

Part 1: Binary MPEG format for XML

Technologies de l'information — Technologies des systèmes MPEG —
Partie 1: Format binaire de MPEG pour XML

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2006

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword.....	iv
Introduction	v
1 Scope	1
2 Normative references	1
3 Terms and definitions.....	2
3.1 Conventions	2
3.2 Definitions	5
4 Symbols and abbreviated terms	11
4.1 Abbreviations	11
4.2 Mathematical operators.....	12
4.3 Mnemonics	14
5 System architecture.....	15
5.1 Terminal architecture	15
5.2 General characteristics of the decoder	15
5.3 Sequence of events during decoder initialisation.....	16
5.4 Decoder behaviour	18
5.5 Issues in encoding documents	19
5.6 Characteristics of the delivery layer	20
5.7 Decoding of Fragment References	21
6 Binary format- BiM	22
6.1 Overview	22
6.2 Binary DecoderInit	22
6.3 Binary Access Unit	31
6.4 Binary Fragment Update Unit	32
6.5 Binary Fragment Update Command	34
6.6 Binary Fragment Update Context.....	36
6.7 Binary Schema Update Unit.....	60
7 Binary Fragment Update Payload	81
7.1 Overview	81
7.2 Definitions	81
7.3 Fragment Update Payload syntax and semantics.....	82
7.4 Element syntax and semantics	84
7.5 Element Content decoding process	96
8 Advanced optimised decoders.....	113
8.1 Overview	113
8.2 Decoder behaviour	114
8.3 Advanced Optimised Decoder Initialization.....	116
8.4 Advanced Optimised Decoder Classification scheme.....	118
8.5 UniformQuantizer advanced optimised decoder.....	118
8.6 NonUniformQuantizer optimized decoder	120
8.7 Zlib advanced optimised decoder.....	122
Annex A (normative) MPEG-7 Specific Simple Type Codecs	125
Annex B (informative) Informative Examples	129
Annex C (informative) Patent Statements	132
Bibliography	133

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. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

ISO/IEC 23001-1 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC 23001 consists of the following parts, under the general title *Information technology — MPEG systems technologies*:

— *Part 1: Binary MPEG format for XML*

Introduction

This International Standard provides a standardized set of generic technologies for encoding XML documents. It addresses a broad spectrum of applications and requirements by providing generic methods for transmitting and compressing XML documents.

Part 1 – Binary Format for XML: specifies the tools for preparing XML documents for efficient transport and storage and for compressing XML documents.

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

The ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured ISO and IEC that he is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with the ISO and IEC. Information may be obtained from the companies listed in Annex C.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified in Annex C. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Document Reference: ISO/IEC 23001-1-2006

Information technology — MPEG systems technologies —

Part 1: Binary MPEG format for XML

1 Scope

This part of ISO/IEC 23001 provides a standardized set of technologies for encoding XML documents. It addresses a broad spectrum of applications and requirements by providing a generic method for transmitting and compressing XML documents.

This part of ISO/IEC 23001 specifies system level functionalities for the communication of XML documents. It provides a specification which will:

- enable the development of ISO/IEC 23001-1 receiving sub-systems, called ISO/IEC 23001-1 Terminal, or Terminal in short, to receive and assemble possibly partitioned and compressed XML documents
- provide rules for the preparation of XML documents for efficient transport and storage.

The decoding process within the ISO/IEC 23001-1 Terminal is normative. The rules mentioned provide guidance for the preparation and encoding of XML documents without leading to a unique encoded representation of such documents.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*

Note: The UTF-8 encoding scheme is described in Annex R of ISO/IEC 10646-1:1993, published as Amendment 2 of ISO/IEC 10646-1:1993.

- XML, *Extensible Markup Language (XML) 1.0*, October 2000.
- XML Schema, *W3C Recommendation*, 2 May 2001.
- XML Schema Part 0: *Primer*, W3C Recommendation, 2 May 2001.
- XML Schema Part 1: *Structures*, W3C Recommendation, 2 May 2001.
- XML Schema Part 2: *Datatypes*, W3C Recommendation 2 May 2001.
- XPath, *XML Path Language*, W3C Recommendation, 16 November 1999.
- Namespaces in XML, W3C Recommendation, 14 January 1999.

Note: These documents are maintained by the W3C (<http://www.w3.org>). The relevant documents can be obtained as follows:

- *Extensible Markup Language (XML) 1.0 (Second Edition)*, 6 October 2000, <http://www.w3.org/TR/2000/REC-xml-20001006>
- *XML Schema: W3C Recommendation*, 2 May 2001, <http://www.w3.org/XML/Schema>
 - *XML Schema Part 0: Primer*, W3C Recommendation, 2 May 2001, <http://www.w3.org/TR/xmlschema-0/>
 - *XML Schema Part 1: Structures*, W3C Recommendation, 2 May 2001, <http://www.w3.org/TR/xmlschema-1/>
 - *XML Schema Part 2: Datatypes*, W3C Recommendation 2 May 2001, <http://www.w3.org/TR/xmlschema-2/>
- *xPath, XML Path Language*, W3C Recommendation, 16 November 1999, <http://www.w3.org/TR/1999/REC-xpath-19991116>
- *Namespaces in XML*, W3C Recommendation, 14 January 1999, <http://www.w3.org/TR/1999/REC-xml-names-19990114>
- *RFC 2396, Uniform Resource Identifiers (URI): Generic Syntax.*
- *RFC 1950, ZLIB Compressed Data Format Specification version 3.3.*
- *IEEE Standard for Binary Floating-Point Arithmetic*, Std 754-1985 Reaffirmed 1990, http://standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html

3 Terms and definitions

3.1 Conventions

3.1.1 Naming convention

In order to specify data types and documents model, this part of ISO/IEC 23001 uses constructs specified in XML Schema, such as “element”, “attribute”, “simpleType” and “complexType”. The names associated with these constructs are created on the basis of the following conventions:

If the name is composed of various words, the first letter of each word is capitalized. The rule for the capitalization of the first word depends on the type of construct and is described below.

- **Element naming:** the first letter of the first word is capitalized (e.g. *TimePoint* element of *TimeType*).
- **Attribute naming:** the first letter of the first word is **not** capitalized (e.g. *timeUnit* attribute of *IncrDurationType*).
- **complexType naming:** the first letter of the first word is capitalized, the suffix “Type” is used at the end of the name.
- **simpleType naming:** the first letter of the first word is not capitalized, the suffix “Type” may be used at the end of the name.

3.1.2 Documentation convention

3.1.2.1 Textual syntax

The syntax of each XML schema item is specified using the constructs specified in XML Schema. It is depicted in this document using a specific font and background, as shown in the example below:

```
<complexType name="ExampleType">
  <sequence>
    <element name="Element1" type="string"/>
  </sequence>
  <attribute name="attribute1" type="string" default="attrvalue1"/>
</complexType>
```

Non-normative XML examples are included in separate subclauses. They are depicted in this document using a separate font and background than the normative syntax specifications, as shown in the example below:

```
<Example attribute1="example attribute value">
  <Element1>example element content</Element1>
</Example>
```

3.1.2.2 Binary syntax

3.1.2.2.1 Overview

The binary document stream retrieved by the decoder is specified in Clause 6, Clause 7, and Clause 8. Each data item in the binary document stream is printed in bold type. It is described by its name, its length in bits, and by a mnemonic for its type and order of transmission. The construct “N+” in the length field indicates that the length of the element is an integer multiple of N.

The action caused by a decoded data element in a bitstream depends on the value of the data element and on data elements that have been previously decoded. The following constructs are used to express the conditions when data elements are present:

while (condition) { data_element ... }	If the condition is true, then the group of data elements occurs next in the data stream. This repeats until the condition is not true.
do { data_element ... } while (condition)	The data element always occurs at least once. The data element is repeated until the condition is not true.
if (condition) { data_element ... } else { data_element ... }	If the condition is true, then the first group of data elements occurs next in the data stream. If the condition is not true, then the second group of data elements occurs next in the data stream.

for (i = m; i < n; i++) { data_element ... }	The group of data elements occurs (n-m) times. Conditional constructs within the group of data elements may depend on the value of the loop control variable i, which is set to m for the first occurrence, incremented by one for the second occurrence, and so forth.
/* comment */	Explanatory comment that may be deleted entirely without in any way altering the syntax.

This syntax uses the 'C-code' convention that a variable or expression evaluating to a non-zero value is equivalent to a condition that is true and a variable or expression evaluating to a zero value is equivalent to a condition that is false.

Use of function-like constructs in syntax tables

In some syntax tables, function-like constructs are used in order to pass the value of a certain syntax element or decoding parameter down to a further syntax table. In that table, the syntax part is then defined like a function in e.g. C program language, specifying in brackets the type and name of the passed syntax element or decoding parameter, and the returned syntax element type, as shown in the following example:

datatype Function(datatype parameter_name) {	Number of bits	Mnemonic
if (parameter_name == ...) {		
OtherFunction(parameter_name)		
} else if		
.....		
} else {		
.....		
}		
Return return_value		
}		

Here, the syntax table describing the syntax part called "Function" receives the parameter "parameter_name" which is of datatype "datatype". The parameter "parameter_name" is used within this syntax part, and it can also be passed further to other syntax parts, in the table above e.g. to the syntax part "OtherFunction".

The parsing of the binary syntax is expressed in procedural terms. However, it should not be assumed that Clause 6, 7 and 8 implement a complete decoding procedure. In particular, the binary syntax parsing in this specification assumes a correct and error-free binary document stream. Handling of erroneous binary document streams is left to individual implementations.

Syntax elements and data elements are depicted in this document using a specific font such as the following example: `FragmentUpdatePayload`.

boolean

In some syntax tables, the "true" and "false" constructs are used. If present in the stream "true" shall be represented with a single bit of value "1" and "false" shall be represented with a single bit of value "0".

3.1.2.2.2 Arrays

Arrays of data elements are represented according to the C-syntax as described below. It should be noted that each index of an array starts with the value "0".

data_element[n] is the n+1th element of an array of data.

data_element[m][n] is the m+1, n+1th element of a two-dimensional array of data.

data_element[l][m][n] is the l+1, m+1, n+1th element of a three-dimensional array of data.

3.1.2.2.3 Functions

3.1.2.2.3.1 nextByteBoundary()

The function “nextByteBoundary()” reads and consumes bits from the binary document stream until but not including the next byte-aligned position in the binary document stream.

3.1.2.2.4 Reserved values and forbidden values

The terms “reserved” and “forbidden” are used in the description of some values of several code and index tables.

The term “reserved” indicates that the value shall not occur in a binary document stream. It may be used in the future for ISO/IEC defined extensions.

The term “forbidden” indicates a value that shall not occur in a binary document stream.

3.1.2.2.5 Reserved bits and stuffing bits

ReservedBits: a binary syntax element whose length is indicated in the syntax table. The value of each bit of this element shall be “1”. These bits may be used in the future for ISO/IEC defined extensions.

Stuffing bits: bits inserted to align the binary document stream, for example to a byte boundary. The value of each of these bits in the binary document stream shall be “1”.

ReservedBitsZero: a binary syntax element whose length is indicated in the syntax table. The value of each bit of this element shall be “0”. These bits may be used in the future for ISO/IEC defined extensions.

3.1.2.3 Textual and binary semantics

The semantics of each schema or binary syntax component, is specified using a table format, where each row contains the name and a definition of that schema or binary syntax component:

<i>Name</i>	<i>Definition</i>
ExampleType	Specifies an ...
element1	Describes the ...
attribute1	Describes the ...

3.2 Definitions

3.2.1

access unit

An entity within an XML document that is atomic in time, i.e., to which a composition time can be attached. An access unit is composed of one or more fragment update units.

3.2.2

additional schema

A schema that can be updated after the start of the decoding process.

3.2.3

advanced optimised decoder

An optimised decoder used to decode a simple type. Advanced optimised decoders parameters and their mappings to types can be modified during binary document stream lifetime.

3.2.4

advanced optimised decoder instance

An advanced optimised decoder initialised and ready to be used for the decoding of some data types.

Note - There can be several instances of the same advanced optimised decoder with different or identical parameters.

3.2.5

advanced optimised decoder instances table

A table of all the advanced optimised decoders available at a certain instant in time.

3.2.6

advanced optimised decoder parameters

The parameters of an advanced optimised decoder.

3.2.7

advanced optimised decoder type

The type, identified by a URI, of an advanced optimised decoder.

3.2.8

application

An abstraction of any entity that makes use of the decoded document stream.

3.2.9

binary access unit

An access unit in binary format as specified in Clause 6 and 7.

3.2.10

binary document stream

A concatenation of binary access units as specified in Clause 6 and 7.

3.2.11

binary format document tree

The internal binary decoder model.

3.2.12

byte-aligned

A bit in a binary document stream is byte-aligned if its position is a multiple of 8-bits from the first bit in the binary document stream.

3.2.13

composition time

The point in time when a specific access unit becomes known to the application.

3.2.14

content particle

A particle is a term in the XML Schema grammar for element content, consisting of an element declaration, a wildcard or a model group, together with occurrence constraints. Refers to XML SCHEMA.

3.2.15**context mode**

Information in the fragment update context specifying how to interpret the subsequent context path information.

3.2.16**context node**

The context node is specified by the context path of the current fragment update context. It is the parent of the operand node.

3.2.17**context path**

Information that identifies and locates the context node and the operand node in the current document tree.

3.2.18**contextual optimised decoder**

"An optimised decoder which behavior is dependent on the current context of the decoding.

Note - For instance, the ZLib optimised decoder (see Clause 8) is a contextual optimised decoder.

Note - Upon certain events, the context must be reset. Upon a certain command or events they are flushed to release their contents. Only contextual optimised decoders are flushable."

3.2.19**contextual optimised decoder reset**

An operation that resets the optimised decoder to put it in a defined initial state. All contextual information is discarded.

3.2.20**current context node**

The starting node for the context path in case of relative addressing.

3.2.21**current document**

The document that is conveyed by the initial document and all access units up to a given composition time.

3.2.22**current document tree**

The XML document tree that represents the current document.

3.2.23**deferred fragment reference**

A fragment reference that can be resolved at any time by the application using the terminal.

3.2.24**deferred node**

A node which is present in the document tree at encoder side and for which the following is true: No part of that node has been sent to the decoder but the existence of that node has been signalled to the decoder.

3.2.25**delivery layer**

An abstraction of any underlying transport or storage functionality.

3.2.26**derived type**

A type defined by the derivation of an other type.

3.2.28**document**

Short term for a structured XML document.

3.2.29

document composer

An entity that reconstitutes the current document tree from the fragment update units.

3.2.30

document fragment

A contiguous part of a document attached at a single node. Using the representation model of a document tree, the document fragment is represented by a sub-tree of the document tree.

3.2.31

document fragment reference

A reference to a document fragment.

Note - For instance, a fragment reference can be a URI which serves to locate the fragment on the world wide web.

3.2.32

document stream

The ordered concatenation of either binary or textual access units conveying a single, possibly time-variant, document.

3.2.33

document tree

A model that is used throughout this specification in order to represent documents. A document tree consists of nodes, which represent elements or attributes of an XML document. Each node may have zero, one or more child nodes. Simple content are considered as child nodes in Clause 6 of the specification.

3.2.34

effective content particle

The particle of a complexType used for the validation process.

3.2.35

fixed optimised decoder

An optimised decoder used to decode either a complex type or a simple type. Fixed optimised decoders are set up at decoder initialisation phase and their mapping to types can't be modified during binary document stream lifetime.

3.2.36

fragment reference

short term for document fragment reference.

3.2.37

fragment reference format

An encoding format of fragment references.

3.2.38

fragment reference marker

A specific information used to describe a deferred fragment reference, which is present within the current document tree. It consists of a fragment reference, the name and type of the top most element of the referenced fragment.

3.2.39

fragment reference resolver

An entity that is capable of resolving the fragment reference provided in the fragment update payload.

3.2.40

fragment update command

A command within a fragment update unit expressing the type of modification to be applied to the part of the current document tree that is identified by the associated fragment update context.

3.2.41**fragment update component extractor**

An entity that de-multiplexes a fragment update unit, resulting in the unit's components: fragment update command, fragment update context, and fragment update payload.

3.2.42**fragment update context**

Information in a fragment update unit that specifies on which node in the current document tree the fragment update command shall be executed. Additionally, the fragment update context specifies the data type of the element encoded in the subsequent fragment update payload.

3.2.43**fragment update decoder parameters**

Configuration parameters conveyed in the DecoderInit (see 6.2) that are required to specify the decoding process of the fragment update decoder.

3.2.44**fragment update payload**

Information in a fragment update unit that conveys the information which is added to the current document or which replaces a part of the current document.

3.2.45**fragment update payload decoder**

The entity that decodes the fragment update payload information of the fragment update.

3.2.46**fragment update unit**

Information in an access unit, conveying a document or a portion thereof. Fragment update units provide the means to modify the current document. They are nominally composed of a fragment update command, a fragment update context and a fragment update payload.

3.2.47**initial document**

A document that initialises the current document tree without conveying it to the application (see 5.3). The initial document is part of the DecoderInit (see 6.2).

3.2.48**initial schema**

The schema that is known by the decoder before the decoding process starts.

3.2.49**initialisation extractor**

An entity that de-multiplexes the DecoderInit (see 6.2), resulting in its components initial document, fragment update decoder parameters and schema URI.

3.2.51**non-deferred fragment reference**

A fragment reference that shall be resolved by the terminal at the composition time of the access unit containing the fragment reference.

3.2.52**operand node**

The node in the binary format document tree that is either added, deleted or replaced according to the current fragment update command and fragment update payload. The operand node is always a child node of the context node.

3.2.53

optimised decoder

A decoder associated to a type and dedicated to certain encoding methods better suited than the generic ones.

3.2.54

optimised decoder mapping

An association between a type and a set of optimised decoders.

3.2.55

schema

A schema is represented in XML by one or more “schema documents”, that is, one or more “<schema>” element information items. A “schema document” contains representations for a collection of schema components, e.g. type definitions and element declarations, which have a common target namespace. A schema document which has one or more “<import>” element information items corresponds to a schema with components with more than one target namespace. Refer also to XML Schema.

3.2.56

schema resolver

An entity that is capable of resolving the schema identification provided in the DecoderInit (see 6.2), and to possibly retrieve the specified schemas.

3.2.57

schema update unit

Information in an access unit, conveying a schema or a portion thereof. Schema update units provide the means to modify the current decoder schema knowledge.

3.2.58

schema URI

A URI that uniquely identifies a schema.

3.2.59

schema valid

A document that is schema valid satisfies the constraints embodied in the Schema to which it should conform.

3.2.60

selector node

The parent node of the topmost node of a document tree. It artificially extends the document tree to allow the addressing of the topmost node.

3.2.61

skippable subtree

A subtree of an XML document that the decoder is permitted not to decode.

3.2.62

super type

The parent of a type in its type hierarchy.

3.2.63

systems layer

An abstraction of the tools and processes specified in this specification.

3.2.64

terminal

The entity that makes use of a coded representation of a document.

3.2.67**topmost node**

The node specified by the first element in the document, instantiating one of the global elements declared in the schema.

3.2.68**type codec**

Synonym to optimised decoder.

3.2.69**type hierarchy**

The hierarchy of type derivations.

3.2.70**validation**

The process of parsing an XML document to determine whether it satisfies the constraints embodied in the Schema to which it should conform.

3.2.71**XML Schema parser**

An application that is capable of validating document schemes (content and structure) and descriptor data types against their schema definition.

4 Symbols and abbreviated terms

4.1 Abbreviations

AU	Access Unit
BiM	Binary format for document streams
D	Descriptor
DL	Delivery Layer
FU	Fragment Update
FUU	Fragment Update Unit
FSAD	Finite State Automaton Decoder
MPC	Multiple element Position Code
SBC	Schema Branch Code
SPC	Single element Position Code
TBC	Tree Branch Code
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF	Universal Character Set Transformation Formats
XML	Extensible Markup Language
XPath	XML Path Language
MSB	Most Significant Bit
SU	Schema Update
SUU	Schema Update Unit

4.2 Mathematical operators

The mathematical operators used to describe this specification are similar to those used in the C programming language. However, integer divisions with truncation and rounding are specifically defined. Numbering and counting loops generally begin from zero.

4.2.1 Arithmetic operators

+ Addition.

- Subtraction (as a binary operator) or negation (as a unary operator).

++ Increment. i.e. $x++$ is equivalent to $x = x + 1$

-- Decrement. i.e. $x--$ is equivalent to $x = x - 1$

***** Multiplication.

^ Power.

sign()
$$\text{sign}(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

abs() $\text{abs}(x) = x \cdot \text{sign}(x)$

log2(..) $\log_2(x) = \log_2(x)$

Ceil (x) denotes the smallest integer larger than or equal to x.

int(..) truncation of the argument to its integer value, e.g. 1.3 is truncated to 1 and -3.7 is truncated to -3.

$\sum_{i=a}^{i=b} f(i)$ the summation of the $f(i)$ with i taking integral values from a up to, but not including b .

4.2.2 Logical operators

|| Logical OR.

&& Logical AND.

! Logical NOT.

4.2.3 Relational operators

> Greater than.

>= Greater than or equal to.

< Less than.

<= Less than or equal to.

== Equal to.

!= Not equal to.

max (, ...,) the maximum value in the argument list.

min (, ... ,) the minimum value in the argument list.

4.2.4 Assignment

= Assignment operator.

4.2.5 Character string comparison

Many phases of the fragment encoding rely on a string comparison method. This method is based on the Unicode value of each character in the strings. The following defines the notion of lexicographic ordering:

Two strings are different if they have different characters at some index that is a valid index for both strings, or if their lengths are different, or both.

If they have different characters at one or more index positions, let k be the smallest such index; then the string whose character at position k has the smaller value, as determined by using the $<$ operator, lexicographically precedes the other string.

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string.

This string comparison is described by each method that is functionally equivalent to the following procedure:

```
compare_strings(string1, string2) {
    len1 = length(string1);
    len2 = length(string2);
    n = min(len1, len2);
    i = 0;
    j = 0;

    while (n-- != 0) {
        c1 = string1[i++];
        c2 = string2[j++];
        if (c1 != c2) {
            return c1 - c2;
        }
    }

    return len1 - len2;
}
```

4.3 Mnemonics

The following mnemonics are defined to describe the different data types used in the coded bitstream.

Name	Definition
bslbf	Bit string, left bit first, where “left” is the order in which bit strings are written in this specification. Bit strings are generally written as a string of 1s and 0s within single quote marks, e.g. '1000 0001'. Blanks within a bit string are for ease of reading and have no significance. For convenience large strings are occasionally written in hexadecimal, in this case conversion to a binary in the conventional manner will yield the value of the bit string. Thus the left most hexadecimal digit is first and in each hexadecimal digit the most significant of the four bits is first.
uimsbf	Unsigned integer, most significant bit first.
vlclbf	Variable length code, left bit first, where “left” refers to the order in which the VLC codes are written. The byte order of multibyte words is most significant byte first.
vluimsbf8	Variable length code unsigned integer, most significant bit first. The size of vluimsbf8 is a multiple of one byte. The first bit (Ext) of each byte specifies if set to 1 that another byte is present for this vluimsbf8 code word. The unsigned integer is encoded by the concatenation of the seven least significant bits of each byte belonging to this vluimsbf8 code word An example for this type is shown in Figure 1.
vluimsbf5	Variable length code unsigned integer, most significant bit first. The first n bits (Ext) which are 1 except of the n-th bit which is 0, indicate that the integer is encoded by n times 4 bits. An example for this type is shown in Figure 2.
vlurmsbf5	Variable length code unsigned rational number, most significant bit first. The first n bits (Ext) which are '1' except of the nth bit which is '0', indicate that the rational number R in the interval $0 \leq R < 1$ is encoded by n times 4 bits. The ith bit of the n times 4 bits representing the rational number corresponds to a value of 2^{-i}. Thus the (n+1)st bit of the vlurmsbf5 code word (which corresponds to the MSB of the rational number) represents a value of $\frac{1}{2}$, the (n+2)nd bit of the vlurmsbf5 code word represents a value of $\frac{1}{4}$, and so forth. An example for this type is shown in Figure 3. Note - Comparing two rational numbers A and B represented by a vlurmsbf5 code word can be done by comparing bit by bit the rational numbers starting from their respective MSBs. Then the rational number A is bigger if there is a '1' bit at a position at which there is a '0' for B. A is also bigger if there is a '1' bit at a position which is not present for B and when A is longer than B.

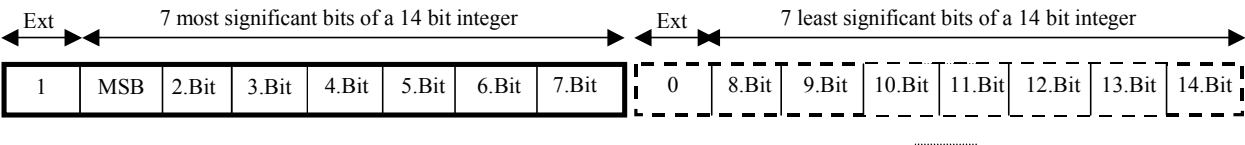


Figure 1 — Informative example for the vluimsbf8 data type

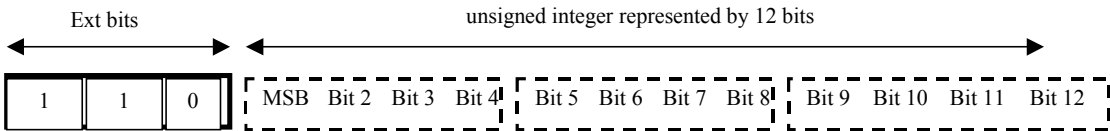


Figure 2 — Informative example for the `vluimbsbf5` data type

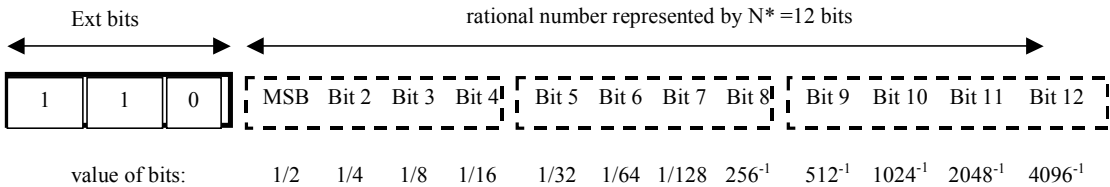


Figure 3 — Informative example for the `vlurmsbf5` data type

5 System architecture

5.1 Terminal architecture

ISO/IEC 23001-1 provides the means to represent coded XML documents. The entity that makes use of such coded representations of XML documents is generically referred to as the “ISO/IEC 23001-1 terminal” or just “terminal” in short. This terminal may correspond to a standalone application or be part of an application system.

This and the following three subclauses provide the description of an ISO/IEC 23001-1 terminal, its components, and their operation. The architecture of such a terminal is depicted in Figure 4. The following subclauses introduce the tools specified in this specification.

In Figure 4, there are three main layers outlined: the application, the normative systems layer, and the delivery layer. ISO/IEC 23001-1 is not concerned with any storage and/or transmission media (whose behaviours and characteristics are abstracted by the delivery layer) or the way the application processes the current document. This specification does make specific assumptions about the delivery layer, and those assumptions are outlined in subclause 5.5.4. The systems layer defines a decoder whose architecture is described here to provide an overview and to establish common terms of reference. A compliant decoder need not implement the constituent parts as visualised in Figure 4, but shall implement the normative decoding process specified in Clauses 6 through 8.

5.2 General characteristics of the decoder

5.2.1 General characteristics of document streams

An ISO/IEC 23001-1 terminal consumes document streams and outputs a – potentially dynamic – representation of the document called the current document tree. Document streams shall consist of a sequence of one or more individually accessible portions of data named access units. An Access Unit (AU) is the smallest data entity to which “terminal-oriented” (as opposed to “described-media oriented”) timing information can be attributed. This timing information is called the “composition” time, meaning the point in time when the resulting current document tree corresponding to a specific access unit becomes known to the application. The timing information shall be carried by the delivery layer (see subclause 5.6). The current document tree shall be schema-valid after processing each access unit.

A document stream consists of binary access units is termed a binary document stream and is processed by a binary decoder (see subclause 5.2.2 and Clauses 6 and 7).

5.2.2 Principles of the binary decoder

Using the ISO/IEC 23001-1 generic method for binary encoding, called BiM, a document (nominally in a textual XML form) can be compressed, partitioned, streamed, and reconstructed at terminal side. The reconstructed XML document will not be byte-equivalent to the original document. Namely, the binary encoding method does not preserve processing instructions, attribute order, comments, or non-significant whitespace. However, the encoding process ensures that XML element order is preserved.

The BiM, in order to gain its compression efficiency, relies on a schema analysis phase. During this phase, internal tables are computed to associate binary code to schema components (XML elements, types and attributes). BiM defines two methods to address schema components.

The first method allows the decoder to resolve a schema, possibly including schema components originating from several namespaces, at initialization phase. This set of schema components form the *initial schema*. In this schema, all type and substitution codes are merged together no matter the namespace they belong to. This results in shorter codes in the binary document stream. The initial schema can't be updated and is considered fixed for the binary document stream lifetime. It contains by default and at minimal the type codes of the xml schema types: anyType, anySimpleType, and all xml schema simple types. In this specification, anySimpleType is considered as a subtype of anyType.

The second method allows the decoder both to resolve a schema at initialization phase (the *initial schema*) and to receive updated schema information called *additional schemas*. Additional schemas differ from the initial schema as the codes of their schema components defined in different namespaces are defined in different code spaces. This results in larger code size but has the required flexibility for late updating. For full flexibility it is also possible to receive exclusively additional schemas and thus to operate without initial schema.

Both initial schemas and additional schemas are part of a unique table in which each entry identifies a specific schema. The first entries identify schemas that are part of the initial schema. The following ones identify additional schemas.

To further improve compression, BiM allows the association of specific codecs to specific data types instead of using the generic mechanisms defined in Clause 7. These encoding schemes can be optimised with the full knowledge of the properties of that data type.

The resulting current document tree may be topologically equivalent to the original document if desired by the encoder, but it may also exhibit dynamic characteristics such that certain parts of the document are present at the decoder only at chosen times, are never present at all, can be acquired on application demand, or appear in a different part of the tree.

Note - The schema update capabilities provided by the BiM framework defined in this specification aims at upgrading BiM decoder. It is not a mean to transmit an XML schema as is. To do so, one should use the W3C schema of schema to encode its schema.

5.3 Sequence of events during decoder initialisation

The decoder set-up is signalled by the initialisation extractor receiving a binary `DecoderInit` (specified in 6.2). The `DecoderInit` shall be received by the systems layer from the delivery layer. The `DecoderInit` will typically be conveyed by a separate delivery channel compared to the document stream, which is also received from the delivery layer. The component parts of the document stream are discussed in subclause 5.4.

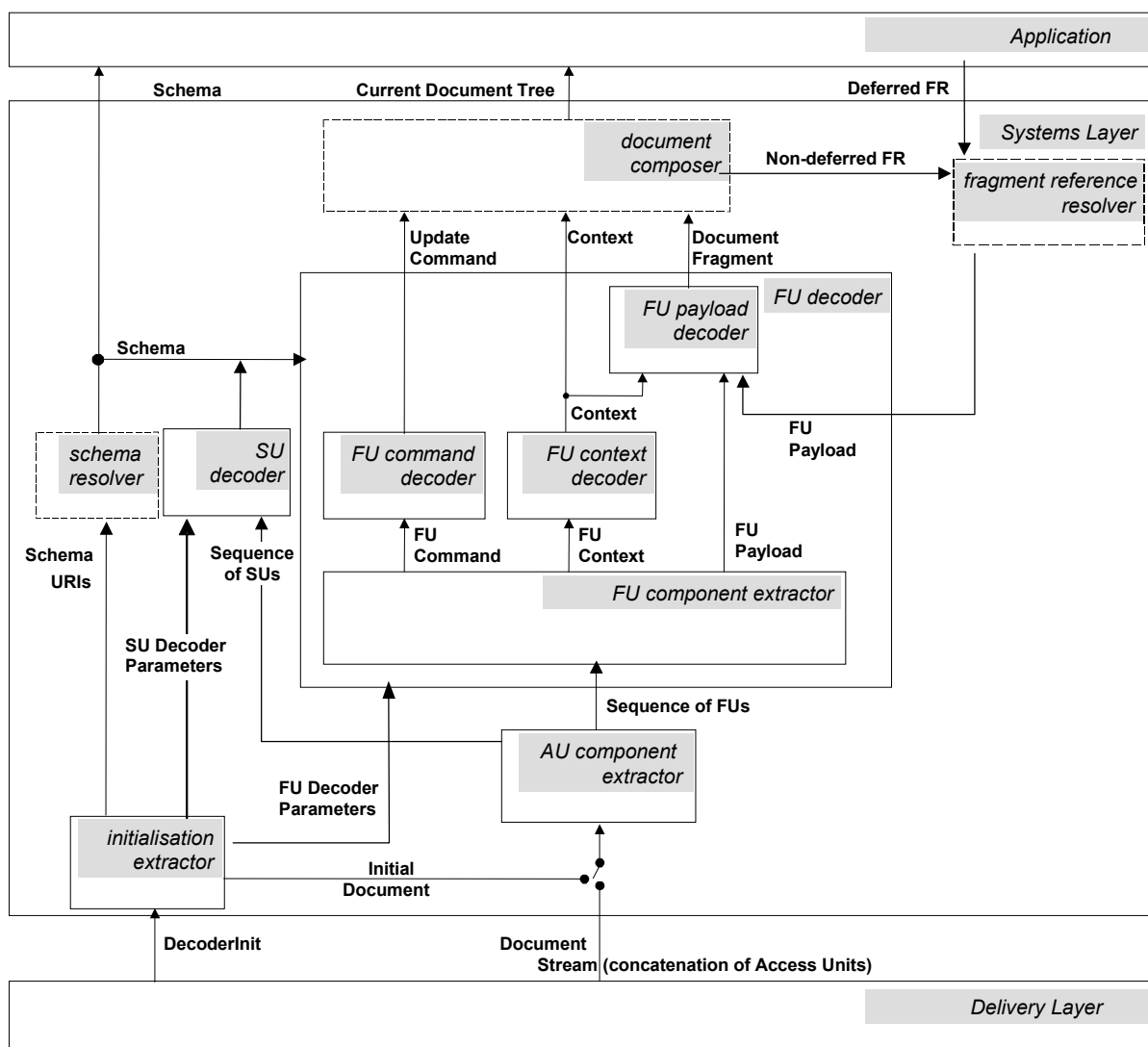


Figure 4 — Terminal Architecture.

Dashed boxes in the systems layer are non-normative. FU is an abbreviation for Fragment Update.

The `DecoderInit` contains a list of URIs that identifies schemas, miscellaneous parameters to configure the decoder (FU Decoder Parameters and SU Decoder Parameters, in Figure 4), and an initial document. There shall be only one `DecoderInit` per document stream. The list of URIs (Schema URIs, in Figure 4) is passed to a schema resolver that associates the URIs with schemas to be passed into the fragment update decoder (see subclause 6.2). The schema resolver is non-normative and may, for example, retrieve schema documents from a network or refer to pre-stored schemas. The resulting schemas are used by the binary decoder specified in Clauses 6 and 7 and by any textual XML Schema parser that may be used for schema validation. If a given Schema URI is unknown to the schema resolver, the corresponding data types in a document stream shall be ignored (i.e., “skipped” and not processed).

The initial document has the same general syntax and semantics as an access unit, but with restrictions, as described in subclauses 6.2. The initial document initialises the current document tree without conveying it to the application. The current document tree is then updated by the access units that comprise the document stream. The initial document may be empty, since a schema-valid current document tree for consumption by the application need only be generated after the first access unit is decoded.

5.4 Decoder behaviour

The document stream shall be processed only after the decoder is initialised. The behaviour of the decoder when access units are received before the decoder is initialised is non-normative. Specifically, there is no requirement to buffer such “early AUs.”

In the case of BiM, an access unit is composed of any number of schema update units followed by any number of fragment update units which are extracted by the access unit component extractor.

A schema update unit carries parts of an additional schema and is composed of

- a namespace identifier,
- a set of code tables to represent global elements, global types and global attributes,
- a binary encoded schema carrying the schema components definitions.

The full schema is not always necessary for the decoding of a particular binary document stream. To avoid unnecessary transmission, a schema update unit may contain only the definitions that are required for the decoding of the bitstream. In this case the code tables can also be sent partially.

Some further constraints are applied to the acquisition of schema update units, notably to ensure that a decoder will not break in case of a missed schema update unit. A specific schema update unit, the so-called first schema update unit, contains initialization information and shall be acquired by the decoder before any use of a received definition. The decoder behavior in case of such missed schema update units is not normative. A transmitted schema definition shall not change during binary document stream lifetime and there shall not be two schema identifiers associated to the same namespace. Finally, all the optimised decoders associated to existing types are immediately applied to all types they derive from in accordance to the rules defined for the optimized decoders in Clause 8.

Once received by the decoder, a schema update unit immediately updates schema information managed by the decoder. It becomes available for the fragment update unit carried in the same access unit as well as future access units.

Fragment update units are extracted in sequence by the fragment update component extractor. Each fragment update unit consists of:

- a fragment update command that specifies the type of update to be executed (i.e., add, replace or delete content or a node, or reset the current document tree);
- a fragment update context that identifies the data type in a given schema document, and points to the location in the current document tree where the fragment update command applies; and
- a fragment update payload conveying either the coded document fragment (extracted out of the original document) to be added or replaced, or a reference to it.

A fragment update extractor splits the fragment update units from the access units and emits the above component parts to the rest of the decoder. The fragment update command decoder generally consists of a simple table lookup for the update command to be passed on to the document composer. The decoded fragment update context information (‘context’ in Figure 4) is passed along to both the document composer and the fragment update payload decoder. The fragment update payload decoder embodies the BiM Payload decoder (Clause 7), which decodes a fragment update payload (aided by context information) to yield a document fragment (see Figure 4).

The corresponding update command and context are processed by the non-normative document composer, which either places the document fragment extracted out of the original document or a reference to it received from the fragment update payload decoder at the appropriate node of the current document tree, or sends a reconstruction event containing this information to the application. If the payload consists of a fragment reference, depending on its nature, the referenced fragment is either immediately acquired (non-deferred

fragment reference) or its acquisition is left to the application (deferred fragment references). In case of a deferred fragment reference, a fragment reference marker is available to the application to help further acquisition. This marker consists of the fragment reference itself, the name and type of the top most element of the referenced fragment. The fragment reference marker is added to the current document tree at the location defined by the fragment update context.

The actual reconstruction of the current document tree by the document composer is implementation-specific, i.e., the application may direct the document composer to prune or ignore unwanted elements as desired. There is no requirement on the format of this current document tree, e.g. it may remain a binary representation.

5.5 Issues in encoding documents

5.5.1 Fragmenting documents

A document stream serves to convey an XML document, as available from a (non-normative) sender or encoder, to the receiving terminal, possibly by incremental transmission in multiple access units. Any number of decompositions of the source document may be possible and it is out of scope of this specification to define such decompositions. Figure 5 illustrates an example of a document, consisting of a number of nodes, that is broken into two document fragments.

If multiple document fragments corresponding to a specific node of the document are sent (e.g., a node is replaced) then the previous data within the nodes of the document represented by that document fragment become unavailable to the terminal. Replacing a single node of the document shall effectively overwrite all children of that node.

Note If an application wishes to retain such updated node information, it may do so. However, access to such outdated portions of the document is outside the scope of this specification.

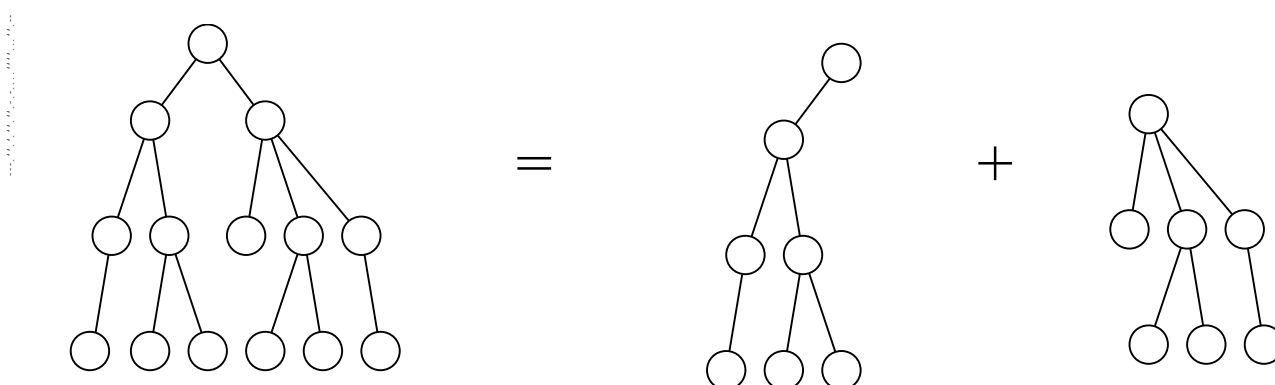


Figure 5 — Decomposition of a document into two document fragments

5.5.2 Deferred nodes, fragment references and their use

With BiM, there exists the possibility for the encoder to indicate that a node in the current document tree is “Deferred.” A deferred node shall not contain content, but shall have a type associated with it. A deferred node is addressable on the current document tree (there is a fragment update context that unambiguously points to it), but it shall not be passed on to any further processing steps, such as a parser or an application. In other words, a deferred node is a placeholder that is rendered “invisible” to subsequent processing steps.

The typical use of deferred nodes by the encoder is to establish a desired tree topology without sending all nodes of the tree. Nodes to be sent later are marked as “deferred” and are therefore hidden from a parser. Hence, the current document tree minus any deferred nodes must be schema-valid at the end of each access unit. The deferred nodes may then be replaced in any subsequent access unit or on application demand without changing the tree topology maintained internally in the decoder. However, there is no guarantee that a deferred node will ever be filled by a subsequent fragment update unit within the document stream.

Some deferred nodes are marked with a fragment reference marker that specifies where the fragment can be acquired. It is then left to the application to decide when to acquire it.

5.5.3 Managing schema version compatibility with ISO/IEC 23001-1

It is very conceivable that a given schema will be updated during its lifetime. Therefore, ISO/IEC 23001-1 provides, with some constraints, interoperability between different versions of ISO/IEC 23001-1 schema definitions, without the full knowledge of all schema versions being required.

Two different forms of compatibility between different versions of schema are distinguished. In both cases, it is assumed that the updated version of a schema imports the previous version of that schema. Backward compatibility means that a decoder aware of an updated version of a schema is able to decode a document conformant to a previous version of that schema. Forward compatibility means that a decoder only aware of a previous version of a schema is able to partially decode a document conformant to an updated version of that schema.

With both the textual and binary format, backward compatibility is provided by the unique reference of the used schema in the `DecoderInit` using its Schema URI as its namespace identifier.

When using the binary format, forward compatibility is ensured by a specific syntax defined in Clause 6 and 7. Its main principle is to use the namespace of the schema, i.e., the Schema URI, as a unique version identifier. The binary format allows one to keep parts of a document related to different schema in separate chunks of the binary document stream, so that parts related to unknown schema may be skipped by the decoder. In order for this approach to work, an updated schema should not be defined using the ISO/IEC 23001-1 “redefine” construct but should be defined in a new namespace. The Decoder Initialisation identifies schema versions with which compatibility is preserved by listing their Schema URIs. A decoder that knows at least one of the Schema URIs will be able to decode at least part of the binary document stream.

Note – Forward compatibility can also be used to generate bitstreams that can be decoded even in case of a schema update unit has not been received (for example because an error occurred) or because the decoder is not able to accept schema update units.

5.5.4 Reference consistency (informative)

The standard itself cannot guarantee reference (link) consistency in all cases. In particular, XPath-style references cannot be guaranteed to point to the correct node, especially when the topology of the tree changes in a dynamic or progressive transmission environment. With ID/IDRef, the system itself cannot guarantee that the ID element will be present, but during the validation phase, all such links are checked, and thus their presence falls under the directive that the current document tree must always be schema-valid. URI and HREF links are typically to external documents, and should be understood not to be under control by the referrer (and therefore not guaranteed).

5.6 Characteristics of the delivery layer

The delivery layer is an abstraction that includes functionalities for the synchronization, framing and multiplexing of document streams with other data streams. Document streams may be delivered independently or together with the described multimedia content. No specific delivery layer is specified or mandated by ISO/IEC 23001-1.

Provisions for two different modes of delivery are supported by this specification:

- Synchronous delivery – each access unit shall be associated with a unique time that indicates when the document fragment conveyed within this access unit becomes available to the terminal. This point in time is termed “composition time.”
- Asynchronous delivery – the point in time when an access unit is conveyed to the terminal is not known to the producer of this document stream nor is it relevant for the usage of the reconstructed document. The composition time is understood to be “best effort,” and the order of decoding AUs, if prescribed by the

producer of the document, shall be preserved. Note, however, that this in no way precludes time related information to be present within the document.

A delivery layer (DL) suitable for conveying ISO/IEC 23001-1 document streams shall have the following properties:

- The DL shall provide a mechanism to communicate a document stream from its producer to the terminal.
- The DL shall provide a mechanism by which at least one entry point to the document stream can be identified. This may correspond to a special case of a random access point, typically at the beginning of the stream.
- For applications requiring random access to document streams, the DL shall provide a suitable random access mechanism.
- The DL shall provide delineation of the access units within the document stream, i.e., AU boundaries shall be preserved end-to-end.
- The DL shall preserve the order of access units on delivery to the terminal, if the producer of the document stream has established such an order.
- The DL shall provide either error-free access units to the terminal or an indication that an error occurred.
- The DL shall provide a means to deliver the `DecoderInit` information (see subclause 6.2) to the terminal before any access unit decoding occurs and signal the coding format (textual/binary) of said information.
- The DL shall provide signalling of the association of a document stream to one or more media streams.
- In synchronous delivery mode, the DL shall provide time stamping of access units, with the time stamps corresponding to the composition time (see section on synchronous delivery earlier in this subclause) of the respective access unit.
- If an application requires access units to be of equal or restricted lengths, it shall be the responsibility of the DL to provide that functionality transparently to the systems layer.

5.7 Decoding of Fragment References

5.7.1 Decoding of Non-Deferred Fragment References

The result of decoding a non-deferred fragment reference shall be, passed to a mechanism (fragment reference resolver) which returns fragment update payload data to the FU payload decoder. This fragment update payload data is in the following form:

- a BiM fragment update payload containing the document fragment data in case of a BiM bitstream.

Note - Examples of possible fragment resolver are:

- An HTTP communication session to a WEB server
- A DSM-CC Object carousel

5.7.2 Decoding of Deferred Fragment References

The result of decoding a deferred fragment reference shall be a fragment reference marker which consists of a fragment reference, the name and type of its top most element.

Note - This fragment reference marker is signaled to the application and can be used to acquire the fragment through the fragment reference resolver at any instant of the document stream.

6 Binary format- BiM

6.1 Overview

The following subclauses specify the syntax elements and associated semantics of the binary format for ISO/IEC 23001-1 documents, abbreviated BiM. The binary DecoderInit (6.2), the binary access unit (6.3), the binary fragment update unit (6.4) and its constituent parts, the binary fragment update command (6.5), binary fragment update context (6.6) and binary schema update unit (6.7) are covered by Clause 6. The specification of the binary fragment update payload follows in Clause 7.

Identifying schema components in the BiM framework

As described in Clause 5, BiM relies upon schema knowledge. In this specification, schema components (elements, types and attributes) are identified by both a *schema identifier* and a *component identifier*.

The decoder manages both a unique initial schema and several additional schemas. From the decoder point of view, both initial schemas and additional schemas are indentified through a unique table in which each entry identifies a specific schema: the first 'NumberOfSchemas' entries identify schemas that are part of the initial schema. The following ones identify additional schemas (starting at the 'NumberOfSchemas' entry and ending at the 'NumberOfSchemas + NumberOfAdditionalSchemas - 1').

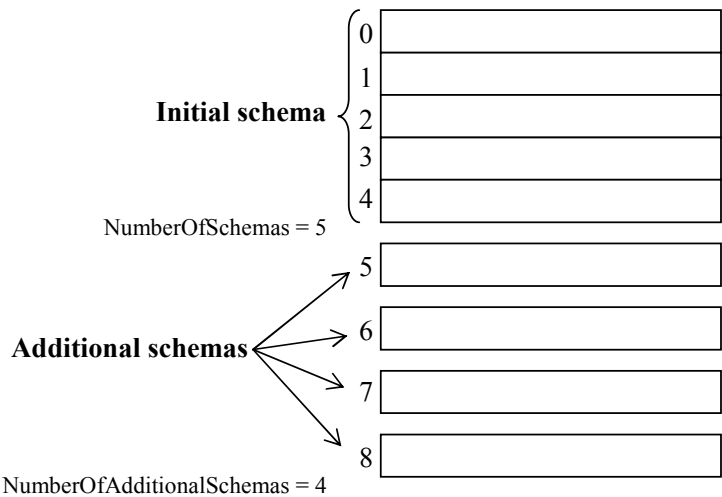


Figure 6 — Addressing the initial schema and the additional schemas

The schema component codes (type codes, element codes or attribute codes) are accessible through all these schemas. However codes are constructed differently depending on which schema they are defined in. The initial schema aggregates all schema components possibly coming from different namespaces in a single code space. On the contrary, additional schemas contain only schema components which are defined in their namespace.

6.2 Binary DecoderInit

6.2.1 Overview

The binary DecoderInit specified in this subclause is used to configure parameters required for the decoding of the binary access units. There is only one DecoderInit associated with one document stream.

Main components of the `DecoderInit` are an indication of the profile and level of the associated document stream, a list of schema URIs and optimised type codecs associated to certain data types as well as the initial document.

An optimised type codec specifies an encoding for a data type not using the generic mechanism specified in Clause 7 but optimised with full knowledge of the properties of that data type. There are two kinds of optimised type codec (or optimised decoders). A fixed optimised decoder associates a specific encoding scheme to a type of the schema (complex as well as simple) and this association is fixed for the entire stream. An advanced optimised decoder associates a specific encoding scheme to any simple type and this association can be changed during the transmission of the bitstream. Moreover, several advanced optimised decoders can be associated to a single type and can accept parameters. Some fixed optimised type codecs are specified in ISO/IEC 15938-3. Some advanced optimised decoders are defined in Clause 8.

Several other coding modes are initialised in the `DecoderInit` related to the features used by the binary description stream: the insertion of elements, the transmission of schema information, references to fragments and a fixed length context path.

Transmission of additional schema is specified for two different use cases: The retrieval of schema information in binary format from a location indicated by a URI, the transmission of schema information in a binary document stream jointly or not with the transmission of a document. In the latter case there is a requirement that all schema information needed for the decoding of a fragment of the transmitted document must have been received before such fragment arrives.

The fixed length context path mechanism provides a simplified addressing of nodes for usage scenarios where only a limited number of nodes need to be addressed. This is done by a table that uniquely maps fixed length codes to full context paths.

6.2.2 Syntax

<code>DecoderInit () {</code>	Number of bits	Mnemonic
SystemsProfileLevelIndication	8+	vluimsbf8
UnitSizeCode	3	bslbf
NoAdvancedFeatures	1	bslbf
ReservedBits	4	bslbf
<code>If (! NoAdvancedFeatures) {</code>		
AdvancedFeatureFlags_Length	8+	vluimsbf8
<code>/** FeatureFlags **/</code>		
InsertFlag	1	bslbf
AdvancedOptimisedDecodersFlag	1	bslbf
AdditionalSchemaFlag	1	bslbf
AdditionalSchemaUpdatesOnlyFlag	1	bslbf
FragmentReferenceFlag	1	bslbf
MPCOnlyFlag	1	bslbf
HierarchyBasedSubstitutionCodingFlag	1	bslbf
ContextPathTableFlag	1	bslbf
ReservedBitsZero	FeatureFlags_Length*8-8	bslbf
<code>/** FeatureFlags end **/</code>		
<code>}</code>		
<code>/** Start FUUConfig **/</code>		

If (! AdditionalSchemaUpdatesOnlyFlag) {		
NumberOfSchemas	8+	vluimsbf8
for (k=0; k< NumberOfSchemas; k++) {		
SchemaURI_Length[k]	8+	vluimsbf8
SchemaURI[k]	8* SchemaURI _Length[k]	bslbf
LocationHint_Length[k]	8+	vluimsbf8
LocationHint[k]	8* LocationHin t_Length[k]	bslbf
NumberOfTypeCodecs[k]	8+	vluimsbf8
for (i=0; i< NumberOfTypeCodecs[k]; i++) {		
TypeCodecURI_Length[k][i]	8+	vluimsbf8
TypeCodecURI[k][i]	8* TypeCodec URI _Length[k][i]	bslbf
NumberOfTypes[k][i]	8+	vluimsbf8
for (j=0; j< NumberOfTypes[k][i]; j++) {		
TypeIdentificationCode[k][i][j]	8+	vluimsbf8
}		
}		
}		
If (ContextPathTableFlag) {		
ContextPathTable()		
}		
<i>/** FUUConfig - Advanced optimised decoder framework **/</i>		
If (AdvancedOptimisedDecodersFlag) {		
NumOfAdvancedOptimisedDecoderTypes	8+	vluimsbf8
for (i=0; i< NumOfAdvancedOptimisedDecoderTypes; i++) {		
AdvancedOptimisedDecoderTypeURI_Length[i]	8+	vluimsbf8
AdvancedOptimisedDecoderTypeURI[i]	8* AdvancedO ptimisedDe coderTypeU RI_Length[i]	bslbf
}		
AdvancedOptimisedDecodersConfig ()		
}		
<i>/** FUUConfig - Fragment reference framework **/</i>		
If (FragmentReferenceFlag) {		
NumOfSupportedFragmentReferenceFormat	8	uimsbf
for (i=0; i< NumOfSupportedFragmentReferenceFormat; i++)		
{		
SupportedFragmentReferenceFormat[i]	8	blsbf
}		

}		
}		
<i>/** end FUUConfig */</i>		
If (AdditionalSchemaFlag) {		
AdditionalSchemaConfig ()		
}		
<i>/** Initial document */</i>		
If (!AdditionalSchemaUpdateOnlyFlag) {		
InitialDocument_Length	8+	vluimsbf8
InitialDocument()		
}		
}		

ContextPathTable {		
ContextPathTable_Length	8+	vluimsbf8
ContextPathCode_Length	8+	vluimsbf8
NumberOfContextPaths	8+	vluimsbf8
CompleteContextPathTable	1	bslbf
for(i=0;i<NumberOfContextPaths;i++){		
ContextPath_Length[i]	5+	vluimsbf5
ContextPath()[i]	ContextPath_Length[i]	
If(!CompleteContextPathTable){		
ContextPathCode[i]	ContextPathCode_Length	bslbf
}		
}		
nextByteBoundary()		
}		

Semantics

Name	Definition
SystemsProfileLevelIndication	Indicates the profile and level as defined in ISO/IEC 23001-1 to which the document stream conforms. Table 1 lists the indices and the corresponding profile and level.
UnitSizeCode	This is a coded representation of UnitSize, as specified in Table 2. UnitSize is used for the decoding of the binary fragment update payload as specified in Clause 7.
NumberOfSchemas	Indicates the number of schemas on which the document stream is based. These schemas compose the initial schema. A zero-value is forbidden.
SchemaURI_Length[k]	Indicates the size in bytes of the SchemaURI[k]. A value of zero is forbidden.

SchemaURI[k]	This is the UTF-8 representation of the URI to unambiguously reference one of the schemas that are needed for the decoder to decode the document stream. The SchemaURI identifies the schema that declares this SchemaURI as being its targetNamespace. The identified schema is the one composed of all schema components defined in its targetNamespace and all schema components imported from other namespaces. To support forward compatibility, multiple SchemaURIs are also used to identify imported schemas. Decoders that are aware of any of these schemas will be able to process at least the corresponding parts of the document. The SchemaID (see 6.6.3 and 7.4.4) as well as SchemaIDOfSubstitution (see 7.4.3) refer to the entries with the corresponding indices in this SchemaURI list. The SchemaURI[0] shall be assigned to the schema which imports all the namespaces that are identified by a SchemaURI[k] with an index $k > 1$. Note In order to maximize forward compatibility, it is recommended to list the SchemaURI for as many imported namespaces as practical.
LocationHint_Length[k]	Indicates the size in bytes of the LocationHint[k] syntax element.
LocationHint[k]	This is the UTF-8 representation of the URI referencing the location of the schema with index k. The LocationHint[k] shall be present except if the corresponding SchemaURI[k] already provides the location reference. In that case it may be omitted by setting the corresponding LocationHint_Length[k] to the value "0".
NumberOfTypeCodecs[k]	Indicates the number of optimised data type codecs that are subsequently associated with data types contained in the schema referred to by the index k.
TypeCodecURI[k][i]	This is the UTF-8 representation of a URI referencing an optimised binary data type codec. This codec shall be used for all data types listed subsequently.
NumberOfTypes[k][i]	Indicates the number of data types which shall be coded with the optimised data type codec referenced by TypeCodecURI[k][i].
TypeIdentificationCode[k][i][j]	Selects one data type from the set of all data types contained in the schema with index k. This data type shall be coded with the optimised data type codec referenced by TypeCodecURI[k][i] for all instantiations of this data type in the document stream. The syntax and semantics of TypeIdentificationCode[k][i][j] is the same as of the type identification code defined in subclause 6.6.5.4 except that here it is represented using vluimsbf8. The TypeIdentificationCode[k][i][j] assumes the "anyType" as base type. There shall not be more than one data type codec associated to the same data type.

	Note In order to maximise forward compatibility the value of the index k should refer to the targetNamespace in which the data type is defined.
NumOfAdvancedOptimisedDecoderTypes	Defines the number of advanced optimised decoder types that are necessary to properly decode the binary document stream.
AdvancedOptimisedDecoderTypeURI_Length[i]	Indicates the size in bytes of the AdvancedOptimisedDecoderTypeURI [i] syntax element.
AdvancedOptimisedDecoderTypeURI[i]	Defines the UTF-8 representation of the URI referencing the advanced optimised decoder type with index i.
AdvancedOptimisedDecodersConfig()	See subclause 8.2.
NumOfSupportedFragmentReferenceFormat	Specifies the number of fragment reference format that shall be supported by the decoder.
SupportedFragmentReferenceFormat[i]	Specifies the i th fragment reference format, according to Table 3, that shall be supported by the decoder. The SupportedFragmentReferenceFormat[0] indicates the default fragment reference format.
AdditionalSchemaConfig()	See subclause 6.2.3.
AdditionalSchemaUpdatesOnlyFlag	Signals that the document stream contains only additional schema updates i.e. no fragment update units. The AdditionalSchemaFlag shall be set to true when this flag is set to true.
FragmentReferenceFlag	Signals that fragment references are supported.
MPCOnlyFlag	Signals that position codes in the fragment update context are encoded in MPC mode only.
HierarchyBasedSubstitutionCodingFlag	Signals that element substitution codes are computed taking into account their substitution hierarchy. If additional schemas are supported (i.e. AdditionalSchemaFlag==true) this flag shall be set to true.
ContextPathTableFlag	Signals the presence of a context path table in the decoderInit.
ContextPathTable_Length	Defines the number of bytes used for the indication of the ContextPathTable. Note – This length provides a simple framework to skip the table.
ContextPathCode_Length	Signals the length of the context path codes in number of bits.
NumberOfContextPaths	Signals the number of ContextPaths contained in the ContextPathTable.

CompleteContextPathTable	Signals if the ContextPathTable is complete and ordered according to the assignment of ContextPathCodes. If CompleteContextPathTable is set to '1' the ContextPathCodes are assigned in the order the ContextPaths as specified in the ContextPathTable starting from '1'. If CompleteContextPathTable is set to '0' the ContextPathCodes are assigned explicitly. The ContextPathCode '0' is reserved.
ContextPath_Length	Signals the number of bits used for the following ContextPath[i]()
ContextPath[i]()	Signals the ContextPath as specified in subclause 6.6.2 with the following restrictions: - ContextModeCode is set to '001' - PositionCode() is an empty bitfield
ContextPathCode[i]	Signals the ContextPathCode of ContextPath[i]

Table 1 — Index Table for SystemsProfileLevelIndication

<i>Index</i>	<i>Systems Profile and Level</i>
0	no profile specified
1 – 127	Reserved for ISO Use

Table 2 — Code Table for UnitSize

<i>Unit Size Code</i>	<i>Unit Size</i>
000	default
001	1
010	8
011	16
100	32
101	64
110	128
111	reserved

Table 3 — Fragment Reference Formats

<i>Fragment Reference Type</i>	<i>Fragment Reference Format</i>	<i>Description</i>
0		ISO reserved
1	URIFragmentReference	This fragment reference format should be used where the reference can be expressed as a URI.
2 - 224		ISO reserved
225 – 255		Private Use

6.2.3 Syntax of AdditionalSchemaConfig

AdditionalSchemaConfig () {	Number of bits	Mnemonic
NumberOfAdditionalSchemas	8+	vluimsbf8
NumberOfKnownAdditionalSchemas	8+	vluimsbf8
for (int t=1;t<NumberOfKnownAdditionalSchemas-1;t++){		
KnownAdditionalSchemaID	8+	
AdditionalSchemaURI_Length[KnownAdditionalSchemaID]	8+	vluimsbf8
AdditionalSchemaURI[KnownAdditionalSchemaID]	8* AdditionalSchema URI_Length [KnownAdditional SchemaID]	bslbf
BinaryLocationHint_Length[KnownAdditionalSchemaID]	8+	vluimsbf8
BinaryLocationHint[KnownAdditionalSchemaID]	8*BinaryLocationHint_Length[KnownAdditionalSchemaID]	bslbf
NumberOfTypeCodecs[KnownAdditionalSchemaID]	8+	vluimsbf8
for (i=0; i< NumberOfTypeCodecs[KnownAdditionalSchemaID]; i++) {		
TypeCodecURI_Length[KnownAdditionalSchemaID][i]	8+	vluimsbf8
TypeCodecURI[KnownAdditionalSchemaID][i]	8* TypeCodecURI _Length[KnownA dditionalSchem alD][i]	bslbf
NumberOfTypes[KnownAdditionalSchemaID][i]	8+	vluimsbf8
for (j=0; j< NumberOfTypes[KnownAdditionalSchemaID][i]; j++) {		
TypeIdentificationCode[KnownAdditionalSchemaID][i][j]	8+	vluimsbf8
}		
}		
ExternallyCastableTypeTable(KnownAdditionalSchemaID)		
ExternallySubstitutableElementTable(KnownAdditionalSchemaID)		
}		
SchemaEncodingMethod	8	blsbf
ExternallyCastableTypeTable(InitialSchema)		
ExternallySubstitutableElementTable(InitialSchema)		
ReservedBitsZero	7	blsbf
}		

Name	Definition
NumberOfAdditionalSchemas	Indicates the number of schemas that can be transmitted and that are not declared in the list of <code>schemaURI</code>. If additional schemas are not supported, this value is set to zero. If additional schemas are supported then the value of <code>GlobalSchemaID = NumberOfSchemas</code> is reserved for a virtual namespace for attributes that do not belong to any namespace. The value of <code>GlobalSchemaID = NumberOfSchemas + NumberOfAdditionalSchemas - 1</code> is reserved for ISO usage.
NumberOfKnownAdditionalSchemas	Indicates the number of additional schemas that are known to be updated in the bitstream.
KnownAdditionalSchemaID	Identifies a schema known to be updated in the bitstream. This identifier shall only address an additional schema. The value <code>KnownAdditionalSchemaID = NumberOfSchemas</code> shall be reserved for attributes that do not belong to any namespace. The value of <code>KnownAdditionalSchemaID = NumberOfSchemas + NumberOfAdditionalSchemas - 1</code> shall be reserved for ISO Use.
AdditionalSchemaURI_Length[KnownAdditionalSchemaID]	Indicates the size in bytes of the <code>AdditionalSchemaURI[KnownAdditionalSchemaID]</code> length. A value of zero is forbidden.
AdditionalSchemaURI[KnownAdditionalSchemaID]	Indicates the UTF-8 representation of the URI of the additional schema identified by <code>KnownAdditionalSchemaID</code>. Note – This field allows to identify some of the additional schemas that are expected to be updated. This information allows one decoder not to monitor the schema updates for which it already knows the schema.
BinaryLocationHint_Length[KnownAdditionalSchemaID]	Indicates the size in bytes of the <code>BinaryLocationHint_Length[KnownAdditionalSchemaID]</code> . A value of zero indicates that for the schema that is referenced by the index <code>KnownAdditionalSchemaID</code> there is no binary encoded schema available.
BinaryLocationHint[KnownAdditionalSchemaID]	This is the UTF-8 representation of the URI to unambiguously reference the location of the binary encoded schema that is referenced by the index <code>KnownAdditionalSchemaID</code>. The schema can be fetched by the schema resolver and is then received as a document stream composed only of schema update units i.e. for which the <code>SchemaOnlyFlag</code> is set to true.
NumberOfTypeCodecs[KnownAdditionalSchemaID]	see <code>NumberOfTypeCodecs[k]</code> in 6.2.2.
TypeCodecURI_Length[KnownAdditionalSchemaID]	see <code>TypeCodecURI_Length[k]</code> in 6.2.2.
TypeCodecURI[KnownAdditionalSchemaID][i]	see <code>TypeCodecURI[k][i]</code> in 6.2.2.
NumberOfTypes[KnownAdditionalSchemaID][i]	see <code>NumberOfTypes[k][i]</code> in 6.2.2.

TypeIdentificationCode[KnownAdditionalSchemaID][i][j]	see TypeIdentificationCode[k][i][j] in 6.2.2.
SchemaEncodingMethod	Indicates the encoding method of the schema update units.
ExternalCastableTypeTable	Defines the types of the initial schema that are externally castable as defined in subclause 6.7.5.4 and 6.7.5.5.
ExternalSubstitutableElementTable	Defines the elements of the initial schema that are substitutable as defined in subclause 6.7.6.4 and 6.7.6.5.

Table 4 — Schema encoding method

<i>SchemaEncodingMethod</i>	<i>definition</i>
0	ISO reserved
1	BiM encoded schema as described in subclause 6.7.8
2-224	ISO reserved
225-255	Private use

6.3 Binary Access Unit

6.3.1 Overview

A binary access unit is composed of one or more binary fragment update units that represent one or more document fragments. Therefore, an access unit may convey updates for several distinct parts of a document simultaneously. Multiple fragment update units in an access unit are ordered and shall be processed by the terminal such that the result of applying the commands is equivalent to having executed them sequentially by the document composer in the order specified within the access unit. Syntax and semantics of a fragment update unit are described in subclause 6.4.

6.3.2 Syntax

AccessUnit () {	Number of bits	Mnemonic
If (AdditionalSchemaFlag) {		
NumberOfSUU	8+	vluimsbf8
for (i=0; i< NumberOfSUU ; i++) {		
SchemaUpdateUnit()		
}		
}		
If (! AdditionalSchemaUpdateOnlyFlag) {		
NumberOfFUU	8+	vluimsbf8
for (i=0; i< NumberOfFUU ; i++) {		
FragmentUpdateUnit()		
}		
}		
}		

6.3.3 Semantics

Name	Definition
NumberOfSUU	Indicates the number of schema update units in this access unit. Value '0' signifies that no schema update unit is carried.
NumberOfFUU	Indicates the number of fragment update units in this access unit. Value '0' signifies that no fragment update unit is carried.

6.4 Binary Fragment Update Unit

6.4.1 Overview

For the specification of the syntax and semantics of a binary fragment update unit it is recalled that documents are hierarchically defined, and therefore, they can be interpreted as a document tree. The elements and attributes in the document tree can generically be also referred to as “nodes”.

The topmost node is the node corresponding to the first element in the document. It instantiates one of the global elements declared in the schema. The selector node is defined to be the parent node of the topmost node artificially extending the hierarchy at the top. Figure 7 shows an example document tree.

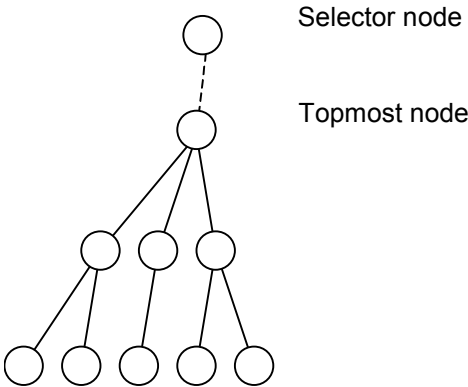


Figure 7 — Example for the tree representation of a document

Two different notions of document trees are used in this Clause: the “current document tree” (see Clause 5) and the “binary format document tree”.

The binary format document tree is used for addressing the nodes. The addressing relies upon schema knowledge, i.e. the shared knowledge of encoder and decoder about the existence and position of potential/allowed elements within the schema. The address information specifies a node within the binary format document tree built from all these possible - and not necessarily instantiated - elements as defined in the schema. Moreover, each node has a specific and fixed address which allows an unambiguous identification not depending on the current document as present at the decoder. Note that it is possible to address nodes which do not correspond to an instantiated element. Nodes corresponding to instantiated elements are called “instantiated nodes”. Deferred nodes shall be considered as instantiated nodes.

The current document tree is defined, immediately after the decoding of any AU, as the set of instantiated nodes in the binary format document tree.

Each binary fragment update unit consists of 3 sections:

- the fragment update command defining which kind of operation shall be performed on the binary format document tree, i.e. if a document fragment shall be added, replaced or deleted or if the complete binary format document tree shall be reset;
- the fragment update context signals on which node in the binary format document tree the fragment update command shall be executed. Fragment update context is present unless the fragment update command is “reset”;
- the fragment update payload containing a document fragment. Fragment update payload is present unless the fragment update command is “DeleteContent” or “Reset”. A special mode called “MultiplePayloadMode” also allows there to be multiple instances of fragment update payloads of the same type within one fragment update unit.

Additionally, each fragment update unit carries the information about its length in bytes, which allows a decoder to skip. This mechanism may be used in case the decoder does not know the corresponding schema required to decode this fragment update unit.

6.4.2 Syntax

FragmentUpdateUnit () {	Number of bits	Mnemonic
FUU_Length	8+	vluimsbf8
FragmentUpdateCommand	4	bslbf
If (AdvancedOptimisedDecodersFlag){		
OptimisedDecoderReparameterization	2	bslbf
if (OptimisedDecoderReparameterization == '00') {		
AdvancedOptimisedDecodersConfig ()		
}		
}		
if (FragmentUpdateCommand != '0100') {		
/* '0100' corresponds to “Reset” */		
FragmentUpdateContext()		
if (FragmentUpdateCommand != '0011') {		
/* '0011' corresponds to “DeleteContent” */		
for (i=0;i<NumberOfFragmentPayloads;i++) {		
FragmentUpdatePayload(startType)		
}		
}		
}		
nextByteBoundary()		
}		

6.4.3 Semantics

Name	Definition
FUU_Length	Indicates the length in bytes of the remainder of this fragment update unit (excluding the FUU_Length syntax element).
OptimisedDecodersReparameterization	This 2-bit flag signals if the parameters of the optimised decoders shall be updated. It can take the following values: — '00' – the optimised decoder instance table and mappings shall be redefined; — '01' – the optimised decoder instance table and mappings shall not be redefined; — '10' – the optimised decoder instance table and mappings are reset to the default table and mappings defined in the DecoderInit; '11' – reserved.
AdvancedOptimisedDecodersConfig()	See subclause 8.2.
startType	This internal variable indicates the effective data type of the first element that is conveyed in the fragment update payload. The startType variable is of type SchemaComponent as specified in 7.3.1. Its value is derived from the Operand_TBC in the FragmentUpdateContext as specified in 6.6.
NumberOfFragmentPayloads	The value of this internal variable is derived from FragmentUpdateContext as specified in 6.6.
FragmentUpdatePayload()	See 7.3.1

6.5 Binary Fragment Update Command

The `FragmentUpdateCommand` code word specifies the command that shall be executed on the binary format document tree. Table 5 defines the code words and the semantics of the fragment update command.

Table 5 — Code Table of fragment update commands

Code Word	Command Name	Specification
0000	---	Reserved
0001	AddContent	Add the document fragment contained in the fragment update payload at the node indicated by the operand node (see 6.6). The operand node shall not be an instantiated node but it turns into an instantiated node after processing this fragment update unit. Additionally, all nodes which are part of the context path specified in the fragment update context turn into instantiated nodes after processing this fragment update unit if these had not been instantiated nodes before. Note In the current document tree this is equivalent to either appending or inserting the corresponding nodes.
0010	ReplaceContent	Replace the document fragment at the node indicated by the operand node with document fragment contained in the fragment update payload. The operand node shall be an instantiated node. This command is equivalent to the command sequence of “DeleteContent” and “AddContent”. Note In the current document tree this is equivalent to replacing the corresponding node.
0011	DeleteContent	Delete the document fragment at the node that is indicated by the operand node. The respective node and all its child nodes are reverted to “not instantiated”. Note In the current document tree this is equivalent to deleting the corresponding node.
0100	Reset	Reset the complete binary format document tree, i.e. revert all nodes in the binary format document tree to “not instantiated” and decode the InitialDocument conveyed in the DecoderInit. After processing a reset command the current context is set to the topmost node of the current document tree. Note This is equivalent to deleting the complete current document tree.
0101 – 1111	---	Reserved

6.6 Binary Fragment Update Context

6.6.1 Overview

The fragment update context specifies on which node of the binary format document tree the fragment update command shall be executed. This node is called the “operand node”. Additionally, the fragment update context specifies the data type of the node encoded in the subsequent fragment update payload(s).

The operand node is addressed by building a path (called “Context Path”) through the binary format document tree. The context path consists of a sequence of local navigation information called Tree Branch Code (TBC). A TBC represents tree branch information from a node to a child node on the path to the operand node (see Figure 8).

A set of TBCs associated to the same complexType is called a TBC table. A TBC table is composed of one TBC for each possible child node of the complexType. Child nodes are defined as the attribute nodes of the complex type as well as, either the contained element nodes or a dedicated node representing a simple content. For the selector node there is a special TBC table containing TBCs corresponding to the global elements defined in the schema. Other TBCs are added to the TBC tables for specific purposes as described below. The algorithm for generating the TBC tables is described in 6.6.5.

A TBC is composed of four parts:

- 1) a Schema Branch Code (SBC) by which one node among the possible child nodes is selected (see 6.6.5.2),
- 2) a Substitution Code which is used if the element declaration addressed by the SBC is a reference to an element which is the head of a substitution group (see 6.6.5.3),
- 3) a Path Type Code which is used if the type of the element identified by the SBC and the Substitution Code is the base type of other named derived types (see 6.6.5.4), and
- 4) a Position Code which is used if multiple occurrences of the element addressed by the SBC are possible (see 6.6.5.5).

The TBCs for the selector node have no Substitution Code and no Position Code.

NOTE In the syntax definition of the context path this concatenation of TBCs is partly reordered (i.e. the Position Codes from all TBCs are shifted to the end of the context path as described further below).

Two types of Context Path exist. In both cases the Context Path is built from concatenated TBCs and leads to the operand node. In case of an absolute Context Path, the Context Path starts from the selector node. In case of a relative Context Path, the Context Path starts from a “current context node”. The current context node in BiM is defined by the previous `FragmentUpdateUnit` as specified in the following paragraphs. Figure 8 shows the principle of absolute and relative addressing.