[MS-BINXML]:
SQL Server Binary XML Structure Specification

Revision Summary

Date	Revision History	Revision Class	Comments
04/04/2008	0.1	Major	Initial Availability.
04/25/2008	0.2	Editorial	Revised and edited the technical content.
06/27/2008	1.0	Editorial	Revised and edited the technical content.
12/12/2008	1.01	Editorial	Revised and edited the technical content.
08/07/2009	1.1	Minor	Updated the technical content.
11/06/2009	1.1.2	Editorial	Revised and edited the technical content.

Contents

1 Introduction

This document specifies the Microsoft® Server Binary XML structure, a Microsoft proprietary format. This format is used to encode the text form of an XML document into an equivalent binary form, which can be parsed and generated more efficiently. The format provides full fidelity with the original XML documents.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

code page
little-endian
stream
Unicode
universally unique identifier (UUID)
URI
UTF-16
UTF-16LE
XML

The following terms are specific to this document:

Parser: A parser is any application that reads a Binary XML formatted stream and extracts information out of it. Parsers are also referred to as readers, processors or consumers.

Writer: A writer is any application that writes Binary XML format. Writers are also referred to as producers.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as described in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[CODEPG] Microsoft Corporation, "Code Pages", http://www.microsoft.com/globaldev/reference/cphome.mspx

If you have any trouble finding [CODEPG], please check here.

[IEEE754] Institute of Electrical and Electronics Engineers, "Standard for Binary Floating-Point Arithmetic", IEEE 754-1985, October 1985, http://ieeexplore.ieee.org/servlet/opac?punumber=2355.

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary", March 2007.

[MS-SSAS9] Microsoft Corporation, "SQL Server Analysis Services Protocol Specification", June 2008.

[MS-TDS] Microsoft Corporation,"Tabular Data Stream Protocol Specification", February 2008.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.ietf.org/rfc/rfc2119.txt

[RFC2781] Hoffman, P., and Yergeau, F., "UTF-16, an encoding of ISO 10646", RFC 2781, February 2000, http://www.ietf.org/rfc/rfc2781.txt

[RFC5234] Crocker, D., Ed., and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008, http://www.ietf.org/rfc/rfc5234.txt

[XML10] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Third Edition)", February 2004, http://www.w3.org/TR/REC-xml

[XMLNS] World Wide Web Consortium, "Namespaces in XML 1.0 (Second Edition)", August 2006, http://www.w3.org/TR/REC-xml-names/

1.2.2 Informative References

[ISO-8601] International Organization for Standardization, "Data Elements and Interchange Formats - Information Interchange - Representation of Dates and Times", ISO 8601:2004, December 2004, http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=40874&ICS1=1&ICS2=140&ICS3=30

Note There is a charge to download the specification.

[RFC3548] Josefsson, S., Ed., "The Base16, Base32, and Base64 Data Encodings", RFC 3548, July 2003, http://www.ietf.org/rfc/rfc3548.txt

[XMLSCHEMA1] Thompson, H.S., Ed., Beech, D., Ed., Maloney, M., Ed., and Mendelsohn, N., Ed., "XML Schema Part 1: Structures", W3C Recommendation, May 2001, http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[XMLSCHEMA2] Biron, P.V., Ed. and Malhotra, A., Ed., "XML Schema Part 2: Datatypes", W3C Recommendation, May 2001, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

1.3 Structure Overview (Synopsis)

Binary XML is used to encode the text form of an **XML** document into an equivalent binary form which can be parsed and generated more efficiently. The format employs the following techniques to achieve this efficiency:

- Values (for example, attribute values or text nodes) are stored in a binary format, which means that a **parser** or a **writer** is not required to convert the values to and from string representations.
- XML element and attribute names are declared once and they are later referenced by numeric identifiers. This is in contrast to the text representation of XML which repeats element and attribute names wherever they are used in an XML document.

1.4 Relationship to Protocols and Other Structures

An XML document encoded in the binary XML format is a **stream** of bytes which can be transmitted by various network protocols. Such network protocols can choose to wrap the binary XML data within other byte streams. The specification of such network protocols and the formats they use to transmit data (including binary XML) is not part of this document.

Binary XML is used by [MS-SSAS9] and [MS-TDS].

1.5 Applicability Statement

Binary XML is suitable for use when it is important to minimize the cost of producing or consuming XML data and all consumers of the XML can agree on this format. It is not appropriate for scenarios where interoperability with consumers using plain-text XML or other binary XML formats is required.

Binary XML can represent any XML document as defined by [XML10] including support for namespaces as defined in [XMLNS].

1.6 Versioning and Localization

The Binary XML format has two versions: Version 1 and Version 2, as defined in Structures (section 2).

Binary XML supports a fixed set of features for each version. The version number in the header of a binary XML document specifies the version of the binary XML format it uses. Document Root Level (section 2.1.1) describes the binary XML document header in detail.

1.7 Vendor-Extensible Fields

Binary XML supports extension tokens, which allow applications to embed application-specific information into the data stream. The format does not specify how to process these values or how to distinguish values from multiple vendors or layers. It also does not provide any capability to negotiate the set of extensions in use. Parsers of the format MUST ignore extension tokens which they do not expect or do not understand.

2 Structures

The structures described in the following sections are applicable to Binary XML Versions 1 and 2, unless otherwise specified.

The following is an Augmented Backus-Naur Form (ABNF) description of the Binary XML format. ABNF is specified in [RFC5234]. In accordance with section 2.4 of that RFC, this description assumes no external encoding because the terminal values of this grammar are bytes.

```
document  =  signature version encoding [xmldecl] *misc
             [doctypedecl *misc] content
signature  =  %xDF %xFF
version  =  %x01 / %x02    ; x01 means Version 1, x02 means Version 2
encoding  =  %xB0 %x04     ; 1200 little-endian = UTF-16LE
xmldecl     =  XMLDECL-TOKEN textdata [ENCODING-TOKEN textdata]
               standalone
misc  =  comment / pi / metadata
doctypedecl  =  DOCTYPEDECL-TOKEN textdata [SYSTEM-TOKEN textdata]
                [PUBLIC-TOKEN textdata] [SUBSET-TOKEN textdata]
content  =  *(element / cdsect / pi / comment / atomicvalue /
            metadata / nestedbinaryxml)
textdata  =  length32  *(byte byte)     ; length is in UTF-16LE
             characters
textdata64  =  length64 *(byte byte)    ; length is in UTF-16LE
               characters
standalone  =  %x00 /    ; the standalone attribute was not specified
               %x01 /    ; yes
               %x02      ; no
comment  =  COMMENT-TOKEN textdata
pi  =  PI-TOKEN name textdata
metadata  =  namedef / qnamedef / extension /
             FLUSH-DEFINED-NAME-TOKENS
namedef  =  NAMEDEF-TOKEN textdata
name  =  mb32    ; 0 is reserved for empty name/zero length string
qnamedef  =  QNAMEDEF-TOKEN namespaceuri prefix localname
qname  =  mb32    ; index to the (NsUri, Prefix and LocalName) table
                    assigned starting from 1, 0 is invalid
extension  =  EXTN-TOKEN length32 *byte
namespaceuri  =  name
prefix  =  name
localname  =  name
element  =  ELEMENT-TOKEN qname [1*attribute ENDATTRIBUTES-TOKEN]
            content ENDELEMENT-TOKEN
cdsect    =  1*(CDATA-TOKEN textdata) CDATAEND-TOKEN
nestedbinaryxml  =  NEST-TOKEN document ENDNEST-TOKEN
attribute  =  *metadata ATTRIBUTE-TOKEN qname
              *(metadata / atomicvalue)
atomicvalue   =  (SQL-BIT byte) /
                 (SQL-TINYINT byte) /
                 (SQL-SMALLINT 2byte) /
                 (SQL-INT 4byte) /
                 (SQL-BIGINT 8byte) /
                 (SQL-REAL 4byte) /
                 (SQL-FLOAT 8byte) /
                 (SQL-MONEY 8byte) /
                 (SQL-SMALLMONEY 4byte) /
                 (SQL-DATETIME 8byte) /
```

```
                    (SQL-SMALLDATETIME 4byte) /
                    (SQL-DECIMAL decimal) /
                    (SQL-NUMERIC decimal) /
                    (SQL-UUID 16byte) /
                    (SQL-VARBINARY blob64) /
                    (SQL-BINARY blob) /
                    (SQL-IMAGE blob64) /
                    (SQL-CHAR codepagetext) /
                    (SQL-VARCHAR codepagetext64) /
                    (SQL-TEXT codepagetext64) /
                    (SQL-NVARCHAR textdata64) /
                    (SQL-NCHAR textdata) /
                    (SQL-NTEXT textdata64) /
                    (SQL-UDT blob) /
                    (XSD-BOOLEAN byte) /
                    (XSD-TIME 8byte) /
                    (XSD-DATETIME 8byte) /
                    (XSD-DATE 8byte) /
                    (XSD-BINHEX blob) /
                    (XSD-BASE64 blob) /
                    (XSD-DECIMAL decimal) /
                    (XSD-BYTE byte) /
                    (XSD-UNSIGNEDSHORT 2byte) /
                    (XSD-UNSIGNEDINT 4byte) /
                    (XSD-UNSIGNEDLONG 8byte) /
                    (XSD-QNAME qname) /
                    (XSD-DATE2 sqldate) /
                    (XSD-DATETIME2 sqldatetime2) /
                    (XSD-TIME2 sqldatetime2) /
                    (XSD-DATEOFFSET sqldatetimeoffset) /
                    (XSD-DATETIMEOFFSET sqldatetimeoffset) /
                    (XSD-TIMEOFFSET sqldatetimeoffset)
  byte  =  OCTET    ; 8 bits stored as one byte
  lowbyte   =  %x00-7F
  highbyte  =  %x80-FF
  mb32  =  *4highbyte lowbyte    ; unsigned integer in
                                   little-endian
                                   multi-byte encoding
  mb64  =  *9highbyte lowbyte    ; unsigned integer in
                                   little-endian
                                   multi-byte encoding
  sqldate  =  3byte    ; little-endian 3 byte integer
  sqltime  =  (%x00-02 3byte) / (%x03-04 4byte) / (%x05-07 5byte)
  sqltimezone  =  2byte    ; little-endian 2 byte integer
  sqldatetime2  =  sqltime sqldate
  sqldatetimeoffset  =  sqltime sqldate sqltimezone
  decimaldata  =  4byte / 8byte / 12byte / 16byte
  sign  =  %x00 / %x01    ; 1 is positive, 0 is negative
  decimal  =  length32 byte sign decimaldata
  length32  =  mb32
  length64  =  mb64
  blob  =  length32 *byte
  blob64  =  length64 *byte
  codepage  = 4byte
  codepagetext  = length32 codepage *byte
  codepagetext64  = length64 codepage *byte
  SQL-SMALLINT  = %x01
  SQL-INT  =  %x02
  SQL-REAL  =  %x03
```

```
SQL-FLOAT  =  %x04
SQL-MONEY  =  %x05
SQL-BIT  =  %x06
SQL-TINYINT  =  %x07
SQL-BIGINT  =  %x08
SQL-UUID  =  %x09
SQL-DECIMAL  =  %x0A
SQL-NUMERIC  =  %x0B
SQL-BINARY  =  %x0C      ; Binary data
SQL-CHAR  =  %x0D        ; Codepage encoded string
SQL-NCHAR  =  %x0E       ; Unicode encoded string
SQL-VARBINARY  =  %x0F   ; Binary data
SQL-VARCHAR  =  %x10     ; Codepage encoded string
SQL-NVARCHAR  =  %x11    ; Unicode encoded string
SQL-DATETIME  =  %x12
SQL-SMALLDATETIME  =  %x13
SQL-SMALLMONEY  =  %x14
SQL-TEXT  =  %x16     ; Codepage encoded string
SQL-IMAGE  =  %x17    ; Binary data
SQL-NTEXT  =  %x18    ; Unicode encoded string
SQL-UDT  =  %x1B      ; Binary data
XSD-TIMEOFFSET  =  %x7A
XSD-DATETIMEOFFSET =  %x7B
XSD-DATEOFFSET  =  %x7C
XSD-TIME2  =  %x7D
XSD-DATETIME2  =  %x7E
XSD-DATE2  =  %x7F
XSD-TIME  =  %x81
XSD-DATETIME  =  %x82
XSD-DATE  =  %x83
XSD-BINHEX  =  %x84
XSD-BASE64  =  %x85
XSD-BOOLEAN  =  %x86
XSD-DECIMAL  =  %x87
XSD-BYTE  =  %x88
XSD-UNSIGNEDSHORT  =  %x89
XSD-UNSIGNEDINT  =  %x8A
XSD-UNSIGNEDLONG  =  %x8B
XSD-QNAME  =  %x8C
FLUSH-DEFINED-NAME-TOKENS  =  %xE9
EXTN-TOKEN  =  %xEA
ENDNEST-TOKEN  =  %xEB
NEST-TOKEN  =  %xEC
QNAMEDEF-TOKEN  =  %xEF
NAMEDEF-TOKEN  =  %xF0
CDATAEND-TOKEN  =  %xF1
CDATA-TOKEN  =  %xF2
COMMENT-TOKEN  =  %xF3
PI-TOKEN  =  %xF4
ENDATTRIBUTES-TOKEN  =  %xF5
ATTRIBUTE-TOKEN  =  %xF6
ENDELEMENT-TOKEN  =  %xF7
ELEMENT-TOKEN  =  %xF8
SUBSET-TOKEN  =  %xF9
PUBLIC-TOKEN  =  %xFA
SYSTEM-TOKEN  =  %xFB
DOCTYPEDECL-TOKEN  =  %xFC
ENCODING-TOKEN  =  %xFD
```

```
XMLDECL-TOKEN  =  %xFE
```

Note that the values of constant tokens (for example **SQL-SMALLINT**) are not sequential. The values which are not defined in the above grammar are not used by Binary XML Versions 1 and 2.

XML documents encoded in Binary XML MUST conform to the grammar of the document.

The byte order of the entire Binary XML document is defined by the application which uses it. The order in which Binary XML data is stored or transferred is not part of this document. Thus any reference to byte order (for example, **little-endian**) in this document is relative to the order of the entire Binary XML document.

A parser of Binary XML MUST fail if it encounters data which does not follow the grammar or the conformance rules specified in this section.

A writer of Binary XML MUST fail if it is requested to write data which would break any of the rules in the grammar or the conformance rules specified in this section.

Binary XML does not impose any restrictions other than those implied or explicitly stated in this section. An implementation of a parser or writer MAY<1> impose additional restrictions. Examples of such restrictions can be derived from limitations on available resources or of a targeted system.

Dates and times in this section are specified by using the notation from [ISO8601]. Dates and times are specified by using the proleptic Gregorian calendar.

2.1 XML Structures

The following sections describe the Binary XML representation of basic XML structures.

2.1.1 Document Root Level

The root level of each document contains the header (for example, signature, version, and declaration) followed by the content of the document.

```
signature  =  %xDF %xFF
version  =  %x01 / %x02
document  =  signature version encoding [xmldecl] *misc
             [doctypedecl *misc] content
misc   =  comment / pi / metadata
```

The document MUST start with a 2-byte signature (0xDF, 0xFF) followed by a 1-byte version, which MUST be either 1 or 2. A parser MAY<2> choose to support version value 0 and treat it as Version 1. It MUST be followed by 2 bytes that specify the document encoding **code page**. In Versions 1 and 2 this value MUST be the UTF-16 code page (0x04B0 or 1200 in decimal).

2.1.2 XML Declaration

The XML declaration token can be used to preserve the XML declaration specified in the original XML document when encoding it in Binary XML.

```
xmldecl  =  XMLDECL-TOKEN textdata [ENCODING-TOKEN textdata]
            standalone

standalone  =  %x00 /    ; standalone attribute was not specified
```

```
%x01 /    ; yes
%x02      ; no
```

XML declaration is included only to preserve the information in text XML documents. The contents of the XML declaration in Binary XML map to the XML declaration in the original text document as follows:

- The first **textdata** value MUST contain the content of the version attribute.
- The **textdata** following the **ENCODING-TOKEN** MUST contain the value of the **encoding** attribute.
- The **standalone** token MUST store the value of the **standalone** attribute.

2.1.3 Document Type Declaration

The **Document Type Declaration (DTD)** token can be used to preserve the information from the **DOCTYPE** tag specified in the original XML document when encoding it in Binary XML.

```
doctypedecl  =  DOCTYPEDECL-TOKEN textdata [SYSTEM-TOKEN textdata]
                [PUBLIC-TOKEN textdata] [SUBSET-TOKEN textdata]
```

DTD is included only to preserve the information in text XML documents. The contents of DTD in Binary XML map to DTD in the original text document as follows:

- The first **textdata** MUST contain the name of the **DOCTYPE** declaration.
- The **textdata** following the **SYSTEM-TOKEN** MUST contain the **SYSTEM ID**.
- The **textdata** following the **PUBLIC-TOKEN** MUST contain the **PUBLIC ID**.
- The **textdata** following the **SUBSET-TOKEN** MUST contain the internal DTD subset.

2.1.4 Comments and Processing Instructions

Comments and processing instructions can be used to preserve comments and processing instructions specified in the original XML document when encoding it in Binary XML.

```
comment  =  COMMENT-TOKEN textdata
pi  =  PI-TOKEN name textdata
```

Comments and processing instructions are included only to preserve the information in text XML documents. The contents of comments and processing instructions in Binary XML map to comments and processing instruction in the original text document as follows:

- The **textdata** following the **COMMENT-TOKEN** MUST contain the value of the comment.
- The **name** following the **PI-TOKEN** MUST contain the target of the processing instruction.
- The **textdata** following the **name** MUST contain the data of the processing instruction.

2.1.5 Content

Each document can have content which can consist of any number of elements or values interleaved with metadata.

```
content   =  *(element / cdsect / pi / comment / atomicvalue / metadata /
             nestedbinaryxml)
metadata  =  1*(namedef / qnamedef / extension /
             FLUSH-DEFINED-NAME-TOKENS)
```

Note that Binary XML allows more than one element at the document root level. However, a parser of Binary XML MAY<3> choose to enforce the XML conformance rules and not allow atomic values, CDATA sections and more than one element at the document root level.

2.1.6 Elements and Attributes

This section describes Binary XML representation of XML elements and attributes.

```
element   =  ELEMENT-TOKEN qname [1*attribute ENDATTRIBUTES-TOKEN]
             content ENDELEMENT-TOKEN
attribute  =  *metadata ATTRIBUTE-TOKEN qname
              *(metadata / atomicvalue)
```

An element is defined by a **qname** token followed by an optional sequence of attributes. Attributes MUST be followed by an **ENDATTRIBUTES-TOKEN** to mark the start of an element's content. The **ENDELEMENT-TOKEN** specifies the end of the current element.

The value of an attribute is optional. If no value is specified, it defaults to an empty string. A parser MUST be able to accept inputs which have zero or one atomic value after **ATTRIBUTE-TOKEN**. A parser MAY<4> choose to also accept inputs which have more than one atomic value after **ATTRIBUTE-TOKEN**.

The **qname** token of elements and attributes can contain a prefix to a namespace **Uniform Resource Identifier (URI)** mapping that is not explicitly declared by an 'xmlns' attribute. Prefix to namespace URI mappings MUST conform to [XMLNS]. This includes but is not limited to the following restrictions:

- A prefix MUST NOT be mapped to two different namespaces within one element
- A prefix MUST NOT be mapped to an empty namespace
- An empty prefix MUST NOT be mapped to a non-empty namespace used on an attribute

For better compatibility, a parser of Binary XML MAY<5> choose to add the missing xmlns declarations when presenting data to an application.

2.1.7 Namespace Declarations

XML namespace declarations are transported as attributes. The local name and namespace URI tokens of all namespace declaration attributes MUST be 0 (empty string). A parser SHOULD report such attributes as having a namespace URI of http://www.w3.org/2000/xmlns/, but it MAY<6> choose to report it as an empty URI. If a namespace declaration is to define a default namespace (empty prefix), the prefix token MUST be defined as "xmlns". If a namespace declaration is to define

a non-empty prefix, the prefix token MUST be defined as a string starting with "xmlns:" followed by the new prefix being declared.

For example a namespace declaration of xmlns:p="ns" is serialized with these properties:

```
Local name      ""
URI             ""
Prefix          "xmlns:p"
Value           "ns"
```

A default namespace declaration of xmlns="ns" is serialized with these properties:

```
Local name      ""
URI             ""
Prefix          "xmlns"
Value           "ns"
```

A non-empty prefix MUST NOT be mapped to an empty namespace URI.

The value of a namespace declaration attribute MUST consist of only zero or one atomic value. A parser MUST accept SQL-NVARCHAR, SQL-NCHAR and SQL-NTEXT as the value of a namespace declaration attribute. A parser MAY<7> accept other atomic value types as the value of a namespace declaration attribute, in which case it MUST convert its value to a **Unicode** string.

2.1.8 CDATA Sections

CDATA sections are used in text XML documents to simplify the storing of code or markup sections. The **CDATA** token can be used to preserve the **CDATA** sections specified in the original XML document when encoding in binary XML.

```
cdsect  =  1*(CDATA-TOKEN textdata) CDATAEND-TOKEN
```

Multiple **CDATA** chunks (**CDATA-TOKEN** and **textdata**) MUST be considered as a single **CDATA** section until **CDATAEND-TOKEN** is reached.

2.1.9 Nested Documents

Binary XML allows a document to be nested in another document. Nesting of documents is useful when constructing an XML document from XML fragments that are already encoded in Binary XML. Nesting allows for fast concatenation of such XML fragments.

```
nestedbinaryxml  =  NEST-TOKEN document ENDNEST-TOKEN
```

Nested documents MUST have their own scope of name and **qname** tokens (separate tables). Subsequent definitions of name and **qname** inside the nested document MUST start from index 1. However, they MUST share the same XML namespace scope as their parent document.

2.1.10 Extensions

Extensions provide a way to embed application-specific information into a Binary XML data stream.

```
extension  =  EXTN-TOKEN length32 *byte
```

Extension is a block of binary data. The length32 specifies its length in bytes followed by the extension data.

The set of supported extensions and their formats is not specified by this document.

A parser of Binary XML MUST ignore an extension which it does not expect or it does not understand. If a parser recognizes an extension but its content is not valid, the parser MAY<8> generate an error and fail.

2.2 Names

During parsing or writing of Binary XML a parser or writer MUST keep a table of **name** tokens and another table of **qname** tokens. Any string that is used as a local name, a prefix or a namespace URI of an XML element or attribute MUST be added to the name table and the qname table. Any string that is used as a processing instruction target MUST be added in the name table and the qname table. The scope of these tables is the current document. Nested documents MUST have separate **name** and **qname** token tables.

Name and **qname** tokens can be declared on the document root level, in the element content, before an attribute, or between atomic values. See the grammar for all the possible locations.

FLUSH-DEFINED-NAME-TOKENS instructs both parser and writer to discard all previously defined **names** and **qnames** at the current nesting level. Subsequent definition of **name** or **qname** MUST start from index 1. Usage of this token can reduce the amount of memory used by parsers and writers. A writer MAY<9> choose to use this token in any place it is allowed by the grammar or it MAY choose not to use it at all.

2.2.1 Name Definition

Each **name** MUST be defined and added into the table of **names** before it is referenced in an element or attribute. Binary XML uses **NAMEDEF-TOKEN** to define a new **name**.

```
namedef  =  NAMEDEF-TOKEN textdata
```

A **name** MUST be stored on the next available position in the current **name** token table and MUST be assigned its index in that table. The index MUST be sequential and MUST start from 1 (inclusive). The index number MUST be used when referring to this **name**. Index 0 MUST be reserved for an empty name (zero-length string).

Note that the index of a **name** is not specified in its definition, it is implied by the current state of the name table. Both parser and writer will derive the index number from the number of **names** in the current name table. As both are using the same algorithm to build their name tables, they will produce the same result.

2.2.2 Name Reference

When a defined **name** is used it MUST be only referenced by its index in the table of names.

```
name    =  mb32  ; assigned starting from 1           ; 0 is reserved for empty name/zero
length string
```

A **name** is referenced by encoding its index in the current name table as an **mb32** token.

Note that the above implies that a ***name*** MUST be defined before it is referenced.

2.2.3 QName Definition

A **qname** MUST be defined by a triplet of a namespace URI, a prefix and a local name.

```
 qnamedef  =  QNAMEDEF-TOKEN namespaceuri prefix localname namespaceuri  =  name prefix  =
name localname  =  name
```

A parser or writer MUST keep a table of **qname** tokens. **qnames** are used for element and attribute names. When a **qname** is defined it MUST be added to the qname table and MUST be assigned a number, which is its index into this table. The indexes MUST be assigned sequentially starting from 1 (inclusive).

2.2.4 QName Reference

When a defined **qname** is used, it MUST only be referenced by its index in the table of qnames.

```
qname  =  mb32  ; index to the qname table          assigned starting from 1, 0 is invalid
```

A **qname** is referenced by encoding its index in the current qname table as an **mb32** token. Note that the above implies that the **qname** MUST be defined before it is referenced.

2.3 Atomic values

2.3.1 Integral Numeric Types

Atomic types **SQL-TINYINT**, **SQL-SMALLINT**, **SQL-INT** and **SQL-BIGINT** are signed integers.

Atomic types **XSD-BYTE**, **XSD-UNSIGNEDSHORT**, **XSD-UNSIGNEDINT** and **XSD-UNSIGNEDLONG** are unsigned integers.

2.3.2 Multi-byte Integers

Multi-byte integers MUST represent unsigned values and use variable length storage to represent numbers. Each byte stores 7 bits of the integer. The high-order bit of each byte indicates whether the following byte is a part of the integer. If the high-order bit is set, the lower seven bits are used and a next byte MUST be consumed. If a byte has the high-order bit cleared (meaning that the value of the byte is less than 0x80) then that byte is the last byte of the integer. The least significant byte (LSB) of the integer appears first.

The following table shows the number of bytes used to store a value in a certain range:

Range from	Range to	Encoding used
0x00000000	0x0000007F	1 byte
0x00000080	0x00003FFF	2 bytes, LSB stored first
0x00004000	0x001FFFFF	3 bytes, LSB stored first
0x00200000	0x0FFFFFFF	4 bytes, LSB stored first

Range from	Range to	Encoding used
0x10000000	0x7FFFFFFF	5 bytes, LSB stored first

For mb32 integers the resulting number MUST fit into a signed 32bit integer.

For mb64 integers the resulting number MUST fit into a signed 64bit integer. A parser or writer MAY<10> choose to limit the valid range of the resulting number even more.

2.3.3 Single Precision Floating Number

A single precision floating number is used to store floating point values with a limited range. The value MUST be a single precision 32bit [IEEE754] value stored as little-endian.

This is used by the **SQL-REAL** atomic value.

2.3.4 Double Precision Floating Number

A double precision floating number is used when the limited range of a single precision floating number is insufficient. The value MUST be a double precision 64bit [IEEE754] value stored as little-endian.

This is used by the **SQL-FLOAT** atomic type.

2.3.5 Decimal Number

A value MUST be stored as:

- **Length (mb32)** – The size of the atomic value in bytes. Length MUST include the number of bytes required to represent precision, scale, sign, and value (as defined below). The value of this field MUST be one of the following values: 7 (4-byte value), 11 (8-byte value), 15 (12-byte value) and 19 (16-byte value).
- **Precision (byte)** – The maximum number of digits in base 10. The maximum value is 38.
- **Scale (byte)** – The number of digits to the right of the decimal point. This MUST be less than or equal to the precision.
- **Sign (byte)** – The sign of the value. 1 is for positive numbers, 0 is for negative numbers, other values MUST NOT be used.
- **Value (4, 8, 12, or 16 bytes)** – The number stored as either a 4- or 8- or 12- or 16-byte integer (little-endian). The size is determined by the **Length** field.

For example, to specify the base 10 number 20.003 with a scale of 4, the number is scaled to an integer of 200030 (20.003 shifted by four tens digits), which is 30D5E in hexadecimal. The value stored in the 16-byte integer is 5E 0D 03 00 00 00 00 00 00 00 00 00 00 00 00 00, the precision is the maximum precision, the scale is 4, and the sign is 1. Or it can also be a 4-byte integer of 5E 0D 03 00. So the complete representation of this number could be for example:

```
07 06 04 01 5E 0D 03 00
```

This is used by the **SQL-DECIMAL**, **SQL-NUMERIC** and **XSD-DECIMAL** atomic types.

2.3.6 Money

Money is stored as an 8 byte signed integer number (little-endian). **Money** MUST be a decimal number with a fixed scale of 4. This means that it is stored as the original value multiplied by 10000.

For example, 10.3001 will be stored as 103001.

This is used by the **SQL-MONEY** atomic type.

2.3.7 Small Money

Small money is stored as a 4-byte signed integer number (little-endian). **Small money** MUST be a decimal number with a fixed scale of 4. This means that it is stored as the original value multiplied by 10000.

This is used by the **SQL-SMALLMONEY** atomic type.

2.3.8 Unicode Encoded Text

Tokens **textdata** and **textdata64** represent)UTF-16LE (Unicode Transformation Format, 16-bits, little endian) encoded strings. The length of a string MUST be stored as either mb32 (in case of **textdata**) or mb64 (in case of textdata64). The length MUST be the number of UTF-16LE characters.

The strings SHOULD<11> be valid UTF-16LE strings. A parser MAY<12> choose not to check this constraint.

These are used for atomic types **SQL-NCHAR**, **SQL-NVARCHAR** and **SQL-NTEXT**.

2.3.9 Code Page Encoded Text

Tokens **codepagetext** and **codepagetext64** represent a string encoded in a specified code page. First, the length of the string MUST be stored. The length MUST be in bytes and MUST include the 4 bytes for the code page number. Next, the code page number MUST be stored as a little-endian 32bit unsigned integer (4 bytes). The code page number specifies which encoding to use to decode the string which follows. The mapping between code page number and the encoding is defined as follows:

- Code page number 1200 means UTF-16LE encoding.
- Other code page numbers are defined in [CODEPG].

These are used for atomic types **SQL-CHAR**, **SQL-VARCHAR** and **SQL-TEXT**.

2.3.10 Boolean

Boolean types are used to store logical true or false values.

An **XSD-BOOLEAN** value MUST be stored as a byte. If the value of the byte is 0, the result is "false", if the value is 1, the result is "true". A parser SHOULD<13> recognize all nonzero values as "true" but it MAY choose to only support 0 and 1.

A **SQL-BIT** value MUST be stored as a byte. Its value SHOULD<14> be either 0 or 1. A parser MAY<15> choose to support all possible values and report it as a number.

2.3.11 XSD Date

XSD Date is used to store date information originating from XML. The type does not include time information. For more information about XSD, see [XMLSCHEMA1] and [XMLSCHEMA2].

An **XSD Date** value MUST be stored as an 8-byte little-endian integer, where the lower 2 bits store number 1. The algorithm for computing the value is:

```
Value  =  1 + 4 * ((60 * 14 + TimeZoneAdj) + (60 * 29 * DayMonthYear))
TimeZoneAdj  =  -Sign * (Minutes + 60 * Hour)
DayMonthYear  =  Day - 1 + 31 * ( Month - 1 + 12 * ( Year + 9999 ) )
```

- Day MUST be in range 1 - 31 depending on the Month.
- Month MUST be in the range 1 - 12.
- Year MUST be in the range -9999 - 9999.
- Minutes MUST be in range 0 - 59.
- Hour MUST be in range 0 - 23.
- Sign MUST be 1 for positive time zones and -1 for negative time zones.

A parser SHOULD fail if the specified Year, Month and Day combination is not valid but it MAY<16> choose to report the value to the application. Hour and Minutes are adjustments for time zone. TimeZoneAdj is positive/negative depending on which direction the adjustment shifts the time. A time zone adjustment, such as 2003-11-9T00:00-4:30, is actually a positive TimeZoneAdj, while 2003-11-9T00:00+4:30 would mean a negative TimeZoneAdj.

This is used by the atomic type **XSD-DATE**.

2.3.12 XSD DateTime

XSD DateTime is used to store both date and time information originating from XML. For more information about XSD, see [XMLSCHEMA1] and [XMLSCHEMA2].

An **XSD DateTime** value MUST be stored as an 8-byte integer, where the lower 2 bits store number 2. The algorithm for computing the value is:

```
Value = 2 + 4 * (
                 Milliseconds + 1000 *  (
                   Seconds + 60 * (
                      Minutes + 60 * (
                        Hour + 24 * (
                          Day - 1 + 31 * (
                            Month - 1 + 12 * (
                              Year + 9999 ) ) ) ) ) ) )
```

- Day MUST be in range 1 - 31 depending on the Month.
- Hour MUST be in range 0 - 23.
- Milliseconds MUST be in range 0 - 999.
- Minutes MUST be in range 0 - 59.

- Month MUST be in range 1 - 12.
- Seconds MUST be in range 0 - 59.

A parser SHOULD fail if the specified Year, Month and Day combination is not valid but it MAY<17> choose to report the value to the application. In supporting years from -9999 – 9999, the year -9999 is considered to be 0th year so an offset of 9999 MUST be applied to Year.

This is used by the atomic type **XSD-DATETIME**.

2.3.13 XSD Time

XSD Time is used to store time information originating from XML in cases where the date does not need to be preserved. For more information about XSD, see [XMLSCHEMA1] and [XMLSCHEMA2].

An **XSD Time** value MUST be stored as an 8-byte integer, where the lower 2 bits store number 0. The algorithm for computing the value is:

```
  Value = 4 * (             Milliseconds + 1000 *  (               Seconds + 60 * (
Minutes + 60 * (                      Hour ) ) ) )
```

- Hour MUST be in range 0-23.
- Milliseconds MUST be in range 0-999.
- Minutes MUST be in range 0-59.
- Seconds MUST be in range 0-59.

This is used by the **XSD-TIME** atomic type.

2.3.14 SQL DateTime and SmallDateTime

SQL DateTime and **SmallDateTime** are used to store date and time information originating from the database date and time values.

```
DayTicks = number of days since 1900-1-1
DateTicks = signed 4 byte little-endian integer with value of DayTicks
SmallDateTicks = unsigned 2 byte little-endian integer
                 with value of DayTicks
SQLTicksPerMillisecond = 0.3
SQLTicksPerSecond = 300
SQLTicksPerMinute = SQLTicksPerSecond * 60
SQLTicksPerHour = SQLTicksPerMinute * 60
TicksForMilliseconds = round-off(Milliseconds *
                       SQLTicksPerMillisecond + 0.5)
  ; round-off means disregard decimal points, so 1.9 is turned into 1
TotalTimeTicks = Hours * SQLTicksPerHour +
                 Minutes * SQLTicksPerMinute +
                   Seconds * SQLTicksPerSecond +
                     TicksForMilliseconds
TimeTicks = unsigned 4 byte little-endian integer
            with value of TotalTimeTicks
  ; This is the number of seconds times 300
SmallTotalTimeTicks = Hours * 60 + Minutes
SmallTimeTicks = unsigned 2 byte little-endian integer
                 with value of SmallTotalTimeTicks
```

```
  ; This is the number of minutes
DateTime = DateTicks TimeTicks
SmallDateTime = SmallDateTicks SmallTimeTicks
```

- Hours MUST be in range 0-23.
- Milliseconds MUST be in range 0-999.
- Minutes MUST be in range 0-59.
- Seconds MUST be in range 0-59.

Note that for the TimeTicks there are cases where two different inputs are stored as the same value due to round off. For example time 00:59:59.999 and time 01:00:00.000 are both stored as value 1080000. A parser SHOULD<18> round up during the parsing of such values and thus report the time of value 1080000 as 01:00:00.000.

The **DateTime** is used by the **SQL-DATETIME** atomic type.

The **SmallDateTime** is used by the **SQL-SMALLDATETIME** atomic type.

2.3.15 Uuid

Uuid is a sequence of 16 bytes (stored as little-endian) that specifies a universally unique identifier (UUID).

The UUID is used by the **SQL-UUID** atomic type.

2.3.16 Base64

Base64 is used to encode binary data in the text XML format. **Base64** is a way to encode binary data into a string representation and is defined in [RFC3548].

From the perspective of Binary XML this is a block of binary data. A parser SHOULD<19> report the value as binary data. In addition to that it MAY<20> choose to expose this as a Base64 (see [RFC3548]) encoded string as well. For the definition of a binary block of data see section 2.3.18.

This is used by the **XSD-BASE64** atomic type.

2.3.17 BinHex

BinHex is used to store binary data in the text XML format. From the perspective of Binary XML this is a block of binary data. A parser SHOULD<21> report the value as binary data. In addition to that it MAY<22> choose to expose this as a BinHex-encoded string as well. For the definition of a binary block of data see section 2.3.18.

BinHex is a method for encoding binary data into a string. To encode binary data into a BinHex string a parser MUST process binary data one byte at a time starting with the first byte. For each byte a parser MUST convert the value of the byte into a hexadecimal representation using uppercase letters. A single byte will be converted into two characters from this set:

```
character =  "0" / "1" / "2" / "3" / "4" / "5" /  "6" / "7" / "8" / "9" /
             "A" / "B" / "C" / "D" / "E" / "F"
```

A parser MUST write the character representing the high 4 bits of the byte value to the string output followed by the character representing the low 4 bits of the byte value.

For example byte values %x42 %xAC %EF will produce a BinHex string "42ACEF".

This is used by the **XSD-BINHEX** atomic type.

2.3.18 Binary

Atomic types **SQL-VARBINARY**, **SQL-BINARY**, **SQL-IMAGE**, and **SQL-UDT** are all treated by Binary XML as a block of binary data. Both parser and writer MUST treat them as such and MUST NOT perform any validation on their content.

The block of binary data MUST be encoded as specified by the following grammar:

```
length = mb32
length64 = mb64
data = *byte
blob = length
data blob64 = length64 data
```

Binary blocks MUST be represented by an mb32/mb64 encoded length in bytes and then followed by the binary data itself.

A parser SHOULD<23> report the value as binary data. In addition to that it MAY<24> choose to expose this as a Base64-encoded string (see [RFC3548]) as well.

Aside from the atomic types listed above, binary large object (BLOB) is also used to store atomic types **XSD-BASE64** and **XSD-BINHEX**.

2.3.19 XSD QName

The value of the token **XSD-QNAME** is stored as a qname reference encoded as mb32. A parser MUST use the same mechanism as described in QName Reference (section 2.2.4).

This is used by the **XSD-QNAME** atomic type.

2.4 Atomic Values in Version 2

Version 2 introduced new types for dates and times. These types provide better precision over existing types for date and time and allow for specification of a time zone (offset).

If the version specified in the beginning of the input is 2, a parser SHOULD<25> recognize types described in this section. If the version specifies 1, a parser SHOULD<26> fail on these.

2.4.1 Date

```
 SqlDate = 3byte    ; unsigned little-endian integer representing                    the
number of days since 0001-1-1
```

SqlDate values MUST be within the range 0001-1-1 to 9999-12-31.

SqlDate is used by the **XSD-DATE2** atomic type.

2.4.2 DateTime2

```
SqlTime  =  (%x00-02 3byte) / (%x03-04 4byte) / (%x05-07 5byte)
```

A SqlTime value consists of a precision (first byte), which MUST be a number from 0 to 7, and 3-5 bytes of value. **SqlTime** is stored as an unsigned little-endian integer.

The value of SqlTime SHOULD<27> be a value from 00:00:00.0000000 through 23:59:59.9999999 with a variable level of fractional precision. For a given precision x, the value will represent the number of 1/10x seconds. The precision can be specified for the full range from 0 (that is, no fractions of a second) to 7 (that is, 100 ns precision). For precision 0, an integer value indicating the number of seconds since 00:00:00 will be returned. For precision 7, an integer value indicating the number of 100 ns since 00:00:00.0000000 will be returned. The value is strictly non-negative. The table below shows the number of bytes used for each precision and varies from 3 to 5 bytes.

	Time							
Precision	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Bytes	3	3	3	4	4	5	5	5

```
SqlDateTime2 = SqlTime SqlDate
```

The **SqlDateTime2** is used by the **XSD-DATETIME2** atomic type. If the SqlTime part overflows 24:00:00 the parser MUST adjust the SqlDate part accordingly.

It is also used by the **XSD-TIME2** atomic type in which case the date part MUST be equal to 1900-1-1. If the SqlTime part overflows 24:00:00 the parser MUST modify the date accordingly and thus report a date after 1900-1-1 in case the date is also reported.

2.4.3 DateTimeOffset

```
SqlTimeZone = 2byte   ; signed little-endian integer - zone in minutes SqlDateTimeOffset =
SqlTime SqlDate SqlTimeZone
```

SqlDateTimeOffset is similar to **SqlDateTime2** except that it additionally provides the time zone offset through a 2 byte signed integer. Two bytes is sufficient as an offset to specify the number of minutes from UTC and MUST be within the range of +14:00 and -14:00 hours. Also, the SqlTime portion of the data type represents the time in UTC, not local time. Since the size of the SqlTime can vary based on its precision the size of the SqlDateTimeOffset can vary from 8 to 10 bytes.

The **SqlDateTimeOffset** is used by the **XSD-DATETIMEOFFSET** atomic type.

It is also used by the **XSD-DATEOFFSET** atomic type, in which case the SqlTime portion MUST be ignored.

It is also used by the **XSD-TIMEOFFSET** atomic type, in which case the SqlDate portion MUST be ignored.

3 Structure Examples

3.1 Document

This example illustrates a simple XML document encoded in Binary XML format.

The textual XML for this example is:

```
<root>
    <?pi text?>
    <!--comment-->
</root>
```

Binary XML:

Token	Binary	Description
Signature	DF FF	
Version	01	
Encoding	B0 04	UTF-16LE code page
NAMEDEF-TOKEN 4 "root"	F0 04 72 00 6F 00 6F 00 74 00	Name "root" id 1
QNAMEDEF-TOKEN 0 0 1	EF 00 00 01	QName "root" id 1
ELEMENT-TOKEN 1	F8 01	<root>
SQL-NVARCHAR 2 "\n\t"	11 02 0A 00 09 00	new-line and tab
NAMEDEF-TOKEN 2 "pi"	F0 02 70 00 69 00	Name "pi" id 2
PI-TOKEN 2 4 "text"	F4 02 04 74 00 65 00 78 00 74 00	<?pi text?>
SQL-NVARCHAR 2 "\n\t"	11 02 0A 00 09 00	new-line and tab
COMMENT-TOKEN 7 "comment"	F3 07 63 00 6F 00 6D 00 6D 00 65 00 6E 00 74 00	<!--comment-->
SQL-NVARCHAR 1 "\n"	11 01 0A 00	new-line
ENDELEMENT-TOKEN	F7	</root>

3.2 Names

This example illustrates the way names are defined and referenced in Binary XML.

Consider the following piece of text XML:

```
<prefix:localName xmlns:prefix="ns"/>
```

The fragment of Binary XML representing this would be the following:

Binary token	Name table ID	QName table ID
NAMEDEF-TOKEN 2 "ns"	1	
NAMEDEF-TOKEN 6 "prefix"	2	
NAMEDEF-TOKEN 9 "localName"	3	
QNAMEDEF-TOKEN 1 2 3		1
ELEMENT-TOKEN 1		
NAMEDEF-TOKEN 12 "xmlns:prefix" 4		
QNAMEDEF-TOKEN 0 4 0		2
ATTRIBUTE-TOKEN 2		
SQL-NVARCHAR 2 "ns"		
ENDATTRIBUTES-TOKEN		
ENDELEMENT-TOKEN		

4 Security Considerations

None.

5 Appendix A: Product Behavior

The information in this specification is applicable to the following Microsoft products and technologies:

- 2007 Microsoft Office system
- Microsoft® SQL Server® 2005

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies the aforementioned Microsoft products' behavior is in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that these Microsoft products do not follow the prescription.

<1> Section 2: The Microsoft implementation imposes limits based on system resources like available memory.

<2> Section 2.1.1: The Microsoft implementation accepts a version value of 0 and treats it as Version 1.

<3> Section 2.1.5: The Microsoft implementation accepts a setting which specifies if the input is to be considered as a document or a fragment. If it is a document, the Microsoft implementation fails in case the root level contains more than one element, any atomic value or **CDATA**. In case of a fragment, the Microsoft implementation allows any number of elements, atomic values or CDATA sections at root level.

<4> Section 2.1.6: The Microsoft implementation accepts multiple atomic values after the **ATTRIBUTE-TOKEN**.

<5> Section 2.1.6: The Microsoft implementation reports namespace declarations which were not present in the input but would be required by a text representation of the XML as additional attributes.

<6> Section 2.1.7: The Microsoft implementation reports empty string as the namespace URI for namespace declaration attributes.

<7> Section 2.1.7: The Microsoft implementation only accepts **SQL-NVARCHAR**, **SQL-NCHAR** and **SQL-NTEXT** as the value of a namespace declaration attribute.

<8> Section 2.1.10: The Microsoft implementation does not recognize any extensions so it does not process the content of the extensions in any way.

<9> Section 2.2: The Microsoft implementation of a writer uses **FLUSH-DEFINED-NAME-TOKENS** to prevent excessive usage of memory by both writer and parser.

<10> Section 2.3.2: The Microsoft implementation only supports **mb32** and treats **mb64** as **mb32**.

<11> Section 2.3.8: The Microsoft implementation does not check for valid surrogate pairs in UTF-16LE strings.

<12> Section 2.3.8: The Microsoft implementation does not check for valid surrogate pairs.

<13> Section 2.3.10: The Microsoft implementation reports all value other than 0 as "true".

<14> Section 2.3.10: The Microsoft implementation supports all possible values and if an application asks for the value as a number it will return the actual value.

<15> Section 2.3.10: The Microsoft implementation supports all possible values and if an application asks for the value as a number it will return the actual value.

<16> Section 2.3.11: The Microsoft implementation only checks the validity of a date if an application asks for the value to be returned as a data type which it would not be able to store. Otherwise the Microsoft implementation will return the value to an application regardless of whether the value is valid or not.

<17> Section 2.3.12: The Microsoft implementation only checks the validity of a date if an application asks for the value to be returned as a data type which it would not be able to store. Otherwise the Microsoft implementation will return the value to an application regardless of whether the value is valid or not.

<18> Section 2.3.14: The Microsoft implementation returns the value rounded up so the original TimeTicks value of 1080000 is reported as time 01:00:00.000.

<19> Section 2.3.16: The Microsoft implementation returns the value as Base64 encoded string if an application asks for the value as a string data type. If an application asks for a binary data type, the Microsoft implementation returns the value as binary data.

<20> Section 2.3.16: The Microsoft implementation returns the value as Base64 encoded string if an application asks for the value as a string data type. If an application asks for a binary data type, the Microsoft implementation returns the value as binary data.

<21> Section 2.3.17: The Microsoft implementation returns the value as a BinHex encoded string if an application asks for the value as a string data type. If an application asks for a binary data type, the Microsoft implementation returns the value as binary data.

<22> Section 2.3.17: The Microsoft implementation returns the value as a BinHex encoded string if an application asks for the value as a string data type. If an application asks for a binary data type, the Microsoft implementation returns the value as binary data.

<23> Section 2.3.18: The Microsoft implementation returns the value as Base64 encoded string if an application asks for the value as a string data type. If an application asks for a binary data type, the Microsoft implementation returns the value as binary data.

<24> Section 2.3.18: The Microsoft implementation returns the value as Base64 encoded string if an application asks for the value as a string data type. If an application asks for a binary data type, the Microsoft implementation returns the value as binary data.

<25> Section 2.4: The Microsoft implementation treats the value of the **Version** field as the current state of a document and thus if you nest a Version 2 document in a Version 1 document, the rest of the parent document, after the nested document, will be treated as Version 2.

<26> Section 2.4: The Microsoft implementation treats the value of the **Version** field as the current state of a document and thus if you nest a Version 2 document in a Version 1 document, the rest of the parent document, after the nested document, will be treated as Version 2.

<27> Section 2.4.2: The Microsoft implementation does not produce values outside of the range 00:00:00.0000000 through 23:59:59.9999999, but it will accept values outside of the range as described in the INTREFERENCE:[Section 2.4.2] DateTime2 (section 2.4.2).

6 Change Tracking

No table of changes is available. The document is either new or has had no changes since its last release.

7 Index