParameterizedEncodingObjectAssignment|
ParameterizedEncodingClassAssignment|
ParameterizedEncodingObjectSetAssignment

8.3 **ParameterList ::= "{<" Parameter "," + ">}"**

Governor ::=
EncodingClassFieldType|
REFERENCE|
DefinedOrBuiltinEncodingClass|
#ENCODINGS

A "DummyReference" in "Parameter" may stand for:

a) an encoding class, in which case there shall be no "ParamGovernor";

b) 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");

c) an "identifier", in which case the "ParamGovernor" shall be present as a "Governor" that is "REFERENCE";

d) 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");

e) an encoding object set, in which case the "ParamGovernor" shall be present as a "Governor" that is "#ENCODINGS".

NOTE – "DummyGovernor"s are not allowed in ECN.

C.2 Parameterized encoding assignments

*The following productions are added to ITU-T Rec. X.683 | ISO/IEC 8824-4, **8.2**:*

ParameterizedEncodingClassAssignment ::=
encodingclassreference
ParameterList
"::="
EncodingClass

ParameterizedEncodingObjectAssignment ::=
encodingobjectreference
ParameterList
DefinedOrBuiltinEncodingClass
"::="
EncodingObject

ParameterizedEncodingObjectSetAssignment ::=
encodingobjectsetreference
ParameterList
#ENCODINGS
"::="
EncodingObjectSet

*ITU-T Rec. X.683 | ISO/IEC 8824-4, **8.4**, is modified as follows:*

8.4 The scope of a "DummyReference" appearing in a "ParameterList" is the "ParameterList" itself, together with that part of the "ParameterizedAssignment" which follows the "::=". In case of a "ParameterizedEncodingObjectAssignment", the scope extends to the "DefinedOrBuiltinEncodingClass" which precedes the "::=". The "DummyReference" hides any other "Reference" with the same name in that scope.

NOTE – The special case for "ParameterizedEncodingObjectAssignment" is intended to be used in common with renames clauses (see D.3.3.3). It allows to write an assignment such as the following in which the dummy parameter "#Any-Class" of the encoding object "new-component-encoding" is used as an actual parameter for the encoding class "#New-component":

new-component-encoding {< #Any-class >} #New-component {< #Any-class >} ::=
{ -- encoding object definition -- }

C.3 Referencing parameterized definitions

*The production "ParameterizedReference" of ITU-T Rec. X.683 | ISO/IEC 8824-4, **9.1**, is modified as follows:*

ParameterizedReference ::=
Reference|
Reference "{<" ">}"

*The following productions are added to ITU-T Rec. X.683 | ISO/IEC 8824-4, **9.2**:*

ParameterizedEncodingObject ::=
SimpleDefinedEncodingObject
ActualParameterList

SimpleDefinedEncodingObject ::=
ExternalEncodingObjectReference |
encodingobjectreference

ParameterizedEncodingObjectSet ::=
SimpleDefinedEncodingObjectSet
ActualParameterList

SimpleDefinedEncodingObjectSet ::=
ExternalEncodingObjectSetReference |
encodingobjectsetreference

ParameterizedEncodingClass ::=
SimpleDefinedEncodingClass
ActualParameterList

SimpleDefinedEncodingClass ::=
ExternalEncodingClassReference |
encodingclassreference

C.4 Actual parameter list

*ITU-T Rec. X.683 | ISO/IEC 8824-4, **9.5**, is modified as follows:*

9.5 The "ActualParameterList" is:

ActualParameterList ::=
"{<" ActualParameter "," + ">}"

ActualParameter ::=
Value |
ValueSet |
OrderedValueList |
DefinedOrBuiltinEncodingClass|
EncodingObject|
EncodingObjectSet|
OrderedEncodingObjectList|
identifier|
STRUCTURE|
OUTER

If the corresponding dummy parameter is:

a) a value, the "Value" alternative;

b) a value set, the "ValueSet" alternative;

c) a fixed-type ordered value list, the "OrderedValueList" alternative;

d) an encoding class, the "DefinedOrBuiltinEncodingClass" alternative;

e) an encoding object, the "EncodingObject" alternative;

f) an encoding object set, the "EncodingObjectSet" alternative;

g) an ordered encoding object list, the "OrderedEncodingObjectList" alternative;

h) a reference, the "identifier", "STRUCTURE" or "OUTER" alternative;

shall be selected. "STRUCTURE" shall only by selected 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.2.15).
- A legacy protocol example which shows three ways of handling the problem of a traditional "more-bit" approach to sequence-of termination (D.3.8).
- 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 24.3.6 -- }}
        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, 23.12.1 and 23.13.1) 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 23.13.1) 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 {
        aBOOLEAN,
        bINTEGER OPTIONAL,
        cBinaryFile
                -- "BinaryFile" is defined in D.1.8.1 --}
```

D.1.11.3 The ECN assignments (see 17.5 and 23.10.1) 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 23.14.1) are:

```
choiceEncoding #Choice ::= {
        ENCODE STRUCTURE {
        boolean [tagEncoding] USE-SET,
        integer [tagEncoding] USE-SET,
        string [tagEncoding] USE-SET
        STRUCTURED WITH {
        ALTERNATIVE
        DETERMINED BY handle
        HANDLE "Tag"}}
WITH PER-BASIC-UNALIGNED}

tagEncoding #TAG ::= {
        ENCODING-SPACE
        SIZE 3
        MULTIPLE OF bit
        EXHIBITS HANDLE "Tag" AT {0 | 1 | 2}}
```

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

```
MyEncodings #ENCODINGS ::= { tagEncoding } COMPLETED BY PER-BASIC-UNALIGNED

choiceEncoding #Choice ::= {
        ENCODE STRUCTURE {
        STRUCTURED WITH {
        ALTERNATIVE
        DETERMINED BY handle
        HANDLE "Tag"}}
WITH MyEncodings}
```

D.1.13 Encoding a bitstring containing another encoding

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

D.1.13.2 The ASN.1 assignment are:

```
Sequence2 ::= SEQUENCE {
        aBOOLEAN,
        bBIT STRING (CONTAINING Sequence3) }

Sequence3 ::= SEQUENCE {
        aINTEGER(0..10),
        bBOOLEAN }
```

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 {containerEncoding}
                                        COMPLETED BY PER-BASIC-UNALIGNED}}
                WITH PER-BASIC-UNALIGNED}

        containerEncoding #OUTER ::= {
                PADDING
                        MULTIPLE OF octet}
```

D.1.14 An encoding object set

This encoding object set contains encoding definitions for some types specified in the ASN.1 module of D.1.15.

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

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.1.12.4 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-module(2)}
        DEFINITIONS AUTOMATIC TAGS : :=
        BEGIN

        MyPDU ::= CHOICE {
                marriedMessageMarried,
                altitudeMessageAltitude
                -- 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.1.12.4 together.

```
Example1-EDM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module(3)}
        ENCODING-DEFINITIONS ::=
        BEGIN

        EXPORTS Example1Encodings;
        IMPORTS #Married, #Altitude, #EvenPositiveInteger, #EvenNegativeInteger, #IntegerRightAligned,
                #IntegerWithHole, #PositiveInteger, #NegativeInteger, #PositiveIntegerBCD, #Fax,
                #BinaryFile, #Password, #CharacterStringToBit, #Sequence1, #Choice, #Sequence2
                FROM Example1-ASN1-Module { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(2) };

        Example1Encodings #ENCODINGS ::= {
                marriedEncoding-1|
```

```
                -- etc
        sequence2Encoding}

-- etc

END
```

D.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-module(1)}
        LINK-DEFINITIONS ::=
        BEGIN

        IMPORTS
                Example1Encodings FROM Example-EDM
                        {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module(3)}
                #MyPDU FROM Example1-ASN1-Module
                        {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(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
        MAPPING DISTRIBUTION {
        0..63TO small,
        REMAINDER TO large }
        WITH PER-BASIC-UNALIGNED}
```

D.2.1.4 The encoding structure assignment is:

```
#NormallySmallValuesStruct ::= #CHOICE {
        small#INT (0..63),
        large#INT (64..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 #NormallySmallValuesStruct2
        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:

```
#NormallySmallValuesStruct2 ::= #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 #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 {
        aINTEGER (0..4),
        bINTEGER (1..10),
        cBOOLEAN OPTIONAL
                -- Condition: "c" is present  if "a" is 0, otherwise "c" is absent --,
        dBOOLEAN 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 {
        cUSE-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 {
        aBOOLEAN 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 {
        aUSE-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 {
        listVariableLengthList}

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-lengthlist-lengthEncoding,
        list  {
        ENCODE STRUCTURE {
        STRUCTURED WITH {
                                                REPETITION-ENCODING {
                                                        REPETITION-SPACE
                                                                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 {
        list1List1,
        list2List2}
        (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 {
        list1list1Encoding{< aux-length >},
        list2list2Encoding{< 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 {
        choiceUnevenChoiceProbability }

UnevenChoiceProbability ::= CHOICE {
        frequent1INTEGER (1..2),
        frequent2BOOLEAN,
        common1INTEGER (1..2),
        common2BOOLEAN,
        common3BOOLEAN,
        rare1BOOLEAN,
        rare2INTEGER (1..2),
        rare3INTEGER (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-selectorauxSelectorEncoding,
        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 PER-BASIC-UNALIGNED }
-- Second mapping: Map determinant indexes to bitstrings
```

D.2.10.5 In the above, we quantified "frequent", "common", and "rare" as 60%, 30%, and 10%, respectively, and used the public domain ECN Huffman generator (see E.8) to determine the optimal bit-patterns to be used for each 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-1BOOLEAN,
        ie-2INTEGER (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
        SIZEencoder-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|
            equalLenghListsEncoding|
            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-module(5)}
        DEFINITIONS AUTOMATIC TAGS : :=
        BEGIN

        ExampleMessages ::= CHOICE {
                firstExampleNormallySmallValues,
                secondExampleSparseEvenlyDistributedValueSet
                -- etc.
}

        NormallySmallValues ::= INTEGER (0..1024)

        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-module(6)}
        ENCODING-DEFINITIONS ::=
        BEGIN

        EXPORTS Example2Encodings;

        IMPORTS #NormallySmallValues, #SparseEvenlyDistributedValue,
                #SparseUnevenlyDistributedValueSet, #ConditionalPresenceOnValueSet,
                #ConditionalPresenceOnExternalCondition,
                #EnclosingStructureForList, #EqualLengthLists, #EnclosingStructureForChoice,
                #Version1Message
                FROM Example2-ASN1-Module
                {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(5)};

        Example2Encodings #ENCODINGS ::= {
                normallySmallValuesEncoding   |
                        -- etc
                extensibleMessageEncoding}
        -- etc

        END
```

D.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-module(4)}
        LINK-DEFINITIONS ::=
        BEGIN

        IMPORTS
                Example2Encodings FROM Example2-EDM
                        {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module(6)}
                #ExampleMessages FROM Example2-ASN1-Module
                        {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(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{
component1INTEGER OPTIONAL,
component2INTEGER OPTIONAL,
component3VisibleString }
```

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 >}}
        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{
component1BOOLEAN OPTIONAL,
component2INTEGER,
component3VisibleStringOPTIONAL }
```

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#VisibleStringOPTIONAL-ENCODING #Sequence2-optional}
```

where:

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

Then an encoding object of class "#Sequence2-optional" is defined; that object, using the replacement group, replaces the component encoding definition (see 23.10.3.2) with the class "Optional-with-determinant".

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

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

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

where:

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

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

```
optional-with-determinant-encoding {< #Element, #ENCODINGS: Sequence2-combined-encoding-object-set >}
        #Optional-with-determinant {< #Element >}::= {
ENCODE STRUCTURE {
determinantdeterminant-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
        DETERMINED BY field-to-be-set
        USING length} }
```

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

```
#Sequence3-optional ::= #OPTIONAL

sequence3-optional-encoding  #Sequence3-optional ::= {
PRESENCE
        DETERMINED BY container
                USING OUTER}

#TAG-4-bits ::= #TAG

tag-4-bits-encoding  #TAG-4-bits::= {
ENCODING-SPACE
SIZE 4}
```

D.3.4 Sequence-of type with a count

D.3.4.1 The ASN.1 assignment is:

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

D.3.4.2 The 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
        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 set

The encoding object set contains encoding objects of classes defined in the EDM module.

```
Example3Encodings#ENCODINGS ::= {
sequence1-encoding |
concat|
sequence2-optional-encoding |
octets3-encoding|
sequence3-optional-encoding|
tag-4-bits-encoding |
sequenceOf-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-module(9)}
DEFINITIONS
AUTOMATIC TAGS ::=
BEGIN

Sequence1 ::=SEQUENCE{
component1BOOLEANOPTIONAL,
component2INTEGEROPTIONAL,
component3VisibleString OPTIONAL }

 -- etc

END
```

D.3.7 EDM definitions

```
Example3-EDM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module(10)}
ENCODING-DEFINITIONS ::=
BEGIN

EXPORTS Example3Encodings;
RENAMES
```

```
#OPTIONAL AS #Sequence2-optional
IN #Sequence2
#OCTET-STRING AS #Octets3
IN ALL
#OPTIONAL AS #Sequence3-optional
IN #Sequence3
#TAG AS #TAG-4-bits
IN #Sequence3
FROM Example3-ASN1-Module { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(9)};

Example3Encodings #ENCODINGS::= {
sequence1-optional-encoding |
        -- 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-module(8)}
LINK-DEFINITIONS ::=
BEGIN

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

ENCODE #Sequence1, #Sequence2, #Octet3, #Sequence3, #SequenceOfIntegers
WITH Example3Encodings
COMPLETED BY PER-BASIC-UNALIGNED

END
```

D.4 A more-bit encoding example

D.4.1 Description of the problem

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
ProfileIndication ::= SEQUENCE OF
        SEQUENCE {
                more-bit            BOOLEAN,
                reserved            BIT STRING (SIZE (2)),
                protocol-Profile-ID    INTEGER (0..32) }
```

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..32) } }
```

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..32)
```

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..32) }
```

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..32) } }
```

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
        PATTERN bits:'00'B }
```

NOTE – 23.11.4.2 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-module(11)}

    DEFINITIONS AUTOMATIC TAGS ::=
    BEGIN

    LegacyProtocolMessages ::= SEQUENCE {
            message-idENUMERATED {message1, message2, message3},
            messagesCHOICE {
            message1Message1,
```

```
            message2Message2,
            message3Message3}}
            -- The CHOICE is constrained by the value of message-id.

        Message1 ::= SEQUENCE {
            aA,
            b-flagBOOLEAN,
            c-lenINTEGER (0..max-c-len),
            bBOPTIONAL,-- determined by "b-flag"
            cC,-- determined by "c-len"
            dDOPTIONAL}-- 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 {
            b1ENUMERATED { e0, e1, e2, e3 },
            b2BOOLEAN,
            b3INTEGER (0..3) }

        C ::= SEQUENCE (SIZE (0..max-c-len)) OF C-elem

        C-elem ::= SEQUENCE {
            c1BIT STRING (SIZE (4)),
            c2INTEGER (0..1024) }

        D ::= SEQUENCE (SIZE (0..max-d-len)) OF D-elem

        D-elem ::= SEQUENCE {
            d1BOOLEAN,
            d2ENUMERATED { f0, f1, f2, f3, f4, f5, f6, f7 },
            d3INTEGER (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 {
        bb-encoding
        OPTIONAL-ENCODING {
        PRESENCE
        DETERMINED BY field-to-be-used
        USING b-flag},
        coctet-aligned-seq-of-with-ext-determinant{< c-len >},
        doctet-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 {
                        ALIGNED TO NEXT nibble
                        ENCODING {
                                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
        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) module(13)}
        ENCODING-DEFINITIONS ::=
        BEGIN

        EXPORTS LegacyProtocolEncodings;

        IMPORTS #LegacyProtocolMessages
        FROMLegacyProtocol-ASN1-Module
        { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) module(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-module(12) }
        LINK-DEFINITIONS ::=
        BEGIN

        IMPORTS
                LegacyProtocolEncodings FROM LegacyProtocol-EDM-Module
                        { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module(13) }
                #LegacyProtocolMessages FROM LegacyProtocol-ASN1-Module
                        { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(11) };

        ENCODE #LegacyProtocolMessages WITH LegacyProtocolEncodings
                COMPLETED BY PER-BASIC-UNALIGNED

        END
```

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 can be obtained from the Web site cited in Annex F).

E.7 The following text contains three examples of ECN Huffman specification.

E.8 In the version with the macro, double clicking the button below:

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 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 .. '0000000101010001'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://asn1.elibel.tm.fr/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:

anystringexceptuserfunctionend
encodingobjectreference
encodingobjectsetreference
encodingclassreference
"::="
".."
"{"
"}"
"("
")"
","
"."
"|"
ALL
AS
BEGIN
BER
BITS
BY
CER
COMPLETED
DECODE
DER
DISTRIBUTION
ENCODE
ENCODE-DECODE
ENCODING-CLASS
ENCODING-DEFINITIONS
END
EXCEPT
EXPORTS
FALSE
FIELDS
FROM
GENERATES
IF
IMPORTS
IN
LINK-DEFINITIONS
MAPPING
MAX
MIN
MINUS-INFINITY
NON-ECN-BEGIN
NON-ECN-END
NULL
OPTIONAL-ENCODING
ORDERED
OUTER
PER-BASIC-ALIGNED
PER-BASIC-UNALIGNED
PER-CANONICAL-ALIGNED
PER-CANONICAL-UNALIGNED
PLUS-INFINITY
REFERENCE
REMAINDER
RENAMES
SIZE
STRUCTURE
STRUCTURED
TO
TRANSFORMS
TRUE
UNION
USE
USE-SET
VALUES
WITH

G.1.2 The following item is defined in Annex A:

REFERENCE

G.1.3 The following items are defined in ITU-T Rec. X.680 | ISO/IEC 8824-1:

bstring
cstring
hstring
identifier
modulereference
number
realnumber
typereference
"-"
";"
":"
ALL
EXCEPT
EXPORTS
FALSE
FROM
IMPORTS
MINUS-INFINITY

NULL
PLUS-INFINITY
TRUE

G.1.4 The following items are defined in ITU-T Rec. X.681 | ISO/IEC 8824-2 :

word
valuefieldreference
valuesetfieldreference

G.1.5 The following items are defined in ITU-T Rec. X.683 | ISO/IEC 8824-4:

"{<"
">}"

G.2 Productions

G.2.1 The following productions are used in 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 ?
        "]"
        EncodingStructure

UntaggedStructure ::=
        DefinedEncodingClass|
        EncodingStructureField|
        EncodingStructureDefn

TagClass ::=
        DefinedEncodingClass|
        TagClassReference

TagValue ::=
        "(" number ")"

EncodingStructureDefn ::=
        AlternativesStructure   |
        RepetitionStructure     |
        ConcatenationStructure

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

AlternativesClass ::=
        DefinedEncodingClass|
        AlternativesClassReference

NamedFields ::= NamedField "," +

NamedField ::=
        identifier
        EncodingStructure

RepetitionStructure ::=
        RepetitionClass
        "{"
        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|
        #GeneralizedTime|
        #UTCTime|
        #BMPString|
        #GeneralString|
        #GraphicString|
        #IA5String|
        #NumericString|
        #PrintableString|
        #TeletexString|
        #UniversalString|
        #UTF8String|
        #VideotexString|
        #VisibleString

AlternativesClassReference ::=
        #ALTERNATIVES|
        #CHOICE

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

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

OptionalityClassReference ::=
        #OPTIONAL

TagClassReference ::=
        #TAG

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

EncodingStructureField ::=
        #NUL              |
        #BOOL       |
        #INT   Bounds?        |
        #BITS    Size?     |
        #OCTETS   Size?|
        #CHARS      Size?|
        #PAD|
        #BIT-STRING  Size?|
        #BOOLEAN|
        #CHARACTER-STRING|
        #EMBEDDED-PDV|
        #ENUMERATEDBounds?|
        #EXTERNAL|
        #INTEGERBounds?|
```

```
            #NULL|
            #OBJECT-IDENTIFIER|
            #OCTET-STRINGSize?|
            #OPEN-TYPE|
            #REAL|
            #RELATIVE-OID|
            #GeneralizedTime|
            #UTCTime|
            #BMPStringSize?|
            #GeneralStringSize?|
            #GraphicStringSize?|
            #IA5StringSize?|
            #NumericStringSize?|
            #PrintableStringSize?|
            #TeletexStringSize?|
            #UniversalStringSize?|
            #UTF8StringSize?|
            #VideotexStringSize?|
            #VisibleStringSize?

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 ITU-T Rec. X.680 | ISO/IEC 8824-1, as modified by Annex A, with the items defined in G.1 as terminal symbols:

NOTE – Struck productions are not allowed in ECN.

```
ModuleIdentifier ::=
modulereference
DefinitiveIdentifier ?

DefinitiveIdentifier ::=
        "{" DefinitiveObjIdComponentList "}"

DefinitiveObjIdComponentList ::=
DefinitiveObjIdComponent|
DefinitiveObjIdComponent DefinitiveObjIdComponentList

DefinitiveObjIdComponent ::=
NameForm|
DefinitiveNumberForm|
DefinitiveNameAndNumberForm

NameForm ::= identifier

DefinitiveNumberForm ::= number

DefinitiveNameAndNumberForm ::= identifier "(" DefinitiveNumberForm ")"


Exports ::=
        EXPORTS SymbolsExported? ";" |
        EXPORTS ALL ";"

SymbolsExported ::= SymbolList

Imports ::= IMPORTS SymbolsImported? ";"

SymbolsImported ::= SymbolsFromModuleList

SymbolsFromModuleList ::=
SymbolsFromModule|
SymbolsFromModuleList  SymbolsFromModule

SymbolsFromModule ::=
SymbolList
        FROM
        GlobalModuleReference
```

GlobalModuleReference ::=
modulereference AssignedIdentifier

AssignedIdentifier ::= DefinitiveIdentifier

SymbolList ::=
Symbol|
SymbolList "," Symbol

Symbol ::=
Reference|
ParameterizedReference|
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 ITU-T Rec. X.681 | ISO/IEC 8824-2, as modified by Annex B, with the items defined in G.1 as terminal symbols:

```
DefinedSyntax ::= "{" DefinedSyntaxList ? "}"

DefinedSyntaxList ::= DefinedSyntaxToken DefinedSyntaxList ?

DefinedSyntaxToken ::=
Literal|
Setting
Literal ::=
word|
","
```

Setting ::=
Value|
ValueSet|
OrderedValueList|
EncodingObject|
EncodingObjectSet|
OrderedEncodingObjectList|
DefinedOrBuiltinEncodingClass|
OUTER

OrderedValueList ::= "{" Value "," + "}"

OrderedEncodingObjectList ::= "{" EncodingObject "," + "}"

InstanceOfValue ::= Value

EncodingClassFieldType ::=
DefinedEncodingClass
"."
FieldName

FieldName ::= PrimitiveFieldName "." +

PrimitiveFieldName ::=
valuefieldreference|
valuesetfieldreference

G.2.4 The following productions are defined ITU-T Rec. X.683 | ISO/IEC 8824-4 as modified by Annex C, with the items defined in G.1 as terminal symbols:

ParameterizedAssignment ::=
ParameterizedEncodingObjectAssignment|
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

DummyGovernor ::= DummyReference

DummyReference ::= Reference


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
```

ITU-T Recommendation X.693
International Standard 8825-4

Information Technology –
ASN.1 encoding rules:
XML Encoding Rules (XER)

INTERNATIONAL STANDARD 8825-4

ITU-T RECOMMENDATION X.693

Information Technology – ASN.1 Encoding Rules – XML Encoding Rules (XER)

Summary

This Recommendation | International Standard specifies rules for encoding values of ASN.1 types using the Extensible Markup Language (XML).

Source

ITU-T Recommendation X.693 was prepared by ITU-T Study Group 17 (2001-2004) and approved on 22 December 2001. An identical text is also published as ISO/IEC 8825-4.

CONTENTS

1 Scope 1
2 Normative references 1
2.1 Identical Recommendations | International Standards 1
2.2 Additional references 1
3 Definitions 2
3.1 Basic Encoding Rules 2
3.2 Additional definitions 2
4 Abbreviations 2
5 Notation 3
6 Encodings specified by this Recommendation | International Standard 3
7 Conformance 3
8 Basic XML encoding rules 3
8.1 Production of a complete XER encoding 3
8.2 The XML prolog 4
8.3 The XML document element 4
8.4 Encoding of the EXTERNAL type 4
9 Canonical XML encoding rules 4
9.1 General rules for canonical XML encodings 5
9.2 Real values 5
9.3 Bitstring value 5
9.4 Octetstring value 5
9.5 Sequence value 5
9.6 Set value 5
9.7 Set-of value 6
9.8 Object identifier value 6
9.9 Relative object identifier value 6
9.10 GeneralizedTime 6
9.11 UTCTime 6
10 Object identifier values referencing the encoding rules 7
Annex A Example of encodings 8
A.1 ASN.1 description of the record structure 8
A.2 ASN.1 description of a record value 8
A.3 Basic XML representation of this record value 8
A.4 Canonical XML representation of this record value 9

Introduction

The publications ITU-T Rec. X.680 | ISO/IEC 8824-1, ITU-T Rec. X.681 | ISO/IEC 8824-2, ITU-T Rec. X.682 | ISO/IEC 8824-3, ITU-T Rec. X.683 | ISO/IEC 8824-4 together describe Abstract Syntax Notation One (ASN.1), a notation for the definition of messages to be exchanged between peer applications.

This Recommendation | International Standard defines encoding rules that may be applied to values of ASN.1 types defined using the notation specified in ITU-T Rec. X.680 | ISO/IEC 8824-1 and ITU-T Rec. X.681 | ISO/IEC 8824-2. Application of these encoding rules produces a transfer syntax for such values. It is implicit in the specification of these encoding rules that they are also to be used for decoding.

There is more than one set of encoding rules that can be applied to values of ASN.1 types. This Recommendation | International Standard defines two sets of encoding rules that use the Extensible Markup Language (XML). These are called the XML Encoding Rules (XER) for ASN.1, and both produce an XML document compliant to W3C XML 1.0. The first set is called the Basic XML Encoding Rules. The second set is called the Canonical XML Encoding Rules because there is only one way of encoding an ASN.1 value using these encoding rules. (Canonical encoding rules are generally used for applications using security-related features such as digital signatures.)

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

Information Technology – ASN.1 Encoding Rules – XML Encoding Rules

1 Scope

This Recommendation | International Standard specifies a set of Basic XML Encoding Rules (XER) that may be used to derive a transfer syntax for values of types defined in ITU-T Rec. X.680 | ISO/IEC 8824-1 and ITU-T Rec. X.681 | ISO/IEC 8824-2. This Recommendation | International Standard also specifies a set of Canonical XML Encoding Rules which provide constraints on the Basic XML Encoding Rules and produce a unique encoding for any given ASN.1 value. It is implicit in the specification of these encoding rules that they are also used for decoding.

The encoding rules specified in this Recommendation | International Standard:

- are used at the time of communication;
- are intended for use in circumstances where displaying of values and/or processing them using commonly available XML tools (such as browsers) is the major concern in the choice of encoding rules;
- allow the extension of an abstract syntax by addition of extra values for all forms of extensibility described in ITU-T Rec. X.680 | ISO/IEC 8824-1.

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 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.

2.1 Identical Recommendations | International Standards

- ITU-T Recommendation X.680 (2002) | ISO/IEC 8824-1:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.*
- ITU-T Recommendation X.681 (2002) | ISO/IEC 8824-2:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.*
- ITU-T Recommendation X.682 (2002) | ISO/IEC 8824-3:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*
- ITU-T Recommendation X.683 (2002) | ISO/IEC 8824-4:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications.*
- ITU-T Recommendation X.690 (2002) | ISO/IEC 8825-1:2002, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).*
- ITU-T Recommendation X.691 (2002) | ISO/IEC 8825-2:2002, *Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER).*

2.2 Additional references

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

- ISO/IEC 10646-1:1993/Amd.2:1996, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane – Amendment 2: UCS Transformation Format 8 (UTF-8).*
- W3C XML 1.0:2000, *Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, Copyright © [6 October 2000] World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University), http://www.w3.org/TR/2000/REC-xml-20001006.*

  NOTE – The reference to a document within this Recommendation | International Standard does not give it, as a stand-alone document, the status of a Recommendation or International Standard.

3 Definitions

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

3.1 Basic Encoding Rules

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.690 | ISO/IEC 8825-1:

a) data value;

b) dynamic conformance;

c) encoding (of a data value);

d) receiver.

e) sender;

f) static conformance;

3.2 Additional definitions

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

3.2.1 ASN.1 schema: The definition of the content and structure of data using an ASN.1 type definition.

NOTE – This enables encoding rules to produce binary encodings of the values of an ASN.1 type, or encodings using XML.

3.2.2 canonical encoding: A complete encoding of an abstract value obtained by the application of encoding rules that have no implementation-dependent options. Such rules result in the definition of a 1-1 mapping between unambiguous and unique encodings and values in the abstract syntax.

3.2.3 valid XML document (for an ASN.1 schema): An XML document which is well-formed (see W3C XML 1.0) and whose content conforms to the XER specification for the encoding of the ASN.1 type specified by an ASN.1 schema.

3.2.4 XML document: A sequence of characters which conforms to W3C XML 1.0 definition of document.

4 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
PDU	Protocol Data Unit
UCS	Universal Multiple-Octet Coded Character Set
UTC	Coordinated Universal Time
UTF-8	UCS Transformation Format, 8-bit form
XML	Extensible Markup Language
XER	XML Encoding Rules

5 Notation

This Recommendation | International Standard references the notation defined by ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 5.

Encodings specified by this Recommendation | International Standard

6.1 This Recommendation | International Standard specifies two sets of encoding rules:

- Basic XML Encoding Rules (BASIC-XER)
- Canonical XML Encoding Rules (CANONICAL-XER)

NOTE – Where this Recommendation | International Standard uses "XER" without qualification, the text applies to both BASIC-XER and CANONICAL-XER.

6.2 The most general set of encoding rules specified in this Recommendation | International Standard is BASIC-XER, which does not in general produce a canonical encoding.

6.3 A second set of encoding rules specified in this Recommendation | International Standard is CANONICAL-XER, which produces encodings that are canonical. This is defined as a restriction of implementation-dependent choices in the BASIC-XER encoding.

NOTE 1 – Any implementation conforming to CANONICAL-XER for encoding is conformant to BASIC-XER for encoding. Any implementation conforming to BASIC-XER for decoding is conformant to CANONICAL-XER for decoding. Thus, encodings made according to CANONICAL-XER are encodings that are permitted by BASIC-XER.

NOTE 2 – CANONICAL-XER produces encodings that have applications when authenticators need to be applied to abstract values.

6.4 If a type encoded with CANONICAL-XER contains "EMBEDDED PDV", "EXTERNAL" or "CHARACTER STRING" types, then the outer encoding ceases to be canonical unless the encoding used for all the "EMBEDDED PDV", "EXTERNAL" and "CHARACTER STRING" types is canonical.

7 Conformance

7.1 Dynamic conformance for the Basic XML Encoding Rules is specified by clause 8, and dynamic conformance for the Canonical XML Encoding Rules is specified by clause 9 inclusive.

7.2 Static conformance is specified by those standards which specify the application of one or more of these encoding rules.

7.3 Alternative encodings are permitted by the Basic XML Encoding Rules as an encoder’s option. Decoders that claim conformance to XER shall support all alternatives.

7.4 No alternative encodings are permitted by the Canonical XML Encoding Rules for the encoding of an ASN.1 value.

8 Basic XML encoding rules

8.1 Production of a complete XER encoding

8.1.1 A conforming XER encoding is a valid XML document which shall consist of:

a) an XML prolog (which may be empty) as specified in 8.2;

b) an XML document element which is the complete encoding of a value of a single ASN.1 type as specified in 8.3.

8.1.2 The specification in 8.2 to 8.4 completely defines the XER encoding.

NOTE – Other constructs of W3C XML 1.0, such as processing instructions and comments are not allowed by those sub-clauses, and can never appear in an XER encoding.

8.1.3 The XML document shall be encoded using UTF-8 to produce a string of octets which forms the encoding specified in this Recommendation | International Standard. The ASN.1 object identifier for these encoding rules is specified in clause 10.

8.1.4 Where this Recommendation | International Standard uses the term "white-space", this means one or more of the following characters: HORIZONTAL TABULATION (9), LINE FEED (10), CARRIAGE RETURN (13),

SPACE (32). The numbers in parentheses are the decimal value of the ISO/IEC 10646-1 characters. The number and choice of characters that constitutes "white-space" is an encoder's option.

8.2 The XML prolog

8.2.1 The XML prolog shall either:

a) be empty; or

b) shall consist of the following character sequences in order, and as an encoder's option the last character sequence may be followed by "white-space" (see 8.1.4):

```
<?xml
version="1.0"
encoding="UTF-8"?>
```

8.2.2 The character sequences listed in 8.2.1 shall not contain "white-space", but shall be separated by a single SPACE (32) character.

8.3 The XML document element

8.3.1 The XML document element shall be an "XMLTypedValue" as specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 15.2, with the changes and restrictions specified in the following sub-clauses.

8.3.2 The ASN.1 "comment" lexical item (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.6) shall not be present. If an XER encoding contains a pair of adjacent hyphens, or "/*", or "*/", these shall be treated as part of the data, and not as ASN.1 comment delimiters.

8.3.3 Where ITU-T Rec. X.680 | ISO/IEC 8824-1 permits the use of ASN.1 white-space between lexical items, the characters used shall be restricted to the "white-space" specified in 8.1.4.

8.3.4 The "XMLIntegerValue" specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 18.9, shall only be "SignedNumber".

8.3.5 The "XMLBitStringValue" specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 21.9, shall not be "XMLIdentifierList".

8.3.6 The "XMLExternalValue" specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 34, shall be replaced by the "XMLExternalValue" specified in 8.4.

8.4 Encoding of the EXTERNAL type

8.4.1 The "XMLExternalValue" production used for an XER encoding of an "EXTERNAL" type shall be the "XMLValue" for the encoding of the sequence type specified in ITU-T Rec. X.691 | ISO/IEC 8825-2, 26.1, with a value as specified in 26.2 to 26.4 of that Recommendation | International Standard.

NOTE – For historical reasons, the XER encoding of an "EXTERNAL" type is not the same as the XML value notation specified in ITU-T Rec. X.680 | ISO/IEC 8824-1.

8.4.2 ITU-T Rec. X.691 | ISO/IEC 8825-2, 26.5 to 26.8 shall apply, except that the provisions of 26.6 shall be replaced by 8.4.3 of this Recommendation | International Standard.

8.4.3 If the data value is the value of a single ASN.1 type, and if the encoding rules for this data value are those specified in this (XER) Recommendation | International Standard, then the sending implementation shall use the "single-ASN1-type" alternative.

8.4.4 ITU-T Rec. X.691 | ISO/IEC 8825-2, 26.9 to 26.11 shall apply, except that the provisions of 26.9 shall be replaced by 8.4.5 of this Recommendation | International Standard. The note in ITU-T Rec. X.691 | ISO/IEC 8825-2, 26.9 applies.

8.4.5 If the "encoding" choice is "single-ASN1-type", then the ASN.1 type shall be the "XMLTypedValue" of the type encoded in the "EXTERNAL", with a value equal to the data value to be encoded.

9 Canonical XML encoding rules

Where "XMLTypedValue" contains options, this clause specifies precisely one of those options in order to produce a unique encoding. The provisions of this clause determine the canonical XML encoding rules.

9.1 General rules for canonical XML encodings

9.1.1 The XML prolog shall be empty (see 8.2.1).

9.1.2 All lexical items forming the "XMLTypedValue" shall have no "white-space" between them (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.1.4).

NOTE – This ensures that the digital signature of a document can be easily generated without considering any possible insertion of "white-space" between the lexical items of the "XMLTypedValue".

9.1.3 The escape sequences specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.15.8, shall not be used.

9.1.4 If the XML value notation permits the use of an XML empty-element tag (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 15.5 and 16.8), then this empty-element tag shall be used.

9.2 Real values

9.2.1 The real value zero shall be encoded as "0".

9.2.2 For all other values, the following sub-clauses specify restrictions that apply to "realnumber" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.9).

9.2.3 The "realnumber" shall be normalized so that the integer part consists of a single, non-zero digit. The decimal point shall be present and shall be followed by a fractional part containing at least one digit (which may be zero). The fractional part shall not contain any trailing zeros after the first digit.

9.2.4 The fractional part shall be followed by an "E" (not an "e") and by an exponent (which may be zero).

NOTE – Leading zeros in the exponent are already forbidded by ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.9.

9.2.5 No "+" sign shall be present either before the integer part or before the exponent.

9.3 Bitstring value

9.3.1 If the "XMLTypedValue" alternative of "XMLBitStringValue" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 21.9) can be used (as specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 21.10), then it shall be used. Otherwise, the "xmlbstring" alternative shall be used with all white-space removed.

9.3.2 If the bitstring type has a "NamedBitList", there shall be no trailing zero bits (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 21.7).

9.4 Octetstring value

If the "XMLTypedValue" alternative of "XMLOctetStringValue" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 22.3) can be used (as specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 22.4), then it shall be used. Otherwise, the "xmlhstring" alternative shall be used with all white-space removed, and all letters in upper-case.

9.5 Sequence value

All components of a sequence which have default values, and which have an abstract value set to those default values, shall have the encoding of the default value textually present. There shall always be an encoding for those components.

9.6 Set value

9.6.1 The set type shall have the elements in its "RootComponentTypeList" sorted into the canonical order specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 8.6, and additionally for the purposes of determining the order in which components are encoded when one or more component is an untagged choice type, each untagged choice type is ordered as though it has a tag equal to that of the smallest tag in the "RootAlternativeTypeList" of that choice type or any untagged choice types nested within.

9.6.2 The set elements that occur in the "RootComponentTypeList" shall then be encoded in the resulting sorted order. After the elements in the "RootComponentTypeList", if any, have been encoded, the set elements that occur in the "ExtensionAdditionList" shall be encoded in the order in which they are defined. (An example of this ordering of elements is provided in ITU-T Rec. X.691 | ISO/IEC 8825-2, clause 20.)

9.6.3 All components of a set which have default values, and which have an abstract value set to those default values, shall have the encoding of the default value textually present. There shall always be an encoding for those components.

9.7 Set-of value

9.7.1 The order of the elements of an "XMLSetOfValue" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 27.3) shall be determined by sorting the character strings which represent the CANONICAL-XER encoding for each element as specifed in 9.7.2 and 9.7.3.

9.7.2 The sort order for the character strings is determined using the 32-bit value of characters specified in ISO/IEC 10646-1, with lower numbered characters preceding higher numbered characters.

9.7.3 A conceptual "pad" character is used in specifying the sort order. This character precedes all other characters. When determining whether a string "A" sorts before a string "B", the shorter string has conceptual "pad" characters added at its end if necessary. String "A" sorts before string "B" if and only if the character in string "A" precedes the corresponding character in string "B" in the first character position in which they have different characters.

9.8 Object identifier value

The "XMLObjIdComponent" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 31.3) shall be "XMLNumberForm".

9.9 Relative object identifier value

The "XMLRelativeOIDComponent" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 32.3) shall be "XMLNumberForm".

9.10 GeneralizedTime

9.10.1 The encoding of a value of type "GeneralizedTime" shall terminate with the character "Z" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 42.3).

9.10.2 The string representing the seconds shall always be present.

9.10.3 The string representing fractions of seconds, if present, shall omit all trailing zeros. If the fractional part corresponds to 0, this string shall be wholly omitted together with the decimal point.

EXAMPLE

Seconds represented with the string "26.000" shall be encoded as "26". Seconds represented with the string "26.5200" shall be encoded as "26.52".

9.10.4 The decimal point, if present, shall be ".".

9.10.5 Midnight (GMT) shall be encoded as a string of the form:

"YYYYMMDD000000Z"

where "YYYYMMDD" represents the day following the midnight in question.

EXAMPLE

The followings encodings are valid:

"19920521000000Z"

"19920622123421Z"

"19920722132100.3Z"

The following encodings are invalid:

"19920520240000Z" (midnight represented incorrectly)

"19920622123421.0Z" (spurious trailing zeros)

"19920722132100.30Z" (spurious trailing zeros)

9.11 UTCTime

9.11.1 The encoding of a value of type "UTCTime" shall terminate with the character "Z" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 43.3).

9.11.2 The string representing the seconds shall always be present.

9.11.3 Midnight (GMT) shall be encoded as a string of the form:

"YYMMDD000000Z"

where "YYMMDD" represents the day following the midnight in question.

EXAMPLE

The following encodings are valid:

"920521000000Z"

"920622123421Z"

"920722132100Z"

The following encodings are invalid:

"920520240000Z"(midnight represented incorrectly)

"9207221321Z"(seconds of "00" omitted)

10 Object identifier values referencing the encoding rules

10.1 The encoding rules specified in this Recommendation | International Standard can be referenced and applied whenever there is a need to specify an unambiguous character string representation for the values of a single identified ASN.1 type.

10.2 The following object identifier and object descriptor values are assigned to identify the encoding rules specified in this Recommendation | International Standard:

For BASIC-XER:

```
{joint-iso-itu-t asn1 (1) xml-encoding (5) basic (0) }
"Basic XML encoding of a single ASN.1 type"
```

For CANONICAL-XER:

```
{joint-iso-itu-t asn1 (1) xml-encoding (5) canonical (1) }
"Canonical XML encoding of a single ASN.1 type"
```

Annex A

Example of encodings

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

This annex illustrates the use of the XML Encoding Rules specified in this Recommendation | International Standard by showing XML Markup representations of a (hypothetical) personnel record which is defined using ASN.1.

A.1 ASN.1 description of the record structure

The structure of the hypothetical personnel record is formally described below using ASN.1 specified in ITU-T Rec. X.680 | ISO/IEC 8824-1. This is identical to the example defined in Annex A of ITU-T Rec. X.690 | ISO/IEC 8825-1.

```
PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET {
            name              Name,
            title             [0] VisibleString,
            number            EmployeeNumber,
            dateOfHire        [1] Date,
            nameOfSpouse      [2] Name,
            children          [3] IMPLICIT
                  SEQUENCE OF ChildInformation DEFAULT {} }

ChildInformation ::= SET
            { name                Name,
              dateOfBirth         [0] Date}

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
            { givenName           VisibleString,
              initial             VisibleString,
              familyName          VisibleString}

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER

Date ::= [APPLICATION 3] IMPLICIT VisibleString  -- YYYYMMDD
```

NOTE – Tags are used in this example only because it was felt appropriate to use the identical example to that which appeared in the earliest version of ITU-T Rec. X.680 | ISO/IEC 8824-1. They have no effect on the XML encodings.

A.2 ASN.1 description of a record value

The value of John Smith's personnel record is formally described below using the basic ASN.1 value notation:

```
{     name                  {givenName "John", initial "P", familyName "Smith"},
      title                 "Director",
      number                51,
      dateOfHire            "19710917",
      nameOfSpouse          {givenName "Mary", initial "T", familyName "Smith"},
      children
            {{name {givenName "Ralph", initial "T", familyName "Smith"},
                        dateOfBirth "19571111"},
             {name {givenName "Susan", initial "B", familyName "Jones"},
                        dateOfBirth "19590717"}}}
```

A.3 Basic XML representation of this record value

The representation of the record value given above (after applying the Basic XML Encoding Rules defined in this Recommendation | International Standard) is shown below assuming an empty prolog.

The length of this encoding in BASIC-XER is 653 octets ignoring all "white-space". For comparison, the same PersonnelRecord value encoded with the UNALIGNED variant of PER (see ITU-T Rec. X.690 | ISO/IEC 8825-1) is 84 octets, with the ALIGNED variant of PER it is 94 octets, with BER (see ITU-T Rec. X.691 | ISO/IEC 8825-2) using the definite length form it is a mininum of 136 octets, and with BER using the indefinite length form it is a minimum of 161 octets.

```
<PersonnelRecord>
      <name>
            <givenName>John</givenName>
            <initial>P</initial>
            <familyName>Smith</familyName>
      </name>
      <title>Director</title>
      <number>51</number>
      <dateOfHire>19710917</dateOfHire>
      <nameOfSpouse>
            <givenName>Mary</givenName>
            <initial>T</initial>
            <familyName>Smith</familyName>
      </nameOfSpouse>
      <children>
            <ChildInformation>
                  <name>
                        <givenName>Ralph</givenName>
                        <initial>T</initial>
                        <familyName>Smith</familyName>
                  </name>
                  <dateOfBirth>19571111</dateOfBirth>
            </ChildInformation>
            <ChildInformation>
                  <name>
                        <givenName>Susan</givenName>
                        <initial>B</initial>
                        <familyName>Jones</familyName>
                  </name>
                  <dateOfBirth>19590717</dateOfBirth>
            </ChildInformation>
      </children>
</PersonnelRecord>
```

A.4 Canonical XML representation of this record value

The representation of the record value given above (after applying the Canonical XML Encoding Rules defined in this Recommendation | International Standard) is shown below:

```
<PersonnelRecord><name><givenName>John</givenName><initial>P</initial><familyName>Smith</fam
ilyName></name><number>51</number><title>Director</title><dateOfHire>19710917</dateOfHire><n
ameOfSpouse><givenName>Mary</givenName><initial>T</initial><familyName>Smith</familyName></n
ameOfSpouse><children><ChildInformation><name><givenName>Ralph</givenName><initial>T</initia
l><familyName>Smith</familyName></name><dateOfBirth>19571111</dateOfBirth></ChildInformation
><ChildInformation><name><givenName>Susan</givenName><initial>B</initial><familyName>Jones</
familyName></name><dateOfBirth>19590717</dateOfBirth></ChildInformation></children></Personn
elRecord>
```

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
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	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
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
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 and open system communications**
Series Y	Global information infrastructure and Internet protocol aspects
Series Z	Languages and general software aspects for telecommunication systems