Open GIS Consortium Inc.

Date: 2003-05-07

Reference number of this OpenGIS© Project Document: **OGC 03-002r8**

Version: 0.0.8

Category: OpenGIS© OGC Interoperability Program Report-Engineering Specification

Editor: Craig Bruce (CubeWerx Inc.)

Binary-XML Encoding Specification

Copyright notice

Warning

This document is not an OGC Standard or Specification. This document presents a discussion of technology issues considered in an Interoperability Initiative of the OGC Interoperability Program. The content of this document is presented to create discussion in the geospatial information industry on this topic; the content of this document is not to be considered an adopted specification of any kind. This document does not represent the official position of the OGC nor of the OGC Technical Committee. It is subject to change without notice and may not be referred to as an OGC Standard or Specification. However, the discussions in this document could very well lead to the definition of an OGC Implementation Specification.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type:	OpenGIS© Interoperability Program Report-Engineering Specification
Document subtype:	Critical Infrastructure Protection Initiative, Phase-1 (CIPI1)
Document stage:	Draft
Document language:	English

Contents

i. Preface

The OpenGIS Consortium (OGC) is an international industry consortium of more than 220 companies, government agencies, and universities participating in a consensus process to develop publicly available geo-processing specifications. This Interoperability Program Report (IPR) is a product of the OGC Web Services Initiative, the objective of which is to provide a vendor-neutral interoperable framework for the web-based discovery and exploitation of geo-processing functions.

The OGC Web Services Initiative is part of the OGC's Interoperability Program: a global, collaborative, hands-on engineering and testing program designed to deliver prototype technologies and proven candidate specifications into the OGC's Specification Development Program. In OGC Interoperability Initiatives, international teams of technology providers work together to solve specific geo-processing interoperability problems posed by Initiative sponsors.

ii. Submitting organizations

This draft Interoperability Program Report – Engineering Specification is being submitted to the OGC Interoperability Program by the following organizations:

CubeWerx Inc.
200 rue Montcalm, Suite R-13
Gatineau, QC, J8Y 3B5
Canada

Galdos Systems Inc.
#200 – 1155 West Pender Street
Vancouver, BC, V6E 2P4
Canada

Document Contributor Contact Points

All questions regarding this submission should be directed to the editor or the submitters:

Dr. Craig S. Bruce
CubeWerx Inc.
csbruce@cubewerx.com

Bill Lalonde
CubeWerx Inc.
wlalonde@cubewerx.com

Panagiotis (Peter) A. Vretanos
CubeWerx Inc.
pvretano@cubewerx.com

Aleksandar Milanovic
Galdos Systems Inc.
amilanovic@galdosinc.com

iii. Revision history

Date	Release	Description
2003-01-08	03-002	Initial version
2003-01-17	03-002r1	Updated content tokens to allow all literal characters, etc.
2003-01-20	03-002r2	Changed format name to “WKBXML”, updated formatting
2003-01-21	03-002r3	Changed format name to “BXML”
2003-01-24	03-002r4	Added **`ushort`** integer type, updated String Table usage
2003-03-31	03-002r5	Added Annex A with Galdos’s report on the binary-XML-encoding work item
2003-04-03	03-002r6	Added ‘**`isValidated`**’ header flag; “pre-page” formatting
2003-05-02	03-002r7	Disallowed arrays of strings; added ‘**`id`**’ field to trailer token
2003-05-07	03-002r8	Collapsed header booleans into ‘flags’ fields

iv. Changes to the OpenGIS® Abstract Specification

No revisions to the OGC Abstract Specification are required.

v. Changes to OpenGIS® Implementation Specifications

As noted in Clause 9.2, revisions are suggested for the GML feature-encoding format. The GML format is defined in OGC document 02-069 and others. Specifically, the encoding of coordinate values needs to be simplified to using a list of numbers as defined by XML Schema for efficient number-array storage in Binary XML.

Foreword

Attention is drawn to the possibility that some of the elements of OGC 03-002r8 may be the subject of patent rights. Open GIS Consortium Inc. shall not be held responsible for identifying and or all such patent rights.

This edition cancels and replaces the previous edition (OGC 03-002r7), which has been technically revised.

Introduction

GML (Geography Markup Language) [GML], and other scientific-data formats, as presently encoded using plain-text XML [XML] have three major performance problems: the text in the XML structure is highly redundant and bulky, making it slow to transfer over the Internet; the lexical scanning of XML (turning free text into tokens for internal processing) is unexpectedly costly; and the conversion of text-encoded numerical coordinate and observation values is also very costly.

The bulkiness of GML encoding can be reduced by using general compression methods such as GZIP [GZIP], but this does not address the scanning and numeric-conversion costs. XML generally compresses quite well, since it is so redundant; repeated blocks of text (such as closing tags) compress down to practically nothing. However, scanning-cost and further space reductions can be achieved by using a binary encoding method for the XML representation plus a minor change to the method for specifying coordinate values.

The binary encoding method mirrors the typical in-memory representation of XML as nodes in a parse tree by representing the stream as a sequence of node-equivalent 'tokens'. There are tokens defined to represent element openings, closings (necessary here unlike in a tree), empty elements, in-line content, entity references, special tags, and even comments. The byte lengths of various structures are given in advance at the head of their byte sequences for efficient processing, and element/attribute names are represented as integer indexes into a global symbol/string table, for both space and time efficiency.

There are also special, efficient representations available for content and attribute values that represent numbers or arrays of numbers. This is crucially important to use with GML for coordinate values. The existing GML representations of using special-character-delimited sequences of textual numbers or of using hierarchical structures for coordinate values should either be scrapped or be augmented by using an XML-Schema [XMLSCHEMA] '**`list`**' representation of '**`double`**' values. This type can be represented and processed very efficiently using a raw array of IEEE '**`double`**' floating-point numbers.

This raw list can be read straight into an array in memory and be used directly by all modern processors with at most a swapping of the byte order (endian) of the values. No costly parsing or conversion of the coordinate values is necessary with an efficient programming language. This direct numeric representation by itself would make GML practically as efficient to use as any other feature-encoding format, such as ESRI® Shapefile format [SHAPE].

The binary-encoding method can also directly represent raw binary data without any indirect textual-encoding methods (such as base64), and a backward-compatibility mechanism is provided to enable the translation raw-binary blocks into an equivalent textual representation when necessary. Binary XML also can be compressed using

general compression methods, though it will already be significantly smaller and less redundant that text encoding.

This binary-encoding method is applicable to all XML documents and not just GML and scientific-data formats, though many XML documents are not necessarily bulky enough to benefit greatly from binary encoding. The binary encoding is directly equivalent to the textual encoding and it is possible to translate any lone XML document to and from the binary representation with no loss of information. The binary stream is also parseable and generable sequentially on-the-fly, as textual XML is, but optional indexing and direct-random-access capabilities (e.g., of "**`ID`**" attributes) are also available.

Binary-XML Encoding Specification

1 Scope

This OpenGIS® Interoperability-Program report specifies a binary encoding format for the efficient representation of XML data, especially scientific data that is characterized by arrays of numbers. This encoding format is applicable to any application that uses XML format.

2 Conformance

Not required in an IP DIPR, IPR or Discussion Paper.

3 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this Interoperability Program Report. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this document (OGC 03-002r8) are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies.

[GML] OGC 02-069 (2002), *OpenGIS® Geography Markup Language (GML) Implementation Specification, version 2.1.2*, <http://www.opengis.net/gml/02-069/GML2-12.pdf>.

[XML] W3C (October 2000), *Extensible Markup Language (XML) 1.0 (Second Edition)*, <http://www.w3.org/TR/REC-xml>.

[GZIP] IETF RFC 1952 (May 1996), *GZIP File Format Specification Version 4.3*, L. Peter Deutsch, <http://www.ietf.org/rfc/rfc1952.txt>.

[IEEE] IEEE 754-1985 (1985), *Standard for Binary Floating-Point Arithmetic*, <http://grouper.ieee.org/groups/754/>.

[XPATH] W3C (November 1999), *XML Path Language (XPath)*, Version 1.0, <http://www.w3.org/TR/xpath>.

4 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

4.1
byte
a group of eight bits usually processed as the basic unit of addressable memory in a computer or in a data file; this is synonymous with the term “octet”

4.2
endian
the byte order of encoding for multi-byte numerical values, normally described as “big” or “little,” indicating that the most-significant or least-significant byte appears first, respectively

4.3
whitespace
defined in XML 1.0 [XML] as the Unicode characters **`#x20`** (Space), **`#x9`** (TAB), **`#xD`** (Carriage Return), **`#xA`** (Line Feed), or sequences or combinations thereof

4.4
blob
a “binary large object”, i.e., a opaque sequence of bytes of arbitrary length

4.5
BXML
shorthand for “Binary XML”, the format defined in this document

4.6
token
an encoded data structure in the BXML format

4.7
reader
(unqualified) a parser or application that reads a BXML data stream

4.8
writer
(unqualified) an application that writes a BXML data stream

4.9
translator
(unqualified) an application that reads either textual XML or BXML and writes the opposite format

5 Conventions

5.1 Requirement levels

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [WORDS].

5.2 Symbols (and abbreviated terms)

The following symbols and abbreviations are used in this document:

API	Application-Program Interface
ASCII	American Standard Code For Information Interchange
BXML	Binary Extensible Markup Language
C/C++	C and/or C++ programming languages
DTD	Document Type Definition
GML	Geography Markup Language
GNU	“GNU’s Not Unix”, the Free Software Foundation
GZIP	GNU Zip compression format
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
MIME	Multipurpose Internet Mail Extensions
PNG	Portable Network Graphics
RFC	Request For Comments
URL	Uniform Resource Locator
UTF-8	Universal Character Set Transformation Format 8(-bit)
UTF-16	Universal Character Set Transformation Format 16(-bit)
WAP	Wireless Application Protocol
WFS	Web Feature Service
WMS	Web Map Service
XML	Extensible Markup Language
ZLIB	Compression Library

5.3 Binary-structure declarations

The binary structures in this document are described using a declaration language modelled after C/C++ and derived programming languages. The declarations have the following forms for structures, enumerated data types, and union data types:

```
structure_name {
    data_type   field_name;      // optional comment
    data_type   array_name[array_size];
    …
}

integer_type enum enum_name {
    identifier1 = value1,        // optional comment
```

```
    …
}

union union_name {
    sub_type1,                  // optional comment
    …
}
```

This format is well-suited toward describing binary structures and should be obvious to anyone who writes software. The fields in the structure definition are byte-packed; i.e., fields are not aligned on word boundaries. Unions are byte-packed for the sub-type used.

6 WAP binary XML

A binary encoding for XML was developed for the WAP environment [WAPXML]. It was designed for use in a limited environment and has some design problems with primitive encoding, content encoding, and tag representation.

Numeric and string primitives are encoded in awkward ways that make them less efficient to parse. Only the representation of unsigned integers is provided and they are encoded with a variable-length string of only seven active bits within bytes. The most-significant bit of each byte is used as a continuation marker from one byte to another. This is inefficient because the input stream must be scanned a single byte at a time to extract the value and it is awkward to handle since the seven bits that constitute part of the result number must be extracted and shifted into position in the result in a bit-wise fashion. A more efficient method would identify the type of number being encoded up-front and would then represent the number in its common raw-binary format. More numeric types than just unsigned integers are also needed, plus arrays of numbers.

Strings are represented as a raw sequence of bytes with no length or standard terminator. This information must be interpreted from whatever character-encoding method is used. For example, with UTF-8, a null-terminator character (single-byte code 0x00) is used to delimit the end of a string. In addition to being awkward to deal with, this method is inefficient to process since the incoming stream must be scanned a single character at a time in order to locate the terminator for each string. A more efficient method would be to encode the byte length before the string so that the string content can be handled as a contiguous block.

All content in WAP is represented as strings (or equivalently as string references). This does not facilitate efficient processing of GML coordinate or observation values.

Tags in WAP-XML are may be represented as either literal strings or equivalently as string references. Using string references allows more efficient processing, but the mandatory use of a special namespace of element/attribute identifiers would be more efficient for parsers.

Finally, the WAP-XML specification does not address the issue of compression within itself. Compression is an important issue for efficient data transfer over the Internet.

7 Design overview

This section discusses important overall design issues.

7.1 File structure

The representation used in BXML consists of a header followed by a stream of tokens. The header contains the information necessary to identify and process the stream, and the tokens encode the content in discrete "packets" of information. These tokens are sequenced in an order that makes the format "streamable"; i.e., so that it can be read and processed properly in sequential order. This is important since data is normally passed as a sequential stream of bytes over the Internet.

However, random access to the BXML data can be very useful also. For example, if an XML file is used as a "database" of objects of some type with an "**`ID`**" attribute used as a primary key for the objects, then if only a few objects need to be accessed for a particular application, it will be greatly more efficient to randomly seek directly to the data tokens that encode those objects and to ignore the rest of the file.

It is also crucially important for a BXML file to be able to be translated to and from the textual XML format with no loss of information. Various systems may support only the textual XML format, so some mechanism will need to be employed to translate a BXML file before it can be processed. Consequentially, a BXML file must also be able to stand alone, in the same way that a textual XML file can stand alone—not in the validation sense, but in the functionality sense. The file must be parseable, by a non-validating parser at least, without dependence on external files or definitions. A sensible way to achieve this is to have a one-to-one correspondence between BXML content and XML constructs. This also allows parsers to more easily support both formats. However, some of the unfortunate compromises in textual XML with respect to character encoding and escaping are alleviated.

7.2 The war of the endians

The term "endian" comes from the Jonathan Swift novel *Gulliver's Travels* [SWIFT] which contains a segment that describes a war between two peoples who could not agree whether to break open soft-boiled eggs starting with the big end or the little end. That war continues today over the byte ordering of data words within computers, since different processor manufacturers use different byte-order endians.

For BXML, the endian issue boils down to a matter of theory versus practice. In theory, BXML should be defined to be big-endian only since this was selected as the "network order" presumably because it is easier for a human to read in a hexadecimal dump, if not because the computers that were first networked together happened to be big-endian. This would slightly simplify the format. However, in practice, probably most of the computers connected to the Internet are Intel®/AMD® x86-based Personal Computers, and always requiring the translation to and from big endian on these machines will make applications run less efficiently than they otherwise would.

The compromise selected for BXML format and used in other formats such as OGC WKB Simple Features [WKB-SF] is to allow a writer of the format to use either endian and to require the reader of the format to be able to handle both endians. An indicator is written into the header of the file to tell which endian is used. This allows greater efficiency to in the common case that a BXML file is shared between two machines that use the same endian, which may be quite likely within an organization and will always be the case when a file is generated and consumed on the same machine. The LAN and local-machine cases are especially important to consider for this issue since the file will be transferred the most efficiently in this environment, making endian conversion have a greater relative impact on performance.

7.3 Compression

Compression of data is a very important capability for a data format that is intended to be streamed over the Internet. This is because the bandwidth of the an Internet connection frequently will be a major bottleneck to performance in distributed data processing, and data can be compressed and decompressed at a low relative cost with modern computers.

However, the difficult issue to decide is whether to specify that an external compression method optionally may be applied to a BXML stream, or whether the compression should be built-in. With HTTP and perhaps other communication protocols, streamed data optionally may be compressed if the client requests compression and the server supports compression. However, in practice, this compression seems to be available and applied only haphazardly, and the benefits usually are not realized. Also, if a data source is relatively static (written once and read many times), the compression step, if available, would be applied many times instead of only once.

For these reasons, optional built-in compression is selected for use with BXML. Writers may optionally support compression, but all readers are required to support decompression. However, a compressed BXML file is implemented as a non-compressed header followed by a compressed stream of bytes for the body. This makes it easy for a reader that does not implement decompression directly to split off the BXML header and then run the file body through an external application that supports the compressed-stream format.

Non-compressed BXML files are also allowed so that files may be accessed randomly and because compression will likely make processing less efficient in a LAN environment, because the network will be much less of a performance bottleneck, and processing will definitely be less efficient with a local file.

The GZIP compression format [GZIP] is selected for use for the file-body compression stream. GZIP is a ubiquitous, open, free, and patent-unencumbered general-purpose compressed-data format. The format is supported by the ZLIB software library [ZLIB], which is a multi-platform, multi-computer-language, open-source, and free implementation of the GZIP format and the “Deflate” compression algorithm that is used within GZIP, ZIP, PNG, and other formats. All BXML readers are required to support this compression format by some method.

7.4 String table

There can be little doubt that a string/symbol table is very well suited for use with an XML document. A great deal of lexical-scanning time is spent extracting and comparing symbol strings (e.g., element names) from text-based XML, and these symbols are also responsible for much of the bulk of the format. The use of a string table allows all of the element and attribute names to be defined in an array and for the symbols to be referenced by merely using the integer index of the symbol string in the string table. The use of an array also allows a reader (and a writer) to efficiently associate auxiliary information with the symbol names, such as a reference to an object that represents an XML namespace that is included in the symbol string or parsing/validation-state information.

In addition to providing element, etc. symbol names, general strings that are repeatedly used in textual content also easily can be stored into the global string table. This allows the uncompressed document to be smaller and perhaps to be processed more efficiently. It is up to the discretion of the BXML writer which strings to use by reference and which to provide in-line. WAP-XML also provides a general-string-table mechanism.

A vexing issue with the use of a string table in a streaming format is whether to require that the string table to placed in its entirety at the front of the file stream, or whether to allow strings to be defined throughout the stream before they are used. The former approach simplifies readers and allows random access to be supported more easily because all of the strings will be defined up-front. The latter approach simplifies writers, since the symbols/strings that can be or will be used may not be known in the programming environment in advance.

The spread-out string-table approach is selected for use in BXML. However, it is required that the string table be (optionally) randomly accessible so that string references encountered when accessing a BXML file randomly may be resolved without first scanning the file sequentially (thereby defeating the benefits of random access). Therefore, in BXML files that are designated as being randomly accessible, an index of all of the fragments of the string table is included so that the string-table references can be resolved in random order. They may be resolved all at once or as needed.

7.5 XML-content-string limitations

Textual XML defines that various character strings must not contain certain literal characters or sequences because they are used as markup characters in the same context. The characters and sequences with special limitations include "**<**", "**>**", "**&**", "**]]>**", "**"**", and "**'**" in various kinds of content. These literal characters must be specially escaped when they appear in content, such as with "**<**" for "**<**". The sequence of two hyphens ("**--**") is also disallowed in comments.

It has been decided that it would be awkward and wasteful to obey all of these unfortunate allowed-character constraints that XML places on its content structures. Textual XML imposes these constraints to allow it to tokenize the text stream, but this consideration is

irrelevant in BXML because **`String`** content is length-encoded and the stream is tokenized in an "out-of-band" fashion. Textual XML uses entity references to encode these special characters, but it is recognized here that these entities are semantically no more significant than just escape sequences, so the literal characters are allowed in BXML content tokens, with a few considerations. Really, both reader and writer applications work more efficiently with the literal characters than with entities. However, the equivalent entity structures still may be used at the discretion of the writer.

However, there may be an issue if textual XML needs to be regenerated, then content strings will need to be scanned by the translator and special characters will need to be escaped using the entity-encoding method wherever the textual-XML definition requires it. While this string scanning will make the process less efficient, this overhead will be insignificant compared to the costs of outputting the textual-XML representation and most likely re-parsing it.

On the other hand, if a traditional API is supplied on top of a BXML stream, it may not be known whether that application that is making use of the API needs to have special characters appear as entity constructs or whether it can handle literal strings. For most applications, it will not matter, but if it does matter, then a special flag is provided in the file header to indicate whether content strings in the BXML file conform to the textual-XML constraints or whether they will need to be scanned for special characters.

8 Representation

This section defines the byte-for-byte representation of BXML files.

8.1 Data-value encoding

The following primitive numeric types are used in this specification:

Name	No. of Bits	Base Type	Range
`byte`	8	unsigned integer	0 to 255
`short`	16	signed integer	-32768 to 32767
`ushort`	16	unsigned integer	0 to 65535
`int`	32	signed integer	-2^{31} to $2^{31}-1$
`long`	64	signed integer	-2^{63} to $2^{63}-1$
`float`	32	floating-point	IEEE 32-bit-float range
`double`	64	floating-point	IEEE 64-bit-float range

These primitive data types are very common in binary-data encoding and reflect what modern computers use internally. The signed integers all use 2's-complement encoding for negative numbers and the floating-point numbers use IEEE-754 representation [IEEE].

Arguably, unsigned versions of 32 and 64-bit integers could also be useful, but their marginal utility in this environment is not considered to be significant enough to justify the additional complexity they would impose on BXML readers. Also, in instances where a larger number

type must be used, such as using a 64-bit **long** to represent an unsigned 32-bit value such as **3000000000**, the 'unused' bits in the larger integer will be highly compressible. An unsigned version of the 16-bit integer, **ushort**, is provided since this type is commonly used for sample/observation values in scientific and imagery data, (along with **byte**, **short**, and **float**).

In many places, the type of data that is present must be identified. This is accomplished by preceding the value with a single-byte code defined as follows:

```
byte enum ValueType {     // value-type identifier codes
    BoolCode    = 0xF0,   // boolean value
    ByteCode    = 0xF1,   // 'byte' numeric value
    ShortCode   = 0xF2,   // 'short' numeric value
    UShortCode  = 0xF3,   // 'ushort' numeric type
    IntCode     = 0xF4,   // 'int' numeric value
    LongCode    = 0xF6,   // 'long' numeric value
    FloatCode   = 0xF8,   // 'float' numeric value
    DoubleCode  = 0xF9,   // 'double' numeric value
    StringCode  = 0xFA,   // character string
    ArrayCode   = 0xFB,   // string of scalar values
}
```

Value-type code values **0xF5**, **0xF7**, and **0xFC** to **0xFF** are reserved for future expansion. Codes from **0x00** to **0xEF** are reserved for use with the **SmallNum** type defined below.

A Boolean type is defined as being a single byte as follows:

```
byte enum Bool {                  // raw boolean value
    FALSE = 0x00,
    TRUE  = 0x01
}
```

In cases where the type must be identified (in user data), the following structure is used:

```
BoolValue {                       // type-identified boolean value
    ValueType type = BoolCode;    // value-type identifier code
    Bool      bool;               // literal value
}
```

In cases where the type must be identified for integer types, the following definitions are used:

```
ByteNum {                         // 8-bit unsigned integer value
    ValueType type = ByteCode;    // value-type identifier code
    byte      num;                // number
}

ShortNum {                        // 16-bit integer value
    ValueType type = ShortCode;   // value-type identifier code
```

```
    short     num;                // number
}

UShortNum {                       // 16-bit unsigned integer value
    ValueType type = UShortCode; // value-type identifier code
    ushort    num;                // number
}

IntNum {                          // 32-bit integer value
    ValueType type = IntCode;    // value-type identifier code
    int       num;                // number
}

LongNum {                         // 64-bit integer value
    ValueType type = LongCode;   // value-type identifier code
    long      num;                // number
}
```

Since the numbers that will be used in general most frequently will be small numbers, the following type is defined:

```
SmallNum {                        // compact, limited-range int
    byte      num;                // number: only values 0 to 239
}
```

Only values from **0** to **239** (**0x00** to **0xEF**) are allowed to be used and these values double as the value-type-identification code relative to the **ValueType** enumeration: codes **0x00** to **0xEF** identify the **SmallNum** type. This allows type-identified small numbers to have a compact, single-byte, literal representation.

The **Count** type is used to represent object counts, lengths, sizes, and file offsets in many places. It is defined as:

```
union Count {   // numeric value type for counts, offsets
    SmallNum,   // negative numbers are invalid
    UShortNum,
    IntNum,
    LongNum
}
```

Eight exabytes ought to be enough for anybody. This definition comprises various type-identified integer types to allow a compact representation of count values. The type used is identified by the first byte, and small numbers will only require a single byte. However, there is no requirement that the minimal-space sub-type be used. The **ByteNum** type was excluded from this definition to simplify readers, since it does not add sufficient compactness beyond the **SmallNum** type to justify its use. Negative numbers are not valid to be used with this type.

Type-identified versions of the floating-point numeric types are defined as follows:

```
FloatNum {                        // single-precision floating point
    ValueType type = FloatCode;   // value-type identifier code
    float     num;                // number
}

DoubleNum {                       // double-precision floating point
    ValueType type = DoubleCode; // value-type identifier code
    double    num;                // number
}
```

Character strings are represented in BXML as follows:

```
String {                        // raw character string
    Count  byteLength;          // length in bytes
    byte   chars[byteLength]; // characters in proper encoding
}
```

The byte length is given up front so that the data may be efficiently processed as a block. The length gives the raw number of bytes used by the string, which is not necessarily equal to the number of characters present. The characters are represented using the character-set-encoding definition that is supplied in the BXML file header. No termination character is required to be present in the string, but any new-line characters or sequences, such as CR-LF (**#xD #xA**), must be normalized to a single line-feed (**#xA**) character. This normalization is defined in the XML 1.0 specification for use with all XML processors, and adding the requirement to all BXML writers allows readers to be more efficient.

The type-identified version of the string type is:

```
StringValue {                      // type-identified string value
    ValueType type = StringCode;  // value-type identifier code
    String    string;             // string value
}
```

Arrays of values are allowed for user content. The **`Array`** type is defined as:

```
Array {                              // array of numbers/strings
    ValueType    type = ArrayCode;  // type id for this object
    ValueType    elementType;       // type of elements
    Count        length;            // length of array
    ArrayElement elements[length];  // elements of array
}

union ArrayElement {   // types allowed as an array element
    Bool,
    byte, short, ushort, int, long, float, double
}
```

The **`String`** type has been intentionally left out of the **`ArrayElement`** definition because arrays of strings are infrequently used in practice and the inclusion of this type would

unnecessarily complicate BXML processors, since strings are significantly structurally different from numbers and booleans. The **elementType** is given in the **Array** structure so that raw primitive types may be used for the elements instead type-identified values. This saves one byte per element but also allows the array contents to be processed as a single block. In some programming languages, such as C/C++, a numeric file array can even be directly read into or written out of an array as used in the language, for exceptional efficiency. The array-encoding structure also combines nicely with the XML-Schema [XMLSCHEMA] "**list**" type for representing XML content.

The **Value** type encodes generic, type-identified user data and is defined as follows:

```
union Value {      // union of all user-data 'value' types
    BoolValue,
    SmallNum,
    ByteNum, ShortNum, UShortNum, IntNum, LongNum,
    FloatNum, DoubleNum,
    StringValue,
    Array
}
```

8.2 File structure

A BXML file is encoded simply as a fixed-format header followed by any number of token objects:

```
BXMLFile {             // BXML file format
    Header  header;    // file header
    Token   tokens[];  // sequence of tokens
}
```

The last token must be a special **TrailerToken**, which is defined in Clause 8.11.

8.3 Header

The file header is used to identify the type of the file and provide the critical information necessary for a reader to process the file. It has the following structure:

```
Header {
    Identifier  identifier;          // file-format identifier
    Version     version;             // BXML version number
    byte        flags1;              // header bit flags
    byte        flags2;              // more header bit flags
    Compression compression;         // file-body compression
    String      charEncoding;        // character encoding used
}
```

The **identifier** portion of the header identifies the file as being of the BXML type and is defined as:

```
Identifier {                                        // format identifier
    byte  nonText = 0x01;                           // ascii SOH
    byte  name[5] = { 'B', 'X', 'M', 'L', 0x00 };
    byte  binaryCheck[3] = { 0xFF, 0x0D, 0x0A }; // high-bit, CR+LF
}
```

Many systems check the identification portion or "magic number" of a file to determine or verify its type. The design here was inspired by the PNG image-format [PNG] identification section. The first byte is used to help insure that the file is not mistaken for a text file. The apropos ASCII SOH or "Start of Header" character code is used for this purpose. The next five bytes identify the file type with a human-readable string. The trailing zero is used to help prevent name conflicts with other potential formats (in the event that a different file format had "BXML" as a prefix of its full name). Finally, the **`binaryCheck`** field helps to insure the early detection of a file that has been mangled by being improperly processed and translated in a text or a 7-bit file mode. This check is especially important with BXML since it will likely be used in conjunction with the plain-text XML format.

The version of the BXML file is identified using three sequential bytes in the following structure:

```
Version {              // version of BXML (not XML)
    byte  major = 0;   // x.y.z (0-255 for each component)
    byte  minor = 0;
    byte  point = 8;
}
```

This identifies the version of the BXML file format and not the version of the XML content (which is identified in the **`XmlDeclarationToken`**). The BXML version for this experimental stage is "0.0.8". The values are coded simply as raw bytes. A BXML reader must reject any file with a version that it does not specifically support, since the BXML structures and semantics may change arbitrarily between versions. In fact, even the header format can change arbitrarily between versions, so only the **`identifier`** and **`version`** fields of the header should be checked to determine compatibility. No specific mechanism is included in this specification to help a writer to select a version that a reader is known to support.

The **`flags1`** and **`flags2`** fields encode various control-information bit flags for the BXML file. The bits of the **`flags1`** field have the following meanings:

Name	Bit	Pattern	True / 1	False / 0
`isLittleEndian`	**`0`**	**`0x01`**	little endian	big endian
`charsAreLittleEndian`	**`1`**	**`0x02`**	little endian	big endian
`hasRandomAccessInfo`	**`2`**	**`0x04`**	random access	not random access
`hasStrictXmlStrings`	**`3`**	**`0x08`**	strings are strict	strings are unrestricted
`isValidated`	**`4`**	**`0x10`**	XML is validated	XML is not validated

Bit positions **5** (**0x20**), **6** (**0x40**) and **7** (**0x80**) of **flags1** are presently unused and must be assigned the value of zero. The **flags2** field is presently completely unused and all bits must be set to zeros.

The **isLittleEndian** bit flag is used to specify the endian (byte order) that is used for all multi-byte binary numbers throughout the file except for multi-byte character values. A value of **1** means that little endian is used (least-significant byte first) and a value of **0** means that big endian is used (most-significant byte first).

The **charsAreLittleEndian** bit flag indicates the 'default' byte order for multi-byte character codes, such as UTF-16. A value of **1** means that little endian is used and **0** means big endian. Some character encoding schemes may have internal byte-order specifications that override this default value, and others such as ISO-8859-1 do not require a byte order, in which case this value may be ignored.

The **hasRandomAccessInfo** bit flag indicates whether the requisite information is available for a reader to process the file (or the uncompressed version of the file) in a random-access fashion. A value of **1** indicates that at least the string-table index is set up in the **TrailerToken**, which is defined in Clause 8.11. A value of **0** indicates that random-access information is unavailable.

The **hasStrictXmlStrings** bit flag indicates whether all strings which contain content or comments contain strictly what textual XML allows them to contain (value of **1**) or whether they may contain any and all characters and sequences (value of **0**). This issue is discussed in Clause 7.5.

The **isValidated** bit flag indicates whether or not the XML content of the document has been positively validated to be syntactically correct relative to the XML-Schema or other definition of its format. This flag can be used as an optimization to avoid validating the document more than once.

The **compression** type is specified in the header using the following code values:

```
byte enum Compression {    // compression code value
    NoCompression = 0x00,  // no compression is used
    GZIP          = 0x01   // everything after header is GZIP stream
}
```

In the case of **NoCompression**, the sequence of tokens that constitute the body of the document shall be encoded literally as is specified in the following sections. In the case of **GZIP** compression, the content shall be encoded as a complete GZIP-compressed [GZIP] byte stream of the tokens, starting at the first byte after the file header. If file offsets are specified within the compressed body, the file offsets shall refer to the file positions that would be used in an uncompressed file. If a compressed file is to be used for random access, it will need to be uncompressed first.

The **charEncoding** field of the header identifies the character-set encoding that is used for all of the **String**s in the file with the exception of this string itself. This string is encoded in US-ASCII format. This string also supplies the value for the implied **encoding** attribute of the equivalent structure to the XML "**<?xml…?>**" construct, **XmlDeclarationToken**. A typical value for this string will be "**UTF-8**". Note that the inclusion of a **String** field makes the file header variable-sized.

8.4 Tokens

Tokens are used to encode the content of the BXML file in discrete "packets" that correspond roughly to XML markups. The type-identification codes for the various token types are as follows:

```
byte enum TokenType {                   //  token-type code value
    EmptyElementCode       = 0x00,      //  <element/>
    EmptyAttrElementCode   = 0x01,      //  <element .../>
    ContentElementCode     = 0x02,      //  <element> ...
    ContentAttrElementCode = 0x03,      //  <element ...> ...
    ElementEndCode         = 0x04,      //  </element>
    AttributeStartCode     = 0x05,      //  attr="
    AttributeListEndCode   = 0x06,      //  end of attributes
    CharContentCode        = 0x10,      //  character content
    CharContentRefCode     = 0x11,      //  string-table char content
    CDataSectionCode       = 0x12,      //  <![CDATA[content]]>
    WhitespaceCode         = 0x13,      //  whitespace character content
    BlobSectionCode        = 0x14,      //  raw-binary data
    EntityRefCode          = 0x15,      //  &entity_ref;
    CharEntityRefCode      = 0x16,      //  &#char_ref;
    CommentCode            = 0x17,      //  <!--comment-->
    XmlDeclarationCode     = 0x20,      //  <?xml ...?>
    BangCode               = 0x21,      //  <!name ...>
    BangBracketCode        = 0x22,      //  <![name[...]]>
    ProcessingInstrCode    = 0x23,      //  <?name ...?>
    StringTableCode        = 0x30,      //  string table (fragment)
    IndexTableCode         = 0x31,      //  index table
    TrailerCode            = 0x32       //  trailer
}
```

The numeric-code values are separated into groups so that any code values added in the future may appear close to the other members of their logical group.

The union of all of the tokens (referenced in the file-format definition) is:

```
union Token {    // union of all tokens
    EmptyElementToken, EmptyAttrElementToken, ContentElementToken,
    ContentAttrElementToken, ElementEndToken, AttributeStartToken,
    AttributeListEndToken, CharContentToken, CharContentRefToken,
    CDataSectionToken, WhitespaceToken, BlobSectionToken,
    EntityRefToken, CharEntityRefToken, CommentToken,
```

```
    XmlDeclarationToken, BangToken, BangBracketToken,
    ProcessingInstrToken, StringTableToken, IndexTableToken,
    TrailerToken
}
```

8.5 Elements and attributes

Elements and attribute definitions are split into the pieces that normally represent nodes in a tree representation of an XML document. The element and attribute definitions are modelled after the WAP-XML representation. An element starts with one of: **`EmptyElementToken`**, **`EmptyAttrElementToken`**, **`ContentElementToken`**, or **`ContentAttrElementToken`**. The type of token identifies whether there are attributes or content present within the element. The least-significant two bits of the code value can be used to identify whether the content and/or attributes are present.

8.5.1 Empty-element token

The empty-element (with no attributes) token is defined as:

```
EmptyElementToken {              // <element/>
    TokenType  type = EmptyElementCode; // token-type code
    Count      stringRef;        // symbol code for element name
}
```

This token is equivalent to the XML empty-element markup, e.g., "**`<blah/>`**" (without quotations). This token is self-complete and does not need to be followed by any other specific tokens. A symbol-string reference is used instead of a literal element name for space and processing efficiency. The string reference is an index into the global string table, where index values start from zero. Element-name values must conform to XML constraints.

8.5.2 Empty-element-with-attributes token

The empty-element-with-attributes token is defined as:

```
EmptyAttrElementToken {          // <element ... />
    TokenType  type = EmptyAttrElementCode; // token-type code
    Count      stringRef;        // symbol code for element name
}
```

This token is equivalent to the XML empty-element markup, e.g., "**`<blah attr="value"/>`**". This token must be followed by a list of at least one **`AttributeStartToken`** plus associated content, and the attribute list must be terminated by an **`AttributeListEndToken`**, or, optionally, a **`StringTableToken`** may precede any **`AttributeStartToken`**, to simplify string-table handling in writers.

8.5.3 Content-element token

The content-element (without attributes) token is defined as:

```
ContentElementToken {            // <element>
    TokenType  type = ContentElementCode; // token-type code
    Count      stringRef;        // symbol code for element name
}
```

This token may be followed by any number of content tokens and embedded sub-elements, including zero, and must be terminated by an **`ElementEndToken`**.

8.5.4 Content-element-with-attributes token

The content-element-with-attributes token is defined as:

```
ContentAttrElementToken {        // <element ...>
    TokenType  type = ContentAttrElementCode; // token-type code
    Count      stringRef;        // symbol code for element name
}
```

This token must be followed by a list of at least one **`AttributeStartToken`** plus associated content (with optional **`StringTableToken`** prefixes), and the attribute list must be terminated by an **`AttributeListEndToken`**. The attribute list may be followed by any number of content tokens and embedded sub-elements, including zero, and the token sequence must be terminated by an **`ElementEndToken`**.

8.5.5 Element-end token

The element-end token is defined as:

```
ElementEndToken {                    // </element>
    TokenType  type = ElementEndCode;  // token-type code
}
```

This token is equivalent to the XML element-closing markup, e.g., "**`</blah>`**" and is used to terminate only element-token types that explicitly include content, i.e., **`ContentElementToken`** and **`ContentAttrElementToken`**.

The element-name equivalent is omitted since it is redundant in textual XML and it is not useful in a binary environment, since humans are unlikely to edit binary XML by hand. WAP-XML also omits the element name in its equivalent construct.

8.5.6 Attribute-start token

The attribute-start token is used to define an attribute and defined as:

```
AttributeStartToken {       // attr="
    TokenType  type = AttributeStartCode; // token-type code
    Count      stringRef;   // symbol code for attribute name
}
```

This token is equivalent to the starting portion of the XML attribute-definition markup, e.g., "**attr="**". The attribute name is referenced from the global string table for efficiency. This token must be followed by some number of content tokens, including zero, that define the attribute value. If the **strictXmlStrings** header flag is set, then this token is interpreted to enclose the content in the double-quotation character (**"**) and the attribute content therefore must not include this character literally (an entity reference must be used instead). If the header flag is not set, then character content may include any literal characters. If textual XML is regenerated from the attribute content with a non-strict string, the translator will need to select a quotation character to use to embed the content (the double-quotation mark is suggested) and it will need to escape any instances of that character in the attribute content using "**"**" or "**'**" as appropriate.

The attribute-value-definition content tokens must be followed by either another **AttributeStartToken** to continue the attribute list or an **AttributeListEndToken** to terminate the attribute list of an element. However, as noted elsewhere, any **AttributeStartToken** may be directly preceded by a **StringTableToken**, to simplify writers.

8.5.7 Attribute-list-end token

The attribute-list-end token is used to terminate the attribute list of an element and is defined as:

```
AttributeListEndToken {              // end marker of attribute list
    TokenType  type = AttributeListEndCode;  // token-type code
}
```

8.6 Content Representation

Several token types are used to represent various types of literal XML content. A content segment of XML may be represented by any sequence and combination of these content tokens. The validity of a sequence of content tokens within a BXML file is defined as the validity of the textual equivalent to the tokens as it would appear in textual XML relative to the content-type definition in XML-Schema or other definition languages.

It is recognized that the ideal processing environment for BXML is one in which binary content is passed directly from generator to parser to application without ever being translated into text in between. Therefore, it is recommended that the XML parser scan and store equivalent structures to the content tokens defined in this section and that it avoid translating numbers and "blobs" into a textual equivalent if at all possible in passing the content information to the reading application.

8.6.1 Character content token

The token that corresponds to regular textual XML content is defined as:

```
CharContentToken {                      // regular character content
    TokenType type = CharContentCode;   // token-type code
    Value     content;                  // single content value
}
```

The **content** field can include any **Value** sub-type, including numbers and arrays. These special types are considered to be equivalent to the character representation for the purpose of XML validation. If the **strictXmlStrings** flag in the header is set, then the content must conform to textual XML constraints, which means that the literal characters "**<**" and "**&**" must not appear in the string and the sequence "**]]>**" must have its final "**>**" character changed. These characters must be generated using entity references. Otherwise, if the flag is not set, then the content may include any and all literal characters. Leading and trailing whitespace characters that are included in a string may be considered to be significant by a reader.

However, the representation of numbers and especially arrays of numbers can be much more efficient than the equivalent text, and it also can be processed much more efficiently if the parser and the application are capable of carrying the raw numeric representation through the parsing and interpretation process.

The representation of numeric values used in content is independent of the DTD/XML-Schema/RDF-Schema that defines a format. The most efficient or convenient representation of a numeric value may be chosen by the writer. For instance, if XML Schema defines the content to be a list of **double** numbers, an array of **float** values may be used instead for greater efficiency if the data will be adequately represented as **float**s. The content could also be provided as a character **String**, an array of **byte**s (if sufficient), or as being spread out over multiple content tokens.

8.6.2 Character-string-reference content token

The character-string-reference content token is defined as:

```
CharContentRefToken {                       // character content
    TokenType type = CharContentRefCode;    // token-type code
    Count     stringRef;                    // string-table ref
}
```

This token is equivalent to the **CharContentToken**, except that instead of using content that is stored in-line, it refers a string that is stored in the global string table. This offers greater compactness than the **CharContentToken** for representing content strings that are repeated very frequently in a document (by consuming as little as two bytes per use regardless of the string length).

8.6.3 CDATA content token

The CDATA content token is defined as:

```
CDataSectionToken {                       // <![CDATA[content]]>
    TokenType type = CDataSectionCode;    // token-type code
    Value     content;                    // single content value
}
```

This is equivalent to the "**<![CDATA[content]]>**" structure in textual XML. This token is essentially equivalent to the **CharContentToken**, except that its use may be regarded as a hint to a translator to regenerate a **CDATA** section in textual XML. If the **strictXmlStrings** header flag is set, then the content string must not include the character sequence "**]]>**". If this header flag is not set, then the content may include the sequence. However, since XML **CDATA** sections must not include the character sequence "**]]>**", it may not be possible to regenerate a valid **CDATA** section in textual XML in all cases. If it is not, then regular character content must be regenerated with appropriate escape sequences. A **CDATA** section is normally used in XML to represent strings with literal "**<**", "**>**", or "**&**" characters, where these characters are used literally for the purposes of visual appearance or convenience.

8.6.4 Whitespace content token

The token that encodes potentially insignificant whitespace is defined as:

```
WhitespaceToken {                 // possibly insignificant whitespace
    TokenType type = WhitespaceCode;  // token-type code
    Count     nBlankLines;            // number of blank lines
    String    content;                // literal whitespace content
}
```

Whitespace is defined in XML as strings of the Unicode characters **#x20**, **#x9**, **#xD**, and **#xA**. The XML specification defines that all whitespace in an XML document is significant and must be passed to the reading application. However, in practice with most applications, much of the whitespace included for formatting and visual presentation of a textual XML document is actually insignificant. Usage of the **WhitespaceToken** allows BXML readers to remove insignificant whitespace efficiently. The writer is not required to use this token, but whitespace that is included in other content tags may be considered to be significant by the reader. A textual-XML translator may also wish to remove potentially insignificant whitespace from a stream anyway, except maybe for completely blank lines.

"Potentially insignificant whitespace" is defined as all sequences of exclusively whitespace characters that separate markup items in textual XML. This includes life-feeds (**#xA** character) and indentation characters that are normally inserted before element opening and closing tags.

The **nBlankLines** field records the number of completely blank lines that are included within the whitespace token. These are normally inserted into an XML file for visual separation between file structures and can make a file much more readable and essentially can be considered to be comments. The number of completely blank lines can be counted as being one less than the number of newline characters in the whitespace string. This

information can be useful to record and reproduce blank lines without any extraneous other whitespace characters when generating textual XML from BXML.

8.6.5 Blob content token

A "blob" is an opaque block of raw binary user data. The blob content token is defined as:

```
BlobSectionToken {                  // raw-binary blob
    TokenType    type = BlobSectionCode; // token-type code
    TextBlobType textEncoding;     // encoding to use in text XML
    Count        length;           // length in bytes
    byte         content[length];  // raw bytes of blob
}

byte enum TextBlobType {  // equivalent text-encoding code
    None      = 0x00,     // there is no text-encoded equivalent
    HexCoded  = 0x01,     // text would use hexadecimal encoding
    Base64    = 0x02,     // text would use base-64 encoding
    ByteArray = 0x03      // text would use number array
}
```

Binary user data cannot be represented directly in textual XML, which is a major limitation and source of headaches, but it can be in BXML. Binary data can either be referenced indirectly with URIs or it can be stored in-line in a text encoding in textual XML. Base-64 encoding [BASE64] is frequently used for this purpose.

It is important to be able to translate a blob into a textual-XML representation if that should become necessary, so the **`textEncoding`** field is provided to indicate what textual representation to use when translating. An encoding value of **`None`** means that no text-encoding equivalent is available and a translator must report an error if the attempt is made.

Normally, an application writing a blob will write it directly into a BXML file, but it is possible for a general translator to detect blobs in some textual XML files. XML Schema includes definitions of the intrinsic types "**`hexBinary`**" and "**`Base64Binary`**" which correspond to the **`HexCoded`** and **`Base64`** code values above in the **`TextBlobType`**. A validating translator could read in the text-encoded blob from an XML file and emit a **`BlobSectionToken`** into the BXML file.

8.6.6 Entity-reference token

The entity-reference token is defined as:

```
EntityRefToken {                    // &entity;
    TokenType type = EntityRefCode; // token-type code
    Count     stringRef;            // name reference
}
```

This is equivalent to the XML "**&name;**" construct to reference an environmentally-defined entity object. A string-table reference is used to identify the entity name, as with attribute and element names. The entity references of "**&**", "**<**", "**>**" , "**"**", and "**'**" are normally used as character-escape sequences in textual XML, but it is suggested that the characters that these entities represent be used literally in BXML content, for efficiency and convenience.

8.6.7 Character-entity-reference token

The character-entity-reference token is defined as:

```
CharEntityRefToken {                    // &#char_code;
    TokenType type = CharEntityRefCode; // token-type code
    Count     unicodeChar;              // unicode character number
}
```

This is equivalent to the XML "**&#char_code;**" Unicode-character-code entity reference.

8.7 Comment token

The comment token is defined as:

```
CommentToken {                      // <!--comment-->
    TokenType type = CommentCode;   // token-type code
    PosHint   positioningHint;      // comment positioning
    String    content;              // content of comment
}

byte enum PosHint {
    PosIndented    = 0x00,  // on fresh line but indented
    PosStartOfLine = 0x01,  // at start of fresh line
    PosEndOfLine   = 0x02   // after previous content
}
```

This is equivalent to the XML "**<!--comment-->**" comment construct and it is provided so that comments can be retained in a BXML file to better mirror the XML content. If the **strictXmlStrings** header flag is set, then the **content** string must not include the character sequence "**--**" (two hyphens). If the header flag is not set, then the content may include this sequence, but if it does, then the sequence must be substituted with something else if this comment is generated into textual XML, perhaps "**-=**". This case is not considered a breach of translating "with no loss of information" since the object in question is only a comment and it could not have originated from a textual-XML document.

The whitespace that is included in the **content** string may be considered to be significant by an application, including leading and trailing whitespace. If there is no leading or trailing whitespace in a string, a textual-XML writer may choose to insert a single space after and before the "**<!--**" and "**-->**" sequences.

The **positioningHint** field is provided to give a hint of what line position in textual XML a comment should be regenerated. Comments will normally be indented on a fresh line, but options are provided to suggest that they be generated at the start of a fresh line in case the comment has full-width visual formatting, or at the end of the previous content/markup object, in the case that the comment describes that object.

8.8 XML control tokens

8.8.1 XML-declaration token

The XML-declaration token is defined as follows:

```
XmlDeclarationToken {           // <?xml ...?>
    TokenType type = XmlDeclarationCode;  // token-type code
    String    version;          // XML version
    Bool      standalone;       // document is standalone
    Bool      standaloneIsSet;  // "standalone" is used
}
```

This is equivalent to the "**<?xml …?>**" XML-declaration construct (where the substring "**xml**" is case-insensitive) and it should normally be the first token present in the BXML token stream. The semantics for the **version** and **standalone** fields are the same as for the attributes of the same names for the XML-declaration. The **version** field may be logically marked as not being present for the XML semantics by assigning them a zero-length string, and the logical presence of the **standalone** field is indicated by the **standaloneIsSet** field. No character-set "**encoding**" value is given here, but the value implied for that attribute of the XML declaration is the **charEncoding** value from the **Header** structure.

8.8.2 Bang token

The "bang" token is defined as:

```
BangToken {                     // <!name ...>, e.g., DOCTYPE
    TokenType type = BangCode;  // token-type code
    Count     nameRef;          // name of tag, string-table ref
    String    content;          // verbatim character content
}
```

This is equivalent to the XML construct of the form "**<!name …>**". "Bang" is a synonym used sometimes for the exclamation mark (!). This construct is infrequently used in practice. The name is given by a string-table reference and the **content** is represented simply as a verbatim unparsed string. The name and content have the same semantics and restrictions as in textual XML. An example name is "**DOCTYPE**" and this type has a complex structure that is not worth tokenizing in the BXML file since non-validating parsers/generators likely will not understand it, and the parsers that do will most likely also support the textual XML

format and will therefore be able to handle the unparsed string anyway. "**ENTITY**" declarations also use this token, among others.

8.8.3 Bang-bracket token

The bang-bracket token is defined as:

```
BangBracketToken {                      // <![name[...]]>, not CDATA
    TokenType type = BangBracketCode;   // token-type code
    Count     nameRef;                  // name of tag, string ref
    String    content;                  // verbatim character content
}
```

This token is equivalent to the XML construct "**<![name[…]]>**", except that the **CDataSectionToken** is used instead for the name being "**CDATA**". This type of markup is used in XML conditional sections which are rarely used in practice. The name is given by a reference into the global string table and the **content** is an unparsed verbatim string and these values have the same semantics and restrictions as in textual XML.

8.8.4 Processing-instruction token

The processing-instruction token is defined as:

```
ProcessingInstrToken {              // <?name ...?> excl. "<?xml?>"
    TokenType type = ProcessingInstrCode;  // token-type code
    Count     nameRef;              // name of tag, string-table ref
    String    content;              // verbatim content, or ""
}
```

This is used to represent XML constructs of the form "**<?name …?>**", excluding the XML-declaration construct. Processing instructions are rarely used in practice. The name is given as a reference into the global string table and the **content** is an unparsed verbatim string and these values have the same semantics and restrictions as in textual XML.

8.9 String table

The string-table-fragment structure is defined as:

```
StringTableToken {                      // string table (fragment)
    TokenType type = StringTableCode;   // token-type code
    Count     nStrings;                 // number of strings in frag.
    String    strings[nStrings];        // string values
}
```

The global string table may be split into many string-table fragments. This is to make it more convenient to implement a sequential writer by not requiring that it know every symbol/string it might produce in advance. This may also allow the global string table to be more compact, since only a small subset of available symbols/strings may be used in any particular XML

document, and it may not be practical to pre-compute this limited subset of symbols before emitting the first XML tag. The writer has the choice of emitting all strings up-front, or in batches as different portions of the generator program are executed, or individually on an as-used basis (though this approach may be inefficient in terms of space and parsing time).

Some constraints are placed on the logical global string table. All string-table fragments define global string/symbol codes sequentially starting from zero, and any literal string must appear in the global string table only once, even if it is used in different contexts (e.g., as an element name and as an attribute name). Also, each symbol string must be defined in the BXML stream before it is first referenced when the token stream is read sequentially; however, there is no requirement that a particular string ever actually be referenced. The strings when used as element/attribute symbol names are subject to the constraints on names in textual XML.

The string-table index in the **`TrailerToken`** may be used to reassemble the global string table for use during random access, or to access only selected string definitions on an as-needed basis.

8.10 Index table

The index-table structure is defined as:

```
IndexTableToken {                        // index table
    TokenType    type = IndexTableCode; // token-type code
    Count        skipSize;              // size of rest of token
    String       xpathExpr;             // XPath expression
    Count        nEntries;              // number of index entries
    IndexTableEntry entries[nEntries];  // index-table entries
}

IndexTableEntry {                       // index value-match entry
    Value          value;               // match value
    Count          nOffsets;            // number of matching offsets
    Count          offsets[nOffsets];   // offsets to match elements
}
```

An index table is used to provide a simple mechanism to use to randomly access elements in the token stream that have properties of certain values. The **`xpathExpr`** value defines the property to be tested using an XPath expression [XPATH], and the **`entries`** give match values (equality comparison) for the test property and file offsets to the start of the element tokens that include the referenced property. The **`skipSize`** is provided to allow a reader to easily skip over the index table if it is not interested in using the index.

Any number of index tables may be provided within a BXML file, but the XPath expression must be unique for each. Index tables will normally be placed at the end of the BXML stream, since this will be the point that the writer will have collected all of the necessary

information. There is no obligation for a writer to generate any index tables at all, but a processing system may add them later (to the end of the stream) if it wishes.

8.11 Trailer Token

The trailer token is defined as:

```
TrailerToken {                      // last token of every file
    TokenType        tokenType = TrailerCode; // token-type code
    byte             id[4] = { 0x01, 'T', 'R', 0x00 }; // id
    StringTableIndex stringIndex;  // index of string tables
    IndexTableIndex  indexIndex;   // index of index-tables
    int              tokenLength;  // length of this token
}
```

This token marks the end of the BXML file for the reader and also provides a collected set of string-table references that logically constitutes a global string table and a collected set of index-table references. The four-byte **`tokenLength`** records the complete length of the **`TrailerToken`** and is required to be the fixed-size last field so that the start of the **`TrailerToken`** may be located to facilitate random access by reading the last four bytes of the BXML file. (Consequently, the **`TrailerToken`** is restricted to being at most approximately 2GB in size.) This token is required to be present at the end of every BXML file, even if the **`StringTableIndex`** and **`IndexTableIndex`** are marked as being 'unused'.

The **`id`** field serves a similar purpose to the **`identifier`** field of the **`Header`** structure at the beginning of the BXML file. It is included in the trailer token to help assure the detection of a truncated BXML file. If the BXML file is truncated, then random access will not work, and the file can probably be discarded as a whole. To check for truncation, the reader must first access the **`tokenLength`** field to locate the start of the **`TrailerToken`**, and then check that the **`tokenType`** and **`id`** fields have the correct values.

The string-table index is defined as:

```
StringTableIndex {           // index of string-table fragments
    Bool    isUsed;          // flag for whether this is active
    Count   nFragments;      // number of fragments in index
    StringTableIndexEntry fragments[nFragments];  // string tables
}

StringTableIndexEntry {      // string-table index fragment
    Count  nStringsDefined;  // number of strings defined in frag.
    Count  fileOffset;       // file offset to string-table token
}
```

This gives an index for all of the string-table fragments by storing the file offset to each fragment token that is present in the file. However, its use is optional, and it need not be given by setting the **`isUsed`** flag to **`FALSE`** and **`nFragments`** to zero. In this case, the **`hasRandomAccessInfo`** in the header must be set to **`FALSE`**.

The index-table index is defined as:

```
IndexTableIndex {            // index of index tables
    byte    isUsed;          // flag for whether this is active
    Count   nEntries;        // number of index-tables
    IndexTableIndexEntry entries[nEntries]; // index-table indexes
}

IndexTableIndexEntry {        // entry for index
    String       xpathExpr;   // XPath expression that is indexed
    Count        fileOffset;  // file offset of index-table token
}
```

This gives an index of all index tables present in the file. If no information is available, **`isUsed`** must be set to **`FALSE`** and **`nEntries`** to zero.

9 Interoperability

9.1 MIME & file types

When used in a MIME type, the BXML format should be identified by substituting the substring "**`xml`**" which normally identifies textual XML format with "**`x-bxml`**". For example, the type "**`text/xml`**" would be "**`text/x-bxml`**" for BXML, and "**`application/vnd.ogc.wms+xml`**" would be "**`application/vnd.ogc.wms+x-bxml`**".

Since "BXML" is a four-letter acronym, it faces the same problem as HTML when it comes to filename extensions, as different systems and people use either a three-letter or four-letter version. The shorter name is more compact and compatible but the longer name gives more information. Therefore, it is suggested that both "**`.BML`**" and "**`.BXML`**" be accepted as filename extensions for BXML files, using the appropriate letter case for the environment. It is important to distinguish this format from textual "**`.XML`**" since they are not compatible.

9.2 Application to GML

It is crucially important for efficiency to encode GML coordinates as a raw binary array of double (or float) numbers. For the sake of simplicity, the GML coordinate structure should be redefined to have the following XML-Schema type:

```
<xs:simpleType name="CoordinatesType">
    <xs:list itemType="xs:double"/>
</xs:simpleType>
```

A list of space-separated numeric values is required for GML coordinate values to be stored and processed efficiently in BXML, since this is how XML Schema defines lists of numbers. The GML practice of allowing user-defined coordinate and point separators is really quite strange, especially considering that in a production environment humans rarely will ever even look at coordinate values and almost certainly will not edit them by hand. Also, the user-

definable decimal-point representation is problematic since this is completely arbitrary relative to the definitions of numbers in XML Schema, which defines the period character to be the only lexical representation of a decimal point. One is forced to speculate that the representations provided, especially the element-structured encoding optionally provided, is intended for low-volume demonstrations only and not for production work. The element-structured encoding, while 'pretty', is enormously wasteful, especially after being parsed into a tree structure in memory. One might also speculate, given the arbitrary definition of the decimal point and given the nature of real-world implementations, that most if not all existing implementations of GML processors are broken.

The standard GML practice of using a comma character as a dimensional separator is not compatible with the BXML numeric-array representation, but if a dimensional separator of a space is selected in addition to the default point separator of a space character, then the numeric-array encoding could actually be used, even with the current specification, assuming that installed parsing applications do not require the two types of separators to be different. However, this would be formally quite awkward to work with, and automatic detection of numeric lists in general by analyzing XML Schema would not be possible with GML data.

These same issues apply to any GML observation values that may be efficiently represented in an array. As an indirect example, there has been informal discussion in various forums about how ridiculously wasteful XML would be at representing imagery data, but with raw-numeric arrays, a compact element type, and GZIP compression, BXML would be fundamentally as efficient at storing imagery data as the lossless PNG imagery format [PNG].

9.3 XML interoperability

It is important to be able to translate back and forth to and from the textual-XML format without loss of information. BXML has been designed to make this translation easy, with the minor exception of literal-character escaping in some cases.

Annex A: Galdos's Report on the Binary-XML-Encoding Work Item

OGC is assessing the use of binary XML encodings as a means of overcoming the performance and scalability problems associated with the inherent verbosity of XML data in the data interchange between OGC web services and their clients. In this context, the proclaimed goals of the CIPI-1.1 project were:

1. Determine the enhancements to the WFS 1.0.0 specification necessary for the efficient use of binary XML encodings in the data interchange between WFS and WFS Clients;

2. Evaluate the usability of an existing binary XML encoding and respective Coder/Decoder (CODEC) API.

Galdos believes that OGC should not tie its web services to any particular binary XML encoding. Rather, the Clients should be able to negotiate with OGC web services the binary XML encoding of their choosing. In accordance with this, Galdos is focusing on the two aforementioned goals of CIPI-1.1.

WFS 1.0.0 Specification Enhancements

For the WFS specification to allow for the use of binary XML encodings, the WFS has to be able to advertise its support for it, and WFS Clients need a way to request the data to be returned in binary XML. In addition, we should allow WFS Clients to submit data encoded in binary XML. Below is an outline of what needs to be done to support these scenarios.

Publishing support for binary XML encodings

We propose that the capabilities document contain a list of supported binary XML encodings along with their respective MIME types (the MIME types could be used as their identifiers). It is assumed that both requests and responses may be encoded using those encodings.

Similar approach can be taken for the detection of supported compression methods (e.g., GZIP).

Use of binary XML encodings

When sending a request encoded in binary XML, a WFS Client must set the content encoding of the data stream to the appropriate MIME type. This will signal the encoding of the request to the target WFS.

To request that the response be encoded in binary XML, a WFS Client can specify the desired MIME type of the response in the request. This will necessitate adding an optional parameter such as **`outputEncoding`**, in which Clients can specify the MIME type of their choosing.

Similar approach can be taken for the use of supported compression methods (e.g., GZIP).

Evaluation of BinXML™ Binary XML Encoding and CODEC

Galdos has evaluated the binary XML CODEC called BinXML™. This CODEC is being successfully used for video streaming, as well as for improving the efficiency of SOAP messaging and SVG. [*Editor's note: BinXML™ is a proprietary product.*]

The transparency of use was deemed a key evaluation criterion, one that would largely determine the acceptance of a particular binary XML Decoder among WFS Client builders. Without a binary XML encoding, a WFS Client would access the XML data using standardized interfaces such as SAX and DOM. For a binary XML CODEC to be transparent, it has to support these interfaces. Simply put, it enables WFS Clients to easily switch between the "binary XML" and regular modes of operations. Without transparency, WFS Clients would have to contain much more specialized code and thus their development would be far more complex.

It should be noted that the transparency of use might not be of interest to all WFS Clients. There can be WFS Clients that are built with a particular binary XML encoding in mind. Such WFS Clients should have the option of direct access to binary-encoded data.

Galdos's evaluation of BinXML™ indicated that this CODEC largely satisfies these criteria. It supports DOM, SAX and JAXP (Java API for XML Processing) as a transparency layer on top of its binary XML encoding. The BinXML™ Decoder also allows accessing the binary-encoded XML directly without going through DOM or SAX. Figures A-1 and A-2 illustrate what our view on the role of the transparency of binary XML encoding in the communication between WFS and Clients.

BinXML™ also fares well in the departments of speed and compression. The decoding time at the client side is much faster than for regular XML because the data is preparsed and prevalidated, while overhead at the encoder side is reasonable. BinXML™ is a schema-aware CODEC that harnesses the knowledge of the data content model carried in a XML/GML schema to provide optimized compression. ZLIB compression library is used to compress the textual content of elements. What is very important for GML and SVG, BinXML™ allows customized encoding/decoding for selected elements such as **`gml:coordinates`** in GML. This means that in addition to the existing compression, vendors can benefit from compression specifically tailored for GML/SVG applications.

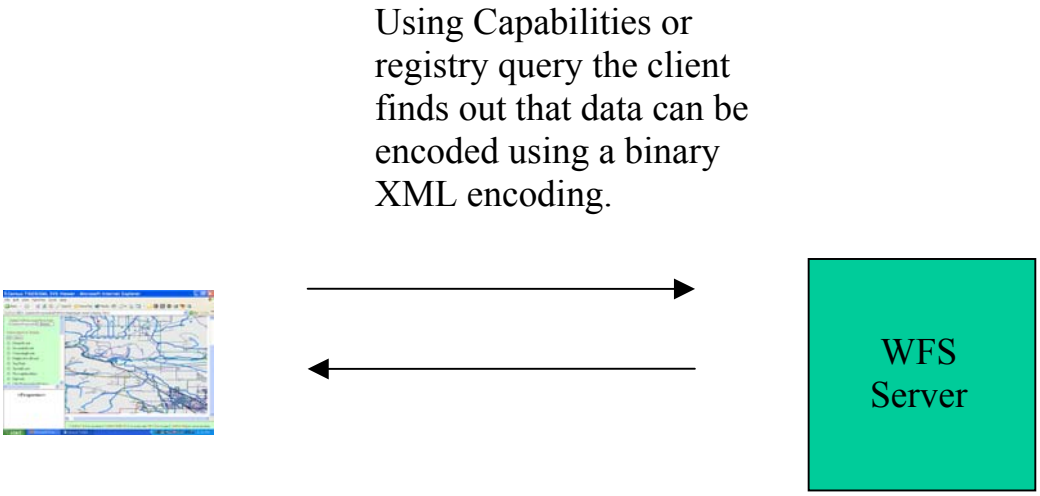

Figure A-1 Focus on supporting interfaces

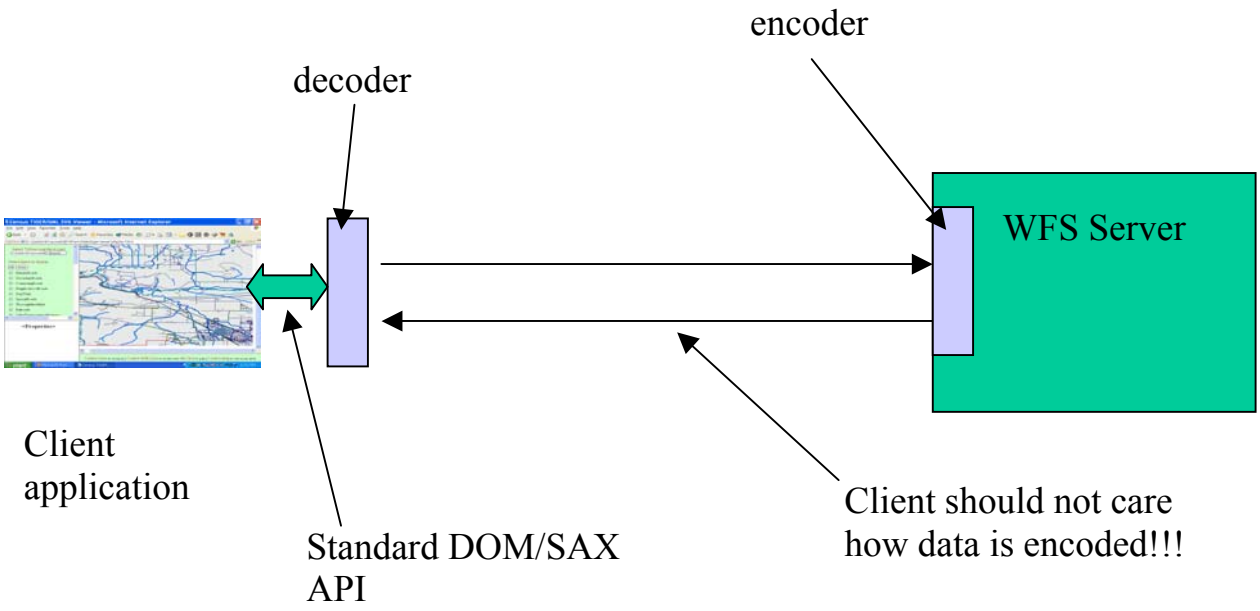

Figure A-2 Encoding should be hidden

Bibliography

[WAPXML] W3C (June 1999), *WAP Binary XML Content Format*, <http://www.w3.org/TR/wbxml/>.

[KEYWORDS] IETF RFC 2119 (March 1997), *Key words for use in RFCs to Indicate Requirement Levels*, Scott Bradner, <http://www.ietf.org/rfc/rfc2119.txt".

[PNG] PNG (2003), *PNG: Portable Network Graphics: A Turbo-Studly Image Format with Lossless Compression*, Greg Roelofs, et al, <http://www.libpng.org/pub/png/>.

[ZLIB] ZLIB (2003), *zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library*, Jean-loup Gailly, et al., <http://www.gzip.org/zlib/>.

[SHAPE] ESRI® (July 1988), *ESRI® Shapefile Technical Description*, <http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf>.

[SWIFT] Publisher unknown (1726), *Gulliver's Travels*, Jonathan Swift, <ftp://sailor.gutenberg.org/pub/Gutenberg/etext97/gltrv10.txt>.

[WKB-SF] OGC 99-049 (May 1999), *OpenGIS® Simple Features Specification for SQL*, Revision 1.1, <http://www.opengis.org/techno/specs/99-049.pdf>.

[BASE64] IETF RFC 1521 (September 1993), *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, N. Borenstein, et al., <http://www.ietf.org/rfc/rfc1952.txt>.

[XMLSCHEMA] W3C (May 2001), *XML Schema Part 0: Primer*, David C. Fallside (ed.), <http://www.w3.org/TR/xmlschema-0/>.