WAP Binary XML Content Format

Version 30-Apr-1998

Wireless Application Protocol
Binary XML Content Format Specification

Disclaimer:

This document is subject to change without notice.

Contents

1. Scope

Wireless Application Protocol (WAP) is a result of continuous work to define an industry-wide specification for developing applications that operate over wireless communication networks. The scope of the WAP Forum is to define a set of specifications to be used by service applications. The wireless market is growing very quickly and reaching new customers and services. To enable operators and manufacturers to meet the challenges in advanced services, differentiation and fast/flexible service creation, WAP defines a set of protocols in transport, session and application layers. For additional information on the WAP architecture, refer to "*Wireless Application Protocol Architecture Specification"* [WAP].

This specification defines a compact binary representation of the Extensible Markup Language [XML]. The binary XML content format is designed to reduce the transmission size of XML documents, allowing more effective use of XML data on narrowband communication channels. Refer to the [WML] specification for one example use of the binary XML content format.

The binary format was designed to allow for compact transmission with no loss of functionality or semantic information. The format is designed to preserve the element structure of XML, allowing a browser to skip unknown elements or attributes. The binary format encodes the parsed physical form of an XML document, ie, the structure and content of the document entities. Meta-information, including the document type definition and conditional sections, is removed when the document is converted to the binary format.

2. Document Status

This document is available online in the following formats:

- PDF format at http://www.wapforum.org/.

2.1 Copyright Notice

2.2 Errata

Known problems associated with this document are published at http://www.wapforum.org/.

2.3 Comments

Comments regarding this document can be submitted to the WAP Forum in the manner published at http://www.wapforum.org/.

3. References

3.1 Normative References

[ISO10646] "Information Technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane", ISO/IEC 10646-1:1993.

[RFC822] "Standard for the Format of ARPA Internet Text Messages", STD 11, RFC 822, D. Crocker, August 1982. URL: ftp://ds.internic.net/rfc/rfc822.txt

[RFC2119] "Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997. URL: ftp://ds.internic.net/rfc/rfc2119.txt

[WAP] "Wireless Application Protocol Architecture Specification", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/

[XML] "Extensible Markup Language (XML), W3C Proposed Recommendation 10-February-1998, REC-xml-19980210", T. Bray, et al, February 10, 1998. URL: http://www.w3.org/TR/REC-xml

3.2 Informative References

[ISO8879] "Information Processing - Text and Office Systems - Standard Generalised Markup Language (SGML)", ISO 8879:1986.

[UNICODE] "The Unicode Standard: Version 2.0", The Unicode Consortium, Addison-Wesley Developers Press, 1996. URL: http://www.unicode.org/

[WML] "Wireless Markup Language", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/

4. Definitions and Abbreviations

4.1 Definitions

The following are terms and conventions used throughout this specification.

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 [RFC2119].

Author - an author is a person or program that writes or generates WML, WMLScript or other content.

Content - subject matter (data) stored or generated at an origin server. Content is typically displayed or interpreted by a user agent in response to a user request.

Resource - a network data object or service that can be identified by a URL. Resources may be available in multiple representations (eg, multiple languages, data formats, size and resolutions) or vary in other ways.

SGML - the Standardised Generalised Markup Language (defined in [ISO8879]) is a general-purpose language for domain-specific markup languages.

User - a user is a person who interacts with a user agent to view, hear or otherwise use a resource.

User Agent - a user agent is any software or device that interprets the binary XML content format or other resources. This may include textual browsers, voice browsers, search engines, etc.

XML - the Extensible Markup Language is a World Wide Web Consortium (W3C) standard for Internet markup languages, of which WML is one such language. XML is a restricted subset of SGML.

4.2 Abbreviations

For the purposes of this specification, the following abbreviations apply.

API	Application Programming Interface
BNF	Backus-Naur Form
LSB	Least Significant Bits
MSB	Most Significant Bits
MSC	Mobile Switch Centre
RFC	Request For Comments
SGML	Standardised Generalised Markup Language [ISO8879]
UCS-4	Universal Character Set - 4 byte [ISO10646]
URL	Universal Resource Locator
UTF-8	UCS Transformation Format 8 [ISO10646]
W3C	World Wide Web Consortium
WAP	Wireless Application Protocol [WAP]
WML	Wireless Markup Language [WML]
XML	Extensible Markup Language [XML]

5. Binary XML Content Structure

The following data types are used in the specification of the XML tokenized format.

Table 1. Data types used in tokenized format

Data Type	***Definition***
bit	1 bit of data
byte	8 bits of opaque data
u_int8	8 bit unsigned integer
mb_u_int32	32 bit unsigned integer, encoded in multi-byte integer format.

Network byte order is "big-endian". In other words, the most significant byte is transmitted on the network first, followed by the less significant bytes. Network bit ordering within a byte is "big-endian". In other words, bit fields described first are placed in the most significant bits of the byte.

5.1 Multi-byte Integers

This encoding uses a multi-byte representation for integer values. A multi-byte integer consists of a series of octets, where the most significant bit is the *continuation* flag and the remaining seven bits are a scalar value. The continuation flag indicates that an octet is not the end of the multi-byte sequence. A single integer value is encoded into a sequence of `N` octets. The first `N-1` octets have the continuation flag set to a value of one (`1`). The final octet in the series has a continuation flag value of zero (`0`).

The remaining seven bits in each octet are encoded in a big-endian order, eg, most significant bit first. The octets are arranged in a big-endian order, eg, the most significant seven bits are transmitted first. In the situation where the initial octet has less than seven bits of value, all unused bits must be set to zero (`0`).

For example, the integer value `0xA0` would be encoded with the two-byte sequence `0x81 0x20`. The integer value `0x60` would be encoded with the one-byte sequence `0x60`.

5.2 Character Encoding

The encoding of all strings in the XML binary content format is specified by transport or container meta-information. Specifically, it is assumed that a *charset* declaration accompanies the binary XML content and indicates the encoding of all strings. The XML binary representation can support any string encoding, but requires that all strings include an encoding-specific termination character (eg, a `NULL` terminator) which can be reliably used to detect the end of a string. If a character encoding includes a `NULL` (eg, Unicode, ASCII, ISO-8859-1, etc.), the `NULL` character must be used as the termination character. As with the textual format of XML, it is also assumed that all tag and attribute names can be represented in the target character encoding.

5.3 BNF for Document Structure

A binary XML document is composed of a sequence of elements. Each element may have zero or more attributes and may contain embedded content. This structure is very general and does not have explicit knowledge of XML element structure or semantics. This generality allows user agents and other consumers of the binary format to skip elements and data that are not understood.

The following is a BNF-like description of the tokenized structure. The description uses the conventions established in [RFC822], except that the "`|`" character is used to designate alternatives and capitalised words indicate single-byte tokens, which are defined later. Briefly, "`(`" and "`)`" are used to group elements, optional elements are enclosed in "`[`" and "`]`". Elements may be preceded with `<N>*` to specify N or more repetitions of the following element (`N` defaults to zero when unspecified).

```
start     = version publicid strtbl 1*content
strtbl    = length *byte
content   = element | string | extension | entity | pi

element   = stag [ 1*attribute END ] [ *content END ]
stag      = TAG | ( LITERAL index )
attribute = attrStart *attrValue
attrStart = ATTRSTART | ( LITERAL index )
attrValue = ATTRVALUE | string | extension | entity

extension = ( EXT_I termstr ) | ( EXT_T index ) | EXT

string    = inline | tableref
inline    = STR_I termstr
tableref  = STR_T index

entity    = ENTITY entcode
entcode   = mb_u_int32     // UCS-4 character code

pi        = PI attrStart *attrValue END

version   = u_int8 containing WBXML version number
publicid  = mb_u_int32 | ( zero index )
termstr   = charset-dependent string with termination
index     = mb_u_int32       // integer index into string table.
length    = mb_u_int32     // integer length.
zero      = u_int8         // containing the value zero (0)
```

5.4 Version Number

```
version   = u_int8 containing WBXML version number
```

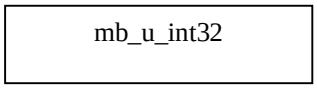
All WBXML documents contain a version number in their initial byte. This version specifies the WBXML version. The version byte contains the major version minus one in the upper four bits and the minor version in the lower four bits. For example, the version number `2.7` would be encoded as `0x17`. This document specifies WBXML version 1.0.

5.5 Document Public Identifier

```
publicid  = mb_u_int32 | ( zero index )
zero      = u_int8         // containing the value zero (0)
```

The binary XML format contains a representation of the XML document public identifier. This `publicid` is used to identify the well-known document type contained within the WBXML entity.

The first form of `publicid` is a multi-byte positive integer value, greater than zero, representing a well-known XML document type (eg, `-//WAPFORUM//DTD WML 1.0//EN`).



Public identifiers may also be encoded as strings, in the situation where a pre-defined numeric identifier is not available.

See section 7.2 for numeric constants related to public identifiers.

5.6 String Table

```
strtbl    = length *byte
```

A binary XML document must include a string table immediately after the public identifier. Minimally, the string table consists of a `mb_u_int32` encoding the string table length in bytes, not including the length field (eg, a string table containing a two-byte string is encoded with a length of two). If the length is non-zero, one or more strings follow. The encoding of the strings follows the current *charset* specified by transport meta-information.

Various tokens encode references to the contents of the string table. These references are encoded as scalar byte offsets from the first byte of the first string in the string table. For example, the offset of the first string is zero (`0`).

5.7 Token Structure

Tokens are split into a set of overlapping *code spaces*. The meaning of a particular token is dependent on the context in which it is used. Tokens are organised in the following manner:

- There are two classifications of tokens: global tokens and application tokens.
- Global tokens are assigned a fixed set of codes in all contexts and are unambiguous in all situations. Global codes are used to encode inline data (eg, strings, entities, opaque data, etc.) and to encode a variety of miscellaneous control functions.
- Application tokens have a context-dependent meaning and are split into two overlapping *code spaces*. These two code spaces are the *tag code space* and the *attribute code space*. A given token value (eg, `0x99`) will have a different meaning depending on whether it represents a token in the tag or attribute code space.
- The tag code space represents specific tag names. Each tag token is a single-byte code and represents a specific tag name (eg, `CARD`).
- The attribute code space is split into two numeric ranges representing attribute prefixes and attribute values respectively.

Each code space is further split into a series of 256 *code pages*. Code pages allow for future expansion of the well-known codes. A single token (`SWITCH_PAGE`) switches between the code pages.

The definition of tag and attribute codes is document-type-specific. Global codes are divided between a generic set of codes common to all document types and a set reserved for document-type-specific extensions.

5.7.1 Parser State Machine

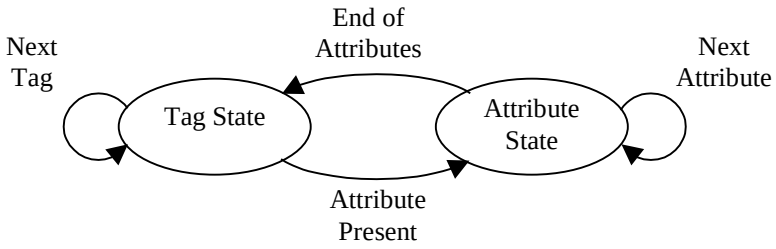
The tokenized format has two states, each of which has an associated code space. The states are traversed according to the syntax described in section 5.3. Code spaces are associated with parser states in the following manner:

Table 2. Parser states

Parser State	***Code Space***
`stag`	Tags
`attribute`	Attributes

Any occurrence of code page switch tokens (`SWITCH_PAGE`) while in a given state changes the current code page for that state. This new code page remains as the current code page until another `SWITCH_PAGE` is encountered in the same state or the document end is reached. Each parser state maintains a separate "current code page". The initial code page for both parser states is zero (`0`).

The following state machine is an alternative representation of the state transitions and is provided as a reference model.



5.7.2 Tag Code Space

Tag tokens are a single `u_int8` and are structured as follows:

Table 3. Tag format

Bit(s)	***Description***
7 (most significant)	Indicates whether attributes follow the tag code. If this bit is zero, the tag contains no attributes. If this bit is one, the tag is followed immediately by one or more attributes. The attribute list is terminated by an END token.
6	Indicates whether this tag begins an element containing content. If this bit is zero, the tag contains no content and no end tag. If this bit is one, the tag is followed by any content it contains and is terminated by an END token.
5 - 0	Indicates the tag identity.

For example:

- Tag value `0xC6`: indicates tag six (`6`), with both attributes and content following the tag, eg,

  ```
  <TAG arg="1">foo</TAG>
  ```

- Tag value `0x46`: indicates tag six (`6`), with content following the start tag. This element contains no attributes, eg

  ```
  <TAG>test</TAG>
  ```

- Tag value `0x06`: indicates tag six (`6`). This element contains no content and has no attributes, eg,

  ```
  <TAG/>
  ```

The globally unique code `LITERAL` (see section 5.7.4.5) represents unknown tag names. An XML tokenizer should avoid the use of the `LITERAL` or string representations of a tag when a more compact form is available.

Tags containing both attributes and content always encode the attributes before the content.

5.7.3 Attribute Code Space (ATTRSTART and ATTRVALUE)

Attribute tokens are a encoded as a single `u_int8`. The attribute code space is split into two ranges (in addition to the global range present in all code spaces):

- Attribute Start - tokens with a value less than `128` indicate the start of an attribute. The attribute start token fully identifies the attribute name, eg, `URL=`, and may optionally specify the beginning of the attribute value, eg, `PUBLIC="TRUE"`. Unknown attribute names are encoded with the globally unique code `LITERAL` (see section 5.7.4.4). `LITERAL` must not be used to encode any portion of an attribute value.
- Attribute Value - tokens with a value of `128` or greater represent a well-known string present in an attribute value. These tokens may only be used to represent attribute values. Unknown attribute values are encoded with string, entity or extension codes (see section 5.7.4).

All tokenized attributes must begin with a single attribute start token and may be followed by zero or more attribute value, string, entity or extension tokens. An attribute start token, a LITERAL token or the END token indicates the end of an attribute value. This allows a compact encoding of strings containing well-known sub-strings and entities.

For example, if the attribute start token TOKEN_URL represents the attribute name "URL", the attribute value token TOKEN_COM represents the string ".com" and the attribute value token TOKEN_HTTP represents the string "http://", the attribute URL="http://foo.com/x" might be encoded with the following sequence:

```
TOKEN_URL TOKEN_HTTP STR_I "foo" TOKEN_COM STR_I "/x"
```

In another example, if the attribute start token TOKEN_PUBLIC_TRUE represents the attribute name "PUBLIC" and the value prefix "TRUE", the attribute PUBLIC="TRUE" might be encoded with the following sequence:

```
TOKEN_PUBLIC_TRUE
```

An XML tokenizer should avoid the use of the LITERAL or string representations of an attribute name when a more compact form is available. An XML tokenizer should avoid the use of string representations of a value when a more compact form is available.

5.7.4 Global Tokens

Global tokens have the same meaning and structure in all code spaces and in all code pages. The classes of global tokens are:

- Strings - inline and table string references
- Extension - document-type-specific extension tokens
- Opaque - inline opaque data
- Entity - character entities
- Processing Instruction - XML PIs
- Literal - unknown tag or attribute name
- Control codes - miscellaneous global control tokens

5.7.4.1 Strings

```
string    = inline | tableref
inline    = STR_I termstr
tableref  = STR_T index
```

Strings encode inline character data or references into a string table. The string table is a concatenation of individual strings. String termination is dependent on the character document encoding and should not be presumed to include NULL termination. References to each string include an offset into the table, indicating the string being referenced.

Inline string references have the following format:

STR_I	… char data …

String table references have the following format:

STR_T	mb_u_int32

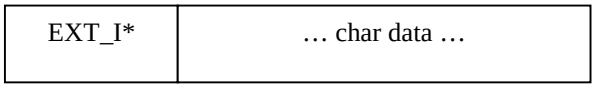
The string table offset is from the first byte of the first string in the table (ie, not a character offset).

5.7.4.2 Global Extension Tokens

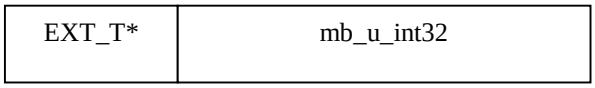
```
extension = ( EXT_I termstr ) | ( EXT_T index ) | EXT
```

The global extension tokens are available for document-specific use. The semantics of the tokens are defined only within the context of a particular document type, but the format is well defined across all documents. There are three classes of global extension tokens: single-byte extension tokens, inline string extension tokens and inline integer extension tokens.

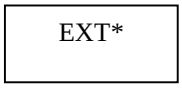
Inline string extension tokens (`EXT_I*`) have the following format:



Inline integer extension tokens (`EXT_T*`) have the following format:



Single-byte extension tokens (`EXT*`) have the following format:

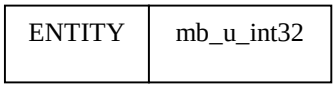


5.7.4.3 Character Entity

```
entity    = ENTITY entcode
entcode   = mb_u_int32     // UCS-4 character code
```

The character entity token (`ENTITY`) encodes a numeric character entity. This has the same semantics as an XML numeric character entity (eg, `&#32;`). The `mb_u_int32` refers to a character in the UCS-4 character encoding. All entities in the source XML document must be represented using either a string token (eg, `STR_I`) or the `ENTITY` token.

The format of the character entity is:

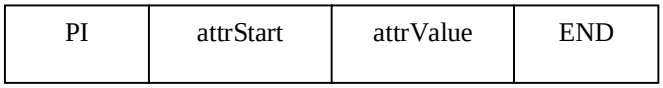


5.7.4.4 Processing Instruction

```
pi        = PI attrStart *attrValue END
```

The processing instruction (PI) token encodes an XML processing instruction. The encoded PI has identical semantics to an XML PI. The `attrStart` encodes the `PITarget` and the `attrValue` encodes the PI's optional value. For more details on processing instructions, see [XML].

The format of the PI tag is:

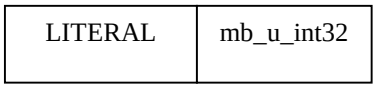


PIs without a value are encoded as:

5.7.4.5 Literal Tag or Attribute Name

The literal token encodes a tag or attribute name that does not have a well-known token code. The actual meaning of the token (ie, tag versus attribute name) is determined by the token parsing state. All literal tokens indicate a reference into the string table, which contains the actual name.

The format of the LITERAL tag is:



5.7.4.6 Miscellaneous Control Codes

5.7.4.6.1 END Token

The END token is used to terminate attribute lists and elements. END is a single-byte token.

5.7.4.6.2 Code Page Switch Token

The code-page switch token (`SWITCH_PAGE`) indicates a switch in the current code page for the current token state. The code-page switch is encoded as a two-byte sequence:

SWITCH	u_int8

5.7.4.7 Reserved Tokens

There are several reserved global tokens. These must not be emitted by a tokenizer and should be treated as a single-byte token by a user agent.

The code page 255 is reserved for implementation-specific or experimental use. The tokens in this code page will never be used to represent standard XML document constructs.

6. Encoding Semantics

6.1 Document Tokenization

The process of tokenizing an XML document must convert all markup and XML syntax (ie, entities, tags, attributes, etc.) into their corresponding tokenized format. All comments must be removed. Processing directives intended for the tokenizer may be removed. Other meta-information, such as the document type definition and unnecessary conditional sections must be removed. All text and entities must be converted to string (eg, `STR_I`) or entity (`ENTITY`) tokens. Entities in the textual markup (eg, `&`) must be converted to string form when tokenized, if the target character encoding can represent the entity. Characters present in the textual form may be encoded using the `ENTITY` token when they can not be represented in the target character encoding. Attribute names must be converted to an attribute start token or must be represented by a single `LITERAL` token. Attribute values may not be represented by a `LITERAL` token.

It is illegal to encode markup constructs as strings. The user agent must treat all text tokens (eg, `STR_I` and `ENTITY`) as CDATA, ie, text with no embedded markup.

6.2 Document Structure Conformance

The tokenized XML document must accurately represent the structure and semantics of the textual source document. This implies that the source document must be well-formed, as defined in [XML]. Document tokenization may validate the document as specified in [XML], but this is not required. If the semantics of a particular DOCTYPE are well known, additional semantic checks may be applied during the tokenization process.

6.3 Encoding Default Attribute Values

The tokenized representation of a XML document may omit any attributes that are implied in the DTD or are specified with their default value. This implies that a user agent implementation must be aware of the attribute defaults of a given version of the DTD. This information can be inferred from the version number in the tokenized data format.

7. Numeric Constants

7.1 Global Tokens

The following token codes are common across all document types and are present in all code spaces and all code pages. All numbers are in hexadecimal.

Table 4. Global tokens

Token Name	***Token***	***Description***
SWITCH_PAGE	0	Change the code page for the current token state. Followed by a single u_int8 indicating the new code page number.
END	1	Indicates the end of an attribute list or the end of an element.
ENTITY	2	A character entity. Followed by a `mb_u_int32` encoding the character entity number.
STR_I	3	Inline string. Followed by a `termstr`.
LITERAL	4	An unknown tag or attribute name. Followed by an `mb_u_int32` that encodes an offset into the string table.
EXT_I_0	40	Inline string document-type-specific extension token. Token is followed by a `termstr`.
EXT_I_1	41	Inline string document-type-specific extension token. Token is followed by a `termstr`.
EXT_I_2	42	Inline string document-type-specific extension token. Token is followed by a `termstr`.
PI	43	Processing instruction.
LITERAL_C	44	Unknown tag, with content.
EXT_T_0	80	Inline integer document-type-specific extension token. Token is followed by a `mb_uint_32`.
EXT_T_1	81	Inline integer document-type-specific extension token. Token is followed by a `mb_uint_32`.
EXT_T_2	82	Inline integer document-type-specific extension token. Token is followed by a `mb_uint_32`.
STR_T	83	String table reference. Followed by a `mb_u_int32` encoding a byte offset from the beginning of the string table.
LITERAL_A	84	Unknown tag, with attributes.
EXT_0	C0	Single-byte document-type-specific extension token.
EXT_1	C1	Single-byte document-type-specific extension token.
EXT_2	C2	Single-byte document-type-specific extension token.
RESERVED_2	C3	Reserved for future use.
LITERAL_AC	C4	Unknown tag, with content and attributes.

7.2 Public Identifiers

The following values represent well-known document type public identifiers. The first 128 values are reserved for use in future WAP specifications. All numbers are in hexadecimal.

Table 5. Public Identifiers

Value	***Public Identifier***
0	String table index follows; public identifier is encoded as a literal in the string table.
1	Unknown or missing public identifier.
2	"-//WAPFORUM//DTD WML 1.0//EN" (WML 1.0)
3	"-//WAPFORUM//DTD WTA 1.0//EN" (WTA Event 1.0)
4 - 7F	Reserved

8. Encoding Examples

8.1 A Simple XML Document

The following is an example of a simple tokenized XML document. It demonstrates basic element, string and entity encoding. Source document:

```
<?xml version="1.0"?>
<!DOCTYPE XYZ [
<!ELEMENT XYZ (CARD)+>
<!ELEMENT CARD (#PCDATA | BR)*>
<!ELEMENT BR EMPTY>
<!ENTITY nbsp " ">
]>
<XYZ>
  <CARD>
    X &amp; Y<BR/>
    X = 1
  </CARD>
</XYZ>
```

The following tokens are defined for the tag code space:

Tag Name	***Token***
BR	5
CARD	6
XYZ	7

Tokenized form (numbers in hexadecimal) follows. This example uses only inline strings and assumes that the character encoding uses a NULL terminated string format. It also assumes that the transport character encoding is US-ASCII. This encoding is incapable of supporting some of the characters in the deck (eg, ` `), forcing the use of the `ENTITY` token.

```
00  01  00  47  46  03  ' ' 'X' ' ' '&' ' ' 'Y' 00  05  03  ' '
'X' 00  02  81  20  03  '=' 00  02  81  20  03  '1' ' ' 00  01  01
```

In an expanded and annotated form:

Table 6. Example tokenized deck

Token Stream	***Description***
`00`	Version number - WBXML version 1.0.
`01`	Unknown public identifier
`00`	String table length
`47`	`XYZ`, with content
`46`	`CARD`, with content
`03`	Inline string follows
`' ', 'X', ' ', '&', ' ', 'Y', 00`	String
`05`	`BR`
`03`	Inline string follows
`' ', 'X', 00`	String

Token Stream	***Description***
02	ENTITY
81 20	Entity value (0x160)
03	Inline string follows
'=', 00	String
02	ENTITY
81 20	Entity value (0x160)
03	Inline string follows
'1', ' ', 00	String
01	END (of CARD element)
01	END (of XYZ element)

8.2 An Expanded Example

The following is another example of a tokenized XML document. It demonstrates attribute encoding and the use of the string table. Source document:

```
<?xml version="1.0"?>
<!DOCTYPE XYZ [
<!ELEMENT XYZ ( CARD )+ >
<!ELEMENT CARD (#PCDATA | INPUT | DO)*>
<!ATTLIST CARD NAME NMTOKEN #IMPLIED>
<!ATTLIST CARD STYLE (LIST|SET) 'LIST'>
<!ELEMENT DO EMPTY>
<!ATTLIST DO TYPE CDATA #REQUIRED>
<!ATTLIST DO URL CDATA #IMPLIED>
<!ELEMENT INPUT EMPTY>
<!ATTLIST INPUT TYPE (TEXT|PASSWORD)'TEXT'>
<!ATTLIST INPUT KEY NMTOKEN #IMPLIED>
<!ENTITY nbsp " ">
]>
<!-- This is a comment -->
<XYZ>
  <CARD NAME="abc" STYLE="LIST">
    <DO TYPE="ACCEPT" URL="http://xyz.org/s"/>
    Enter name: <INPUT TYPE="TEXT" KEY="N"/>
  </CARD>
</XYZ>
```

The following tokens are defined for the tag code space:

Tag Name	***Token***
CARD	5
INPUT	6
XYZ	7
DO	8

The following attribute start tokens are defined:

Attribute Name	***Attribute Value Prefix***	***Token***
STYLE	LIST	5
TYPE		6
TYPE	TEXT	7
URL	http://	8
NAME		9
KEY		B

The following attribute value tokens are defined:

Attribute Value	***Token***
.org	85
ACCEPT	86

Tokenized form (numbers in hexadecimal) follows. This example assumes an UTF-8 character encoding and NULL terminated strings:

```
00  01  12  'a' 'b' 'c' 00 ' ' 'E' 'n' 't' 'e' 'r' ' ' 'n'
'a' 'm' 'e' ':' ' ' 00 47  C5  09  03  00  05  01  88  06
86  08  03  'x' 'y' 'z' 00  85  03  '/' 's' 00  01 83  04
01  83  04  86  07  0A  03  'N' 00  01  01  01
```

In an expanded and annotated form:

Table 7. Example tokenized deck

Token Stream	***Description***
`00`	Version number - WBXML version 1.0
`01`	Unknown public identifier
`12`	String table length
`'a', 'b', 'c', 00, ' ', 'E', 'n', 't', 'e', 'r', ' ', 'n', 'a', 'm', 'e', ':', ' ', 00`	String table
`47`	`XYZ`, with content
`C5`	`CARD`, with content and attributes
`09`	`NAME=`
`83`	String table reference follows
`00`	String table index
`05`	`STYLE="LIST"`
`01`	`END` (of `CARD` attribute list)
`88`	`DO`, with attributes
`06`	`TYPE=`
`86`	`ACCEPT`
`08`	URL="http://"

Token Stream	***Description***
03	Inline string follows
'x', 'y', 'z', 00	string
85	".org"
03	Inline string follows
'/', 's', 00	string
01	END (of DO attribute list)
83	String table reference follows
04	String table index
86	INPUT, with attributes
07	TYPE="TEXT"
0A	KEY=
03	Inline string follows
'N', 00	String
01	END (of INPUT attribute list)
01	END (of CARD element)
01	END (of XYZ element)