# WAP WML

Draft Version 3-Feb-1998

## Wireless Application Protocol
## Wireless Markup Language Specification

***Disclaimer:***

*This document is a draft of the Wireless Markup Language (WML) specification, and is subject to change without notice.*

# Contents

# 1. Scope

Wireless Application Protocol (WAP) is a result of continuous work to define an industry wide standard for developing applications over wireless communication networks. The scope for the WAP working group is to define a set of standards 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 the Wireless Markup Language (WML). WML is a markup language based on [XML], and is intended for use in specifying content and user interface for narrowband devices, including cellular phones and pagers.

WML is designed with the constraints of small narrowband devices in mind. These constraints include:

- Small display and limited user input facilities
- Narrowband network connection
- Limited memory and computational resources

WML includes four major functional areas:

- Text presentation and layout - WML includes text and image support, including a variety of formatting and layout commands. For example, boldfaced text may be specified.
- Deck/card organisational metaphor - all information in WML is organised into a collection of *cards* and *decks*. Cards specify one or more units of user interaction (e.g. a choice menu, a screen of text or a text entry field). Logically, a user navigates through a series of WML cards, reviews the contents of each, enters requested information, makes choices, and moves on to another card.

  Cards are grouped together into decks. A WML deck is similar to an HTML page, in that it is identified by a URL [RFC1738], and is the unit of content transmission.
- Inter-card navigation and linking - WML includes support for explicitly managing the navigation between cards and decks. WML also includes provisions for event handling in the device, which may be used for navigational purposes, or to execute scripts. WML also supports anchored links, similar to those found in [HTML4].
- String parameterization and state management - all WML decks can be parameterised, using a state model. Variables can be used in the place of strings, and are substituted at run-time. This parameterization allows for more efficient use of network resources.

# 2. Document Status

This document is a preliminary draft. It is published to solicit comments from WAP members and other interested parties. The document is subject to change without any notice. It may be updated, replaced, or dropped at any time. Publishing the document does not imply endorsement nor does it imply that it will be part of a published WAP standard or recommendation. Contents of this draft reflect "work in progress." Any references to the document's content should only cite them as "work in progress."

This document is available online in the following formats:

- PDF format at URL, http://www.wapforum.org/TBD/.

## 2.1 Copyright Notice



## 2.2 Errata

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

## 2.3 Comments

Comments regarding this document can be submitted to the WAG working group in the manner published at http://www.wapforum.org/TBD/.

# 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

[RFC1738] "Uniform Resource Locators (URL)", T. Berners-Lee, et al., December 1994. URL: ftp://ds.internic.net/rfc/rfc1738.txt

[RFC1808] "Relative Uniform Resource Locators", R. Fielding, June 1995. URL: ftp://ds.internic.net/rfc/rfc1808.txt

[RFC2045] "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", N. Freed, et al., November 1996. URL: ftp://ds.internic.net/rfc/rfc2045.txt

[RFC2048] "Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures", N. Freed, et al., November 1996. URL: ftp://ds.internic.net/rfc/rfc2048.txt

[RFC2068] "Hypertext Transfer Protocol - HTTP/1.1", R. Fielding, et al., January 1997. URL: ftp://ds.internic.net/rfc/rfc2068.txt

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

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

[WAE] "Wireless Application Environment Specification", WAP Forum, January 30, 1998. URL: http://www.wapforum.org/

[WAP] "Wireless Application Protocol Architecture Specification, version 0.9", Wireless Application Protocol Architecture Working Group, 1997. URL: http://www.wapforum.org/

[WSP] "Wireless Session Protocol", WAP Forum, January 30, 1998. URL: http://www.wapforum.org/

[XML] "Extensible Markup Language (XML), W3C Proposed Recommendation 8-December-1997, PR-xml-971208", T. Bray, et al, December 8, 1997. URL: http://www.w3.org/TR/PR-xml


## 3.2 Informative References


[HDML2] "Handheld Device Markup Language Specification", P. King, et al., April 11, 1997. URL: http://www.uplanet.com/pub/hdml_w3c/hdml20-1.html

[HTML4] "HTML 4.0 Specification, W3C Recommendation 18-December-1997, REC-HTML40-971218", D. Raggett, et al., September 17, 1997. URL: http://www.w3.org/TR/REC-html40

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

# 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.

**Bytecode** - content encoding where the content is typically a set of low-level opcodes (i.e., instructions) and operands for a targeted hardware (or virtual) machine.

**Card** - a single WML unit of navigation and user interface. May contain information to present on the screen, instructions for gathering user input, etc.

**Client** - a device (or application) that initiates a request for connection with a server.

**Client Server Communication** - communication between a client and a server. Typically the server performs a task (such as generating content) on behalf of the server. Results of the task are usually sent back to the client (e.g., generated content.)

**Content** - synonym for resources.

**Content Encoding** - when used as a verb, content encoding indicates the act of converting content from one format to another. Typically the resulting format requires less physical space than the original, is easier to process or store, and/or is encrypted. When used as a noun, content encoding specifies a particular format or encoding standard or process.

**Content Format** - actual representation of content.

**Content Generator** - devices (or applications) that generate or format content. Typically content generators are on origin servers.

**Deck** - a collection of WML cards. A WML deck is also an XML document. May contain WMLScript.

**Device** - a network entity that is capable of sending and receiving packets of information and has a unique device address. A device can act as both a client or a server within a given context or across multiple contexts. For example, a device can service a number of clients (as a server) while being a client to another server.

**Deprecated** - an element, attribute or other construct that is outdated by other constructs and should not be used by applications. Deprecated constructs remain in the specification for a variety of purposes, including ease of application migration, backward compatibility, etc. Deprecated elements may become obsolete in a future specification.

**JavaScript** - a *de facto* standard language that can be used to add dynamic behaviour to HTML documents. Also known as ECMAScript.

**Obsolete** - this term indicates a construct or element that is no longer supported, and for which there is no guarantee of support by a given user agent.

**Origin Server** - the server on which a given resource resides or is to be created. Often referred to as a web server or an HTTP server.

**Peer-to-peer** - direct communication between two terminals typically thought of as clients without involving an intermediate server. Also known as client-to-client communication.

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

**Server** - a device (or application) that passively waits for connection requests from one or more clients. A server may accept or reject a connection request from a client.

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

**Terminal** - a device. Also called a mobile terminal or mobile station.

**Transcode** - the act of converting from one character set to another, e.g., conversion from UCS-2 to UTF-8.

**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 WML. This may include textual browsers, voice browsers, search engines, etc.

**WMLScript** - a scripting language used to program the mobile device. WMLScript is an extended subset of the JavaScript™ scripting language.

**XML** - the Extensible Markup Language is a World Wide Web Consortium (W3C) proposed 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 |
| **CGI** | Common Gateway Interface |
| **ECMA** | European Computer Manufacturer Association |
| **ETSI** | European Telecommunication Standardisation Institute |
| **GSM** | Global System for Mobile Communication |
| **HDML** | Handheld Markup Language [HDML2] |
| **HTML** | HyperText Markup Language [HTML4] |
| **HTTP** | HyperText Transfer Protocol [RFC2068] |
| **IANA** | Internet Assigned Number Authority |
| **IMC** | Internet Mail Consortium |
| **LSB** | Least Significant Bits |
| **MSB** | Most Significant Bits |
| **MSC** | Mobile Switch Centre |
| **PDA** | Personal Digital Assistant |
| **RFC** | Request For Comments |
| **SAP** | Service Access Point |
| **SGML** | Standardised Generalised Markup Language [ISO8879] |
| **SSL** | Secure Socket Layer |
| **TLS** | Transport Layer Security |
| **URI** | Uniform Resource Identifier |
| **URL** | Uniform Resource Locator [RFC1738] |
| **URN** | Uniform Resource Name |
| **W3C** | World Wide Web Consortium |
| **WAE** | Wireless Application Environment |
| **WAP** | Wireless Application Protocol [WAP] |
| **WBMP** | Wireless BitMaP |
| **WSP** | Wireless Session Protocol [WSP] |
| **WTA** | Wireless Telephony Applications |
| **WTAI** | Wireless Telephony Applications Interface |
| **WTP** | Wireless Transport Protocol |
| **WWW** | World Wide Web |
| **XML** | Extensible Markup Language [XML] |

## 4.3 Device Types

WML is designed to meet the constraints of a wide range of small, narrowband devices. These devices are primarily characterised by four constraints:

- Display size - limited screen size and resolution. A small mobile device such as a phone may only have a few lines of textual display, each line containing 8-12 characters.
- Limited input characteristics - a limited, or special-purpose input device. A phone typically has a numeric keypad and a few additional function-specific keys. A more sophisticated device may have software-programmable buttons, but may not have a mouse or other pointing device.
- Limited computational resources - limited CPU and memory, often limited by power constraints.
- Narrowband network connectivity - limited bandwidth and high latency. Devices with 300 baud network connections and 5-10 second round-trip latency are not uncommon.

This document uses the following terms to define broad classes of device functionality:

- **Phone** - a phone-class device is limited in all major areas. The typical display size ranges from two to ten lines. Input is usually accomplished with a combination of a numeric keypad and a few additional function keys. Computational resources and network throughput is typically limited, especially when compared with more general-purpose computer equipment.
- **PDA** - a Personal Digital Assistant is a device with a broader range of capabilities. When used in this document, it specifically refers to devices with additional display and input characteristics. A PDA display often supports resolution in the range of 160x100 pixels. A PDA may support a pointing device, handwriting recognition, and a variety of other advanced features.

These terms are meant to define very broad descriptive guidelines and to clarify certain examples in the document.

# 5. WML and URLs

The World Wide Web is a network of information and devices. Three areas of specification ensure widespread interoperability:

- A unified naming model. Naming is implemented with Uniform Resource Locators (URLs), which provide standard way to name any network resource. See [RFC1738].
- Standard protocols to transport information (e.g. HTTP).
- Standard content types (e.g. HTML, WML).

WML assumes the same reference architecture as HTML and the World Wide Web. Content is named using URLs, and is fetched over standard protocols that have HTTP semantics, such as [WSP]. URLs are defined in [RFC1738]. The character set used to specify URLs is also defined in [RFC1738].

In WML, URLs are used in the following situations:

- When specifying navigation, e.g., hyperlinking.
- When specifying external resources, e.g., an image or a script.

## 5.1 URL Schemes

WML browsers must implement the URL schemes specified in [WAE].

## 5.2 Fragment Anchors

WML has also adopted the HTML *de facto* standard of naming locations within a resource. A WML fragment anchor is specified by the document URL, followed by a hash mark (#), followed by a fragment identifier. WML uses fragment anchors to identify individual WML cards within a WML deck and to identify function names defined in a SCRIPT element (see sections 11.3.4 and 12.5 for more information). If no fragment is specified, a URL names an entire deck. In some contexts, the deck URL also implicitly identifies the first card in a deck.

## 5.3 Relative URLs

WML has also adopted the use of relative URLs, as specified in [RFC1808]. [RFC1808] specifies the method used to resolve relative URLs in the context of a WML deck. The base URL of a WML deck is the URL that identifies the deck.

# 6. WML Character Set

WML is an XML language, and inherits the XML document character set. In SGML nomenclature, a document character set is the set of all logical characters that a document type may contain (e.g. the letter 'T'), and a fixed integer identifying that letter. An SGML or XML document is simply a sequence of these integer tokens, which taken together form a document.

The document character set for XML and WML is the Universal Character set of ISO/IEC-10646 ([ISO10646]). Currently, this character set is identical to Unicode 2.0 ([UNICODE]). WML will adopt future changes and enhancements to the [XML] and [ISO10646] specifications. Within this document, the terms ISO10646 and Unicode are used interchangeably, and indicate the same document character set.

There is no requirement that WML decks be encoded using the full Unicode encoding (e.g. UCS-4). Any character encoding ("charset") that contains an inclusive subset of the characters in Unicode may be used (e.g. US-ASCII, ISO-8859-1, UTF-8, etc.). Documents not encoded using UTF-8 or UTF-16 must declare their encoding as specified in the XML specification.

## 6.1 Reference Processing Model

The WML reference-processing model is as follows. User agents must implement this model, or a model that is indistinguishable from it.

- The user agent must correctly map a document's external character encoding to Unicode before processing the document in any way.
- Any processing of entities is done in the document character set.

A given implementation may choose any internal representation (or representations) that is convenient.

## 6.2 Character Entities

WML supports both named and numeric character entities. An important consequence of the reference processing model is that all numeric character entities are referenced with respect to the document character set (Unicode), and not to the current document encoding (charset).

This means that `&#302;` always refers to the same logical character, independent of the current character encoding.

WML supports the following character entity formats:

- Named character entities, such as `&` and `<`
- Decimal numeric character entities, such as `&#32;`
- Hexadecimal numeric character entities, such as `&#x20;`

Six named character entities are particularly important in the processing of WML:

```
<!ENTITY quot  "&#34;">     <!-- quotation mark -->
<!ENTITY amp   "&#38;#38;"> <!-- ampersand -->
<!ENTITY lt    "&#38;#60;"> <!-- less than -->
<!ENTITY gt    "&#62;">     <!-- greater than -->
<!ENTITY nbsp  " ">    <!-- non-breaking space -->
<!ENTITY shy   "&#173;">    <!-- soft hyphen (discretionary hyphen) -->
```

# 7. WML Syntax

WML borrows most of its syntactic constructs from XML. Refer to [XML] for in-depth information on syntactical issues.

## 7.1 Entities

WML text can contain numeric or named character entities. These entities specify specific characters in the document character set. They are used to specify characters which must be escaped in WML, or which may be difficult to enter in a text editor. For example, the ampersand (&) is represented by the named entity `&`. All entities begin with an ampersand, and end with a semicolon.

WML is an XML language, and this implies that the ampersand and less-than characters must be escaped when they are used in textual data, i.e., these characters may appear in their literal form only when used as markup delimiters, within a comment, etc. See [XML] for more details.

## 7.2 Elements

Elements specify all markup and structural information about a WML deck. Elements may contain a start tag, content, and an end tag. Elements have one of two structures:

```
<tag> content </tag>
```

or

```
<tag/>
```

Elements containing content are identified by a start tag (`<tag>`) and an end tag (`</tag>`). An empty-element tag (`<tag/>`) identifies elements with no content.

## 7.3 Attributes

WML attributes specify additional information about an element. More specifically, attributes specify information about an element that is not part of the element's content. Attributes are always specified in the start tag of an element. For example,

```
<tag attr="abcd"/>
```

Attribute names are an XML `NAME` and are case sensitive.

XML requires that all attribute values be quoted using either double quotation marks (") or single quotation marks (’). Single quote marks can be included within the attribute value when the value is delimited by double quote marks, and vice versa. Character entities may be included in an attribute value.

## 7.4 Comments

WML comments follow the XML commenting style, and have the following syntax:

```
<!-- a comment -->
```

Comments are intended for use by the WML author and should not be displayed by the user agent. WML comments cannot be nested.

## 7.5 Variables

WML cards and decks can be parameterised using variables. To substitute a variable into a card or deck, the following syntaxes are used:

**$***identifier*
**$(***identifier***)**
**$(***identifier*:*conversion***)**

Parentheses are required if white space does not indicate the end of a variable. Variable syntax has the highest priority in WML, i.e., anywhere the variable syntax is legal, an unescaped '$' character indicates a variable substitution. Variable references are legal in any `PCDATA` and in any attribute value identified by the `vdata` entity type (see section 8.3).

A sequence of two dollar signs, e.g., $$ represents a single dollar sign character.

See section 10.3 for more information on variable syntax and semantics.

## 7.6 Case Sensitivity

XML is a case-sensitive language; WML inherits this characteristic. No case folding is performed when parsing a WML deck. This implies that all WML tags and attributes are case sensitive. In addition, any enumerated attribute values are case sensitive.

## 7.7 CDATA Section

WML uses XML CDATA sections to encapsulate non-WML content, e.g. scripts or other literal text. CDATA sections begin with the string "`<![CDATA[`" and end with the string "`]]>`". For example:

```
<![CDATA[ this is <B> a test ]]>
```

Any content contained in a `CDATA` section is treated as literal text for the purposes of parsing WML.

## 7.8 Processing Instructions

WML makes no use of XML processing instructions beyond those explicitly defined in the XML specification.

## 7.9 Errors

Illegal syntax must be treated as an error. Unknown elements or attributes should be ignored. Any WML deck which is not **well-formed** and **valid**, as defined by the [XML] specification must be treated as an error.

# 8. Core WML Data Types

## 8.1 Character Data

All character data in WML is defined in terms of XML data types. In summary:

- `CDATA` - text which may contain numeric or named character entities. `CDATA` is used only in attribute values.
- `PCDATA` - text which may contain numeric or named character entities. This text may contain tags (`PCDATA` is "Parsed `CDATA`"). `PCDATA` is used only in elements.
- `NMTOKEN` - a name token, containing any mixture of name characters, as defined by the XML specification.

See [XML] for more details.

## 8.2 Length

```
<!ENTITY % length  "CDATA">    <!-- nn for pixels or nn% for percentage
                                    length -->
```

The `length` type may either be specified as an integer representing the number of pixels of the canvas (screen, paper) or as a percentage of the available horizontal or vertical space. Thus, the value "50" means fifty pixels. For widths, the value "50%" means half of the available horizontal space (between margins, within a canvas, etc.). For heights, the value "50%" means half of the available vertical space (in the current window, the current canvas, etc.).

The integer value consists of one or more decimal digits ([0-9]) followed by an optional percent character (%). The `length` type is only used in attribute values.

## 8.3 Vdata

```
<!ENTITY % vdata   "CDATA">    <!-- attribute value possibly containing
                                    variable references -->
```

The `vdata` type represents a string that may contain variable references (see section 10.3). This type is only used in attribute values.

## 8.4 Flow and Inline

```
<!ENTITY % layout  "BR">
<!ENTITY % inline   "%text; | %layout;">
<!ENTITY % flow     "%inline; | IMG | A">
```

The `flow` type represents "card-level" information. The `inline` type represents "text-level" information. In general, `flow` is used anywhere general content can be included. The `inline` type indicates areas that only handle pure text or variable references.

## 8.5 URL

```
<!ENTITY % URL     "%vdata;">  <!-- URL or URN designating a hypertext
                                    node. May contain variable references -->
```

The `URL` type refers to either a relative or absolute Uniform Resource Locator [RFC1738]. See section 5 for more information.

## 8.6 Boolean

```
<!ENTITY % boolean "(TRUE|FALSE)">
```

The `boolean` type refers to a logical value of true or false.

## 8.7 Number

```
<!ENTITY % number  "NMTOKEN">  <!-- a number, with format [0-9][0-9]* -->
```

The `number` type represents an integer value.

# 9. Events and Navigation

## 9.1 Navigation and Event Handling

WML includes a navigation and event-handling model, allowing the author to specify the processing of specific user agent events. Events may be bound to *tasks* by the author; when an event occurs, the bound task is executed.

An event binding is scoped to the element in which it is declared, e.g., an event binding declared in a card is local to that card. Any event binding declared in an element is active only within that element. Event bindings specified in sub-elements take precedence over any conflicting event bindings declared in a parent element. Conflicting event bindings within an element are an error.

## 9.2 History

WML includes a simple navigation history model, allowing the author to manage backward navigation in a convenient and efficient manner. The user agent history is modelled as a stack of URLs, representing the navigational path the user traversed to arrive at the current card. There are three operations that may be performed on the history stack:

- Reset - the history stack may be reset to a state where it only contains the current card. See the `NEWCONTEXT` attribute (section 10.2) for more information.
- Push - a new URL is implicitly pushed onto the history stack as a side effect of navigation to a new card.
- Pop - the current card's URL (top of the stack) is popped as an implicit side effect of backward navigation.

The user agent must implement a navigation history. As each card is accessed via an explicitly specified URL, e.g., a `GO` task, the card URL is added to the history stack. The user agent must provide a means for the user to navigate back to the previous card in the history. Authors can depend on the existence of a user interface construct allowing the user to navigate backwards in the history. As a consequence, the author may rely on the user agent to provide default backward navigation support. The user agent must return the user to the previous card in the history if a `PREV` task is executed (see section 9.3). The execution of the PREV task pops the current card URL from the history stack. No additional variable state side effects or semantics are associated with the PREV task.

## 9.3 Tasks

```
<!ENTITY % tasktypes "(GO|PREV|NOOP)">
<!ENTITY % taskattrs "
  URL             %URL;           #IMPLIED
  VARS            %vdata;         #IMPLIED
  SENDREFERER     %boolean;       'FALSE'
  METHOD          (POST|GET)      'GET'
  ACCEPT-CHARSET  CDATA           #IMPLIED
  POSTDATA        %vdata;         #IMPLIED"
  >
<!ENTITY % task "
  TASK            %tasktypes;     'GO'
  %taskattrs;"
  >
```

Tasks specify processing that should be performed in response to an event. Tasks are typically used to specify a transition to another URL, which may name a WML card, or some other content.

*Attributes*

TASK=*tasktypes*

The following table describes the tasks and lists the attributes required for each. Attributes in bold are required for that task.

**Table 1. Task names**

| ***Task*** | ***Description*** | ***Attributes (required if bold)*** |
|---|---|---|
| GO | Navigate to the specified URL. If this URL names a WML card or deck, it is displayed. Implicitly executes a "push" on the history stack. | **URL**, VARS, SENDREFERER, METHOD, ACCEPT-CHARSET, POSTDATA |
| NOOP | Do nothing. | |
| PREV | Navigate to the previous URL in the history stack. Implicitly executes a "pop" on the history stack. | VARS |

URL=*URL*

The URL attribute specifies the destination URL of this task, e.g., the URL of the card to display or the script to invoke.

VARS=*vdata*

The VARS attribute specifies the variables to set in the current browser context as a side effect of executing this task. The variables must be specified in URL query-string format. For example:

```
VARS="var1=value1&amp;var2=value2"
```

The values must be escaped according to URL escaping conventions. The user agent must unescape the VARS attribute before setting the value of the variables. See section 10.3.4 for more information on setting variables.

SENDREFERER=*boolean*

This attribute specifies whether the user agent should specify the URL of the current deck (i.e. the referring deck) when requesting the next URL from a server. This allows a server to perform a form of access control on URLs, based on which decks are linking to them. The URL must be the smallest relative URL possible if it can be relative at all. For example, if SENDREFERER=TRUE, an HTTP based user agent shall indicate the URL of the current deck in the HTTP "Referer" request header [RFC2068].

METHOD=*(POST|GET)*

This attribute specifies the HTTP submission method. Currently, the values of GET and POST are accepted, and cause the user agent to perform an HTTP GET or POST respectively. If METHOD is not specified, the user agent must use the GET method, unless the POSTDATA attribute is present, in which case the user agent must use the POST method.

ACCEPT-CHARSET=*cdata*

This attribute specifies the list of character encodings for data that the origin server must accept when processing input. The value of this attribute is a comma- or space-separated list of character encoding scheme names (charset) as specified in [RFC2045] and [RFC2068]. The IANA Character Set registry defines the public registry for charset values. This list is an exclusive-OR list, i.e., the server must accept any one of the acceptable character encodings.

The default value for this attribute is the reserved string UNKNOWN. User agents should interpret this value as the character encoding that was used to transmit the WML deck containing this attribute.

POSTDATA=*vdata*

This attribute specifies data to be posted to the server. The data is sent to the server as an application/x-www-form-urlencoded entity. The data is formatted as stream of octets, encoding using the URL escaping mechanism specified in [RFC1738].

Specifically, the following occurs:

1. The user agent should transcode the input data to the correct character set, as specified explicitly by ACCEPT-CHARSET, or implicitly by the document encoding.
2. The data is escaped using URL escaping. Any characters outside the legal URL character set will be converted into the sequence %XX, where XX is the octet represented as a hexadecimal number.
3. The resulting string is transmitted to the server in an application/x-www-form-urlencoded entity.

Note that the resulting character set is not indicated in the POST. The client must explicitly or implicitly specify the required character set in any situation where it is ambiguous.

This attribute is ignored if the METHOD attribute has a value of GET.

## 9.4 Card/Deck Task Shadowing

A variety of elements can be used to associate a task with an event. Certain elements specify event-handling behaviour for an entire card, e.g. DO and ONEVENT, and may appear at the card and deck-level:

- Card-level: the event-handling element may appear inside a card element, e.g. FORMCARD, and specify event-processing behaviour for that particular card.
- Deck-level: the event-handling element may appear inside a COMMON element, and specify event-processing behaviour for all cards in the deck. A deck-level event-handling element is equivalent to specifying the event-handling element in each card.

A card-level event-handling element overrides (or "shadows") a deck-level event-handling element if they both specify the same event. A card-level ONEVENT element will shadow a deck-level ONEVENT element if they both have the same TYPE. A card-level DO element will shadow a deck-level DO element if they have the same NAME.

## 9.5 The DO Element

```
<!ELEMENT DO EMPTY>
<!ATTLIST DO
  TYPE         CDATA       #REQUIRED
  LABEL        %vdata;     #IMPLIED
  NAME         NMTOKEN     #IMPLIED
  OPTIONAL     %boolean;   'FALSE'
  %task;
  >
```

The DO element provides a general mechanism for the user to act upon the current card, i.e. a card-level user interface element. The representation of the DO element is user agent dependent, and the author must only assume that the element is mapped to a unique user interface *widget* that the user can activate. For example, the widget mapping may be to a graphically rendered button, a soft or function key, a voice-activated command sequence, or any other interface that has a simple "activate" operation with no inter-operation persistent state.

The TYPE attribute is provided as a hint to the user agent about the author's intended use of the element, and should be used by the user agent to provide a suitable mapping onto a physical user interface construct. WML authors must not rely on the semantics or behaviour of a particular TYPE, or on the mapping of that TYPE to a particular physical construct.

The DO element may appear at both the card and deck-level:

- Card-level: the DO element may appear inside a card element, e.g. FORMCARD, and may be located anywhere in the text flow. If the user agent intends to render the DO element inline (i.e. in the text flow), it should use the element's anchor point as the rendering point. WML authors must not rely on the inline rendering of the DO element, and must not rely on the correct positioning of an inline rendering of the element.

- Deck-level: the DO element may appear inside a COMMON element, indicating a deck-level DO element. A deck-level DO element applies to all cards in the deck (i.e. is equivalent to having specified the DO within each card). For the purposes of inline rendering, the user agent must assume that deck-level DO elements are inserted at the end of the card's text flow.

A card-level DO element overrides (or "shadows") a deck-level DO element if they have the same NAME (see section 9.4 for more details). With the exception of shadowed elements, all DO elements specified in a card or deck must be made accessible to the user in some form, i.e., it must be possible for the user to activate these user interface items when viewing the card containing the element. When the user activates the DO element, the associated task is executed.

## *Attributes*

TYPE=*cdata*

The DO element type. This attribute provides a hint to the user agent about the author's intended use of the element, and how it should be mapped to a physical user interface construct. All types are reserved, except for those explicitly marked as not reserved.

User agents must accept any TYPE, but may treat any unrecognised type as the equivalent of UNKNOWN.

In the following table, the * character represents any string, i.e. Test* indicates any string starting with the word Test.

**Table 2. Pre-defined DO types**

| ***Type*** | ***Description*** |
|---|---|
| ACCEPT | Positive acknowledgement (acceptance) |
| PREV | Backward history navigation |
| HELP | Request for help. May be context-sensitive. |
| RESET | Clearing or resetting state. |
| OPTIONS | Context-sensitive request for options or additional operations. |
| DELETE | Delete item or choice. |
| UNKNOWN | A generic DO element. Equivalent to an empty string (e.g. TYPE=""). |
| X-*, x-* | Experimental types. This set is not reserved. |
| vnd.*, VND.* and any combination of [Vv][Nn][Dd].* | Vendor-specific or user-agent-specific types. This set is not reserved. Vendors should allocate names with the format VND.CO-TYPE, where CO is a company name abbreviation and TYPE is the DO element type. See [RFC2045] for more information. |

LABEL=*vdata*

If the user agent is able to dynamically label the user interface widget, this attribute specifies a textual string suitable for such labelling. The user agent must make a best-effort attempt to label the UI widget, and should adapt the label to the constraints of the widget (e.g. truncate the string). If an element can not be dynamically labeled, this attribute may be ignored.

To work well on a wide variety of user agents, it is suggested that authors limit labels to text strings of six characters or shorter in length

NAME=*nmtoken*

This attribute specifies the name of the DO event binding. If two DO elements are specified with the same name, they refer to the same binding. If DO elements are specified both at the card-level (in a FORMCARD element) and at the deck-level (in a COMMON element) and both elements have the same NAME, the deck-level DO element is ignored. It is an error to specify two or more DO elements with the same NAME in a single card

or in the COMMON element. A NAME with an empty value is equivalent to unspecified NAME attribute. An unspecified NAME defaults to the value of the TYPE attribute.

OPTIONAL=*boolean*

If this attribute has a value of TRUE, the user agent may ignore this element.

### *Attributes Specified Elsewhere*

The following attributes are defined in section 9.3:

%task

## 9.6 Anchored Links - the A Elements

```
<!ELEMENT A (%inline;)*>
<!ATTLIST A
  TITLE        %vdata;      #IMPLIED
  %task;
  >
```

The anchored link element specifies the head of a link. The tail of a link is specified as part of other elements (e.g. a card name attribute). Anchored links may not be nested.

Anchors may be present in any text flow, excluding the text in OPTION elements (i.e. anywhere formatted text is legal, except for OPTION elements). Anchored links have an associated *task* that specifies the behaviour when the anchor is selected.

### *Attributes*

TITLE=*vdata*

This attribute specifies a brief text string identifying the link. The user agent may display it in a variety of ways, including dynamic labelling of a button or key, a *tool tip*, a voice prompt, etc. The user agent may truncate or ignore this attribute depending on the characteristics of the navigational user interface. To work well on a broad range of user agents, the author should limit all labels to 6 characters in length.

### *Attributes Defined Elsewhere*

The following task attributes are defined in section 9.3:

%task

## 9.7 Intrinsic Events

A variety of WML elements are capable of generating events when the user interacts with them. These events are called "intrinsic events", and indicate state transitions inside the user agent. Individual elements specify the events they can generate. WML defines the following intrinsic events:

**Table 3. WML Intrinsic Events**

| ***Event*** | ***Element(s)*** | ***Description*** |
|---|---|---|
| ONENTERFORWARD | Cards: FORMCARD, CHOICE, ENTRY, DISPLAY, NODISPLAY, COMMON | The ONENTERFORWARD event occurs when the user causes the user agent to enter a card using a GO task or any method with identical semantics. This includes card entry caused by a script function or user-agent-specific mechanisms, such as a means to directly enter and navigate to a URL.<br><br>The ONENTERFORWARD intrinsic event may be specified at both the card and deck-level. Event handlers specified in the COMMON element apply to all cards in the deck, and may be overridden as specified in section 9.4. |

| ***Event*** | ***Element(s)*** | ***Description*** |
|---|---|---|
| ONENTERBACKWARD | Cards: FORMCARD, CHOICE, ENTRY, DISPLAY, NODISPLAY, COMMON | The ONENTERBACKWARD event occurs when the user causes the user agent to navigate into a card using a PREV task or any method with identical semantics. In other words, the ONENTERBACKWARD event occurs when the user causes the user agent to navigate into a card by using a URL retrieved from the history stack. This includes navigation caused by a script function or user-agent-specific mechanisms.<br><br>The ONENTERBACKWARD intrinsic event may be specified at both the card and deck-level. Event handlers specified in the COMMON element apply to all cards in the deck, and may be overridden as specified in section 9.4. |
| ONCLICK | OPTION | The ONCLICK event occurs when the user selects or deselects this item. |

The author may specify that certain tasks are to be executed when an intrinsic event occurs. This specification may take one of two forms. The first form specifies a URL to be navigated to when the event occurs. This event binding is specified in a well-defined element-specific attribute, and is the equivalent of a GO task. For example:

```
<FORMCARD ONENTERFORWARD="/url"> hello </FORMCARD>
```

This attribute value may only specific a URL.

The second form is an expanded version of the previous, allowing the author more control over user agent behaviour. An ONEVENT element is declared within a parent element, specifying the full event binding for a particular intrinsic event. For example, the following is identical to the previous example:

```
<FORMCARD>
   <ONEVENT TYPE="ONENTERFORWARD" TASK="GO" URL="/url"/>
   Hello
</FORMCARD>
```

The user agent must treat the attribute syntax as an abbreviated form of the ONEVENT element where the attribute name is mapped to the ONEVENT type.

An intrinsic event binding is scoped to the element in which it is declared, e.g., an event binding declared in a card is local to that card. Any event binding declared in an element is active only within that element. Event bindings specified in sub-elements take precedence over any conflicting event bindings declared in a parent element. Conflicting event bindings within an element are an error.

## 9.7.1 The ONEVENT Element

```
<!ELEMENT ONEVENT EMPTY>
<!ATTLIST ONEVENT
  TYPE         CDATA       #REQUIRED
  %task;
  >
```

The ONEVENT element binds a task to a particular intrinsic event for the immediately enclosing element, i.e., specifying an ONEVENT element inside a "XYZ" element binds an intrinsic event handler to the "XYZ" element.

The user agent must ignore any ONEVENT element specifying a TYPE that does not correspond to a legal intrinsic event for the immediately enclosing element.

### *Attributes*

TYPE=*cdata*

The TYPE attribute indicates the name of the intrinsic event.

*Attributes Defined Elsewhere*

The following task attributes are defined in section 9.3:
`%task`

## 9.7.2 Card/Deck Intrinsic Events

The `ONENTERFORWARD` and `ONENTERBACKWARD` intrinsic events may be specified at both the card- and deck-level, and have the shadowing semantics defined in section 9.4. Intrinsic events may be overridden regardless of the syntax used to specify them. A deck-level event-handler specified with the `ONEVENT` element may be overridden by the `ONEVENTFORWARD` attribute, and vice versa.

# 10. The State Model

WML includes support for managing user agent state, including:

- Variables - parameters used to change the characteristics and content of a WML card or deck
- History - navigational history, which may be used to facilitate efficient backwards navigation
- Implementation-dependent state - other state relating to the particulars of the user agent implementation and behaviour

## 10.1 The Browser Context

WML state is stored in a single scope, known as a *browser context.* The browser context is used to manage all parameters and user agent state, including variables, the navigation history, and other implementation-dependent information related to the current state of the user agent.

## 10.2 The NEWCONTEXT Attribute

The browser context may be initialised to a well-defined state by the `NEWCONTEXT` attribute of the card elements (see section 11.4). This attribute indicates that the browser context should be re-initialised, and must perform the following operations:

- Unset (remove) all variables defined in the current browser context
- Clear the navigational history state
- Reset implementation-specific state to a well-known value

`NEWCONTEXT` is not performed on `PREV` tasks. See section 12.4 for more information on the processing of state during navigation.

## 10.3 Variables

All WML content can be parameterised, allowing the author a great deal of flexibility in creating cards and decks with improved caching behaviour and better perceived interactivity. WML variables can be used in the place of strings and are substituted at run-time with their current value.

A variable is said to be *set* if it has a value not equal to the empty string. A value is *not set* if it has a value equal to the empty string, or is otherwise unknown or undefined in the current browser context.

### 10.3.1 Variable Substitution

The values of variables can be substituted into both the text (`#PCDATA`) of a card and into `%vdata` and `%URL` attribute values in WML elements. Only textual information can be substituted; no substitution of elements or attributes is possible. The substitution of variable values happens at run-time in the user agent. Substitution does not affect the current value of the variable, and is defined as a simple string substitution. If an undefined variable is referenced, it results in the substitution of the empty string.

WML variable names consist of an US-ASCII letter or underscore followed by zero or more letters, digits or underscores. Any other characters are illegal. Variable names are case sensitive.

The following is a BNF-like description of the variable substitution syntax. The description uses the conventions established in [RFC822], except that the "`|`" character is used to designate alternatives. Briefly, "`(`" and "`)`" are used to group elements, optional elements are enclosed in "`[`" and "`]`", and elements may be preceded with `<N>*` to specify N or more repetitions of the following element (`N` defaults to zero when unspecified).

```
var     = ( "$" varname ) |
          ( "$(" varname [ conv ] ")" )

conv    = ":" ( escape | noesc | unesc )
escape  = ("E" | "e") [ ( "S" | "s" ) ( "C" | "c" )
                        ( "A" | "a" ) ( "P" | "p" )
                        ( "E" | "e" ) ]
noesc   = ( "N" | "n" ) [ ( "O" | "o" ) ("E" | "e")
                          ( "S" | "s" ) ( "C" | "c" ) ]
unesc   = ( "U" | "u" ) [ ( "N" | "n" ) ("E" | "e")
                          ( "S" | "s" ) ( "C" | "c" ) ]

varname = ( "_" | alpha ) *[ "_" | alpha | digit ]
alpha   = lowalpha | hialpha
lalpha  = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
          "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
          "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
halpha  = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
          "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
          "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
digit   = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
          "8" | "9"
```

Parentheses are required anywhere the end of a variable can not be inferred from the surrounding context, e.g. an illegal character such as white space.

For example:

```
This is a $var
This is another $(var).
    This is an escaped $(var:e).
Long form of escaped $(var:escape).
Long form of unescape $(var:unesc).
Short form of no-escape $(var:N).
Other legal variable forms: $_X $X32 $Test_9A
```

The value of variables can be converted into a different form as they are substituted. A conversion can be specified in the variable reference following the colon. The following table summarised the current conversions and their legal abbreviations:

**Table 4. Variable escaping methods**

| ***Conversion*** | ***Effect*** |
|---|---|
| noesc | No change to the value of the variable. |
| escape | URL escape the value of the variable. |
| unesc | URL unescape the value of the variable. |

The use of a conversion during variable substitution does not affect the actual value of the variable.

URL escaping is detailed in [RFC1738]. All lexically sensitive characters defined in WML must be escaped. These include all `reserved` and `unsafe` URL characters, and characters reserved by WML-Script syntax (left parenthesis, right parenthesis and comma [ASCII `40`, `41` & `44`]).

If no conversion is specified, the variable is substituted using the conversion format appropriate for the context. The `ONENTERBACKWARD`, `ONENTERFORWARD`, `URL`, `SRC`, and `VARS` attributes default to escape conversion, elsewhere no conversion is done. Specifying the `noesc` conversion disables context sensitive escaping of a variable.

## 10.3.2 Parsing the Variable Substitution Syntax

The variable substitution syntax (e.g. $X) is parsed after all XML parsing is complete. In XML terminology, variable substitution is parsed after the *XML processor* has parsed the document, and provided the resulting parsed form to the *XML application*. In the context of this specification, the WML parser and user agent is the *XML application*.

This implies that all variable syntax is parsed *after* the XML constructs, such as tags and entities, have been parsed. In the context of variable parsing, all XML syntax has a higher precedence than the variable syntax, e.g., entity substitution occurs before the variable substitution syntax is parsed. The following examples are identical references to the variable named X:

```
$X
&#x24;X
$&#x58;
&#36;&#x58;
```

## 10.3.3 The Dollar-sign Character

A side effect of the parsing rules is that the literal dollar sign must be encoded with a pair of dollar sign entities. A single dollar-sign entity, even specified as $, results in a variable substitution.

In order to include a '$' character in a WML deck, it must be explicitly escaped. This can be accomplished with the following syntax:

```
$$
```

Two dollar signs in a row are replaced with a single '$' character. For example:

```
This is a $$ character.
```

This would be displayed as:

```
This is a $ character.
```

To include the '$' character in URL escaped strings (e.g. in a VARS attribute), specify it with the URL escaped form:

```
%24
```

## 10.3.4 Setting Variables

There are a number of ways to set the value of a variable. Input elements set the variable identified by the KEY attribute to any information entered by the user, e.g., an INPUT element assigns the entered text to the variable, and the SELECT element assigns the value present in the chosen OPTION element's VALUE attribute. Variables can also be set as a side effect of card-to-card navigation, by using the VARS attribute.

When a variable is set, and it is already defined in the browser context, the current value is updated.

The VARS attribute allows the author to set variable state as a side effect of navigation, and may be specified in a variety of event handling elements (e.g. A, ONEVENT and DO). The value of the VARS attribute is an ampersand-delimited list of variable names and values. For example:

```
VARS="name=John&amp;location=home&amp;state=$(s:E)"
```

The variables identified in the list (e.g. location) are set as a side effect of navigation. See the discussion of event handling (section 9 and section 12.4) for more information on the processing of the VARS attribute.

Pending user input (e.g. in an INPUT element) is written to variables when a task is executed (e.g. GO or PREV). Input is committed to the activity immediately before the task is executed, allowing the use of the variables in the execution of the task.

# 11. The Structure of WML Decks

WML data are structured as a collection of *cards*. A single collection of cards is referred to as a WML *deck*. Each card contains structured content and navigation specifications. Logically, a user navigates through a series of cards, reviews the contents of each, enters requested information, makes choices, and navigates to another card or returns to a previously visited card.

## 11.1 Document Prologue

A *valid* WML deck is a *valid* XML document, and therefore must contain an XML declaration and a document type declaration (see [XML] for more detail about the definition of a valid document). A typical document prologue contains:

```
<?xml version="1.0"?>
<!DOCTYPE WML PUBLIC "-//WAPFORUM//DTD WML 1.0//EN">
```

It is an error to omit the prologue.

## 11.2 The WML Element

```
<!ENTITY % cards "FORMCARD | DISPLAY | CHOICE | ENTRY | NODISPLAY">
<!ELEMENT WML ( (COMMON, (%cards;)* ) | (%cards;)+ )>
<!ATTLIST WML
  VERSION        NMTOKEN     #FIXED      '1.0'
  >
```

The WML element defines a deck, and encloses all information and cards in the deck.

*Attributes*

`VERSION='1.0'`

The WML language version number. Authors should declare the version number in any content. This attribute defaults to the current WML version ("1.0").

### 11.2.1 A WML Example

The following is a deck containing two cards, each represented by a `FORMCARD` element (see section 11.4 for information on cards). After loading the deck, a user agent displays the first card. If the user activates the DO element, the user agent displays the second card.

```
<WML>
  <FORMCARD>
    <DO TYPE="ACCEPT" TASK="GO" URL="#card2"/>
    Hello world!
    This is the first card...
  </FORMCARD>

  <FORMCARD NAME="card2">
    This is the second card.
    Goodbye.
  </FORMCARD>
</WML>
```

# 11.3 Common Declarations

## 11.3.1 The COMMON Element

```
<!ENTITY % navelmts "DO | ONEVENT">

<!ELEMENT COMMON (ACCESS | META | SCRIPT | %navelmts;)*>
<!ATTLIST COMMON
  %cardev;
  >
```

The COMMON element contains information relating to the deck as a whole. A COMMON element may contain meta-data, scripts, access control specifications, and deck-level navigation and event elements.

Event bindings specified in the COMMON element (e.g. DO or ONEVENT) apply to all cards in the deck. Specifying an event binding in the COMMON element is equivalent to specifying it in every card element. A card element may override the behaviour specified in the COMMON element. In particular:

- DO elements specified in the COMMON element may be overridden in individual cards if both elements have the same NAME attribute value. See section 9.4 for more information.
- Intrinsic event bindings specified in the COMMON element may be overridden by the specification of an event binding in a card element. See section 9.7 for more information.

See section 11.4 for the definition of the card-level intrinsic events (the cardev entity).

### *Attributes Defined Elsewhere*

The following task attributes are defined in section 11.4.1:

%cardev

## 11.3.2 The ACCESS Element

```
<!ELEMENT ACCESS EMPTY>
<!ATTLIST ACCESS
  DOMAIN       CDATA       #IMPLIED
  PATH         CDATA       #IMPLIED
  PUBLIC       %boolean;   'FALSE'
  >
```

The ACCESS element specifies access control information for the entire deck. It is an error for a deck to contain more than one ACCESS element.

### *Attributes*

DOMAIN=*cdata*
PATH=*cdata*

A deck's DOMAIN and PATH attributes specify which other decks may access it. As the user agent navigates from one deck to another, it performs access control checks to determine whether the destination deck allows access from the current deck.

If a deck has a DOMAIN and/or PATH attribute, the referring deck's URL must match the values of the attributes. Matching is done as follows: the access domain is suffix-matched against the domain name portion of the referring URL, and the access path is prefix matched against the path portion of the referring URL.

DOMAIN suffix matching is done using the entire element of each sub-domain, and must match each element exactly (e.g. www.wapforum.org shall match wapforum.org, but shall not match forum.org). PATH prefix matching is done using entire path elements, and must match each element exactly (e.g. /X/Y matches /X, but does not match /XZ).

The DOMAIN attribute defaults to the current deck's domain. The PATH attribute defaults to the value "/".

To simplify the development of applications that may not know the absolute path to the current deck, the PATH attribute accepts relative URLs. The user agent converts the relative path to an absolute path and then performs prefix matching against the PATH attribute.

Given the following access control attributes:

```
DOMAIN="wapforum.org"
PATH="/cbb"
```

The following referring URLs would be allowed to go to the deck:

```
http://wapforum.org/cbb/stocks.cgi
https://www.wapforum.org/cbb/bonds.cgi
http://www.wapforum.org/cbb/demos/alpha/packages.cgi?x=123&y=456
```

The following referring URLs would not be allowed to go to the deck:

```
http://www.test.net/cbb
http://www.wapforum.org/internal/foo.wml
```

DOMAIN and PATH follow URL capitalisation rules.

PUBLIC=*boolean*

This attribute indicates whether deck access control has been disabled for this deck. If disabled, i.e. PUBLIC="TRUE" is specified, cards in any deck can access this deck. If enabled, then the DOMAIN and PATH attributes used to determine which cards or decks can access the deck. By default, access control is enabled.

## 11.3.3 The META Element

```
<!ELEMENT META EMPTY>
<!ATTLIST META
  HTTP-EQUIV     CDATA      #IMPLIED
  NAME           CDATA      #IMPLIED
  USER-AGENT     CDATA      #IMPLIED
  CONTENT        CDATA      #REQUIRED
  SCHEME         CDATA      #IMPLIED
  >
```

The META element contains generic meta-information relating to the WML deck. Meta-information is specified with property names and values. This specification does not define any properties, nor does it define how user agents must interpret meta-data. User agents are not required to support the meta-data mechanism.

### *Attributes*

NAME=*cdata*

This attribute specifies the property name. The user agent must ignore any meta-data named with this attribute. Network servers should not emit WML content containing meta-data named with this attribute.

HTTP-EQUIV=*cdata*

This attribute may be used in place of NAME, and indicates that the property should be interpreted as an HTTP header (see [RFC2068]). Meta-data named with this attribute should be converted to a WSP or HTTP response header if the content is tokenized before it arrives at the user agent (see appendix A).

USER-AGENT=*cdata*

This attribute may be used in place of NAME. This meta-data must be delivered to the user agent, and may not be removed by any network intermediary.

CONTENT=*cdata*

This attribute specifies the property value.

SCHEME=*cdata*

This attribute specifies a form or structure that may be used to interpret the property value. Scheme values vary depending on the type of meta-data.

## 11.3.4 The SCRIPT Element

```
<!ELEMENT SCRIPT (#PCDATA)>
<!ATTLIST SCRIPT
  TYPE     CDATA   #REQUIRED
  >
```

The SCRIPT element allows scripts to be defined in a deck. Scripts are evaluated by a *script interpreter*, which must be known to the user agent. A user agent may ignore any script types that it does not recognise. The semantics of a script are entirely determined by the script interpreter. It is an error to specify more than one script element of a given type, where type is indicated by the value of the TYPE attribute.

A script may be invoked by any event binding, for example, the DO, ONEVENT and A elements. For more information, see section 12.5.

All script data within the SCRIPT element must be escaped so that the *less-than* character ("<") never appears within the script. This allows the WML element parser to locate the SCRIPT end tag successfully. The XML CDATA section syntax may be used for this purpose, e.g.,

```
<SCRIPT TYPE="text/wmlscript"> <![CDATA[
     function foo() { doit("<testing>"); }
]]> </SCRIPT>
```

*Attributes*

TYPE=*cdata*

This attribute defines the scripting language present in this element. The attribute's value must be a media type, as defined by [RFC2048].

# 11.4 The Card Elements

A WML deck contains a collection of cards. There are a variety of card types, each specifying a different mode of user interaction.

## 11.4.1 Card Attributes

```
<!ENTITY % cardev
 "ONENTERFORWARD  %URL;          #IMPLIED
  ONENTERBACKWARD %URL;          #IMPLIED"
  >
<!ENTITY % cardattrs
 "NAME            NMTOKEN       #IMPLIED
  TITLE           %vdata;       #IMPLIED
  NEWCONTEXT      %boolean;     'FALSE'
  %cardev;"
  >
```

The following attributes are available in all types of WML cards.

### *Attributes*

`NAME=`*`nmtoken`*

This attribute gives a name to the card. A card's name may be used as a fragment anchor, allowing it to be linked to. See section 5.2.

`TITLE=`*`vdata`*

The `TITLE` attribute specifies advisory information about the card. The title may be rendered in a variety of ways by the user agent (e.g. suggested bookmark name, pop-up *tooltip*, etc.).

`NEWCONTEXT=`*`boolean`*

This attribute indicates that the current browser context should be re-initialised upon entry to this card. See section 10.2 for more information.

`ONENTERFORWARD=`*`URL`*

The `ONENTERFORWARD` event occurs when the user causes the user agent to navigate into a card using a `GO` task.

`ONENTERBACKWARD=`*`URL`*

The `ONENTERBACKWARD` event occurs when the user causes the user agent to navigate into a card using a `PREV` task.

## 11.4.2 The TABINDEX Attribute

### *Attributes*

`TABINDEX=`*`number`*

This attribute specifies the tabbing position of the current element. The tabbing position indicates the relative order in which elements are traversed when tabbing within a single WML card. A numerically greater `TABINDEX` value indicates an element that is later in the tab sequence than an element with a numerically lesser `TABINDEX` value.

Each input element (i.e., `INPUT` and `SELECT`) in a card is assigned a position in the card's tab sequence. In addition, the user agent may assign a tab position to other elements. The `TABINDEX` attribute indicates the tab position of a given element. Elements that are not designated with an author-specified tab position may be assigned one by the user agent. User agent specified tab positions must be later in the tab sequence than any author-specified tab positions.

Tabbing is a navigational accelerator, and is optional for all user agents. Authors must not assume that a user agent implements tabbing.

## 11.4.3 The FORMCARD Element

```
<!ENTITY % fields  "%flow; | INPUT | SELECT | FIELDSET">
<!ELEMENT FORMCARD (%fields; | %navelmts;)*>
<!ATTLIST FORMCARD
  %cardattrs;
  STYLE           (LIST|SET)    'LIST'
  >
```

The FORMCARD element is a container of text and input elements that is sufficiently flexible to allow presentation and layout in a wide variety of devices, with a wide variety of display and input characteristics. The FORMCARD element indicates the general layout and required input fields, but does not overly constrain the user agent implementation in the areas of layout or user input. For example, a FORMCARD can be presented as a single page on a large-screen device, and as a series of smaller pages on a small-screen device.

A FORMCARD can contain markup, input fields, and elements indicating the structure of the card. The order of elements in the card is significant, and should be respected by the user agent.

User input is committed to variables when any task is executed (see section 10.3.4).

### *Attributes*

STYLE=*(LIST|SET)*

This attribute specifies a hint to the user agent about the organisation of the FORMCARD content. This hint may be used to organise the content presentation or to otherwise influence layout of the card.

- LIST - the card is naturally organised as a linear sequence of field elements, e.g. a set of questions or fields which are naturally handled by the user in the order in which they are specified in the group. This style is best for short forms in which no fields are optional (e.g. sending an email message requires a To: address, a subject and a message, and they are logically specified in this order).

  It is expected that in small-screen devices, LIST groups may be presented as a sequence of screens, with a screen flip in between each field or fieldset. Other user agents may elect to present all fields simultaneously.

- SET - the card is a collection of field elements without a natural order. This is useful for collections of fields containing optional or unordered components or simple record data where the user is updating individual input fields.

  It is expected that in small-screen devices, SET groups may be presented by using a hierarchical or tree organisation. In these types of presentation, the TITLE attribute of each field and fieldset may be used to define the name presented to the user in the top-level summary card.

The user agent may interpret the style attribute in a manner appropriate to its device capabilities (e.g. screen size or input device). In addition, the user agent should adopt user interface conventions for handling the editing of input elements in a manner that best suits the device's input model.

For example, a phone-class device displaying a FORMCARD with STYLE=SET may use a soft key or button to select individual fields for editing or viewing. A PDA-class device might create soft buttons on demand, or simply present all fields on the screen for direct manipulation.

On devices with limited display capabilities, it is often necessary to insert card flips or other user-interface transitions between fields. When this is done, the user agent needs to decide on the proper boundary between fields. User agents may use the following heuristic for determining the choice of a card flip location:

- FIELDSET defines a logical boundary between fields.
- Fields (e.g. INPUT) may be individually displayed. When this is done, the line of markup (flow) immediately preceding the field should be treated as a field prompt, and displayed with the input element.

*Attributes Defined Elsewhere*

The following task attributes are defined in section 11.4.1:

`%cardattrs`

### 11.4.3.1 A FORMCARD Example

The following is an example of a simple `FORMCARD` element embedded within a WML deck. The card contains text, which is displayed by the user agent. In addition, the example demonstrates the use of a simple `DO` element, defined at the deck level.

```
<WML>
   <COMMON>
      <DO TYPE="ACCEPT" TASK="PREV"/>
   </COMMON>
   <FORMCARD>
      Hello World!
   </FORMCARD>
</WML>
```

## 11.4.4 The CHOICE Element

```
<!ELEMENT CHOICE (%inline; | %navelmts; | CE)*>
<!ATTLIST CHOICE
  %cardattrs;
  KEY           NMTOKEN            #IMPLIED
  DEFAULT       %vdata;            #IMPLIED
  IKEY          NMTOKEN            #IMPLIED
  IDEFAULT      %vdata;            #IMPLIED
  >
<!ELEMENT CE (%text;)*>
<!ATTLIST CE
  VALUE        %vdata;     #IMPLIED
  TASK         %tasktypes; #IMPLIED
  %taskattrs;
  >
```

The `CHOICE` element is **Deprecated**. Authors should use the FORMCARD and SELECT elements.

The `CHOICE` element describes a single choice card, which contains choices specified using the `CE` element. Choice cards let users pick from a list of choices, and are identical in behaviour to a single-choice select list. The initial display content is shown to the user, followed by the choices. Each choice item in a choice card can have one line of formatted text (which may be wrapped or truncated by the user agent if too long). The `CHOICE` card is equivalent to a `FORM` card containing only `text` markup and navigation elements, followed by a single select list. See section 11.4.3.1 for more information on the `SELECT` element and select list behaviour.

For example, the following `CHOICE` and `FORMCARD` elements are identical in their behaviour and semantics:

```
<CHOICE KEY="Y">
  Pick a choice:
  <CE VALUE="1">1</CE>
  <CE VALUE="2">2</CE>
</CHOICE>
```

```
<FORMCARD>
  Pick a choice:
  <SELECT KEY="Y">
     <OPTION VALUE="1">1</OPTION>
     <OPTION VALUE="2">2</OPTION>
   </SELECT>
</FORMCARD>
```

*Attributes Defined Elsewhere*

The following task attributes are defined in section 9.3:

`TASK`

The following task attributes are defined in section 11.4.1:

`%cardattrs`

The following task attributes are defined in section 11.5.1.1:

`KEY`
`DEFAULT`
`IKEY`
`IDEFAULT`

The following task attributes are defined in section 11.5.1.2:

`VALUE`

## 11.4.5 The DISPLAY Element

```
<!ELEMENT DISPLAY (%inline; | %navelmts;)* >
<!ATTLIST DISPLAY
  %cardattrs;
  >
```

The `DISPLAY` element is **Deprecated**. Authors should use the `FORMCARD` element.

`DISPLAY` cards contain information to be presented to the user. This information includes `inline` markup, i.e. structured text, images and links. A `DISPLAY` card is equivalent to a `FORMCARD` containing only `inline` markup and navigation elements (i.e. no input elements).

For example, the following `DISPLAY` and `FORMCARD` elements are identical in their behaviour and semantics:

```
<DISPLAY>
  Hello World!
</DISPLAY>
```

```
<FORMCARD>
  Hello World!
</FORMCARD>
```

*Attributes Defined Elsewhere*

The following task attributes are defined in section 11.4.1:

`%cardattrs`

## 11.4.6 The ENTRY Element

```
<!ELEMENT ENTRY (%inline; | %navelmts;)*>
<!ATTLIST ENTRY
  %cardattrs;
  KEY           NMTOKEN         #REQUIRED
  DEFAULT       %vdata;         #IMPLIED
  FORMAT        CDATA           #IMPLIED
  NOECHO        %boolean;       'FALSE'
  EMPTYOK       %boolean;       'FALSE'
  >
```

The `ENTRY` element is **Deprecated**. Authors should use the FORMCARD and INPUT elements.

The `ENTRY` element specifies a request for user input. The text prompt (`flow`) is presented to the user, followed by an input field. The user input is constrained by the optional `FORMAT` attribute. The `ENTRY` element is equivalent to a `FORMCARD` element containing only `inline` markup and navigation elements followed by a single `INPUT` element. See section 11.5.2 for more information on the `INPUT` element.

For example, the following `ENTRY` and `FORMCARD` elements are identical in their behaviour and semantics:

```
<ENTRY KEY="X">
   Enter name:
```

```
</ENTRY>
```

```
<FORMCARD>
   Enter name:
     <INPUT TYPE="TEXT" KEY="X"/>
</FORMCARD>
```

*Attributes*

NOECHO=*boolean*

The NOECHO attribute indicates whether text entry should echo the input as entered, or whether the echoed input should be obscured in some manner. If NOECHO is specified, the semantics are identical to specifying TYPE=PASSWORD in an INPUT element. If NOECHO is not specified, the semantics are identical to TYPE=TEXT in an INPUT element.

EMPTYOK=*boolean*

The EMPTYOK attribute indicates that this text entry accepts empty input even though a non-empty format string has been specified. Typically, the EMPTYOK attribute is indicated for formatted entry fields that are optional. By default, entry elements require the user to input data.

*Attributes Defined Elsewhere*

The following task attributes are defined in section 11.4.1:

%cardattrs

The following task attributes are defined in section 11.5.1.1:

KEY
DEFAULT
IKEY
IDEFAULT

## 11.4.7 The NODISPLAY Element

```
<!ELEMENT NODISPLAY (ONEVENT)* >
<!ATTLIST NODISPLAY
  %cardattrs;
  >
```

The NODISPLAY element is **Deprecated**. Authors should use the FORMCARD elements, with the ONENTERFORWARD and ONENTERBACKWARD intrinsic events.

NODISPLAY cards do not specify information for presentation to the user. Rather, they immediately execute the task bound to the ONENTERFORWARD and ONENTERBACKWARD intrinsic event.

For example, the following NODISPLAY and FORMCARD elements are identical in their behaviour and semantics:

```
<NODISPLAY>
   <ONEVENT  TYPE="ONEVENTFORWARD"
             TASK="GO" URL="/foo"/>
   <ONEVENT  TYPE="ONEVENTBACKWARD"
             TASK="PREV"/>
</NODISPLAY>
```

```
<FORMCARD>
   <ONEVENT  TYPE="ONEVENTFORWARD"
             TASK="GO" URL="/foo"/>
   <ONEVENT  TYPE="ONEVENTBACKWARD"
             TASK="PREV"/>
</FORMCARD>
```

*Attributes Defined Elsewhere*

The following task attributes are defined in section 11.4.1:

%cardattrs

# 11.5 Control Elements

## 11.5.1 Select Lists

Select lists are an input element that specifies a list of options for the user to choose from. Single and multiple choice lists are supported.

### 11.5.1.1 The SELECT Element

```
<!ELEMENT SELECT (OPTGROUP|OPTION)+>
<!ATTLIST SELECT
  TITLE        %vdata;          #IMPLIED
  KEY          NMTOKEN          #IMPLIED
  DEFAULT      %vdata;          #IMPLIED
  IKEY         NMTOKEN          #IMPLIED
  IDEFAULT     %vdata;          #IMPLIED
  MULTIPLE     %boolean;        'FALSE'
  TABINDEX     %number;         #IMPLIED
  >
```

The SELECT element lets users pick from a list of options. Each option is specified by an OPTION element. Each OPTION element may have one line of formatted text (which may be wrapped or truncated by the user agent if too long). OPTION elements may be organised into hierarchical groups using the OPTGROUP element.

*Attributes*

MULTIPLE=*boolean*

This attribute indicates that the select list should accept multiple selections. When not set, the select list should only accept a single selected option.

KEY=*nmtoken*
DEFAULT=v*data*

This KEY attribute indicates the name of the variable to set with the result of the selection. The variable is set to the string value of the chosen OPTION element, which is specified with the VALUE attribute. The KEY variable's value is used to pre-select options in the select list.

The DEFAULT attribute indicates the default value of the variable named in the KEY attribute. When the element is displayed, and the variable named in the KEY attribute is not set, the KEY variable is assigned the value specified in the DEFAULT attribute. If the KEY variable already contains a value, the DEFAULT attribute is ignored. Any application of the default value is done before the list is pre-selected with the value of the KEY variable.

If this element allows the selection of multiple options, the result of the user's choice is a list of all selected values, separated by the semicolon character. The KEY variable is set with this result. In addition, the DEFAULT attribute is interpreted as a semicolon separated list of pre-selected options.

IKEY=*nmtoken*
IDEFAULT=*vdata*

The IKEY attribute indicates the name of the variable to be set with the index result of the selection. The index result is the position of the currently selected OPTION in the select list. An index of zero indicates that no OPTION is selected. Index numbering begins at one, and increases monotonically.

The IDEFAULT attribute indicates the default-selected OPTION element. When the element is displayed, if the variable named in the IKEY attribute is not set, it is assigned the default-selected entry. If the variable already contains a value, the IDEFAULT attribute is ignored. If the IKEY attribute is not specified, the IDEFAULT value is applied every time the element is displayed.

If this element allows the selection of multiple options, the index result of the user's choice is a list of the indices of all the selected options, separated by the semicolon character (e.g. "1;2"). The IKEY variable is set with this result. In addition, the IDEFAULT attribute is interpreted as a semicolon separated list of pre-selected options (e.g. "1;4").

TITLE=*vdata*

This attribute specifies a title for this element, which may be used in the presentation of this object.

### *Attributes Defined Elsewhere*

The following attribute is defined in section 11.4.2:

TABINDEX

On entry into a card containing a SELECT element, the user agent must select the initial options in the following way:

- If the IKEY attribute exists, the indices in the variable named by IKEY are used to select the option. If the specified variable is not set, the index is assumed to be 1. If any index is larger than the number of options in the select list, the last entry is selected.
- If the IKEY attribute does not exist, and the KEY attribute exists, the value of the variable specified by KEY is used to select options. If the variable specified by KEY is not set, or no OPTION has a VALUE attribute matching the value, the first option is selected.

Once an OPTION is selected, the variable named by KEY is updated to the value of the option.

Both KEY and IKEY, or DEFAULT and IDEFAULT may be specified. IDEFAULT takes precedence over DEFAULT, and IKEY takes precedence over KEY.

## 11.5.1.2 The OPTION Element

```
<!ELEMENT OPTION (%text; | ONEVENT)*>
<!ATTLIST OPTION
  VALUE      %vdata;    #IMPLIED
  TITLE      %vdata;    #IMPLIED
  ONCLICK    %URL;      #IMPLIED
  >
```

This element specifies a single choice option in a SELECT element. Text within an OPTION element is rendered in non-breaking mode (see section 11.6.3 for more information on line break modes).

### *Attributes*

VALUE=*vdata*

The VALUE attribute specifies the value to be used when setting the KEY variable. When the user selects this option, the resulting value specified in the VALUE attribute is used to set the SELECT element's KEY variable.

The VALUE attribute may contain variable references, which are evaluated before the KEY variable is set.

TITLE=*vdata*

This attribute specifies a title for this element, which may be used in the presentation of this object.

ONCLICK=*URL*

The ONCLICK event occurs when the user selects or deselects this option. A multiple-selection option list generates an ONCLICK event whenever the user selects or deselects this option. A single-selection option list generates an ONCLICK event when the user selects this option, i.e. no event is generated for the de-selection of any previously selected option.

## 11.5.1.3 The OPTGROUP Element

```
<!ELEMENT OPTGROUP (OPTGROUP|OPTION)+ >
<!ATTLIST OPTGROUP
```

```
  TITLE      %vdata;    #IMPLIED
  >
```

The OPTGROUP element allows the author to group related OPTION elements into a hierarchy. The user agent may use this hierarchy to facilitate layout and presentation on a wide variety of devices.

*Attributes*

TITLE=*vdata*

This attribute specifies a title for this element, which may be used in the presentation of this object.

#### 11.5.1.4 Select list examples

In this example, a simple single-choice select list is specified. If the user were to choose the "Dog" option, the variable "X" would be set to a value of "D".

```
<SELECT KEY="X">
  <OPTION VALUE="D">Dog</OPTION>
  <OPTION VALUE="C">Cat</OPTION>
</SELECT>
```

In this example, a single choice select list is specified. If the user were to choose the "Cat" option, the variable "I" would be set to a value of "2". In addition, the "Dog" option would be pre-selected if the "I" variable had not been previously set.

```
<SELECT IKEY="I" IDEFAULT="1">
  <OPTION VALUE="D">Dog</OPTION>
  <OPTION VALUE="C">Cat</OPTION>
</SELECT>
```

In this example, a multiple choice list is specified. If the user were to choose the "Cat" and "Horse" options, the variable "X" would be set to "C;H", and the variable "I" would be set to "1;3". In addition, the "Dog" and "Cat" options would be pre-selected if the variable "I" had not been previously set.

```
<SELECT KEY="X" IKEY="I" IDEFAULT="1;2" MULTIPLE="TRUE">
  <OPTION VALUE="D">Dog</OPTION>
  <OPTION VALUE="C">Cat</OPTION>
  <OPTION VALUE="H">Horse</OPTION>
</SELECT>
```

### 11.5.2 The INPUT Element

```
<!ELEMENT INPUT EMPTY>
<!ATTLIST INPUT
  KEY        NMTOKEN           #REQUIRED
  TYPE       (TEXT|PASSWORD)   'TEXT'
  VALUE      %vdata;           #IMPLIED
  DEFAULT    %vdata;           #IMPLIED
  FORMAT     CDATA             #IMPLIED
  SIZE       %number;          #IMPLIED
  MAXLENGTH  %number;          #IMPLIED
  TABINDEX   %number;          #IMPLIED
  TITLE      %vdata;           #IMPLIED
  >
```

The INPUT element specifies a text entry object. The user input is constrained by the optional FORMAT attribute.

*Attributes*

KEY=*nmtoken*
DEFAULT=v*data*
VALUE=*vdata*

The KEY attribute specifies the name of the variable to set with the result of the user's text input. The KEY variable's value is used to pre-load the text entry object.

The DEFAULT attribute indicates the default value of the variable named in the KEY attribute. When the element is displayed and the variable named in the KEY attribute is not set, the KEY variable is assigned the value specified in the DEFAULT attribute. If the KEY variable already contains a value, the DEFAULT attribute is ignored. If the DEFAULT attribute specifies a value that does not conform to the input mask specified by the FORMAT attribute, the user agent must ignore the DEFAULT attribute.

The DEFAULT and VALUE attributes are identical in their behaviour and syntax.

TYPE=*(TEXT|PASSWORD)*

This attribute specifies the type of text-input area. The default type is TEXT. The following values are allowed:

- TEXT - a text entry box. Input should be displayed to the user in a readable form, and each character should be echoed in a manner appropriate to the user agent.
- PASSWORD - a text entry box. Input of each character should be echoed in an obscured or illegible form. For example, user agents may elect to display an asterisk in place of a character entered by the user. Typically, the PASSWORD input mode is indicated for password entry or other private data. Note that PASSWORD input is not secure, and should not be depended on for critical applications.

In both cases, the user's input is applied to the KEY variable.

FORMAT=*cdata*

The FORMAT attribute specifies an input mask for user input entries. The string consists of mask control characters and static text that is displayed in the input area. The user agent may use the format mask to facilitate accelerated data input.

The format control characters specify the data format expected to be entered by the user. The default format is "*M". The format codes are:

**A** entry of any upper-case alphabetic or punctuation character (i.e. upper-case non-numeric character)

**a** entry of any lower-case alphabetic or punctuation character (i.e. lower-case non-numeric character)

**N** entry of any numeric character

**X** entry of any upper case character

**x** entry of any lower-case character

**M** entry of any character; the user agent may chose to assume that the character is upper-case for the purposes of simple data entry, but must allow entry of any character

**m** entry of any character; the user agent may chose to assume that the character is lower-case for the purposes of simple data entry, but must allow entry of any character

***f** entry of any number of characters; f is one of the above format codes and specifies what kind of characters can be entered. *Note: This format may only be specified once, and it must appear at the end of the format string*

***nf*** entry of n characters where n is from 1 to 9; f is one of the above format codes and specifies what kind of characters can be entered. *Note: This format may only be specified once, and it must appear at the end of the format string*

**\c** display the next character, c, in the entry field; allows quoting of the format codes so they can be displayed in the entry area

User agents must implement the format codes to the best of their ability given the constraints of the input language and character set. If the input language and character set have clear definitions numbers and character case, they must be followed. Authors must not rely on the interpretation of a particular format code in a given language.

SIZE=*number*

This attribute specifies the width, in characters, of the text-input area. The user agent may ignore this attribute.

`MAXLENGTH=`*`number`*

This attribute specifies the maximum number of characters that can be entered by the user in the text-entry area. The default value for this attribute is an unlimited number of characters.

`TITLE=`*`vdata`*

This attribute specifies a title for this element, which may be used in the presentation of this object.

*Attributes Defined Elsewhere*

The following attribute is defined in section 11.4.2:

`TABINDEX`

#### 11.5.2.1 INPUT Element Examples

In this example, an `INPUT` element is specified. This element accepts any characters, and displays the input to the user in a human-readable form. The maximum number of character entered is 32, and the resulting input is assigned to the variable named `X`.

```
<INPUT KEY="X" TYPE="TEXT" MAXLENGTH="32"/>
```

The following example requests input from the user, and assigns the resulting input to the variable `NAME`. The text field has a default value of "Robert".

```
<INPUT KEY="NAME" TYPE="TEXT" DEFAULT="Robert"/>
```

The following example is a card that prompts the user for a first name, last name and age.

```
<FORMCARD>
  First name: <INPUT TYPE="TEXT" KEY="first"/><BR/>
  Last name: <INPUT TYPE="TEXT" KEY="last"/><BR/>
  Age: <INPUT TYPE="TEXT" KEY="age" FORMAT="*N"/>
</FORMCARD>
```

### 11.5.3 The FIELDSET Element

```
<!ELEMENT FIELDSET (%fields;)* >
<!ATTLIST FIELDSET
  TITLE           %vdata;      #IMPLIED
  >
```

The `FIELDSET` element allows the grouping of related fields and text. This grouping provides information to the user agent, allowing the optimising of layout and navigation. `FIELDSET` elements may nest, providing the user with a means of specifying behaviour across a wide variety of devices. See section 11.4.3 for information on how the `FIELDSET` element may influence layout and navigation.

*Attributes*

`TITLE=`*`vdata`*

This attribute specifies a title for this element, which may be used in the presentation of this object.

### 11.5.3.1 FIELDSET Element Examples

The following example specifies a WML deck that requests basic identity and personal information from the user. It is separated into multiple field sets, indicating the preferred field grouping to the user agent.

```
<WML>
   <FORMCARD>
   <DO TYPE="ACCEPT" TASK="GO"
          URL="/submit?f=$(fname)&amp;l=$(lname)&amp;s=$(sex)&amp;a=$(age)"/>
     <FIELDSET TITLE="Name">
       First name: <INPUT TYPE="TEXT" KEY="fname" MAXLENGTH="32"/><BR/>
       Last name: <INPUT TYPE="TEXT" KEY="lname" MAXLENGTH="32"/><BR/>
     </FIELDSET>
     <FIELDSET TITLE="Info">
       <SELECT KEY="sex">
         <OPTION VALUE="F">Female</OPTION>
         <OPTION VALUE="M">Male</OPTION>
       </SELECT>
       <BR/>
       Age: <INPUT TYPE="TEXT" KEY="age" FORMAT="*N"/>
     </FIELDSET>
   </FORMCARD>
</WML>
```

# 11.6 Text

This section defines the elements and constructs related to text.

## 11.6.1 White Space

WML white space and line break handling is based on [XML], and assumes the default white space handling rules. The WML user agent ignores all *insignificant* white space, as defined by the XML specification. In addition, all other sequences of white space must be compressed into a single inter-word space.

User agents should treat inter-word spaces in a locale-dependent manner, as different written languages treat inter-word spacing in different ways.

## 11.6.2 Emphasis

```
<!ELEMENT EM     (%flow;)*>
<!ELEMENT STRONG (%flow;)*>
<!ELEMENT B      (%flow;)*>
<!ELEMENT I      (%flow;)*>
<!ELEMENT U      (%flow;)*>
<!ELEMENT BIG    (%flow;)*>
<!ELEMENT SMALL  (%flow;)*>
```

The emphasis elements specify text emphasis markup information.

**EM:**
Render with emphasis.

**STRONG:**
Render with strong emphasis.

**I:**
Render with an italic font.

**B:**
Render with a bold font.

**U:**
Render with underline.

**BIG:**
Render with a large font.

**SMALL:**
Render with a small font.

Authors should use the `STRONG` and `EM` elements where possible. `B`, `I`, and `U` elements should not be used except where explicit control over text presentation is required.

## 11.6.3 Line Breaks

```
<!ENTITY % TAlign   "(LEFT|RIGHT|CENTER)" >
<!ENTITY % BRMode    "(WRAP|NOWRAP)" >

<!ELEMENT BR EMPTY>
<!ATTLIST BR
  ALIGN   %TAlign;   'LEFT'
  MODE    %BRMode;   #IMPLIED
  >
```

WML has two line-wrapping modes: breaking and non-breaking. In breaking mode, line breaks should be inserted into a text flow as appropriate for presentation on an individual device, and any inter-word space is a legal line break point. In non-breaking mode, a line of text must not be automatically wrapped.

The non-breaking space entity (` ` or ` `) indicates a space that must not be treated as an inter-word space by the user agent. Authors should use ` ` to prevent undesired line-breaks. The soft-hyphen character entity (`&shy;` or `&#173;`) indicates a location that may be used by the user agent for a line break. If a line break occurs at a soft-hyphen, the user agent must insert a hyphen character (`&#45;`) at the end of the line. In all other operations, the soft-hyphen entity should be ignored. A user agent may choose to entirely ignore soft-hyphens when formatting text lines.

The BR element establishes the beginning of a new line, and specifies the line break and alignment parameters for the new line. If the line break mode is not specified, it is identical to the line break mode of the previous line in the current card. If the text alignment is not specified, it defaults to `LEFT`.

The initial line break mode for a card is `MODE="WRAP"` (breaking mode), and the initial text alignment is `ALIGN="LEFT"` (left alignment). If the first non-whitespace markup in a card is a `BR` element, the `BR` begins the first line in the card. If the first non-whitespace markup in a card is not a `BR` element, a new line is implicitly started with the default line break and alignment modes.

The treatment of a line too long to fit on the screen is specified by the current line-break mode. If `MODE="WRAP"` is specified, the line is word-wrapped onto multiple lines. If `MODE="LINE"` is specified, the line is not wrapped. The user agent must provide a mechanism to view entire non-wrapped lines (e.g. horizontal scrolling or some other user-agent-specific mechanism).

### *Attributes*

`ALIGN=`*`(LEFT|RIGHT|CENTER)`*

This attribute specifies the text alignment mode for the line. Text can be centre aligned, left aligned or right aligned when it is displayed to the user. Left alignment is the default alignment mode. If not explicitly specified, the text alignment is set to the default alignment. For example, a simple `<BR/>` element starts a new line, and sets the alignment to `LEFT`.

`MODE=`*`(WRAP|NOWRAP)`*

This attribute specifies the line-breaking mode for the subsequent text line. `WRAP` specifies breaking text mode and `NOWRAP` specifies non-breaking text mode. If not explicitly specified, the line-break mode is identical to the line-break mode of the previous line in the text flow. For example, a simple `<BR/>` element starts a new line, but does not change the current line-break mode.

## 11.6.3.1 Line Break Examples

The following example demonstrates how the BR element affects text alignment and line break mode.

```
<WML VERSION="1.0">
  <FORMCARD>
    line 1, three-line card      <!-- left alignment, breaking mode -->
    <BR ALIGN="RIGHT"/>line 2    <!-- right alignment, breaking mode -->
    <BR MODE="NOWRAP"/>line 3    <!-- left alignment, non-breaking mode -->
  </FORMCARD>
  <FORMCARD>
    <BR ALIGN="CENTER"/>
    line 1, one-line card        <!-- centre alignment, breaking mode -->
  </FORMCARD>
  <FORMCARD>
    <BR MODE="NOWRAP"/>
    <BR ALIGN="CENTER"/>
    line 2, two-line card        <!-- centre alignment, non-breaking mode -->
  </FORMCARD>
</WML>
```

The following example demonstrates a more complex card and the interaction text alignment and line break modes.

```
<WML VERSION="1.0">
  <FORMCARD>
    <FIELDSET>
      line 1, three-line fieldset  <!-- left alignment, breaking mode -->
      <BR ALIGN="RIGHT"/>line 2    <!-- right alignment, breaking mode -->
      <BR MODE="NOWRAP"/>line 3    <!-- left alignment, non-breaking mode -->
    </FIELDSET>
    <FIELDSET>
      Choose:                      <!-- left alignment, non-breaking mode -->
      <SELECT KEY="X">
        <OPTION VALUE="1">One</OPTION>
        <OPTION VALUE="2">Two</OPTION>
      </SELECT>
    </FIELDSET>
    <FIELDSET>
      line 1, two-line fieldset    <!-- left alignment, non-breaking mode -->
      <INPUT KEY="Y"/>
      <BR MODE="WRAP"/>line 2      <!-- left alignment, breaking mode -->
    </FIELDSET>
  </FORMCARD>
</WML>
```

### 11.6.4 Tab Columns

The following elements specify tab columns.

```
<!ENTITY % tab     "TAB">
<!ENTITY % TAlign   "(LEFT|RIGHT|CENTER)" >
<!ELEMENT TAB EMPTY>
<!ATTLIST TAB
  ALIGN   %TAlign;   'LEFT'
  >
```

The TAB element is used to create aligned columns. Rather than tab to specific character positions, the TAB element separates the text for each column. To ensure the narrowest display width, the user agent should determine the width of each column from the maximum width of the text and images in that column. A non-zero width gutter must be used to separate each non-empty column. Some lines have fewer TAB elements than others, in which case the right hand columns of the line are assumed to be empty.

A column group is defined as the largest set of contiguous lines containing TAB elements that can be formed at any given point in the text flow. Depending on the display characteristics, the user agent may create aligned columns for each column group, or may use a single set of aligned columns for all column groups in a card.

#### *Attributes*

ALIGN=*(LEFT|RIGHT|CENTER)*

This attribute specifies the text layout within a column. Text can be center aligned, left aligned or right aligned when it is displayed to the user. Left alignment is the default.

## 11.7 Images

```
<!ENTITY % IAlign "(TOP|MIDDLE|BOTTOM)" >
<!ELEMENT IMG EMPTY>
<!ATTLIST IMG
  ALT        %vdata;      #IMPLIED
  SRC        %URL;        #IMPLIED
  LOCALSRC   %vdata;      #IMPLIED
  VSPACE     %length;     '0'
```

```
  HSPACE     %length;     '0'
  ALIGN      %IAlign;     'BOTTOM'
  HEIGHT     %length;     #IMPLIED
  WIDTH      %length;     #IMPLIED
  >
```

The IMG element indicates that an image is to be included in the text flow. Image layout is done within the context of normal text layout.

## *Attributes*

ALT=*vdata*

This attribute specifies an alternative textual representation for the image. This representation is used when the image can not be displayed using any other method (i.e. the user agent does not support images, or the image contents can not be found).

SRC=*URL*

This attribute specifies the URL for the image. If the browser supports images, it downloads the image from the specified URL, and renders it when the text is being displayed.

LOCALSRC=*vdata*

This attribute specifies an alternative internal representation for the image. This representation is used if it exists; otherwise the image is downloaded from the URL specified in the SRC attribute, i.e., any LOCALSRC parameter specified takes precedence over the image specified in the SRC parameter.

VSPACE=*length*
HSPACE=*length*

These attributes specify the amount of white space to be inserted to the left and right (HSPACE) and above and below (VSPACE) an image or object. The default value for this attribute is not specified, but is generally a small, non-zero length. If length is specified as a percentage value, the resulting size is based on the available horizontal or vertical space, not on the natural size of the image. These attributes are hints to the user agent, and may be ignored.

ALIGN=*(TOP|MIDDLE|BOTTOM)*

This attribute specifies image alignment within the text flow, and with respect to the current insertion point. ALIGN has three possible values:

- BOTTOM: means that the bottom of the image should be vertically aligned with the current baseline. This is the default value.
- MIDDLE: means that the centre of the image should be vertically aligned with the centre of the current text line.
- TOP: means that the top of the image should be vertically aligned with the top of the current text line.

HEIGHT=*length*
WIDTH=*length*

These attributes give user agents an idea of the size of an image or object so that they may reserve space for it and continue rendering the card while waiting for the image data. User agents may scale objects and images to match these values if appropriate. If length is specified as a percentage value, the resulting size is based on the available horizontal or vertical space, not on the natural size of the image. These attributes are a hint to the user agent, and may be ignored.

# 12. User Agent Semantics

## 12.1 Deck Access Control

The introduction of variables into WML exposes potential security issues that do not exist in other markup languages such as HTML. In particular, certain variable state may be considered private by the user. While the user may be willing to send a credit card number to a secure service, an insecure or malicious service should not be able to retrieve that number from the user agent by other means.

A conforming WML user agent must implement deck-level access control, including the `ACCESS` element, and the `PUBLIC`, `SENDREFERER`, `DOMAIN` and `PATH` attributes.

## 12.2 Low-Memory Behaviour

WML is targeted at devices with limited hardware resources, including significant restrictions on memory size. It is important that the author have a clear expectation of device behaviour in error situations, including those caused by lack of memory.

### 12.2.1 Limited History

The user agent may limit the size of the history stack (i.e. the depth of the historical navigation information). In the case of history size exhaustion, the user agent should delete the least-recently-used history information.

It is recommended that all user agents implement a minimum history stack size of ten entries.

### 12.2.2 Limited Cache

Many user agents implement some form of caching. If a user agent implements deck or card caching, it must implement the following semantics.

In selecting decks to free from the cache, the user agent should refrain from freeing decks that are referenced by the history stack. If cache space remains exhausted after freeing unreferenced cache entries, the user agent should prune the history stack as described in section 12.2.1, and free any unreferenced entries in the cache until there is sufficient space to continue processing. The user agent must never delete the current deck.

### 12.2.3 Limited Browser Context Size

In some situations, it is possible that the author has defined an excessive number of variables in the browser context, leading to memory exhaustion.

In this situation, the user agent should attempt to acquire additional memory by reclaiming cache and history memory as described in sections 12.2.1 and 12.2.2. If this fails, and the user agent has exhausted all memory, the user should be notified of the error.

## 12.3 Error Handling

Conforming user agents must enforce error conditions defined in this specification. User agents must not attempt to infer author or origin server intent upon receipt of illegal WML.

## 12.4 Reference Processing Behaviour - Inter-card Navigation

The following process describes the reference model for inter-card traversal in WML. This process is triggered by both the GO and PREV task (see section 9.3). All user agents must implement this process, or one that is indistinguishable from it.

The process of executing a GO or PREV task constitutes the following steps:

1. If the originating task has specified a VARS attribute, the attribute value is converted into a simple string by substituting all referenced variables. See section 10.3 for more information on variable substitution.
2. The target URL is identified and fetched by the user agent. If the task is a GO, the URL attribute value is converted into a simple string by substituting all referenced variables. If the task is a PREV, the URL attribute is the top of the history stack.
3. The destination card is located using the fragment name specified in the URL.
   a) If the destination deck does not contain a card, the destination card is set to *none*, i.e., no destination.
   b) If no fragment name was specified as part of the URL, the first card in the deck is the destination card.
   c) If a fragment name was identified, and a card has a NAME attribute that is identical to the fragment name, then that card is the destination card.
   d) If the fragment name can not be associated with a specific card, the destination card is set to *none*, i.e., no destination.
4. If there is a destination card, and the task is not a PREV, and the destination card contains a NEWCONTEXT attribute, the current browser context is re-initialised as described in section 10.2.
5. The string resulting from the processing done in step #1 (the VARS attribute value) is interpreted. The string is processed in a left-to-right manner, with each variable set as it is encountered in the string.
6. If there is a destination card:
   a) Any intrinsic event handlers present in the destination card are executed. See section 9.7 for more information.
   b) The destination card is displayed and processing stops.

   If the destination card is *none*, the browser attempts to invoke a script (see section 12.5). If the script invocation fails, the user agent must display the first card in the destination deck, or notify the user of an error if the deck does not contain a card.

## 12.5 Script Invocation

WML contains provisions for integrating script interpreters into the user agent. The SCRIPT element (see section 11.3.4) can be used to embed scripts in a WML deck. The scripting engine determines all semantics of the embedded script.

WML also includes a script invocation mechanism, which can be used in any WML task. When the user agent attempts to resolve a URL fragment name and the URL names a WML deck, the fragment may indicate either a card or a script invocation. The definition, semantics and result of a script invocation are entirely determined by the scripting engine.

When resolving a fragment name, card names take precedence over script names, i.e., if a script and a card share the same name, the fragment always refers to the WML card. If there are multiple scripting engines in the user agent, and they each have scripts with the same name, it is indeterminate which script is invoked.

The following reference process is one example of how a user agent could implement URL fragment name resolution. The user agent must implement a fragment resolution process that performs in a manner indistinguishable from this one, but does not need to literally follow this procedure.

1. If the URL fragment name matches a card name, go to the card and stop processing. See section 12.4 for more information.
2. If the fragment name does not match a card name, then for each scripting engine present in the user agent, the following is performed:
   a) Ask the scripting engine to invoke a script, function or other resource identified by the entire fragment name. Note that the interpretation of the fragment name is entirely at the discretion of the scripting engine. WML makes no assumptions at the format or syntax present in the fragment name, other than the fact that it must be a legal URL fragment.
   b) If the scripting engine returns "Not Found", continue processing.
   c) If the scripting engine returns "OK", stop processing.

This simple model provides the WML author with a powerful and convenient script invocation model that may be used in a variety of ways. For example, the following WML deck contains a WMLScript script, which is executed upon card entry.

```
<WML>
  <COMMON>
    <SCRIPT TYPE="text/wmlscript">
      // the following function is a no-op
      function enter() { return; }
    </SCRIPT>
  </COMMON>
  <FORMCARD ONENTER="#enter()">
    Sample card.
  </FORMCARD>
</WML>
```

# 13. WML Reference Information

WML is an application of [XML] version 1.0.

## 13.1 Document Identifiers

Ed: these identifiers have not yet been registered with the IANA or ISO 9070 Registrar

### 13.1.1 SGML Public Identifier

```
-//WAPFORUM//DTD WML 1.0//EN
```

### 13.1.2 WML Media Type

Textual form:

```
text/x-wml
```

Tokenized form:

```
application/x-wmlc
```

Ed: these types are not yet registered with the IANA, and are consequently *experimental* media types.

## 13.2 Document Type Definition (DTD)

```
<!--
Wireless Markup Language (WML) Document Type Definition.
WML is an XML language.  Typical usage:
   <?xml version="1.0"?>
   <!DOCTYPE WML PUBLIC "-//WAPFORUM//DTD WML 1.0//EN">
   <WML>
   ...
   </WML>
-->

<!ENTITY % length  "CDATA">    <!-- nn for pixels or nn% for percentage
                                    length -->
<!ENTITY % vdata   "CDATA">    <!-- attribute value possibly containing
                                    variable references -->
<!ENTITY % URL     "%vdata;">  <!-- URL or URN designating a hypertext
                                    node. May contain variable references -->
<!ENTITY % boolean "(TRUE|FALSE)">
<!ENTITY % number  "NMTOKEN">  <!-- a number, with format [0-9][0-9]* -->

<!ENTITY % emph    "EM | STRONG | B | I | U | BIG | SMALL">
<!ENTITY % tab     "TAB">
<!ENTITY % layout  "BR">

<!ENTITY % text     "#PCDATA | %emph; | %tab;">
<!ENTITY % inline   "%text; | %layout;">

<!-- flow covers "card-level" elements, such as text and images -->
<!ENTITY % flow     "%inline; | IMG | A">

<!-- card types -->
<!ENTITY % cards "FORMCARD | DISPLAY | CHOICE | ENTRY | NODISPLAY">

<!-- tasks types -->
<!ENTITY % tasktypes "(GO|PREV|NOOP)">

<!-- task attributes -->
<!ENTITY % taskattrs "
  URL             %URL;           #IMPLIED
  VARS            %vdata;         #IMPLIED
  SENDREFERER     %boolean;       'FALSE'
  METHOD          (POST|GET)      'GET'
  ACCEPT-CHARSET  CDATA           #IMPLIED
  POSTDATA        %vdata;         #IMPLIED"
  >
<!ENTITY % task "
  TASK            %tasktypes;     'GO'
  %taskattrs;"
  >

<!-- Navigation and event elements -->
<!ENTITY % navelmts "DO | ONEVENT">
```

```
<!--================ Decks ================-->

<!ELEMENT WML ( (COMMON, (%cards;)* ) | (%cards;)+ )>
<!ATTLIST WML
  VERSION        NMTOKEN     #FIXED      '1.0'
  >

<!--================ Cards ================-->

<!-- card intrinsic events -->
<!ENTITY % cardev
 "ONENTERFORWARD  %URL;          #IMPLIED
  ONENTERBACKWARD %URL;          #IMPLIED"
  >

<!ENTITY % cardattrs
 "NAME            NMTOKEN       #IMPLIED
  TITLE           %vdata;       #IMPLIED
  NEWCONTEXT      %boolean;     'FALSE'
  %cardev;"
  >

<!-- FORMCARD field types -->
<!ENTITY % fields  "%flow; | INPUT | SELECT | FIELDSET">

<!ELEMENT FORMCARD (%fields; | %navelmts;)*>
<!ATTLIST FORMCARD
  %cardattrs;
  STYLE           (LIST|SET)    'LIST'
  >

<!-- DEPRECATED -->
<!ELEMENT DISPLAY (%inline; | %navelmts;)* >
<!ATTLIST DISPLAY
  %cardattrs;
  >

<!-- DEPRECATED -->
<!ELEMENT CHOICE (%inline; | %navelmts; | CE)*>
<!ATTLIST CHOICE
  %cardattrs;
  KEY           NMTOKEN             #IMPLIED
  DEFAULT       %vdata;             #IMPLIED
  IKEY          NMTOKEN             #IMPLIED
  IDEFAULT      %vdata;             #IMPLIED
  >

<!-- DEPRECATED -->
<!ELEMENT ENTRY (%inline; | %navelmts;)*>
<!ATTLIST ENTRY
  %cardattrs;
  KEY           NMTOKEN         #REQUIRED
  DEFAULT       %vdata;         #IMPLIED
  FORMAT        CDATA           #IMPLIED
  NOECHO        %boolean;       'FALSE'
  EMPTYOK       %boolean;       'FALSE'
  >
```

```
<!-- DEPRECATED -->
<!ELEMENT NODISPLAY (ONEVENT)* >
<!ATTLIST NODISPLAY
  %cardattrs;
  >

<!--================ Event Handling ================-->

<!ELEMENT DO EMPTY>
<!ATTLIST DO
  TYPE         CDATA       #REQUIRED
  LABEL        %vdata;     #IMPLIED
  NAME         NMTOKEN     #IMPLIED
  OPTIONAL     %boolean;   'FALSE'
  %task;
  >

<!ELEMENT ONEVENT EMPTY>
<!ATTLIST ONEVENT
  TYPE         CDATA       #REQUIRED
  %task;
  >

<!--================ Common declarations ================-->

<!ELEMENT COMMON (ACCESS | META | SCRIPT | %navelmts;)*>
<!ATTLIST COMMON
  %cardev;
  >

<!ELEMENT ACCESS EMPTY>
<!ATTLIST ACCESS
  DOMAIN       CDATA       #IMPLIED
  PATH         CDATA       #IMPLIED
  PUBLIC       %boolean;   'FALSE'
  >

<!ELEMENT META EMPTY>
<!ATTLIST META
  HTTP-EQUIV     CDATA      #IMPLIED
  NAME           CDATA      #IMPLIED
  USER-AGENT     CDATA      #IMPLIED
  CONTENT        CDATA      #REQUIRED
  SCHEME         CDATA      #IMPLIED
  >

<!ELEMENT SCRIPT (#PCDATA)>
<!ATTLIST SCRIPT
  TYPE     CDATA   #REQUIRED
  >
```

```
<!--================ FORMCARD Fields ================-->

<!ELEMENT SELECT (OPTGROUP|OPTION)+>
<!ATTLIST SELECT
  TITLE        %vdata;           #IMPLIED
  KEY          NMTOKEN           #IMPLIED
  DEFAULT      %vdata;           #IMPLIED
  IKEY         NMTOKEN           #IMPLIED
  IDEFAULT     %vdata;           #IMPLIED
  MULTIPLE     %boolean;         'FALSE'
  TABINDEX     %number;          #IMPLIED
  >

<!ELEMENT OPTGROUP (OPTGROUP|OPTION)+ >
<!ATTLIST OPTGROUP
  TITLE      %vdata;    #IMPLIED
  >

<!ELEMENT OPTION (%text; | ONEVENT)*>
<!ATTLIST OPTION
  VALUE      %vdata;    #IMPLIED
  TITLE      %vdata;    #IMPLIED
  ONCLICK    %URL;      #IMPLIED
  >

<!ELEMENT INPUT EMPTY>
<!ATTLIST INPUT
  KEY        NMTOKEN           #REQUIRED
  TYPE       (TEXT|PASSWORD)   'TEXT'
  VALUE      %vdata;           #IMPLIED
  DEFAULT    %vdata;           #IMPLIED
  FORMAT     CDATA             #IMPLIED
  SIZE       %number;          #IMPLIED
  MAXLENGTH  %number;          #IMPLIED
  TABINDEX   %number;          #IMPLIED
  TITLE      %vdata;           #IMPLIED
  >

<!ELEMENT FIELDSET (%fields;)* >
<!ATTLIST FIELDSET
  TITLE          %vdata;      #IMPLIED
  >

<!--================ Choice elements ================-->

<!-- DEPRECATED -->
<!ELEMENT CE (%text;)*>
<!ATTLIST CE
  VALUE        %vdata;     #IMPLIED
  TASK         %tasktypes; #IMPLIED
  %taskattrs;
  >
```

```
<!--================ Images ================-->

<!ENTITY % IAlign "(TOP|MIDDLE|BOTTOM)" >

<!ELEMENT IMG EMPTY>
<!ATTLIST IMG
  ALT        %vdata;      #IMPLIED
  SRC        %URL;        #IMPLIED
  LOCALSRC   %vdata;      #IMPLIED
  VSPACE     %length;     '0'
  HSPACE     %length;     '0'
  ALIGN      %IAlign;     'BOTTOM'
  HEIGHT     %length;     #IMPLIED
  WIDTH      %length;     #IMPLIED
  >

<!--================ Anchor ================-->

<!ELEMENT A (%inline;)*>
<!ATTLIST A
  TITLE        %vdata;      #IMPLIED
  %task;
  >

<!--================ Text layout and line breaks ================-->

<!-- Text alignment attributes -->
<!ENTITY % TAlign   "(LEFT|RIGHT|CENTER)" >

<!ELEMENT TAB EMPTY>
<!ATTLIST TAB
  ALIGN   %TAlign;   'LEFT'
  >

<!ELEMENT EM     (%flow;)*>
<!ELEMENT STRONG (%flow;)*>
<!ELEMENT B      (%flow;)*>
<!ELEMENT I      (%flow;)*>
<!ELEMENT U      (%flow;)*>
<!ELEMENT BIG    (%flow;)*>
<!ELEMENT SMALL  (%flow;)*>

<!ENTITY % BRMode    "(WRAP|NOWRAP)" >
<!ELEMENT BR EMPTY>
<!ATTLIST BR
  ALIGN   %TAlign;   'LEFT'
  MODE    %BRMode;   #IMPLIED
  >

<!ENTITY quot  "&#34;">     <!-- quotation mark -->
<!ENTITY amp   "&#38;#38;"> <!-- ampersand -->
<!ENTITY lt    "&#38;#60;"> <!-- less than -->
<!ENTITY gt    "&#62;">     <!-- greater than -->
<!ENTITY nbsp  " ">    <!-- non-breaking space -->
<!ENTITY shy   "&#173;">    <!-- soft hyphen (discretionary hyphen) -->
```

# 14. A Compact Binary Representation of WML

WML may be encoded using a compact binary representation. The tokenized format was designed to allow for compact transmission over narrowband channels, with no loss of functionality or semantic information. The format is also designed to allow forward and backward compatibility by preserving the element structure of WML, and allowing a browser to skip unknown elements or attributes.

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

**Table 5. Data types used in tokenized format**

| ***Data Type*** | ***Definition*** |
|---|---|
| bit | 1 bit of data |
| byte | 8 bits of opaque data |
| int8 | 8 bit signed integer |
| u_int8 | 8 bit unsigned integer |
| int16 | 16 bit signed integer |
| u_int16 | 16 bit unsigned integer |
| int24 | 24 bit signed integer |
| u_int24 | 24 bit unsigned integer |
| int32 | 32 bit signed integer |
| u_int32 | 32 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.

## 14.1 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, e.g., most significant bit first. The octets are arranged in a big-endian order, e.g. 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`.

## 14.2 Character Encoding

The encoding of all strings in the WML tokenized format is specified by transport or container meta-information, and is expected to use the same mechanisms as the textual WML format. Specifically, it is assumed that a *charset* declaration accompanies the WML content in any form, and indicates the encoding of all strings (see section 6). The WML tokenized representation can support any string encoding, but requires that all strings include an encoding-specific termination mechanism (i.e. a NULL terminator, length encoding, etc.) which can be reliably used to detect the end of a string. As with the textual format of WML, it is also assumed that all tag and attribute names can be represented in the target character encoding.

## 14.3 BNF for Document Structure

A binary WML deck 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 WML element structure or semantics. This generality allows user agents and other consumers of the tokenized form 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 "]", and elements may be preceded with <N>* to specify N or more repetitions of the following element (N defaults to zero when unspecified).

```
deck      = version strtbl 1*content
strtbl    = mb_u_int32 *byte
content   = element | string | opaque | variable | entity

element   = stag [ 1*attribute ETAG ] [ *content ETAG ]
stag      = TAG | ( UNKNOWN index )
attribute = attrStart *attrValue
attrStart = ATTRSTART | ( UNKNOWN index )
attrValue = ATTRVALUE | string | variable | entity

variable  = ( VAR_I termstr ) | ( VAR_T index )
opaque    = ( OPQ_I length *byte ) | ( OPQ_R mb_u_int32 )

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

entity    = ENTITY mb_u_int32

version   = single u_int8 version number
termstr   = charset-dependent string with termination
index     = mb_u_int32       // integer index into string table.
length    = mb_u_int32       // integer length.
```

## 14.4 Language Version Number

```
version   = single u_int8 version number
```

A binary WML deck consists of a WML language version number followed by one or more elements. The version byte contains the major version minus one in the upper 4 bits and the minor version in the lower 4 bits. For example, the version number 2.7 would be encoded as 0x16. This document describes the 1.0 version of the WML language, and is thus encoded as 0x00. Note that the version number refers to the WML language version (see section 11.2), and not the version of the compiler, browser or any other software.

## 14.5 String Table

```
strtbl    = mb_u_int32 *byte
```

A tokenized WML deck may include a string table immediately after the version number. Minimally, the string table consists of a mb_u_int32 encoding the string table length in bytes, not including the length field (e.g. 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 string in the string table. For example, the offset of the first string is zero (0).

# 14.6 Token Structure

Tokens are organised into two separate *code spaces*, each of which is completely independent and overlapping:

- Tag - single-byte token indicating a specific tag name.
- Attribute - single-byte token indicating a attribute name or value.

Each code space is further organised into a series of *code pages*. Code pages allow for future expansion of the well-known codes. A well-defined token (`SWITCH_PAGE`) causes a switch between 256 possible code pages. This effectively allows for two independent 16-bit token sets.

There is a small set of codes that are identical in all code spaces and across all code pages. These codes are named *global codes*, and are used for the following purposes:

- Encoding inline data (e.g., strings, entities, opaque data and variable references).
- Code page switch and other miscellaneous functions.

## 14.6.1 Parser State Machine

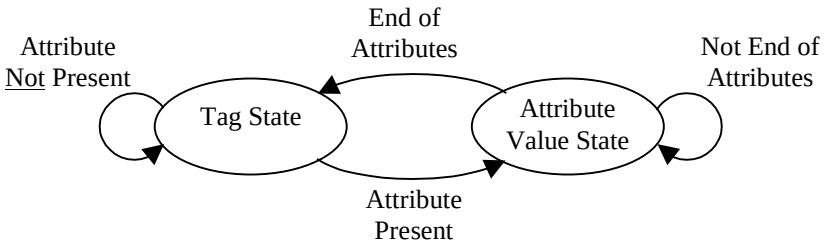
When decoding the tokenized form, a parser must move between two states, each of which has an associated code space. The states are traversed according to the syntax described in section 14.3. Code spaces are associated with parser states in the following manner:

**Table 6. Parser states**

| ***Parser State*** | ***Code Space*** |
|---|---|
| stag | Tags |
| attribute | Attribute name and value |

Any occurrence of code page switch tokens (`SWITCH_PAGE`) while in a given state changes the current code page for that state. Each parser state maintains a separate "current code page".

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



## 14.6.2 Tag Code Space

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

**Table 7. 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 two (`6`), with both attributes and content following the tag, e.g.,

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

- Tag value `0x48`: indicates tag eight (`8`). This element contains content, and this start tag is followed by that content and terminated by an `END` tag. This element contains no attributes, e.g.,

  ```
  <B>test</B>
  ```

- Tag value `0x10`: indicates tag sixteen (`16`). This element contains no content, and has no attributes, e.g.,

  ```
  <BR/>
  ```

The globally unique code `UNKNOWN` (see section 14.6.4.5) represents unknown tag names. It is illegal to use the `UNKNOWN` code to represent a well-known tag.

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

## 14.6.3 Attribute Code Space (ATTRSTART and ATTRVALUE)

Attribute tokens are a single `u_int8`. For example, the value `0x20` indicates attribute token number `32`. 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, e.g., `URL=`, and may optionally specify the beginning of the attribute value, e.g. `PUBLIC="TRUE"`. Unknown attribute names are encoded with the globally unique code `UNKNOWN` (see section 14.6.4.5). It is illegal to use the `UNKNOWN` code to represent a well-known attribute name or to represent any part of an attribute value with the `UNKOWN` code.

- 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 variable references (see section 14.6.4).

All tokenized attributes must begin with a single attribute start token, and may be followed by zero or more attribute value, string, entity or variable tokens. This allows a compact encoding of strings containing well-known sub-strings, as well as variables and entities.

For example, if the attribute start token `TOKEN_URL` represents the attribute name "`URL`" and the attribute value token `TOKEN_HTTP` represents the prefix "`http://`", the attribute `URL="http://foo/"` might be encoded with the following sequence:

```
TOKEN_URL TOKEN_HTTP STR_I "foo/"
```

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
```

## 14.6.4 Global Tokens

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

- Strings - inline and table string references
- Variables - variable references
- Opaque - inline opaque data
- Entity - character entities
- Unknown - unknown tag or attribute name
- Control codes - miscellaneous global control tokens

### 14.6.4.1 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 beginning of the table, and is a byte offset (i.e. not a character offset).

### 14.6.4.2 Variables

```
variable = ( VAR_I termstr ) | ( VAR_T index )
```

Variable references may occur in a variety of places in a WML deck (see section 10.3). There are several codes that indicate variable substitution. Each code has different escaping semantics (e.g. direct substitution, escaped substitution, and unescaped substitution). The variable name is encoded in the current document character encoding.

Inline variable substitution (`VAR_I`) is encoded into the token stream in the following way:

| | |
|---|---|
| VAR_I* | … char data … |

Variable string table references (`VAR_T`) have the following format:

| | |
|---|---|
| VAR_T* | mb_u_int32 |

The `mb_u_int32` string table offset is from the beginning of the table, and is a byte offset (i.e. not a character offset).

#### 14.6.4.3 Opaque Data

```
opaque    = ( OPQ_I length *byte ) | ( OPQ_R mb_u_int32 )
```

An opaque datum encodes non-WML data, and is used to represent a variety of content, e.g., a compiled script, an inline image, etc.

Inline opaque data are encoded as follows:

| OPQ_I | mb_u_int32 | … byte data … |
|---|---|---|

Opaque data references are encoded as follows:

| OPQ_R | mb_u_int32 |
|---|---|

The opaque data reference token (`OPQ_R`) encodes a reference to a previously encoded inline opaque datum (`OPQ_I`). The reference is coded as the offset between the beginning of the `OPQ_R`, and the start of the `OPQ_I` (i.e. the number of bytes between the two tokens).

#### 14.6.4.4 Character Entity

```
entity    = ENTITY mb_u_int32
```

The character entity token (`WML_ENTITY`) encodes a numeric character entity. This has identical semantics as a WML numeric character entity (e.g. `&#32;`). The `mb_u_int32` refers to a character in the UCS-4 character encoding (see section 6). All entities in the source WML deck must be represented using either a string token (e.g., `STR_I`) or the `ENTITY` token.

The format of the character entity is:

| ENTITY | mb_u_int32 |
|---|---|

#### 14.6.4.5 Unknown Tag or Attribute Name

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

The format of the unknown tag is:

| UNKNOWN | mb_u_int32 |
|---|---|

#### 14.6.4.6 Miscellaneous Control Codes

##### 14.6.4.6.1 END Token

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

##### 14.6.4.6.2 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 |
|---|---|

#### 14.6.4.7 Reserved Tokens

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

| Ed. - Should there be additional reserved code pages? Maybe all reserved except for FF, which could be for experimental or vendor-specific use? |
|---|

# 14.7 Encoding Semantics

The process of tokenizing WML must convert all markup and XML syntax (i.e., entities, tags, attributes, etc.) into their corresponding tokenized format. It is illegal to encode markup constructs as strings. The user agent must treat all text tokens (e.g., `STR_I` and `ENTITY`) as CDATA, i.e., text with no embedded markup.

This implies that all XML syntax (e.g. tags, entities, etc.) must be fully parsed and converted to a tokenized form. Tags and attributes must be converted to tokens (e.g., the `WML` token). Text and entities must be converted to string (e.g., `STR_I`) or entity (`ENTITY`) tokens. Entities in the textual markup (e.g., `&`) may be converted to string form when tokenized. Characters present in the textual form may be encoded using the `ENTITY` token when they can not be represented in the string character encoding. All variable references must be converted to variable reference tokens (e.g. `VAR_ESC_I`). Attribute names must be converted to an attribute start token, or must be represented by a single `UNKNOWN` token. Attribute values may not be represented by an `UNKNOWN` token.

A tokenized WML deck must conform to the WML document type definition (DTD), and must have identical semantics to the original textual representation of the deck. For example, this implies that the tokenized content must contain a single, top-level WML element, and all other elements must be included inside this element.

See section 14.9 for an example of the WML tokenized format.

The process of tokenizing WML must also apply a variety of transformations, as specified in the following sections.

## 14.7.1 Encoding the CE Element

Each instance of the `CE` element must be converted to an `OPTION` element. This can be automated with the following process:

1. `CE` is transformed to OPTION.
2. If the element includes a task specification, an `ONEVENT` intrinsic event handler is inserted into the `OPTION` element, specifying the `ONCLICK` event and the same task attributes indicated in the `CE` element.

## 14.7.2 Encoding the CHOICE Element

Each instance of the `CHOICE` element must be converted to a `FORMCARD` element and a `SELECT` element. This can be automated with the following process:

1. CHOICE is transformed to FORMCARD.
2. All PCDATA and elements other than CE are inserted into the FORMCARD element.
3. A select element is inserted at the end of the FORMCARD element, and each CE is inserted into it. Each CE is converted to an OPTION as described in section 14.7.1.

## 14.7.3 Encoding the DISPLAY Card

Each instance of the `DISPLAY` card must be converted to a `FORMCARD` element when tokenized. This is accomplished by literally transforming `DISPLAY` to `FORMCARD`.

### 14.7.4 Encoding the ENTRY Element

The ENTRY element's NOECHO attribute must be transformed to a TYPE attribute. Specifically:

- If NOECHO is TRUE, it must be transformed to TYPE="PASSWORD".
- If NOECHO is FALSE, it must be transformed to TYPE="TEXT".

### 14.7.5 Encoding the NODISPLAY Card

Each instance of the NODISPLAY card must be converted to a FORMCARD element when tokenized. This is accomplished by literally transforming the NODISPLAY to FORMCARD.

### 14.7.6 Encoding the SCRIPT Element

The SCRIPT element contains a mandatory TYPE attribute. This TYPE attribute must represent the actual IANA type of the content embedded within the SCRIPT element. For example, if the WML tokenization process compiles a given scripting language into a bytecode form, the TYPE attribute must be updated to reflect the change in type.

### 14.7.7 Encoding the VERSION Attribute

The WML element contains a VERSION attribute that is encoded differently than other attributes. The VERSION attribute must be encoded at the beginning of the tokenized deck (see section 14.4). The WML VERSION attribute must not be included in the tokenized WML element.

For example, the tokenized form of the following WML tag does not contain attributes:

```
<WML VERSION="1.0">
```

# 14.8 Numeric Constants

## 14.8.1 Global Tokens

The following token codes are present in all code pages. All numbers are in hexadecimal.

**Table 8. 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`. |
| UNKNOWN | 4 | An unknown tag or attribute name. Followed by an `mb_u_int32` that encodes an offset into the string table. |
| VAR_ESC_I | 40 | Variable substitution - escaped. Name of the variable is inline, and follows the token as a `termstr`. |
| VAR_UNESC_I | 41 | Variable substitution - unescaped. Name of the variable is inline, and follows the token as a `termstr`. |
| VAR_DIRECT_I | 42 | Variable substitution - no transformation. Name of the variable is inline, and follows the token as a `termstr`. |
| OPQ_I | 43 | An inline opaque datum. Followed by a length field (`mb_u_int32`) and zero or more bytes of data. |
| UNKNOWN_C | 44 | Unknown tag, with content. |
| VAR_ESC_T | 80 | Variable substitution - escaped. Variable name encoded as a reference into the string table. |
| VAR_UNESC_T | 81 | Variable substitution - unescaped. Variable name encoded as a reference into the string table. |
| VAR_DIRECT_T | 82 | Variable substitution - no transformation. Variable name encoded as a reference into the string table. |
| STR_T | 83 | String table reference. Followed by a `mb_u_int32` encoding a byte offset from the beginning of the string table. |
| UNKNOWN_A | 84 | Unknown tag, with attributes. |
| RESERVED_C0 | C0 | Reserved for future use. |
| RESERVED_C1 | C1 | Reserved for future use. |
| RESERVED_C2 | C2 | Reserved for future use. |
| OPQ_R | C3 | An opaque datum reference. Followed by an `mb_u_int32` that encodes a reverse offset to the beginning of an OPQ_I. |
| UNKNOWN_AC | C4 | Unknown tag, with content and attributes. |

## 14.8.2 Tag Tokens

The following token codes represent tags in code page zero (0). All numbers are in hexadecimal.

Note: token assignments may change before final publication.

**Table 9. Tag tokens**

| ***Tag Name*** | ***Token*** | ***Tag Name*** | ***Token*** |
|---|---|---|---|
| A | 28 | IMG | 34 |
| ACCESS | 29 | INPUT | 35 |
| B | 2A | META | 36 |
| BIG | 2B | ONEVENT | 37 |
| BR | 2C | OPTGROUP | 38 |
| COMMON | 2D | OPTION | 39 |
| DO | 2E | SCRIPT | 3A |
| EM | 2F | SELECT | 3B |
| ENTRY | 30 | SMALL | 3C |
| FIELDSET | 31 | TAB | 3D |
| FORMCARD | 32 | U | 3E |
| I | 33 | WML | 3F |

## 14.8.3 Attribute Start Tokens

The following token codes represent the start of an attribute in code page zero (0). All numbers are in hexadecimal.

Note: token assignments may change before final publication.

**Table 10. Attribute start tokens**

| ***Attribute Name*** | ***Attribute Value Prefix*** | ***Token*** |
|---|---|---|
| ACCEPT-CHARSET | | 5 |
| ALIGN | | 6 |
| ALIGN | BOTTOM | 7 |
| ALIGN | MIDDLE | 8 |
| ALIGN | TOP | 9 |
| ALT | | A |
| CONTENT | | B |
| DEFAULT | | C |
| DOMAIN | | D |
| EMPTYOK | | E |
| FORMAT | | F |
| HEIGHT | | 10 |
| HSPACE | | 11 |
| HTTP-EQUIV | | 12 |
| IDEFAULT | | 13 |
| IKEY | | 14 |
| KEY | | 15 |
| LABEL | | 16 |
| LOCALSRC | | 17 |
| MAXLENGTH | | 18 |
| METHOD | | 19 |
| METHOD | GET | 1A |
| METHOD | POST | 1B |
| MODE | | 1C |
| MODE | NOWRAP | 1D |
| MODE | WRAP | 1E |
| MULTIPLE | | 1F |
| MULTIPLE | FALSE | 20 |
| MULTIPLE | TRUE | 21 |
| NAME | | 22 |
| NEWCONTEXT | | 23 |
| NEWCONTEXT | FALSE | 24 |
| NEWCONTEXT | TRUE | 25 |
| ONCLICK | | 26 |
| ONENTERBACKWARD | | 27 |
| ONENTERFORWARD | | 28 |
| OPTIONAL | | 29 |
| OPTIONAL | FALSE | 2A |
| OPTIONAL | TRUE | 2B |
| PATH | | 2C |
| POSTDATA | | 2D |
| PUBLIC | | 2E |
| PUBLIC | FALSE | 2F |
| PUBLIC | TRUE | 30 |
| SCHEME | | 31 |
| SENDREFERER | | 32 |
| SENDREFERER | FALSE | 33 |
| SENDREFERER | TRUE | 34 |
| SIZE | | 35 |
| SRC | | 36 |
| STYLE | | 37 |
| STYLE | LIST | 38 |
| STYLE | SET | 39 |
| TABINDEX | | 3A |
| TASK | | 3B |
| TASK | GO | 3C |
| TASK | NOOP | 3D |
| TASK | PREV | 3E |
| TITLE | | 3F |
| TYPE | | 45 |

| ***Attribute Name*** | ***Attribute Value Prefix*** | ***Token*** |
|---|---|---|
| TYPE | PASSWORD | 46 |
| TYPE | TEXT | 47 |
| URL | | 48 |
| URL | http:// | 49 |
| URL | https:// | 4A |

| ***Attribute Name*** | ***Attribute Value Prefix*** | ***Token*** |
|---|---|---|
| USER-AGENT | | 4B |
| VALUE | | 4C |
| VARS | | 4D |
| VSPACE | | 4E |
| WIDTH | | 4F |

## 14.8.4 Attribute Value Tokens

The following token codes represent attribute values in code page zero (0). All numbers are in hexadecimal.

Note: token assignments may change before final publication.

NOTE: need to be specific about case folding, e.g. 'vnd.'

**Table 11. Attribute value tokens**

| ***Attribute Value*** | ***Token*** |
|---|---|
| .com | 5 |
| .edu | 6 |
| .net | 7 |
| .org | 8 |
| ACCEPT | 9 |
| BOTTOM | A |
| CENTER | B |
| CLEAR | C |
| DELETE | D |
| FALSE | E |
| GET | F |
| GO | 10 |
| HELP | 11 |
| http:// | 12 |
| http://www. | 13 |
| https:// | 14 |
| https://www. | 15 |
| LEFT | 16 |
| LIST | 17 |
| MIDDLE | 18 |
| NOOP | 19 |

| ***Attribute Value*** | ***Token*** |
|---|---|
| NOWRAP | 1A |
| ONCLICK | 1B |
| ONENTERBACKWARD | 1C |
| ONENTERFORWARD | 1D |
| OPTIONS | 1E |
| PASSWORD | 1F |
| POST | 20 |
| PREV | 21 |
| RESET | 22 |
| RIGHT | 23 |
| SET | 24 |
| TEXT | 25 |
| text/wmlscript | 26 |
| TOP | 27 |
| TRUE | 28 |
| UNKNOWN | 29 |
| vnd. | 2A |
| WRAP | 2B |
| www. | 2C |
| x- | 2D |

# 14.9 WML Encoding Examples

## 14.9.1 A Simple Deck

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

```
<WML>
  <FORMCARD>
    X &amp; Y<BR/>
    X = 1
  </FORMCARD>
</WML>
```

Tokenized form (numbers in hex) 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 (e.g. ` `), forcing the use of the `ENTITY` token.

```
00  00  7F  72  03  ' ' 'X' ' ' 00  02  26  03  ' ' 'Y' 00  2C
03  ' ' 'X' 00  02  81  20  03  '=' 00  02  81  20  03  '1' ' '
00  01  01
```

In an expanded and annotated form:

**Table 12. Example tokenized deck**

| ***Token Stream*** | ***Description*** |
|---|---|
| `00` | Version number |
| `00` | String table length |
| `7F` | `WML`, with content |
| `72` | `FORMCARD`, with content |
| `03` | Inline string follows |
| `' ', 'X', ' ', 00` | String |
| `02` | `ENTITY` |
| `26` | Entity value (`0x26`) |
| `03` | Inline string follows |
| `' ', 'Y', 00` | String |
| `2C` | `BR` |
| `03` | Inline string follows |
| `' ', 'X', 00` | String |
| `02` | `ENTITY` |
| `81 20` | Entity value (`0x160`) |
| `03` | Inline string follows |
| `'=', 00` | String |
| `02` | `ENTITY` |
| `81 20` | Entity value (`0x160`) |

| ***Token Stream*** | ***Description*** |
|---|---|
| 03 | Inline string follows |
| '1', ' ', 00 | String |
| 01 | END (of FORMCARD element) |
| 01 | END (of WML element) |

## 14.9.2 An Expanded Deck

The following is another example of a tokenized WML deck. It demonstrates variable encoding, attribute encoding, and the use of the string table. Source deck:

```
<WML>
  <FORMCARD NAME="abc" STYLE="LIST">
    X: $(X)<BR/>
    Y: $(&#x59;)<BR MODE="NOWRAP"/>
    Enter name: <INPUT TYPE="TEXT" KEY="N"/>
  </FORMCARD>
</WML>
```

Tokenized form (numbers in hex) follows. This example only uses inline strings, and assumes that the character encoding uses a NULL terminated string format. It also assumes that the character encoding is UTF-8:

```
00  04  'X' 00  'Y' 00  7F  F2  E9  03  'a' 'b' 'c' 00  38
01  03  ' ' 'X' ':' ' ' 00  82  00  2C  03  ' ' 'Y' ':' ' '
00  82  02  AC  1E  01  03  ' ' 'E' 'n' 't' 'e' 'r' ' ' 'n'
'a' 'm' 'e' ':' ' ' 00  B5  47  E2  03  'N' 00  01  01  01
```

In an expanded and annotated form:

**Table 13. Example tokenized deck**

| ***Token Stream*** | ***Description*** |
|---|---|
| 00 | Version number |
| 04 | String table length |
| 'X', 00, 'Y', 00 | String table |
| 7F | WML, with content |
| F2 | FORMCARD, with content and attributes |
| 23 | NAME= |
| 03 | Inline string follows |
| 'a', 'b', 'c', 00 | string |
| 38 | STYLE="LIST" |
| 01 | END (of FORMCARD attribute list) |
| 03 | Inline string follows |
| ' ', 'X', ':', ' ', 00 | String |
| 82 | Direct variable reference (VAR_DIRECT_T) |
| 00 | Variable offset 0 |
| 2C | BR |
| 03 | Inline string follows |

| ***Token Stream*** | ***Description*** |
|---|---|
| ' ', 'Y', ':', ' ', 00 | String |
| 82 | Direct variable reference (VAR_DIRECT_T) |
| 02 | Variable offset 2 |
| AC | BR, with attributes |
| 1E | MODE="NOWRAP" |
| 01 | END (of BR attribute list) |
| 03 | Inline string follows |
| ' ', 'E', 'n', 't', 'e', 'r', ' ', 'n', 'a', 'm', 'e', ':', ' ', 00 | String |
| B5 | INPUT, with attributes |
| 47 | TYPE="TEXT" |
| E2 | KEY= |
| 03 | Inline string follows |
| 'N', 00 | String |
| 01 | END (of INPUT attribute list) |
| 01 | END (of FORMCARD element) |
| 01 | END (of WML element) |