INTERNATIONAL STANDARD

ISO/IEC 8825-2

Sixth edition
2021-06

Information technology — ASN.1 encoding rules —

Part 2: Specification of Packed Encoding Rules (PER)

Technologies de l'information — Règles de codage ASN.1 — Partie 2: Spécification des règles de codage compact (PER)

Reference number
ISO/IEC 8825-2:2021(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs)

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 6, *Telecommunications and information exchange between systems,* in collaboration with ITU-T. The identical text is published as ITU-T X.691 (02/2021).

This sixth edition cancels and replaces the fifth edition (ISO/IEC 8825-2:2015), which has been technically revised. It also incorporates ISO/IEC 8825-2:2015/Cor 1:2017.

A list of all parts in the ISO/IEC 8825 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

CONTENTS

Introduction

Specifications Rec. ITU-T X.680 | ISO/IEC 8824-1, Rec. ITU-T X.681 | ISO/IEC 8824-2, Rec. ITU-T X.682 | ISO/IEC 8824-3, Rec. ITU-T X.683 | ISO/IEC 8824-4 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 Rec. ITU-T 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 Rec. ITU-T 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 Rec. ITU-T 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 Rec. ITU-T X.680 | ISO/IEC 8824-1;

– can be modified in accordance with the provisions of Rec. ITU-T X.695 | ISO/IEC 8825-6.

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.

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

2.1 Identical Recommendations | International Standards

– Recommendation ITU-T X.680 (2021) | ISO/IEC 8824-1:2021, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*.

– Recommendation ITU-T X.681 (2021) | ISO/IEC 8824-2:2021, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification*.

– Recommendation ITU-T X.682 (2021) | ISO/IEC 8824-3:2021, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*

– Recommendation ITU-T X.683 (2021) | ISO/IEC 8824-4:2021, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications*.

– Recommendation ITU-T X.690 (2021) | ISO/IEC 8825-1:2021, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).*

– Recommendation ITU-T X.695 (2021) | ISO/IEC 8825-6:2021, *Information technology – ASN.1 encoding rules: Registration and application of PER encoding instructions.*

NOTE – The references above 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 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/IEC 2375:2003, *Information technology – Procedure for registration of escape sequences and coded character sets.*

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

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

3.2 Information Object Specification

For the purposes of this Recommendation | International Standard, all the definitions in Rec. ITU-T X.681 | ISO/IEC 8824-2 apply.

3.3 Constraint Specification

This Recommendation | International Standard makes use of the following terms defined in Rec. ITU-T 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 term defined in Rec. ITU-T 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 Rec. ITU-T 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.

3.6 PER Encoding Instructions

This Recommendation | International Standard makes use of the following term defined in Rec. ITU-T X.695 | ISO/IEC 8825-6:

- identifying keyword.

3.7 Additional definitions

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

3.7.1 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 11.4.

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*.

3.7.2 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**.

3.7.3 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.

3.7.4 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.

3.7.5 composite type: A set, sequence, set-of, sequence-of, choice, embedded-pdv, external or unrestricted character string type.

3.7.6 composite value: The value of a composite type.

3.7.7 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.

3.7.8 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.

3.7.9 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.

3.7.10 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".

3.7.11 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.

3.7.12 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.

3.7.13 indefinite-length: An encoding whose length is greater than 64K-1 or whose maximum length cannot be determined from the ASN.1 notation.

3.7.14 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.

3.7.15 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.

3.7.16 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`**.

3.7.17 length determinant: A count (of bits, octets, characters, or components) determining the length of part or all of a PER encoding.

3.7.18 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.

3.7.19 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.

3.7.20 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 11.3.

NOTE – The value of a non-negative-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 non-negative-binary-number is obtained by summing the numerical values assigned to each bit for those bits which are set to one.

3.7.21 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 11.1.1).

NOTE – PER encodings of an outermost type are always an integral multiple of eight bits.

3.7.22 PER-visible constraint: An instance of use of the ASN.1 constraint notation which affects the PER encoding of a value.

3.7.23 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.

3.7.24 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.

3.7.25 simple type: A type that is not a composite type.

3.7.26 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 10.3.6 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)
```

3.7.27 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 Rec. ITU-T 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 Rec. ITU-T 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 (but see 9.2 and 9.3).

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 Rec. ITU-T 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 PER encoding instructions

9.1 PER encoding instructions can be associated with a type in accordance with the provisions of Rec. ITU-T X.680 | ISO/IEC 8824-1 and Rec. ITU-T X.695 | ISO/IEC 8825-6.

NOTE 1 – The application of some PER encoding instructions can make it impossible to encode all the abstract values of the type. Where this can arise, the specific PER encoding instruction identifies the problem. It is a designers decision, based on the possible need to use multiple encoding rules, whether to add an explicit constraint on the type in order to restrict the range of abstract values to those that can be handled by the encoding using the PER encoding instruction. This can make the specification less readable, but ensures that all encoding rules can encode all allowed abstract values, making relaying possible without errors.

NOTE 2 – Each PER encoding instruction starts with an identifying keyword that unambiguously identifies that encoding instruction.

9.2 If the ALIGNED version of either BASIC-PER or CANONICAL-PER is in use, then all PER encoding instructions shall be silently ignored and have no affect on the encoding.

9.3 If the UNALIGNED version of either BASIC-PER or CANONICAL-PER is in use, then if a type has an associated encoding instruction, the following subclauses shall apply.

9.3.1 If the identifying keyword is not known, then a "not supported" error message shall be issued.

9.3.2 If the identifying keyword is known, the procedures of this Recommendation | International Standard shall be modified by the amendments to those procedures that are specified by the PER encoding instruction (see Rec. ITU-T X.695 | ISO/IEC 8825-6).

NOTE 1 – If multiple PER encoding instructions are associated with a type, then the amendments specified for all of them shall be applied.

NOTE 2 – It is an error in the register of PER encoding instructions if amendments produced by two or more separate encoding instructions conflict and it is not stated that they are mutually exclusive.

10 The approach to encoding used for PER

10.1 Use of the type notation

10.1.1 These encoding rules make specific use of the ASN.1 type notation as specified in Rec. ITU-T 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.

10.1.2 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.

10.2 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 8.6.

10.3 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.

10.3.1 Constraints that are expressed in human-readable text or in ASN.1 comment are not PER-visible.

10.3.2 Variable constraints are not PER-visible (see Rec. ITU-T X.683 | ISO/IEC 8824-4, 10.3 and 10.4).

10.3.3 User-defined constraints (see Rec. ITU-T X.682 | ISO/IEC 8824-3, 9.1) are not PER visible.

10.3.4 Table constraints are not PER-visible (see Rec. ITU-T X.682 | ISO/IEC 8824-3).

10.3.5 Component relation constraints (see Rec. ITU-T X.682 | ISO/IEC 8824-3, 10.7) are not PER-visible.

10.3.6 Constraints whose evaluation is textually dependent on a table constraint or a component relation constraint are not PER-visible (see Rec. ITU-T X.682 | ISO/IEC 8824-3).

10.3.7 Constraints on restricted character string types which are not (see Rec. ITU-T X.680 | ISO/IEC 8824-1, clause 41) known-multiplier character string types are not PER-visible (see 3.7.16).

10.3.8 Pattern constraints are not PER-visible.

10.3.9 Subject to the above, all size constraints are PER-visible.

10.3.10 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.

10.3.11 Permitted-alphabet constraints on known-multiplier character string types which are not extensible after application of Rec. ITU-T X.680 | ISO/IEC 8824-1, 52.3 to 52.5, are PER-visible. Permitted-alphabet constraints which are extensible are not PER-visible.

10.3.12 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.

10.3.13 Property setting constraints on the time type (or on the useful and defined time types) which are not extensible after the application of Rec. ITU-T X.680 | ISO/IEC 8824-1, 52.3 to 52.5, are PER-visible. Property setting constraints which are extensible are not PER-visible.

10.3.14 Constraints applied to real types are not PER-visible.

10.3.15 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 28 and 31).

10.3.16 Constraints on the useful types are not PER-visible.

10.3.17 Single value subtype constraints applied to a character string type are not PER-visible.

10.3.18 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.

10.3.19 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.

10.3.20 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 50.11.

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.

10.3.21 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.

10.3.22 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.

10.4 Type and value model used for encoding

10.4.1 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 10.2. 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.

10.4.2 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.

10.4.3 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.*

10.5 Structure of an encoding

10.5.1 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 11.1).

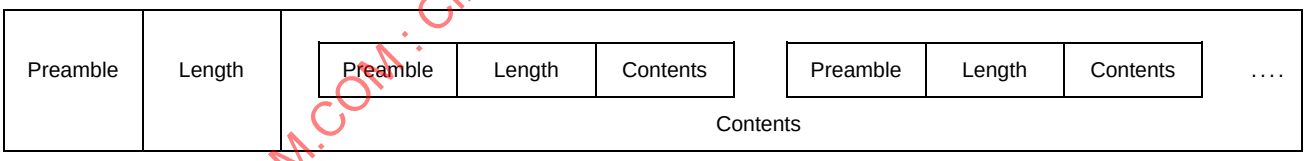
10.5.2 The encoding of a component of a data value either:

a) consists of three parts, as shown in Figure 1, which appear in the following order:

1) a preamble (see clauses 19, 21 and 23);

2) a length determinant (see 11.9);

3) contents; or



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

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 19, 21 and 23) 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 11.9.

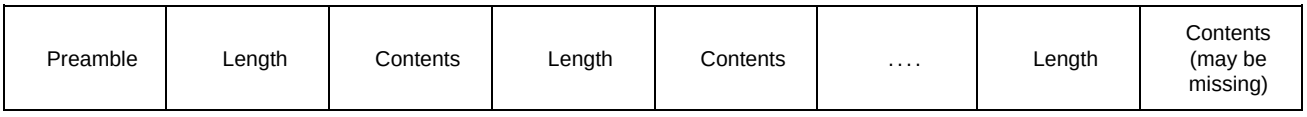


Figure 2 – Encoding of a long data value

10.5.3 Each of the parts mentioned in 10.5.2 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.

10.6 Types to be encoded

10.6.1 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.

10.6.2 The selection type shall be encoded as an encoding of the selected type.

10.6.3 Encoding of tagged types is not included in this Recommendation | International Standard as, except as stated in 10.2, 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.

10.6.4 An encoding prefixed type is encoded according to the type which has been prefixed.

10.6.5 The following "useful types" shall be encoded as if they had been replaced by their definitions given in Rec. ITU-T X.680 | ISO/IEC 8824-1, clause 45:

– 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 Rec. ITU-T X.690 | ISO/IEC 8825-1, 11.7 and 11.8, shall apply here.

10.6.6 A type defined using a value set assignment shall be encoded as if the type had been defined using the production specified in Rec. ITU-T X.680 | ISO/IEC 8824-1, 16.8.

11 Encoding procedures

11.1 Production of the complete encoding

11.1.1 If an ASN.1 type is encoded using any of the encoding rules identified by the object identifiers listed in subclause 33.2 (or by direct textual reference to this Recommendation | International Standard), and the encoding is included in:

a) an ASN.1 octetstring; or

b) an ASN.1 bitstring, or

c) an ASN.1 open type; or

d) any part of an ASN.1 external or embedded pdv type; or

e) 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 subclause 11.1.2 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 except when the UNALIGNED variant is used and the encoding is included in an ASN.1 bitstring (case b)) above).

NOTE 2 – It is possible using the Encoding Control Notation (see Recommendation ITU-T 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 11.1.2. 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 11.1.2), but can imply their presence.

11.1.2 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.

11.1.3 In the UNALIGNED variant of these encoding rules, all fields shall be concatenated without padding. In all the cases of 11.1.1 except case b), subclause 11.1.3.1 applies. In case b) of 11.1.1, subclause 11.1.3.2 applies.

11.1.3.1 (The result of the encoding is not contained in an ASN.1 bitstring) 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, (one to seven) zero bits shall be appended to it to produce a multiple of eight bits.

11.1.3.2 (The result of the encoding is contained in an ASN.1 bitstring) If the result of encoding the outermost value is an empty bit string, the bit string shall be replaced with a single bit set to 0. No padding bits shall be appended.

11.1.4 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 1 – 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.

NOTE 2 – Zero-length octet-aligned bit-fields can never be present in the field-list (see 11.9.3.3).

11.1.5 The resulting bit string is the complete encoding of the abstract syntax value of an outermost type.

11.2 Open type fields

11.2.1 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 11.1 to produce an octet string of length "n" (say).

11.2.2 The field-list for the value in which the open type is to be embedded shall then have added to it (as specified in 11.9) 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 11.2.1.

NOTE – Where the number of octets in the open type encoding is large, the fragmentation procedures of 11.9 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.

11.3 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.

11.3.1 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 (11.3) specifies the precise encoding to be applied when such references are made.

11.3.2 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.

11.3.3 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.

11.3.4 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.

11.3.5 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.

11.3.6 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.

11.4 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.

11.4.1 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 (11.4) specifies the precise encoding to be applied when such references are made.

11.4.2 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.

11.4.3 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.

11.4.4 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.

11.4.5 Any encoding which sums (as defined above) to the value being encoded is an encoding of that value.

11.4.6 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.

11.5 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 11.5.3). 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 11.9) 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.

11.5.1 This subclause (11.5) 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.

11.5.2 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)**.

11.5.3 Let "range" be defined as the integer value ("ub" – "lb" + 1), and let the value to be encoded be "n".

11.5.4 If "range" has the value 1, then the result of the encoding shall be an empty bit-field (no bits).

11.5.5 There are five other cases (leading to different encodings) to consider, where one applies to the UNALIGNED variant and four to the ALIGNED variant.

11.5.6 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 11.3 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.

11.5.7 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).

11.5.7.1 (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 11.3.

"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

11.5.7.2 (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 11.3.

11.5.7.3 (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 11.3.

11.5.7.4 (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 11.3, 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.

11.6 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.

11.6.1 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.

11.6.2 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 11.9 shall be invoked to add it to the field-list preceded by a length determinant.

11.7 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.

11.7.1 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.

11.7.2 The procedures of this subclause (11.7) 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)`**.

11.7.3 The procedures of this subclause always produce the indefinite length case.

11.7.4 (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 11.3, 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.

11.8 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.

11.8.1 This subclause (11.8) 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.

11.8.2 The procedures of this subclause always produce the indefinite length case.

11.8.3 (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 11.4, 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.

11.9 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.

11.9.1 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.

11.9.2 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).

11.9.3 (ALIGNED variant) The procedures for the ALIGNED variant are specified in 11.9.3.1 to 11.9.3.8.4. (The procedures for the UNALIGNED variant are specified in 11.9.4.)

11.9.3.1 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.

11.9.3.2 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.

11.9.3.3 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 11.5. 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 11.5.

11.9.3.4 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 11.9.3.5 to 11.9.3.8.4.

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.

11.9.3.5 Otherwise (unconstrained length, or large "ub"), "n" is encoded and appended to the field-list followed by the associated fields as specified below.

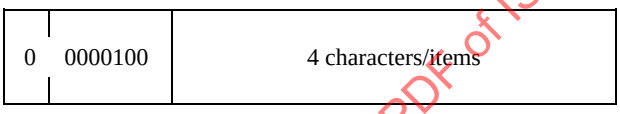
NOTE – The lower bound, "lb", does not affect the length encodings specified in 11.9.3.6 to 11.9.3.8.4.

11.9.3.6 If "n" is less than or equal to 127, then "n" shall be encoded as a non-negative-binary-integer (using the procedures of 11.3) 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:

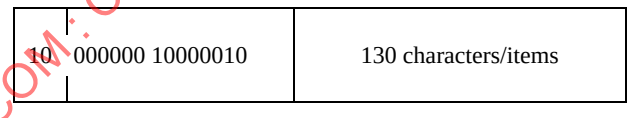
```
A ::= IA5String

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:



11.9.3.7 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 11.3) 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 11.9.3.6 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.



11.9.3.8 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 11.8). 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).

11.9.3.8.1 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 11.9.3.6 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

11.9.3.8.2 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.

11.9.3.8.3 The procedures of 11.9 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.

11.9.3.8.4 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*.

11.9.4 (UNALIGNED variant) The procedures for the UNALIGNED variant are specified in 11.9.4.1 to 11.9.4.2 (the procedures for the ALIGNED variant are specified in 11.9.3).

11.9.4.1 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 11.3) 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" ≤ 2^{m+1}, then the number of bits in the length determinant is m + 1.

11.9.4.2 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 11.9.3.4 to 11.9.3.8.4.

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.

12 Encoding the boolean type

12.1 A value of the boolean type shall be encoded as a bit-field consisting of a single bit.

12.2 The bit shall be set to 1 for **`TRUE`** and 0 for **`FALSE`**.

12.3 The bit-field shall be appended to the field-list with no length determinant.

13 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.

13.1 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 13.2.4 to 13.2.6, completing this procedure. In the latter case, the value shall be encoded as if the extension marker is not present.

13.2 If an extension marker is not present in the constraint specification of the integer type, then the following applies.

13.2.1 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.

13.2.2 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 11.5 (encoding of a constrained whole number), and the procedures of 13.2.5 to 13.2.6 shall then be applied.

13.2.3 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 11.7 (encoding of a semi-constrained whole number), and the procedures of 13.2.6 shall then be applied.

13.2.4 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 11.8 (encoding of an unconstrained whole number), and the procedures of 13.2.6 shall then be applied.

13.2.5 If the procedures invoked to encode the integer value into a field did not produce the indefinite length case (see 11.5.7.4 and 11.8.2), then that field shall be appended to the field-list completing these procedures.

13.2.6 Otherwise, (the indefinite length case) the procedures of 11.9 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 11.5.7.4) 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 in the ALIGNED variant 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 significant bits (bits 2 and 1) of an octet).

b) An unconstrained length determinant equal to "len" (as determined by 11.7 and 11.8) 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.

14 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.

14.1 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 – Rec. ITU-T X.680 | ISO/IEC 8824-1 requires that each successive extension addition shall have a greater enumeration value than the last.

14.2 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.

14.3 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 14.1 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 14.2.

NOTE – There are no PER-visible constraints that can be applied to an enumerated type that are visible to these encoding rules.

15 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.

15.1 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.

15.2 The encoding of **REAL** specified for CER and DER in Rec. ITU-T X.690 | ISO/IEC 8825-1, 11.3 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 11.9 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".

16 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.

16.1 PER-visible constraints can only constrain the length of the bitstring.

16.2 Where there are no PER-visible constraints and Rec. ITU-T X.680 | ISO/IEC 8824-1, 22.7, 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).

16.3 Where there is a PER-visible constraint and Rec. ITU-T X.680 | ISO/IEC 8824-1, 22.7, 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.

16.4 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.

16.5 When a bitstring value is placed in a bit-field as specified in 16.6 to 16.11, 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.

16.6 If the type is extensible for PER encodings (see 10.3.9), then a bit-field consisting of a single bit shall be added to the field-list. 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, 16.11 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 no extension is present in the constraint.

16.7 If an extension marker is not present in the constraint specification of the bitstring type, then 16.8 to 16.11 apply.

16.8 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.

16.9 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.

16.10 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.

16.11 If 16.8-16.10 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 11.9 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.

17 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).

17.1 PER-visible constraints can only constrain the length of the octetstring.

17.2 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.

17.3 If the type is extensible for PER encodings (see 10.3.9), then a bit-field consisting of a single bit shall be added to the field-list. 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 17.8 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 no extension is present in the constraint.

17.4 If an extension marker is not present in the constraint specification of the octetstring type, then 17.5 to 17.8 apply.

17.5 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.

17.6 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.

17.7 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.

17.8 If 17.5 to 17.7 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 11.9 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.

18 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.

19 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.

19.1 If the sequence type has an extension marker in the "ComponentTypeLists" or in the "SequenceType" productions, 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 in the "ComponentTypeLists" or in the "SequenceType" productions, there shall be no extension bit added.

19.2 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.

19.3 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 11.9 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.

19.4 The preamble shall be followed by the field-lists of each of the components of the sequence value which are present, taken in turn.

19.5 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 3.7.25), otherwise it is a sender's option whether or not to encode it.

19.6 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.

19.7 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.

19.8 The procedures of 11.9 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.

19.9 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 11.2.1. Each extension addition that is an "ExtensionAdditionGroup" shall be encoded as a sequence type as specified in 19.2 to 19.6, which is then encoded as if it were the value of an open type field as specified in 11.2.1. 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 19.7 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.

20 Encoding the sequence-of type

20.1 PER-visible constraints can constrain the number of components of the sequence-of type.

20.2 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.

20.3 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.

20.4 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 11.9 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.

20.5 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.

20.6 Otherwise, the procedures of 11.9 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.

21 Encoding the set type

The set type shall have the elements in its "RootComponentTypeList" sorted into the canonical order specified in Rec. ITU-T 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 19.9 (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]`**.

22 Encoding the set-of type

22.1 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.

22.2 For BASIC-PER the set-of shall be encoded as if it had been declared a sequence-of type.

23 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.

23.1 Encoding of choice types are not affected by PER-visible constraints.

23.2 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 Rec. ITU-T 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 – Rec. ITU-T X.680 | ISO/IEC 8824-1, 29.7, requires that each successive extension addition shall have a greater tag value than the last added to the "ExtensionAdditionAlternativesList".

23.3 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.

23.4 If the choice has only one alternative in the extension root, there shall be no encoding for the index if that alternative is chosen.

23.5 If the choice type has an extension marker in the "AlternativeTypeLists" production, 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 in the "AlternativeTypeLists" production, there shall be no extension bit added.

23.6 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 13 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.

23.7 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 13. This shall then be followed by the fields of the chosen alternative, completing the procedures of this clause.

23.8 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 11.2, completing the procedures of this clause.

NOTE – Version brackets in the definition of choice extension additions have no effect on how "ExtensionAdditionAlternatives" are encoded.

24 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 11.9 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 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 24.

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 11.9 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.

26 Encoding the internationalized resource reference type

NOTE – (Tutorial) An internationalized resource reference 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 24.

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 11.9 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.

27 Encoding the relative internationalized resource reference type

NOTE – (Tutorial) A relative internationalized resource reference 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 24.

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 11.9 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.

28 Encoding the embedded-pdv type

28.1 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.

28.2 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 36.3 a).

28.3 In the "general" case, the encoding of a value of the embedded-pdv type shall be the PER encoding of the type defined in Rec. ITU-T X.680 | ISO/IEC 8824-1, 36.5, 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 36.3 a).

29 Encoding of a value of the external type

29.1 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 Rec. ITU-T X.680 | ISO/IEC 8824-1 for historical reasons.

29.2 The value of the components depends on the abstract value being transmitted, which is a value of the type specified in Rec. ITU-T X.680 | ISO/IEC 8824-1, 36.5.

29.3 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.

29.4 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 36.5, to the external type components **`direct-reference`** and **`indirect-reference`** defined in 29.1.

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 ***

29.5 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.

29.6 If the data value is the value of a single ASN.1 data type (see the Note in 29.7), 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.

29.7 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 29.6.

29.8 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**.

29.9 If the **encoding** choice is chosen as **single-ASN1-type**, then the ASN.1 type shall be encoded as specified in 11.2 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**.

29.10 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.

29.11 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.

30 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 10.3.12), 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 3.7.16), 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.

30.1 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.

30.2 The effective size constraint notation may determine an upper bound "aub" for the length of the abstract character string. Otherwise, "aub" is unset.

30.3 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.

30.4 If the type is extensible for PER encodings (see 10.3.18), 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 the effective permitted-alphabet constraint specified in 10.3.12.

NOTE 1 – 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.

NOTE 2 – Effective permitted-alphabet constraints can never be extensible, as extensible permitted-alphabet constraints are not PER-visible (see 10.3.11).

30.5 This subclause applies to known-multiplier character strings. Encoding of the other restricted character string types is specified in 30.6.

30.5.1 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.

30.5.2 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".

30.5.3 A numerical value "v" is associated with each character by reference to Rec. ITU-T X.680 | ISO/IEC 8824-1, clause 43 as follows. For **UniversalString**, the value is that used to determine the canonical order in Rec. ITU-T X.680 | ISO/IEC 8824-1, 43.3 (the value is in the range 0 to $2^{32} - 1$). For **BMPString**, the value is that used to determine the canonical order in Rec. ITU-T X.680 | ISO/IEC 8824-1, 43.3 (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.)

30.5.4 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, clause 43. 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).

30.5.5 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.

30.5.6 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.

30.5.7 If "aub" does not equal "alb" or is greater than or equal to 64K, then 11.9 shall be invoked to add the bit-field preceded by 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 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 30.5.6 and 30.5.7 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, 30.5.6 specifies no alignment and 30.5.7 specifies alignment.

30.6 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.

30.6.1 For BASIC-PER, reference below to "base encoding" means production of the octet string specified in Rec. ITU-T X.690 | ISO/IEC 8825-1, 8.23.5. For CANONICAL-PER it means the production of the same octet string subject to the restrictions specified for CER and DER in Rec. ITU-T X.690 | ISO/IEC 8825-1, 11.4.

30.6.2 The "base encoding" shall be applied to the character string to give a field of "n" octets.

30.6.3 Subclause 11.9 shall be invoked to add the field of "n" octets as a bit-field (octet-aligned in the ALIGNED variant), preceded by an unconstrained length determinant with "n" as a count in octets, completing the procedures of this subclause.

31 Encoding the unrestricted character string type

31.1 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.

31.2 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 44.3 a).

31.3 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 44.5, 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 44.3 a).

32 Encoding the time type, the useful time types, the defined time types and the additional time types

32.1 General

32.1.1 The encoding of the useful time types, the defined time types and the additional time types shall be determined by the property settings of the abstract values of these types. Property settings for the abstract values of the useful and defined time types are specified in Rec. ITU-T X.680 | ISO/IEC 8824-1, 38.4 and Annex B, respectively. Property settings for the abstract values of additional time types are determined by the property settings of the parent type, restricted by any PER-visible constraints that apply (see 10.3.13).

32.1.2 If all the abstract values of the type to be encoded have one of the property settings listed in a row of column 2 of Table 2, then that type shall be encoded as if the type with its constraints (if any) had been replaced by the type specified in the corresponding row of column 3 of Table 2. Otherwise, it shall be encoded as specified in 32.11.

NOTE – If a time property (for example **Midnight**) is not listed in Table 2 for a particular row, there is no constraint on its setting.

32.1.3 For rows 24 to 32 to be applicable, all abstract values of the type are required to have the same value of ***n*** in **F*n***.

32.1.4 The types specified in column 3 of Table 2 are defined (using the ASN.1 notation) in 32.2 to 32.10, and are assumed to be defined in an environment of **AUTOMATIC TAGS**.

NOTE 1 – The use of these type reference names in the specification of PER encodings does not make them available for use by an application designer in an ASN.1 specification, nor are they reserved words in such a specification. However, with the removal of **-ENCODING**, they correspond to the names of the useful time types or defined time types specified in Rec. ITU-T X.680 | ISO/IEC 8824-1, 38.4 and Annex B.

NOTE 2 – All the useful and defined time types satisfy the conditions for one of the rows of Table 2, and hence have optimized encodings. Additional time types may satisfy the conditions for one of the rows, but are otherwise encoded as specified in 32.11. The unconstrained **TIME** type is always encoded as specified in 32.11.

Table 2 – Encoding of a time subtype with all abstract values having specified property settings

Row number	Property settings	ASN.1 type to be encoded
1	"Basic=Date Date=C Year=Basic" or "Basic=Date Date=C Year=Proleptic"	CENTURY-ENCODING (see 32.2.1)
2	"Basic=Date Date=C Year=Negative" or "Basic=Date Date=C Year=L*n*" (for any *n*)	ANY-CENTURY-ENCODING (see 32.2.2)
3	"Basic=Date Date=Y Year=Basic" or "Basic=Date Date=Y Year=Proleptic"	YEAR-ENCODING (see 32.2.3)
4	"Basic=Date Date=Y Year=Negative" or "Basic=Date Date=Y Year=L*n*" (for any *n*)	ANY-YEAR-ENCODING (see 32.2.4)
5	"Basic=Date Date=YM Year=Basic" or "Basic=Date Date=YM Year=Proleptic"	YEAR-MONTH-ENCODING (see 32.2.5)
6	"Basic=Date Date=YM Year=Negative" or "Basic=Date Date=YM Year=L*n*" (for any *n*)	ANY-YEAR-MONTH-ENCODING (see 32.2.6)
7	"Basic=Date Date=YMD Year=Basic" or "Basic=Date Date=YMD Year=Proleptic"	DATE-ENCODING (see 32.2.7)
8	"Basic=Date Date=YMD Year=Negative" or "Basic=Date Date=YMD Year=L*n*" (for any *n*)	ANY-DATE-ENCODING (see 32.2.8)
9	"Basic=Date Date=YD Year=Basic" or "Basic=Date Date=YD Year=Proleptic"	YEAR-DAY-ENCODING (see 32.2.9)
10	"Basic=Date Date=YD Year=Negative" or "Basic=Date Date=YD Year=L*n*" (for any *n*)	ANY-YEAR-DAY-ENCODING (see 32.2.10)
11	"Basic=Date Date=YW Year=Basic" or "Basic=Date Date=YW Year=Proleptic"	YEAR-WEEK-ENCODING (see 32.2.11)
12	"Basic=Date Date=YW Year=Negative" or "Basic=Date Date=YW Year=L*n*" (for any *n*)	ANY-YEAR-WEEK-ENCODING (see 32.2.12)
13	"Basic=Date Date=YWD Year=Basic" or "Basic=Date Date=YWD Year=Proleptic"	YEAR-WEEK-DAY-ENCODING (see 32.2.13)
14	"Basic=Date Date=YWD Year=Negative" or "Basic=Date Date=YWD Year=L*n*" (for any *n*)	ANY-YEAR-WEEK-DAY-ENCODING (see 32.2.14)
15	"Basic=Time Time=H Local-or-UTC=L"	HOURS-ENCODING (see 32.3.1)
16	"Basic=Time Time=H Local-or-UTC=Z"	HOURS-UTC-ENCODING (see 32.3.2)
17	"Basic=Time Time=H Local-or-UTC=LD"	HOURS-AND-DIFF-ENCODING (see 32.3.3)
18	"Basic=Time Time=HM Local-or-UTC=L"	MINUTES-ENCODING (see 32.3.4)

Table 2 – Encoding of a time subtype with all abstract values having specified property settings

Row number	Property settings	ASN.1 type to be encoded
19	"Basic=Time Time=HM Local-or-UTC=Z"	MINUTES-UTC-ENCODING (see 32.3.5)
20	"Basic=Time Time=HM Local-or-UTC=LD"	MINUTES-AND-DIFF-ENCODING (see 32.3.6)
21	"Basic=Time Time=HMS Local-or-UTC=L"	TIME-OF-DAY-ENCODING (see 32.3.7)
22	"Basic=Time Time=HMS Local-or-UTC=Z"	TIME-OF-DAY-UTC-ENCODING (see 32.3.8)
23	"Basic=Time Time=HMS Local-or-UTC=LD"	TIME-OF-DAY-AND-DIFF-ENCODING (see 32.3.9)
24	"Basic=Time Time=HFn Local-or-UTC=L" (but see 32.1.3)	HOURS-AND-FRACTION-ENCODING (see 32.3.10)
25	"Basic=Time Time=HFn Local-or-UTC=Z" (but see 32.1.3)	HOURS-UTC-AND-FRACTION-ENCODING (see 32.3.11)
26	"Basic=Time Time=HFn Local-or-UTC=LD" (but see 32.1.3)	HOURS-AND-DIFF-AND-FRACTION-ENCODING (see 32.3.12)
27	"Basic=Time Time=HMFn Local-or-UTC=L" (but see 32.1.3)	MINUTES-AND-FRACTION-ENCODING (see 32.3.13)
28	"Basic=Time Time=HMFn Local-or-UTC=Z" (but see 32.1.3)	MINUTES-UTC-AND-FRACTION-ENCODING (see 32.3.14)
29	"Basic=Time Time=HMFn Local-or-UTC=LD" (but see 32.1.3)	MINUTES-AND-DIFF-AND-FRACTION-ENCODING (see 32.3.15)
30	"Basic=Time Time=HMSFn Local-or-UTC=L" (but see 32.1.3)	TIME-OF-DAY-AND-FRACTION-ENCODING (see 32.3.16)
31	"Basic=Time Time=HMSFn Local-or-UTC=Z" (but see 32.1.3)	TIME-OF-DAY-UTC-AND-FRACTION-ENCODING (see 32.3.17)
32	"Basic=Time Time=HMSFn Local-or-UTC=LD" (but see 32.1.3)	TIME-OF-DAY-AND-DIFF-AND-FRACTION-ENCODING (see 32.3.18)
33	"Basic=Date-Time" All abstract values are required to have the same additional property settings specified in one of rows 7, 8, 9, 10, 13 and 14 for "Basic=Date" together with the same additional property settings specified in one of the rows 15 to 32 for "Basic=Time".	DATE-TIME-ENCODING {Date-Type, Time-Type} (instantiated as specified in 32.4.1)
34	"Basic=Interval Interval-type=SE SE-point=Date" All abstract values are required to have the same additional property settings specified in one of rows 1 to 14 for "Basic=Date".	START-END-DATE-INTERVAL-ENCODING {Date-Type} (see 32.5.1)
35	"Basic=Interval Interval-type=SE SE-point=Time" All abstract values are required to have the same additional property settings specified in one of rows 15 to 32 for "Basic=Time".	START-END-TIME-INTERVAL-ENCODING {Time-Type} (see 32.5.2)
36	"Basic=Interval Interval-type=SE SE-point=Date-Time" All abstract values are required to have the same additional property settings specified in one of rows 7, 8, 9, 10, 13 and 14 for "Basic=Date" together with the same additional property settings specified in one of rows 15 to 32 for "Basic=Time".	START-END-DATE-TIME-INTERVAL-ENCODING {Date-Type, Time-Type} (see 32.5.3)

Table 2 – Encoding of a time subtype with all abstract values having specified property settings

Row number	Property settings	ASN.1 type to be encoded
37	**"Basic=Interval Interval-type=D"**	**DURATION-INTERVAL-ENCODING** (see 32.6.1)
38	**"Basic=Interval Interval-type=SD SE-point=Date"** All abstract values are required to have the same additional property settings specified in one of rows 1 to 14 for **"Basic=Date"**.	**START-DATE-DURATION-INTERVAL-ENCODING {Date-Type}** (see 32.7.1)
39	**"Basic=Interval Interval-type=SD SE-point=Time"** All abstract values are required to have the same additional property settings specified in one of rows 15 to 32 for **"Basic=Time"**.	**START-TIME-DURATION-INTERVAL-ENCODING {Time-Type}** (see 32.7.2)
40	**"Basic=Interval Interval-type=SD SE-point=Date-Time"** All abstract values are required to have the same additional property settings specified in one of rows 7, 8, 9, 10, 13 and 14 for **"Basic=Date"** together with the same additional property settings specified in one of rows 15 to 32 for **"Basic=Time"**.	**START-DATE-TIME-DURATION-INTERVAL-ENCODING {Date-Type, Time-Type}** (see 32.7.3)
41	**"Basic=Interval Interval-type=DE SE-point=Date"** All abstract values are required to have the same additional properties specified in one of rows 1 to 14 for **"Basic=Date"**.	**DURATION-END-DATE-INTERVAL-ENCODING {Date-Type}** (see 32.7.4)
42	**"Basic=Interval Interval-type=DE SE-point=Time"** All abstract values are required to have the same additional properties specified in one of rows 15 to 32 for **"Basic=Time"**.	**DURATION-END-TIME-INTERVAL-ENCODING {Time-Type}** (see 32.7.5)
43	**"Basic=Interval Interval-type=DE SE-point=Date-Time"** All abstract values are required to have the same additional properties specified in one of rows 7, 8, 9, 10, 13 and 14 for **"Basic=Date"** together with the same additional property settings specified in one of rows 15 to 32 for **"Basic=Time"**.	**DURATION-END-DATE-TIME-INTERVAL-ENCODING {Date-Type, Time-Type}** (see 32.7.6)
44	**"Basic=Rec-Interval Interval-type=SE SE-point=Date"** All abstract values are required to have the same additional property settings specified in one of rows 1 to 14 for **"Basic=Date"**.	**REC-START-END-DATE-INTERVAL-ENCODING {Date-Type}** (see 32.8.1)
45	**"Basic=Rec-Interval Interval-type=SE SE-point=Time"** All abstract values are required to have the same additional property settings specified in one of rows 15 to 32 for **"Basic=Time"**.	**REC-START-END-TIME-INTERVAL-ENCODING {Time-Type}** (see 32.8.2)
46	**"Basic=Rec-Interval Interval-type=SE SE-point=Date-Time"** All abstract values are required to have the same additional property settings specified in one of rows 7, 8, 9, 10, 13 and 14 for **"Basic=Date"** together with the same additional property settings specified in one of rows 15 to 32 for **"Basic=Time"**.	**REC-START-END-DATE-TIME-INTERVAL-ENCODING {Date-Type, Time-Type}** (see 32.8.3)
47	**"Basic=Rec-Interval Interval-type=D"**	**REC-DURATION-INTERVAL-ENCODING** (see 32.9.1)

Table 2 – Encoding of a time subtype with all abstract values having specified property settings

Row number	Property settings	ASN.1 type to be encoded
48	**"Basic=Rec-Interval Interval-type=SD SE-point=Date"** All abstract values are required to have the same additional property settings specified in one of rows 1 to 14 for **"Basic=Date"**.	**REC-START-DATE-DURATION-INTERVAL-ENCODING {Date-Type}** (see 32.10.1)
49	**"Basic=Rec-Interval Interval-type=SD SE-point=Time"** All abstract values are required to have the same additional property settings specified in one of rows 15 to 32 for **"Basic=Time"**.	**REC-START-TIME-DURATION-INTERVAL-ENCODING {Time-Type}** (see 32.10.2)
50	**"Basic=Rec-Interval Interval-type=SD SE-point=Date-Time"** All abstract values are required to have the same additional property settings specified in one of rows 7, 8, 9, 10, 13 and 14 for **"Basic=Date"** together with the same additional property settings specified in one of rows 15 to 32 for **"Basic=Time"**.	**REC-START-DATE-TIME-DURATION-INTERVAL-ENCODING {Date-Type, Time-Type}** (see 32.10.3)
51	**"Basic=Rec-Interval Interval-type=DE SE-point=Date"** All abstract values are required to have the same additional properties specified in one of rows 1 to 14 for **"Basic=Date"**.	**REC-DURATION-END-DATE-INTERVAL-ENCODING {Date-Type}** (see 32.10.4)
52	**"Basic=Rec-Interval Interval-type=DE SE-point=Time"** All abstract values are required to have the same additional properties specified in one of rows 15 to 32 for **"Basic=Time"**.	**REC-DURATION-END-TIME-INTERVAL-ENCODING {Time-Type}** (see 32.10.5)
53	**"Basic=Rec-Interval Interval-type=DE SE-point=Date-Time"** All abstract values are required to have the same additional properties specified in one of rows 7, 8, 9, 10, 13 and 14 for **"Basic=Date"** together with the same additional property settings specified in one of rows 15 to 32 for **"Basic=Time"**.	**REC-DURATION-END-DATE-TIME-INTERVAL-ENCODING {Date-Type, Time-Type}** (see 32.10.6)

32.2 Encoding subtypes with the "Basic=Date" property setting

This subclause defines the ASN.1 types referenced in Table 2, column 3 for types where all the abstract values of the type have the **"Basic=Date"** property setting.

32.2.1 The **CENTURY-ENCODING** type is:

```
CENTURY-ENCODING ::= INTEGER(0..99) -- 7 bits
```

with the integer value set to the value specified by the first two digits of the year component of the abstract value.

32.2.2 The **ANY-CENTURY-ENCODING** type is:

```
ANY-CENTURY-ENCODING ::= INTEGER(MIN..MAX)
```

with the integer value set to the value specified by the year component of the abstract value, ignoring the last two digits.

32.2.3 The **YEAR-ENCODING** type is:

```
YEAR-ENCODING ::= CHOICE { -- 2 bits for choice determinant
        immediate       INTEGER (2005..2020),  -- 4 bits
        near-future     INTEGER (2021..2276),  -- 8 bits
        near-past       INTEGER (1749..2004),  -- 8 bits
        remainder       INTEGER (MIN..1748 | 2277..MAX)}
```

with the integer value set to the year component of the abstract value.

NOTE – This has been optimized to provide a 6-bit or a 10-bit encoding in common cases.

32.2.4 The **ANY-YEAR-ENCODING** type is:

```
ANY-YEAR-ENCODING ::= INTEGER(MIN..MAX)
```

with the integer value set to the year component of the abstract value.

32.2.5 The **YEAR-MONTH-ENCODING** type is:

```
YEAR-MONTH-ENCODING ::= SEQUENCE {
        year        YEAR-ENCODING,
        month       INTEGER (1..12) -- 4 bits -- }
```

with the **YEAR-ENCODING** set according to 32.2.3 and the **month** integer value set to the month component of the abstract value.

NOTE – This has been optimized to provide a 10-bit or a 14-bit encoding in common cases.

32.2.6 The **ANY-YEAR-MONTH-ENCODING** type is:

```
ANY-YEAR-MONTH-ENCODING ::= SEQUENCE {
        year        ANY-YEAR-ENCODING,
        month       INTEGER (1..12) }
```

with the **ANY-YEAR-ENCODING** set according to 32.2.4 and the **month** integer value set to the month component of the abstract value.

32.2.7 The **DATE-ENCODING** type is:

```
DATE-ENCODING ::= SEQUENCE {
        year        YEAR-ENCODING,
        month       INTEGER (1..12), -- 4 bits
        day         INTEGER (1..31) -- 5 bits -- }
```

with the **YEAR-ENCODING** set according to 32.2.3, the **month** integer value set to the month component of the abstract value and the **day** integer value set to the day component of the abstract value.

NOTE – This has been optimized to provide a 15-bit or a 19-bit encoding in common cases.

32.2.8 The **ANY-DATE-ENCODING** type is:

```
ANY-DATE-ENCODING ::= SEQUENCE {
        year        ANY-YEAR-ENCODING,
        month       INTEGER (1..12),
        day         INTEGER (1..31)}
```

with the **ANY-YEAR-ENCODING** set according to 32.2.4, the **month** integer value set to the month component of the abstract value and the **day** integer value set to the day component of the abstract value.

32.2.9 The **YEAR-DAY-ENCODING** type is:

```
YEAR-DAY-ENCODING ::= SEQUENCE {
        year        YEAR-ENCODING,
        day         INTEGER (1..366)}
```

with the **YEAR-ENCODING** set according to 32.2.3 and the **day** integer value set to the day component of the abstract value.

32.2.10 The **ANY-YEAR-DAY-ENCODING** type is:

```
ANY-YEAR-DAY-ENCODING ::= SEQUENCE {
        year        ANY-YEAR-ENCODING,
        day         INTEGER (1..366)}
```

with the **ANY-YEAR-ENCODING** set according to 32.2.4 and the **day** integer value set to the day component of the abstract value.

32.2.11 The **YEAR-WEEK-ENCODING** type is:

```
YEAR-WEEK-ENCODING ::= SEQUENCE {
        year        YEAR-ENCODING,
        week        INTEGER (1..53) -- 6 bits --}
```

with the **YEAR-ENCODING** set according to 32.2.3 and the **week** integer value set to the week component of the abstract value.

NOTE – This has been optimized to provide a 12-bit or a 16-bit encoding in common cases.

32.2.12 The **ANY-YEAR-WEEK-ENCODING** type is:

```
ANY-YEAR-WEEK-ENCODING ::= SEQUENCE {
        year        ANY-YEAR-ENCODING,
        week        INTEGER (1..53)}
```

with the **ANY-YEAR-ENCODING** set according to 32.2.4 and the **week** integer value set to the week component of the abstract value.

32.2.13 The **YEAR-WEEK-DAY-ENCODING** type is:

```
YEAR-WEEK-DAY-ENCODING ::= SEQUENCE {
        year        YEAR-ENCODING,
        week        INTEGER (1..53), -- 6 bits
        day         INTEGER (1..7) -- 3 bits -- }
```

with the **YEAR-ENCODING** set according to 32.2.3, the **week** integer value set to the week component of the abstract value and the **day** integer value set to the day component of the abstract value.

NOTE – This has been optimized to provide a 15-bit or a 19-bit encoding in common cases.

32.2.14 The **ANY-YEAR-WEEK-DAY-ENCODING** type is:

```
ANY-YEAR-WEEK-DAY-ENCODING ::= SEQUENCE {
        year        ANY-YEAR-ENCODING,
        week        INTEGER (1..53),
        day         INTEGER (1..7)}
```

with the **ANY-YEAR-ENCODING** set according to 32.2.4, the **week** integer value set to the week component of the abstract value and the **day** integer value set to the day component of the abstract value.

32.3 Encoding subtypes with the "Basic=Time" property setting

This subclause defines the ASN.1 types referenced in Table 2, column 3 for types where all the abstract values of the type have the **Basic=Time** property setting.

32.3.1 The **HOURS-ENCODING** type is:

```
HOURS-ENCODING ::= INTEGER(0..24) -- 5 bits
```

with the integer value set to the hours component of the abstract value.

NOTE – This has been optimized to provide a 5-bit encoding.

32.3.2 The **HOURS-UTC-ENCODING** type is:

```
HOURS-UTC-ENCODING ::= INTEGER(0..24) -- 5 bits
```

with the integer value set to the hours component of the abstract value.

NOTE – This has been optimized to provide a 5-bit encoding.

32.3.3 The **HOURS-AND-DIFF-ENCODING** type is:

```
HOURS-AND-DIFF-ENCODING ::= SEQUENCE {
        local-hours      INTEGER (0..24),
        time-difference  TIME-DIFFERENCE }
```

where:

```
TIME-DIFFERENCE ::= SEQUENCE {
        sign        ENUMERATED { positive, negative },
        hours       INTEGER (0..15),
        minutes     INTEGER (1..59) OPTIONAL }
```

with the **local-hours** integer value set to the hours component of the local time of the abstract value and the **time-difference** set to the sign, hours and minutes of the time-difference component of the abstract value. If the minutes component of the time-difference is zero, the **TIME-DIFFERENCE minutes** shall be omitted.

32.3.4 The **MINUTES-ENCODING** type is:

```
MINUTES-ENCODING ::= SEQUENCE {
        hours       INTEGER (0..24), -- 5 bits
        minutes     INTEGER (0..59) -- 5 bits -- }
```

with the **hours** integer value set to the hours component of the abstract value and the **minutes** integer value set to the minutes component.

NOTE – This has been optimized to provide a 10-bit encoding.

32.3.5 The **MINUTES-UTC-ENCODING** type is:

```
MINUTES-UTC-ENCODING ::= SEQUENCE {
        hours       INTEGER (0..24), -- 5 bits
        minutes     INTEGER (0..59) -- 5 bits -- }
```

with the **hours** integer value set to the hours component of the abstract value and the **minutes** integer value set to the minutes component.

NOTE – This has been optimized to provide a 10-bit encoding.

32.3.6 The **MINUTES-AND-DIFF-ENCODING** type is:

```
MINUTES-AND-DIFF-ENCODING ::= SEQUENCE {
        local-time SEQUENCE {
          hours    INTEGER (0..24),
          minutes  INTEGER (0..59) },
        time-difference       TIME-DIFFERENCE }
```

with the **local-time** set to the hours and minutes component of the local time of the abstract value and the **time-difference** set to the sign, hours and minutes of the time-difference component of the abstract value as specified in 32.3.3.

32.3.7 The **TIME-OF-DAY-ENCODING** type is:

```
TIME-OF-DAY-ENCODING ::= SEQUENCE {
        hours       INTEGER (0..24), -- 5 bits
        minutes     INTEGER (0..59), -- 5 bits
        seconds     INTEGER (0..60) -- 5 bits -- }
```

with the **hours** integer value set to the hours component of the abstract value, the **minutes** integer value set to the minutes component, and the **seconds** integer value set to the seconds component.

NOTE – This has been optimized to provide a 15-bit encoding.

32.3.8 The **TIME-OF-DAY-UTC-ENCODING** type is:

```
TIME-OF-DAY-UTC-ENCODING ::= SEQUENCE {
        hours       INTEGER (0..24), -- 5 bits
        minutes     INTEGER (0..59), -- 5 bits
        seconds     INTEGER (0..60) -- 5 bits -- }
```

with the **hours** integer value set to the hours component of the abstract value, the **minutes** integer value set to the minutes component, and the **seconds** integer value set to the seconds component.

NOTE – This has been optimized to provide a 15-bit encoding.

32.3.9 The **TIME-OF-DAY-AND-DIFF-ENCODING** type is:

```
TIME-OF-DAY-AND-DIFF-ENCODING ::= SEQUENCE {
        local-time SEQUENCE {
          hours    INTEGER (0..24),
          minutes  INTEGER (0..59),
          seconds  INTEGER (0..60) },
        time-difference       TIME-DIFFERENCE }
```

with the **local-time** set to the hours, minutes and seconds components of the local time of the abstract value and the **time-difference** set to the sign, hours and minutes of the time-difference component of the abstract value as specified in 32.3.3.

32.3.10 The **HOURS-AND-FRACTION-ENCODING** type is:

```
HOURS-AND-FRACTION-ENCODING ::= SEQUENCE {
        hours       INTEGER (0..24), -- 5 bits
        fraction    INTEGER (0..999, ..., 1000..MAX)
          -- 11 bits for up to three digits accuracy -- }
```

with the **hours** integer value set to the hours component of the abstract value and the **fraction** integer value set to the fractional hours multiplied by ten-to-the-power-N, where N is the specified number of digits in the fractional part.

NOTE – This has been optimized to provide a 16-bit encoding for up to 3-digit accuracy.

32.3.11 The **HOURS-UTC-AND-FRACTION-ENCODING** type is:

```
HOURS-UTC-AND-FRACTION-ENCODING ::= SEQUENCE {
        hours       INTEGER (0..24), -- 5 bits
        fraction    INTEGER (0..999, ..., 1000..MAX)
          -- 11 bits for up to three digits accuracy -- }
```

with the **hours** integer value set to the hours component of the abstract value and the **fraction** integer value set to the fractional hours multiplied by ten-to-the-power-N, where N is the specified number of digits in the fractional part.

NOTE – This has been optimized to provide a 16-bit encoding for up to 3-digit accuracy.

32.3.12 The **HOURS-AND-DIFF-AND-FRACTION-ENCODING** type is:

```
HOURS-AND-DIFF-AND-FRACTION-ENCODING ::= SEQUENCE {
        local-hours       INTEGER (0..24), -- 5 bits
        fraction    INTEGER (0..999, ..., 1000..MAX)
          -- 11 bits for up to three digits accuracy -- ,
        time-difference       TIME-DIFFERENCE }
```

with the **local-hours** integer value set to the hours component of the local time of the abstract value, the **fraction** integer value set to the fractional hours multiplied by ten-to-the-power-N (where N is the specified number of digits in the fractional part) and the **time-difference** set to the sign, hours and minutes of the time-difference component of the abstract value as specified in 32.3.3.

32.3.13 The **MINUTES-AND-FRACTION-ENCODING** type is:

```
MINUTES-AND-FRACTION-ENCODING ::= SEQUENCE {
        hours       INTEGER (0..24), -- 5 bits
        minutes     INTEGER (0..59), -- 5 bits
        fraction    INTEGER (0..999, ..., 1000..MAX)
          -- 11 bits for up to three digits accuracy -- }
```

with the **hours** integer value set to the hours component of the abstract value, the **minutes** integer value set to the minutes component and the **fraction** integer value set to the fractional hours multiplied by ten-to-the-power-N, where N is the specified number of digits in the fractional part.

NOTE – This has been optimized to provide a 21-bit encoding for up to 3-digit accuracy.

32.3.14 The **MINUTES-UTC-AND-FRACTION-ENCODING** type is:

```
MINUTES-UTC-AND-FRACTION-ENCODING ::= SEQUENCE {
        hours       INTEGER (0..24), -- 5 bits
        minutes     INTEGER (0..59), -- 5 bits
        fraction    INTEGER (0..999, ..., 1000..MAX)
          -- 11 bits for up to three digits accuracy -- }
```

with the **hours** integer value set to the hours component of the abstract value, the **minutes** integer value set to the minutes component and the **fraction** integer value set to the fractional hours multiplied by ten-to-the-power-N (where N is the specified number of digits in the fractional part).

NOTE – This has been optimized to provide a 21-bit encoding for up to 3-digit accuracy.

32.3.15 The **MINUTES-AND-DIFF-AND-FRACTION-ENCODING** type is:

```
MINUTES-AND-DIFF-AND-FRACTION-ENCODING ::= SEQUENCE {
        local-time SEQUENCE {
          hours    INTEGER (0..24),
          minutes  INTEGER (0..59),
          fraction INTEGER (0..999, ..., 1000..MAX)},
        time-difference TIME-DIFFERENCE }
```

with the **local-time** set to the hours and minutes component of the local time of the abstract value, the **fraction** integer value set to the fractional minutes multiplied by ten-to-the-power-N (where N is the specified number of digits in the fractional part) and the **time-difference** set to the sign, hours and minutes of the time-difference component of the abstract value as specified in 32.3.3.

32.3.16 The **TIME-OF-DAY-AND-FRACTION-ENCODING** type is:

```
TIME-OF-DAY-AND-FRACTION-ENCODING ::= SEQUENCE {
        hours       INTEGER (0..24), -- 5 bits
        minutes     INTEGER (0..59), -- 5 bits
        seconds     INTEGER (0..60), -- 5 bits --
        fraction    INTEGER (0..999, ..., 1000..MAX)
```

-- 11 bits for up to three digits accuracy -- **}**

with the **hours** integer value set to the hours component of the abstract value, the **minutes** integer value set to the minutes component, the **seconds** integer value set to the seconds component and **fraction** integer value set to the fractional seconds multiplied by ten-to-the-power-N, where N is the specified number of digits in the fractional part.

NOTE – This has been optimized to provide a 26-bit encoding.

32.3.17 The **TIME-OF-DAY-UTC-AND-FRACTION-ENCODING** type is:

```
TIME-OF-DAY-UTC-AND-FRACTION-ENCODING ::= SEQUENCE {
        hours       INTEGER (0..24), -- 5 bits
        minutes     INTEGER (0..59), -- 5 bits
        seconds     INTEGER (0..60), -- 5 bits --
        fraction    INTEGER (0..999, ..., 1000..MAX)
          -- 11 bits for up to three digits accuracy -- }
```

with the **hours** integer value set to the hours component of the abstract value, the **minutes** integer value set to the minutes component, the **seconds** integer value set to the seconds component and **fraction** integer value set to the fractional seconds multiplied by ten-to-the-power-N, where N is the specified number of digits in the fractional part.

NOTE – This has been optimized to provide a 26-bit encoding.

32.3.18 The **TIME-OF-DAY-AND-DIFF-AND-FRACTION-ENCODING** type is:

```
TIME-OF-DAY-AND-DIFF-AND-FRACTION-ENCODING ::= SEQUENCE {
        local-time SEQUENCE {
          hours    INTEGER (0..24),
          minutes  INTEGER (0..59),
          secondsINTEGER (0..60),
          fraction INTEGER (0..999, ..., 1000..MAX)},
        time-difference       TIME-DIFFERENCE }
```

with the **local-time** set to the hours, minutes, seconds and fractional part components of the local time of the abstract value and the **time-difference** set to the sign, hours and minutes of the time-difference component of the abstract value as specified in 32.3.3.

32.4 Encoding subtypes with the "Basic=Date-Time" property setting

This subclause defines the ASN.1 type referenced in Table 2, column 3 for types where all the abstract values of the type have the **"Basic=Date-Time"** property setting.

32.4.1 The **DATE-TIME-ENCODING** type is:

```
DATE-TIME-ENCODING {Date-Type, Time-Type} ::= SEQUENCE {
        date        Date-Type,
        time        Time-Type}
```

32.4.2 The encoding shall be the encoding of an instantiation of this type with the **Date-Type** and **Time-Type** actual parameters set to the types specified in Table 2 column 3 of the **"Basic=Date"** and **"Basic=Time"** rows (respectively) that specify the additional property settings of all the abstract values of the type.

NOTE – This has been optimized to provide a 32-bit encoding in common cases.

32.5 Encoding subtypes with the "Basic=Interval Interval-type=SE" property setting

This subclause defines the ASN.1 types referenced in Table 2, column 3 for types where all the abstract values of the type have the **"Basic=Interval Interval-type=SE"** property setting.

32.5.1 The **START-END-DATE-INTERVAL-ENCODING** type is:

```
START-END-DATE-INTERVAL-ENCODING {Date-Type} ::= SEQUENCE {
        start       Date-Type,
        end         Date-Type}
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** actual parameter set to the type specified in Table 2 column 3 of the **"Basic=Date"** row that specifies the additional property settings of all the abstract values of the type. The **start** component shall be set to the start date and the **end** component shall be set to the end date of the interval.

32.5.2 The **START-END-TIME-INTERVAL-ENCODING** type is:

```
START-END-TIME-INTERVAL-ENCODING {Time-Type} ::= SEQUENCE {
        start       Time-Type,
        end         Time-Type}
```

and the encoding shall be the encoding of an instantiation of this type with the **Time-Type** actual parameter set to the type specified in Table 2 column 3 of the **"Basic=Time"** row that specifies the additional property settings of all the abstract values of the type. The **start** component shall be set to the start time and the **end** component shall be set to the end time of the interval.

32.5.3 The **START-END-DATE-TIME-INTERVAL-ENCODING** type is:

```
START-END-DATE-TIME-INTERVAL-ENCODING {Date-Type, Time-Type} ::=
        SEQUENCE {
          start    DATE-TIME-ENCODING {Date-Type, Time-Type},
          end      DATE-TIME-ENCODING {Date-Type, Time-Type}}
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** and **Time-Type** actual parameters set to the types specified in Table 2 column 3 of the **"Basic=Date"** and **"Basic=Time"** rows (respectively) that specify the additional property settings of all the abstract values of the type. The **start** component shall be set (as specified in 32.4) to the start date-time and the **end** component shall be set to the end date-time of the interval.

32.6 Encoding subtypes with the "Basic=Interval Interval-type=D" property setting

This subclause defines the ASN.1 type referenced in Table 2, column 3 for types where all the abstract values of the type have the **"Basic=Interval Interval-type=D"** property setting.

32.6.1 The **DURATION-INTERVAL-ENCODING** type is:

```
DURATION-INTERVAL-ENCODING ::= SEQUENCE { -- 8 bits for optionality
        years           INTEGER (0..31, ..., 32..MAX) OPTIONAL,
                        -- 5 bits for up to 31 years
        months          INTEGER (0..15, ..., 16..MAX) OPTIONAL,
                        -- 4 bits for up to 15 months
        weeks           INTEGER (0..63, ..., 64..MAX) OPTIONAL,
                        -- 6 bits for up to 63 weeks
        days            INTEGER (0..31, ..., 32..MAX) OPTIONAL,
                        -- 5 bits for up to 31 days
        hours           INTEGER (0..31, ..., 32..MAX) OPTIONAL,
                            -- 5 bits for up to 31 hours
        minutes         INTEGER (0..63, ..., 64..MAX) OPTIONAL,
                            -- 6 bits for up to 63 minutes
        seconds         INTEGER (0..63, ..., 64..MAX) OPTIONAL,
                        -- 6 bits for up to 63 seconds
        fractional-part     SEQUENCE {
                    number-of-digits    INTEGER(1..3, ..., 4..MAX),
                        -- 3 bits for up to three digits accuracy
                    fractional-value    INTEGER(0..999, ..., 1000..MAX)
                        -- 11 bits for up to three digits accuracy
                                } OPTIONAL }
```

32.6.2 The **weeks** component shall be present if, and only if, the **years**, **months**, **days**, **hours**, **minutes**, and **seconds** components are all absent.

NOTE – This reflects restrictions that are present for the use of time elements in the definition of the **DURATION** abstract value.

32.6.3 If a time element component of the abstract value is zero, and does not have a fractional part, then the corresponding component of **DURATION-INTERVAL-ENCODING** shall be absent unless this time element is the least significant time element in the abstract value. If a time element of the abstract value has the value zero, and is the least significant time element in the abstract value, or has a fractional part, then the corresponding component shall be present in **DURATION-INTERVAL-ENCODING** with the value zero.

NOTE – This ensures that the encoding is canonical.

32.6.4 The **fractional-part** of **DURATION-INTERVAL-ENCODING** shall be absent if there is no fractional part of any time element, otherwise it shall be set to the fractional part (of the least significant time element) as specified in 32.6.5.

32.6.5 The number of digits in the fractional part shall be placed in **number-of-digits**. If the number of digits is N, then the value of the fractional part shall be multiplied by ten-to-the-power-N and the resulting integer value placed in **fractional-value**.

NOTE 1 – Decoders can recover the original fractional part from these encodings, including any trailing zeros.

NOTE 2 – This encoding has been optimized for the cases where there are only a few non-zero time elements in the abstract value, and where the values of the time elements are small. Encodings of less than 16 bits occur in simple cases.

32.7 Encoding subtypes with the "Basic=Interval Interval-type=SD" or "Basic=Interval Interval-type=DE" property setting

This subclause defines the ASN.1 types referenced in Table 2, column 3 for types where all the abstract values of the type have the "Basic=Interval Interval-type=SD" or "Basic=Interval Interval-type=DE" property setting.

32.7.1 The START-DATE-DURATION-INTERVAL-ENCODING type is:

```
START-DATE-DURATION-INTERVAL-ENCODING {Date-Type} ::= SEQUENCE {
        start       Date-Type,
        duration    DURATION-INTERVAL-ENCODING}
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** actual parameter set to the type specified in Table 2 column 3 of the "Basic=Date" row that specifies the additional property settings of all the abstract values of the type. The **start** component shall be set to the start date and the **duration** component shall be set (as specified in 32.6) to the duration of the interval.

32.7.2 The START-TIME-DURATION-INTERVAL-ENCODING type is:

```
START-TIME-DURATION-INTERVAL-ENCODING {Time-Type} ::= SEQUENCE {
        start       Time-Type,
        duration    DURATION-INTERVAL-ENCODING }
```

and the encoding shall be the encoding of an instantiation of this type with the **Time-Type** actual parameter set to the type specified in Table 2 column 3 of the "Basic=Time" row that specifies the additional property settings of all the abstract values of the type. The **start** component shall be set to the start time and the **duration** component shall be set (as specified in 32.6) to the duration of the interval.

32.7.3 The START-DATE-TIME-DURATION-INTERVAL-ENCODING type is:

```
START-DATE-TIME-DURATION-INTERVAL-ENCODING {Date-Type, Time-Type} ::=
        SEQUENCE {
          start     DATE-TIME-ENCODING {Date-Type, Time-Type},
          durationDURATION-INTERVAL-ENCODING }
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** and **Time-Type** actual parameters set to the types specified in Table 2 column 3 of the "Basic=Date" and "Basic=Time" rows (respectively) that specify the additional property settings of all the abstract values of the type. The **start** component shall be set (as specified in 32.4) to the start date-time and the **duration** component shall be set (as specified in 32.6) to the duration of the interval.

32.7.4 The DURATION-END-DATE-INTERVAL-ENCODING type is:

```
DURATION-END-DATE-INTERVAL-ENCODING {Date-Type} ::= SEQUENCE {
        duration    DURATION-INTERVAL-ENCODING,
        end             Date-Type }
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** actual parameter set to the type specified in Table 2 column 3 of the "Basic=Date" row that specifies the additional property settings of all the abstract values of the type. The **duration** component shall be set (as specified in 32.6) to the duration of the interval and the **end** component shall be set to the end date.

32.7.5 The DURATION-END-TIME-INTERVAL-ENCODING type is:

```
DURATION-END-TIME-INTERVAL-ENCODING {Time-Type} ::= SEQUENCE {
        duration    DURATION-INTERVAL-ENCODING,
        end         Time-Type }
```

and the encoding shall be the encoding of an instantiation of this type with the **Time-Type** actual parameter set to the type specified in Table 2 column 3 of the "Basic=Time" row that specifies the additional property settings of all the abstract values of the type. The **duration** component shall be set (as specified in 32.6) to the duration of the interval and the **end** component shall be set to the end time.

32.7.6 The **DURATION-END-DATE-TIME-INTERVAL-ENCODING** type is:

```
DURATION-END-DATE-TIME-INTERVAL-ENCODING {Date-Type, Time-Type} ::= SEQUENCE {
        duration  DURATION-INTERVAL-ENCODING,
        end       DATE-TIME-ENCODING {Date-Type, Time-Type}}
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** and **Time-Type** actual parameters set to the types specified in Table 2 column 3 of the **"Basic=Date"** and **"Basic=Time"** rows (respectively) that specify the additional property settings of all the abstract values of the type. The **duration** component shall be set (as specified in 32.6) to the duration of the interval and the **end** component shall be set (as specified in 32.4) to the end date-time.

32.8 Encoding subtypes with the "Basic=Rec-Interval Interval-type=SE" property setting

This subclause defines the ASN.1 types referenced in Table 2, column 3 for types where all the abstract values of the type have the **"Basic=Rec-Interval Interval-type=SE"** property setting.

32.8.1 The **REC-START-END-DATE-INTERVAL-ENCODING** type is:

```
REC-START-END-DATE-INTERVAL-ENCODING {Date-Type} ::= SEQUENCE {
        recurrence      INTEGER OPTIONAL,
        start       Date-Type,
        end         Date-Type}
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** actual parameter set to the type specified in Table 2 column 3 of the **"Basic=Date"** row that specifies the additional property settings of all the abstract values of the type. The **recurrence** component shall be absent for an unlimited number of recurrences in the abstract value, and shall otherwise be set to the number of recurrences. The **start** component shall be set to the start date and the **end** component shall be set to the end date of the interval.

32.8.2 The **REC-START-END-TIME-INTERVAL-ENCODING** type is:

```
REC-START-END-TIME-INTERVAL-ENCODING {Time-Type} ::= SEQUENCE {
        recurrence      INTEGER OPTIONAL,
        start       Time-Type,
        end         Time-Type}
```

and the encoding shall be the encoding of an instantiation of this type with the **Time-Type** actual parameter set to the type specified in Table 2 column 3 of the **"Basic=Time"** row that specifies the additional property settings of all the abstract values of the type. The **recurrence** component shall be absent for an unlimited number of recurrences in the abstract value, and shall otherwise be set to the number of recurrences. The **start** component shall be set to the start time and the **end** component shall be set to the end time of the interval.

32.8.3 The **REC-START-END-DATE-TIME-INTERVAL-ENCODING** type is:

```
REC-START-END-DATE-TIME-INTERVAL-ENCODING {Date-Type, Time-Type} ::=
    SEQUENCE {
        recurrence      INTEGER OPTIONAL,
        start       DATE-TIME-ENCODING {Date-Type, Time-Type},
        end         DATE-TIME-ENCODING {Date-Type, Time-Type}}
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** and **Time-Type** actual parameters set to the types specified in Table 2 column 3 of the **"Basic=Date"** and **"Basic=Time"** rows (respectively) that specify the additional property settings of all the abstract values of the type. The **recurrence** component shall be absent for an unlimited number of recurrences in the abstract value, and shall otherwise be set to the number of recurrences. The **start** component shall be set (as specified in 32.4) to the start date-time and the **end** component shall be set to the end date-time of the recurring interval.

32.9 Encoding subtypes with the "Basic=Rec-Interval Interval-type=D" property setting

This subclause defines the ASN.1 type referenced in Table 2, column 3 for types where all the abstract values of the type have the **"Basic=Rec-Interval Interval-type=D"** property setting.

32.9.1 The **REC-DURATION-INTERVAL-ENCODING** type is:

```
REC-DURATION-INTERVAL-ENCODING ::= SEQUENCE {
        recurrence      INTEGER OPTIONAL,
        duration    DURATION-INTERVAL-ENCODING}
```

32.9.2 The **recurrence** component shall be absent for an unlimited number of recurrences in the abstract value, and shall otherwise be set to the number of recurrences. The **duration** component shall be set (as specified in 32.6) to the duration of the recurring interval.

32.10 Encoding subtypes with the "Basic=Rec-Interval Interval-type=SD" or "Basic=Rec-Interval Interval-type=DE" property setting

This subclause defines the ASN.1 types referenced in Table 2, column 3 for types where all the abstract values of the type have the "Basic=Rec-Interval Interval-type=SD" or "Basic=Rec-Interval Interval-type=DE" property setting.

32.10.1 The **REC-START-DATE-DURATION-INTERVAL-ENCODING** type is:

```
REC-START-DATE-DURATION-INTERVAL-ENCODING {Date-Type} ::= SEQUENCE {
        recurrence      INTEGER OPTIONAL,
        start           Date-Type,
        duration        DURATION-INTERVAL-ENCODING}
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** actual parameter set to the type specified in Table 2 column 3 of the "Basic=Date" row that specifies the additional property settings of all the abstract values of the type. The **recurrence** component shall be absent for an unlimited number of recurrences in the abstract value, and shall otherwise be set to the number of recurrences. The **start** component shall be set to the start date and the **duration** component shall be set (as specified in 32.6) to the duration of the interval.

32.10.2 The **REC-START-TIME-DURATION-INTERVAL-ENCODING** type is:

```
REC-START-TIME-DURATION-INTERVAL-ENCODING {Time-Type} ::= SEQUENCE {
        recurrence      INTEGER OPTIONAL,
        start           Time-Type,
        duration        DURATION-INTERVAL-ENCODING }
```

and the encoding shall be the encoding of an instantiation of this type with the **Time-Type** actual parameter set to the type specified in Table 2 column 3 of the "Basic=Time" row that specifies the additional property settings of all the abstract values of the type. The **recurrence** component shall be absent for an unlimited number of recurrences in the abstract value, and shall otherwise be set to the number of recurrences. The **start** component shall be set to the start time and the **duration** component shall be set (as specified in 32.6) to the duration of the interval.

32.10.3 The **REC-START-DATE-TIME-DURATION-INTERVAL-ENCODING** type is:

```
REC-START-DATE-TIME-DURATION-INTERVAL-ENCODING {Date-Type, Time-Type} ::=
    SEQUENCE {
        recurrence          INTEGER OPTIONAL,
        start               DATE-TIME-ENCODING {Date-Type, Time-Type},
        duration            DURATION-INTERVAL-ENCODING }
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** and **Time-Type** actual parameters set to the types specified in Table 2 column 3 of the "Basic=Date" and "Basic=Time" rows (respectively) that specify the additional property settings of all the abstract values of the type. The **recurrence** component shall be absent for an unlimited number of recurrences in the abstract value, and shall otherwise be set to the number of recurrences. The **start** component shall be set (as specified in 32.4) to the start date-time and the **duration** component shall be set (as specified in 32.6) to the duration of the recurring interval.

32.10.4 The **REC-DURATION-END-DATE-INTERVAL-ENCODING** type is:

```
REC-DURATION-END-DATE-INTERVAL-ENCODING {Date-Type} ::= SEQUENCE {
        recurrence      INTEGER OPTIONAL,
        duration        DURATION-INTERVAL-ENCODING,
        end             Date-Type }
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** actual parameter set to the type specified in Table 2 column 3 of the "Basic=Date" row that specifies the additional property settings of all the abstract values of the type. The **recurrence** component shall be absent for an unlimited number of recurrences in the abstract value, and shall otherwise be set to the number of recurrences. The **duration** component shall be set (as specified in 32.6) to the duration of the interval and the **end** component shall be set to the end date.

32.10.5 The **REC-DURATION-END-TIME-INTERVAL-ENCODING** type is:

```
REC-DURATION-END-TIME-INTERVAL-ENCODING {Time-Type} ::= SEQUENCE {
        recurrence      INTEGER OPTIONAL,
        duration  DURATION-INTERVAL-ENCODING,
        end       Time-Type }
```

and the encoding shall be the encoding of an instantiation of this type with the **Time-Type** actual parameter set to the type specified in Table 2 column 3 of the **"Basic=Time"** row that specifies the additional property settings of all the abstract values of the type. The **recurrence** component shall be absent for an unlimited number of recurrences in the abstract value, and shall otherwise be set to the number of recurrences. The **duration** component shall be set (as specified in 32.6) to the duration of the interval and the **end** component shall be set to the end time.

32.10.6 The **REC-DURATION-END-DATE-TIME-INTERVAL-ENCODING** type is:

```
REC-DURATION-END-DATE-TIME-INTERVAL-ENCODING {Date-Type, Time-Type} ::= SEQUENCE {
      recurrence      INTEGER OPTIONAL,
      duration        DURATION-INTERVAL-ENCODING,
      end  DATE-TIME-ENCODING {Date-Type, Time-Type}}
```

and the encoding shall be the encoding of an instantiation of this type with the **Date-Type** and **Time-Type** actual parameters set to the types specified in Table 2 column 3 of the **"Basic=Date"** and **"Basic=Time"** rows (respectively) that specify the additional property settings of all the abstract values of the type. The **recurrence** component shall be absent for an unlimited number of recurrences in the abstract value, and shall otherwise be set to the number of recurrences. The **duration** component shall be set (as specified in 32.6) to the duration of the interval and the **end** component shall be set (as specified in 32.4) to the end date-time.

32.11 Encoding subtypes with mixed settings of the Basic property

This subclause specifies the encoding for the **TIME** type and subsets of that type whose abstract values do not all have the same setting of the **Basic** property or for which there is no applicable row in Table 2 (for example, because of the use of multiple accuracies – see 32.1.3). It defines and uses the types **DATE-TYPE**, **TIME-TYPE**, and **MIXED-ENCODING** (see 32.11.5 to 32.11.7). These types are defined using the ASN.1 types defined in earlier subclauses.

32.11.1 For all abstract values of the **TIME** type, there is exactly one row of Table 2 for which the property settings specified in column 2 match the property settings of the abstract value, for all of those property settings that are listed in column 2. (The abstract value may have additional property settings not listed in column 2.) This is called the main determining row.

32.11.2 If the main determining row is row 33, 34, 36, 38, 40, 41, 43, 44, 46, 48, 50, 51, or 53, there is a requirement that the additional properties match those specified in one of rows 1 to 14. The applicable row 1 to 14 is called the date determining row.

32.11.3 If the main determining row is row 33, 35, 36, 39, 40, 42, 43 45, 46, 49, 50, 52 or 53, there is a requirement that the additional properties match those specified in one of rows 15 to 32. The applicable row 15 to 32 is called the time determining row.

32.11.4 In the **DATE-TYPE**, **TIME-TYPE** and **MIXED-ENCODING** type, the **row-*n*** alternative shall be selected if the date determining row, the time determining row, or the main determining row (respectively) is row *n*.

32.11.5 The encoding of the abstract value shall be the encoding of the **MIXED-ENCODING** type:

```
MIXED-ENCODING ::= CHOICE {
        row-1     CENTURY-ENCODING,
        row-2     ANY-CENTURY-ENCODING,
        row-3     YEAR-ENCODING,
        row-4     ANY-YEAR-ENCODING,
        row-5     YEAR-MONTH-ENCODING,
        row-6     ANY-YEAR-MONTH-ENCODING,
        row-7     DATE-ENCODING,
        row-8     ANY-DATE-ENCODING,
        row-9     YEAR-DAY-ENCODING,
        row-10    ANY-YEAR-DAY-ENCODING,
        row-11    YEAR-WEEK-ENCODING,
        row-12    ANY-YEAR-WEEK-ENCODING,
        row-13    YEAR-WEEK-DAY-ENCODING,
        row-14    ANY-YEAR-WEEK-DAY-ENCODING,
        row-15    HOURS-ENCODING,
        row-16    HOURS-UTC-ENCODING,
```

row-17 HOURS-AND-DIFF-ENCODING,
row-18 MINUTES-ENCODING,
row-19 MINUTES-UTC-ENCODING,
row-20 MINUTES-AND-DIFF-ENCODING,
row-21 TIME-OF-DAY-ENCODING,
row-22 TIME-OF-DAY-UTC-ENCODING,
row-23 TIME-OF-DAY-AND-DIFF-ENCODING,
row-24 FRACTIONAL-TIME{HOURS-AND-FRACTION-ENCODING},
row-25 FRACTIONAL-TIME{HOURS-UTC-AND-FRACTION-ENCODING},
row-26 FRACTIONAL-TIME{HOURS-AND-DIFF-AND-FRACTION-ENCODING},
row-27 FRACTIONAL-TIME{MINUTES-AND-FRACTION-ENCODING},
row-28 FRACTIONAL-TIME{MINUTES-UTC-AND-FRACTION-ENCODING},
row-29 FRACTIONAL-TIME{MINUTES-AND-DIFF-AND-FRACTION-ENCODING},
row-30 FRACTIONAL-TIME{TIME-OF-DAY-AND-FRACTION-ENCODING},
row-31 FRACTIONAL-TIME{TIME-OF-DAY-UTC-AND-FRACTION-ENCODING},
row-32 FRACTIONAL-TIME{TIME-OF-DAY-AND-DIFF-AND-FRACTION-ENCODING},
row-33 DATE-TIME-ENCODING {DATE-TYPE, TIME-TYPE},
row-34 START-END-DATE-INTERVAL-ENCODING {DATE-TYPE},
row-35 START-END-TIME-INTERVAL-ENCODING {TIME-TYPE},
row-36 START-END-DATE-TIME-INTERVAL-ENCODING {DATE-TYPE, TIME-TYPE},
row-37 DURATION-INTERVAL-ENCODING,
row-38 START-DATE-DURATION-INTERVAL-ENCODING {DATE-TYPE},
row-39 START-TIME-DURATION-INTERVAL-ENCODING {TIME-TYPE},
row-40 START-DATE-TIME-DURATION-INTERVAL-ENCODING {DATE-TYPE, TIME-TYPE},
row-41 DURATION-END-DATE-INTERVAL-ENCODING {DATE-TYPE},
row-42 DURATION-END-TIME-INTERVAL-ENCODING {TIME-TYPE},
row-43 DURATION-END-DATE-TIME-INTERVAL-ENCODING {DATE-TYPE, TIME-TYPE},
row-44 REC-START-END-DATE-INTERVAL-ENCODING {DATE-TYPE},
row-45 REC-START-END-TIME-INTERVAL-ENCODING {TIME-TYPE},
row-46 REC-START-END-DATE-TIME-INTERVAL-ENCODING {DATE-TYPE, TIME-TYPE},
row-47 REC-DURATION-INTERVAL-ENCODING,
row-48 REC-START-DATE-DURATION-INTERVAL-ENCODING {DATE-TYPE},
row-49 REC-START-TIME-DURATION-INTERVAL-ENCODING {TIME-TYPE},
row-50 REC-START-DATE-TIME-DURATION-INTERVAL-ENCODING {DATE-TYPE, TIME-TYPE},
row-51 REC-DURATION-END-DATE-INTERVAL-ENCODING {DATE-TYPE},
row-52 REC-DURATION-END-TIME-INTERVAL-ENCODING {TIME-TYPE},
row-53 REC-DURATION-END-DATE-TIME-INTERVAL-ENCODING {DATE-TYPE, TIME-TYPE} }

where the encoding of the type of each alternative shall be as specified in the subclause identified in Table 2, column 3 of the main determining row.

32.11.6 **`FRACTIONAL-TIME`** is defined as follows:

```
FRACTIONAL-TIME{Time-Type} ::= SEQUENCE {
number-of-digits INTEGER (1..MAX),
time-value Time-Type}
```

The **`number-of-digits`** encodes the number of digits in the fractional part of the abstract value.

32.11.7 The **`DATE-TYPE`** is:

```
DATE-TYPE ::= CHOICE {
        row-1   CENTURY-ENCODING,
        row-2   ANY-CENTURY-ENCODING,
        row-3   YEAR-ENCODING,
        row-4   ANY-YEAR-ENCODING,
        row-5   YEAR-MONTH-ENCODING,
        row-6   ANY-YEAR-MONTH-ENCODING,
        row-7   DATE-ENCODING,
        row-8   ANY-DATE-ENCODING,
        row-9   YEAR-DAY-ENCODING,
        row-10  ANY-YEAR-DAY-ENCODING,
        row-11  YEAR-WEEK-ENCODING,
        row-12  ANY-YEAR-WEEK-ENCODING,
        row-13  YEAR-WEEK-DAY-ENCODING,
        row-14  ANY-YEAR-WEEK-DAY-ENCODING }
```

where the encoding of the type of each alternative shall be as specified in the subclause identified in Table 2, column 3 of the date determining row.

32.11.8 The **TIME-TYPE** is:

```
TIME-TYPE ::= SEQUENCE {
        number-of-digits    INTEGER (1..MAX) OPTIONAL,
        time-type CHOICE {
            row-15          HOURS-ENCODING,
            row-16          HOURS-UTC-ENCODING,
            row-17          HOURS-AND-DIFF-ENCODING,
            row-18          MINUTES-ENCODING,
            row-19          MINUTES-UTC-ENCODING,
            row-20          MINUTES-AND-DIFF-ENCODING,
            row-21          TIME-OF-DAY-ENCODING,
            row-22          TIME-OF-DAY-UTC-ENCODING,
            row-23          TIME-OF-DAY-AND-DIFF-ENCODING,
            row-24          HOURS-AND-FRACTION-ENCODING,
            row-25          HOURS-UTC-AND-FRACTION-ENCODING,
            row-26          HOURS-AND-DIFF-AND-FRACTION-ENCODING,
            row-27          MINUTES-AND-FRACTION-ENCODING,
            row-28          MINUTES-UTC-AND-FRACTION-ENCODING,
            row-29          MINUTES-AND-DIFF-AND-FRACTION-ENCODING,
            row-30          TIME-OF-DAY-AND-FRACTION-ENCODING,
            row-31          TIME-OF-DAY-UTC-AND-FRACTION-ENCODING,
            row-32          TIME-OF-DAY-AND-DIFF-AND-FRACTION-ENCODING } }
```

where the encoding of the type of each alternative shall be as specified in the subclause identified in Table 2, column 3 of the time determining row.

32.11.9 The **number-of-digits** shall be present in the **TIME-TYPE** if and only if the **time-type** alternative is one of **row-24** to **row-32**. It shall encode the number of digits in the fractional part of the abstract value.

33 Object identifiers for transfer syntaxes

33.1 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.

33.2 The following object identifier, OID internationalized resource identifier (with assignment of Unicode labels) 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)}
"/ASN.1/Packed-Encoding/Basic/Aligned"
"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)}
"/ASN.1/Packed-Encoding/Basic/Unaligned"
"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)}
"/ASN.1/Packed-Encoding/Canonical/Aligned"
"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)}
"/ASN.1/Packed-Encoding/Canonical/Unaligned"
"Packed encoding of a single ASN.1 type (canonical unaligned)"

33.3 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 33.2 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.

33.4 The names specified in 33.2 shall not be used with an abstract syntax name to identify a transfer syntax if the conditions of 33.3 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 Rec. ITU-T X.680 | ISO/IEC 8824-1 for defining types. This is identical to the example defined in Rec. ITU-T 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                                                                    Length of name.givenName = 4
01001010 01101111 01101000 01101110                                         name.givenName = "John"

00000001                                                                    Length of name.initial = 1
01010000                                                                    name.initial = "P"

00000101                                                                    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"

00001000                                                                    Length of dateOfHire = 8
00110001 00111001 00110111 00110001 00110000 00111001 00110001 00110111     dateOfHire = "19710917"

00000100                                                                    Length of nameOfSpouse.givenName = 4
01001101 01100001 01110010 01111001                                         nameOfSpouse.givenName = "Mary"

00000001                                                                    Length of nameOfSpouse.initial = 1
01010100                                                                    nameOfSpouse.initial = "T"

00000101                                                                    Length of nameOfSpouse.familyName = 5
01010011 01101101 01101001 01110100 01101000                                nameOfSpouse.familyName = "Smith"

00000010                                                                    Number of children

00000101                                                                    Length of children[0].givenName = 5
01010010 01100001 01101100 01110000 01101000                                children[0].givenName = "Ralph"

00000001                                                                    Length of children[0].initial = 1
01010100                                                                    children[0].initial = "T"

00000101                                                                    Length of children[0].familyName = 5
01010011 01101101 01101001 01110100 01101000                                children[0].familyName = "Smith"

00001000                                                                    Length of children[0].dateOfBirth = 8
00110001 00111001 00110101 00110111 00110001 00110001 00110001 00110001     children[0].dateOfBirth = "19571111"

00000101                                                                    Length of children[1].givenName = 5
01010011 01110101 01110011 01100001 01101110                                children[1].givenName = "Susan"

00000001                                                                    Length of children[1].initial = 1
01000010                                                                    children[1].initial = "B"

00000101                                                                    Length of children[1].familyName = 5
01001010 01101111 01101110 01100101 01110011                                children[1].familyName = "Jones"

00001000                                                                    Length of children[1].dateOfBirth = 8
00110001 00111001 00110101 00111001 00110000 00110111 00110001 00110111     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 Rec. ITU-T 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 01001010 01101111 01101000 01101110	Length of name.givenName = 4 name.givenName = "John"
01010000	name.initial = "P"
000100xx 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"
0001 1001 0111 0001 0000 1001 0001 0111	dateOfHire = "19710917"
000011xx 01001101 01100001 01110010 01111001	Length of nameOfSpouse.givenName = 4 nameOfSpouse.givenName = "Mary"
01010100	nameOfSpouse.initial = "T"
000100xx 01010011 01101101 01101001 01110100 01101000	Length of nameOfSpouse.familyName = 5 nameOfSpouse.familyName = "Smith"
00000010	Number of children
000100xx 01010010 01100001 01101100 01110000 01101000	Length of children[0].givenName = 5 children[0].givenName = "Ralph"
01010100	children[0].initial = "T"
000100xx 01010011 01101101 01101001 01110100 01101000	Length of children[0].familyName = 5 children[0].familyName = "Smith"
0001 1001 0101 0111 0001 0001 0001 0001	children[0].dateOfBirth = "19571111"
000100xx 01010011 01110101 01110011 01100001 01101110	Length of children[1].givenName = 5 children[1].givenName = "Susan"
01000010	children[1].initial = "B"
000100xx 01001010 01101111 01101110 01100101 01110011	Length of children[1].familyName = 5 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 0.01011 101.010 10001.1 101001	Length of name.givenName = 4 name.givenName = "John"
0.10001	name.initial = "P"
000.100 01010.0 101000 1.00100 101.111 10001.1	Length of name.familyName = 5 name.familyName = "Smith"
0000000.1 0011001.1	Length of (employee) number = 1 (employee) number = 51
0000100.0 1000100 .1101001 1.110010 11.00101 110.0011 1110.100 11011.11 111001.0	Length of title = 8 title = "Director"
0001 100.1 0111 000.1 0000 100.1 0001 011.1	dateOfHire = "19710917"
000011 0.01110 011.100 10110.1 110100	Length of nameOfSpouse.givenName = 4 nameOfSpouse.givenName = "Mary"
0.10101	nameOfSpouse.initial = "T"
000.100 01010.0 101000 1.00100 101.111 10001.1	Length of nameOfSpouse.familyName = 5 nameOfSpouse.familyName = "Smith"
0000001.0	Number of children
000100 0.10011 011.100 10011.1 101011 1.00011	Length of children[0].givenName = 5 children[0].givenName = "Ralph"
010.101	children[0].initial = "T"
00010.0 010100 1.01000 100.100 10111.1 100011 0.001 1001 0.101 0111 0.001 0001 0.001 0001	Length of children[0].familyName = 5 children[0].familyName = "Smith" children[0].dateOfBirth = "19571111"
0.00100 010.100 11000.0 101110 0.11100 101.001	Length of children[1].givenName = 5 children[1].givenName = "Susan"
00001.1	children[1].initial = "B"
000100 0.01011 101.010 10100.1 100000 1.01110	Length of children[1].familyName = 5 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 Rec. ITU-T 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"
```