WAP WML

Proposed Version 3-Feb-1999

Wireless Application Protocol
Wireless Markup Language Specification
Version 1.1

Disclaimer:

This document is subject to change without notice.

Contents

1. Scope

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

This specification defines 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 (eg, 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 [RFC2396] 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 parameterisation 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 parameterisation allows for more efficient use of network resources.

2. Document Status

This document is available online in the following formats:

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

2.1 Copyright Notice

2.2 Errata

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

2.3 Comments

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

3. References

3.1 Normative References

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

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

[RFC1766] "Tags for the Identification of Languages", H. Alvestrand, March 1995. URL: ftp://ds.internic.net/rfc/rfc1766.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

[RFC2396] "Uniform Resource Identifiers (URI): Generic Syntax", T. Berners-Lee, et al., August 1998. URL: ftp://ds.internic.net/rfc/rfc2396.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, 30-April-1998. URL: http://www.wapforum.org/

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

[WBXML] "WAP Binary XML Content Format", WAP Forum, 30-April-1998. URL: http://www.wapforum.org/

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

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

3.2 Informative References

[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.

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

Character Encoding – when used as a verb, character encoding refers to conversion between sequence of characters and a sequence of bytes. When used as a noun, character encoding refers to a method for converting a sequence of bytes to a sequence of characters. Typically, WML document character encoding is captured in transport headers attributes (eg, Content-Type's "charset" parameter), meta information placed within a document, or the XML declaration defined by [XML].

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

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

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.

Deck - a collection of WML cards. A WML deck is also an XML document.

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.

JavaScript - a *de facto* standard language that can be used to add dynamic behaviour to HTML documents. JavaScript is one of the originating technologies of ECMAScript.

Man-Machine Interface - a synonym for user interface.

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.

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

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 providing the user with user agent capabilities, including the ability to request and receive information. Also called a mobile terminal or mobile station.

Transcode - the act of converting from one character set to another, eg, 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, WMLScript, WTAI or other resources. 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) 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.

BNF	Backus-Naur Form
HDML	Handheld Markup Language [HDML2]
HTML	HyperText Markup Language [HTML4]
HTTP	HyperText Transfer Protocol [RFC2068]
IANA	Internet Assigned Number Authority
MMI	Man-Machine Interface
PDA	Personal Digital Assistant
RFC	Request For Comments
SGML	Standardised Generalised Markup Language [ISO8879]
UI	User Interface
URL	Uniform Resource Locator [RFC2396]
URN	Uniform Resource Name
W3C	World Wide Web Consortium
WAE	Wireless Application Environment [WAE]
WAP	Wireless Application Protocol [WAP]
WSP	Wireless Session Protocol [WSP]
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 in four ways:

- Display size - smaller 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.
- Input devices - 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.
- Computational resources - low power CPU and small memory size; often limited by power constraints.
- Narrowband network connectivity - low bandwidth and high latency. Devices with 300bps to 10kbps 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** - 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 [RFC2396].
- Standard protocols to transport information (eg, HTTP).
- Standard content types (eg, 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 [RFC2396]. The character set used to specify URLs is also defined in [RFC2396].

In WML, URLs are used in the following situations:

- When specifying navigation, eg, hyperlinking.
- When specifying external resources, eg, 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 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. 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 adopted the use of relative URLs, as specified in [RFC2396]. [RFC2396] 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 (eg, 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 (eg, UCS-4). Any character encoding ("charset") that contains a proper subset of the logical characters in Unicode may be used (eg, US-ASCII, ISO-8859-1, UTF-8, Shift_JIS, etc.). Documents not encoded using UTF-8 or UTF-16 must declare their encoding as specified in the XML specification and should include Content-Type meta-information.

6.1 Reference Processing Model

WML documents maybe encoded with any character encoding as defined by [HTML4].

Character encoding of a WML document may be converted to another encoding (or transcoded) to better meet the user agent's characteristics. However, transcoding can lead to loss of information and must be avoided when the user agent supports the document's original encoding. Unnecessary transcoding must be avoided when information loss will result. If required, transcoding should be done before the document is delivered to the user agent.

This specification does not mandate which character encoding a user agent must support.

User agents must determine the character encoding of a WML document according to the following precedence (listed highest to lowest):

- Based on the "charset" parameter of the "Content-Type" transport header (eg, WSP or HTTP).
- Based on meta-information placed within the document (eg, the charset parameter in an `http-equiv meta` element)
- Based on the encoding on the XML declaration.
- Based on some other heuristics or user settings. For example, if content type is text based (ie, `application/vnd.wap.wml`), the charset may be assumed to be US-ASCII; otherwise, the default content encoding can be assumed (ie, UTF-8).

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

- User agents must correctly map to Unicode all characters in any character encoding that they recognise, or they must behave as if they did.
- 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

A given character encoding may not be able to express all characters of the document character set. For such encoding, or when the device characteristics do not allow users to input some document characters directly, authors and users may use character entities (ie, [XML] character references). Character entities are a character encoding-independent mechanism for entering any character from the document character set.

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;`

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

```
<!ENTITY quot  "&#34;">     <!-- quotation mark -->
<!ENTITY amp   "&#38;#38;"> <!-- ampersand -->
<!ENTITY apos  "&#39;">     <!-- apostrophe -->
<!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 inherits 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. Entities are used to specify characters in the document character set either 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. This implies that the ampersand and less-than characters must be escaped when they are used in textual data, ie, 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 syntax is 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, ie, 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 ($$) 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

CDATA sections are used to escape blocks of text and are legal in any PCDATA, eg, inside an element. CDATA sections begin with the string "<![CDATA[" and end with the string "]]>". For example:

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

Any text inside a CDATA section is treated as literal text and will not be parsed for markup. CDATA sections are useful anywhere literal text is convenient.

Refer to the [XML] specification for more information on CDATA sections.

7.8 Processing Instructions

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

7.9 Errors

The [XML] specification defines the concept of a **well-formed** XML document. WML decks that violate the definition of a well-formed document are in error. See section 14.2.2 for related information.

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">    <!-- [0-9]+ for pixels or [0-9]+"%" 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 % flow     "%text; | %layout; | img | anchor | a | table">
```

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

8.5 HREF

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

The `HREF` type refers to either a relative or an absolute Uniform Resource Locator [RFC2396]. 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]+ -->
```

The `number` type represents an integer value greater than or equal to zero.

8.8 xml:lang

The `xml:lang` attribute specifies the natural or formal language of an element or its attributes. The value of the attributes is a language code according to [RFC1766]. See [XML] for details on the syntax and specification of the attribute values. The attribute identifies to the user agent the language used text that may be presented to the user (ie, an element's content and attribute values). The user agent should perform a best effort to present the data according to the specifics of the language. Nested elements can assume the parent's language or use another. Where an element has both text content and text based attribute values that may be presented to the user, authors must use the same language for both. Variable values that are placed in `vdata` should match the language of the containing element.

An element's language must be established according to the following precedence (from highest to lowest):

1. Based on the `xml:lang` attribute specified for the element.
2. Based on the `xml:lang` attribute specified by the closest parent element.
3. Based on any language information included in the transport and document meta data (see sections 6.1 and 11.3.2 for more detail).
4. Based on user agent default preferences.

8.9 The id and class Attributes

All WML elements have two core attributes: `id` and `class` that can be used for such tasks as server-side transformations. The `id` attribute provides an element a unique name within a single deck. The attribute `class` affiliates an element with one or more classes. Multiple elements can be given the same class name. All elements of a single deck with a common class name are considered to be part of the same class. Class names are cases sensitive. An element can be part of multiple classes if it has multiple unique class names listed in its `class` attribute. Multiple class names within a single attribute must be separated by white space. Redundant class names as well as insignificant white space between class names may be removed. The WML user agent should ignore these attributes.

9. Events and Navigation

9.1 Navigation and Event Handling

WML includes navigation and event-handling models. The associated elements allow the author to specify the processing of user agent events. Events may be bound to *tasks* by the author; when an event occurs, the bound task is executed. A variety of tasks may be specified, such as navigation to an author-specified URL. Event bindings are declared by several elements, including `do` and `onevent`.

9.2 History

WML includes a simple navigational history model that allows the author to manage backward navigation in a convenient and efficient manner. The user agent history is modelled as a stack of URLs that represent the navigational path the user traversed to arrive at the current card. Three operations 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 pushed onto the history stack as an effect of navigation to a new card.
- Pop - the current card's URL (top of the stack) is popped as a result of backward navigation.

The user agent must implement a navigation history. As each card is accessed via an explicitly specified URL, eg, a `href` attribute in `go`, 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. 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 `prev` pops the current card URL from the history stack. Refer to section 12.5.2 for more information on the semantics of `prev`.

9.3 The Postfield Element

```
<!ELEMENT postfield EMPTY>
<!ATTLIST postfield
  name         %vdata;           #REQUIRED
  value        %vdata;           #REQUIRED
  %coreattrs;
  >
```

The `postfield` element specifies a field name and value for transmission to an origin server during a URL request. The actual encoding of the name and value will depend on the method used to communicate with the origin server.

Refer to section 9.5.1 for more information on the use of `postfield` in a `go` element.

Attributes

`name=`*`vdata`*

The `name` attribute specifies the field name.

`value=`*`vdata`*

The `value` attribute specifies the field value.

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)

9.4 The Setvar Element

```
<!ELEMENT setvar EMPTY>
<!ATTLIST setvar
  name         %vdata;           #REQUIRED
  value        %vdata;           #REQUIRED
  %coreattrs;
  >
```

The `setvar` element specifies the variable to set in the current browser context as a side effect of executing a task. The element must be ignored if the `name` attribute doe not evaluate to a legal variable name at runtime (see section 10.3). See section 10.3.4 for more information on setting variables.

Attributes

`name=`*`vdata`*

The `name` attribute specifies the variable name.

`value=`*`vdata`*

The `value` attribute specifies the value to be assigned to the variable.

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)

9.5 Tasks

```
<!ENTITY % task   "go | prev | noop | refresh">
```

Tasks specify processing that is performed in response to an event. Tasks are bound to events in the `do`, `a`, `onevent` and `anchor` elements.

9.5.1 The Go Element

```
<!ELEMENT go (postfield | setvar)*>
<!ATTLIST go
  href            %HREF;           #REQUIRED
  sendreferer     %boolean;       "false"
  method          (post|get)      "get"
  accept-charset  CDATA           #IMPLIED
  %coreattrs;
  >
```

The `go` element declares a `go` task, indicating navigation to a URI. If the URI names a WML card or deck, it is displayed. A `go` executes a "push" operation on the history stack (see section 9.2).

Refer to section 12.5.1 for more information on the semantics of `go`.

Attributes

`href=`***`HREF`***

The `href` attribute specifies the destination URI, eg, the URI of the card to display.

`sendreferer=`*`boolean`*

If this attribute is true, the user agent must specify, for the server's benefit, the URI of the deck containing this task (ie, the referring deck). This allows a server to perform a form of access control on URIs, based on which decks are linking to them. The URI must be the smallest relative URI possible if it can be relative at all. For example, if `sendreferer=true`, an HTTP based user agent shall indicate the URI 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.

`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 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, ie, 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.

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)

The `go` element may contain one or more `postfield` elements. These elements specify information to be submitted to the origin server during the request. The submission of field data is performed in the following manner:

1. The field name/value pairs are identified and all variables are substituted.
2. The user agent should transcode the field names and values to the correct character set, as specified explicitly by the `accept-charset` or implicitly by the document encoding.
3. The field names and values are escaped using URL-escaping and assembled into an `application/x-www-form-urlencoded` content type. URL-escaping is defined in [RFC2396] and the assembly of the content is specified in [RFC2070].
4. The request is performed according to the `method` attribute's value:

- `get` – if the `method` attribute has a value of `get` and the `href` attribute value is an HTTP URI, the submission data is added to the query component of the URI (see [RFC2396], section 2.1 for more information on the URI syntax for HTTP). An HTTP GET operation is performed on the resulting URL.
- `post` – if the `method` attribute has a value of `POST` and the `href` attribute value is an HTTP URI, the submission data is sent to the origin server with an HTTP POST operation. The submission is identified with a content type of `application/x-www-form-urlencoded`, with a `charset` parameter indicating the character encoding.

For example, the following `go` element would cause an HTTP GET request to the URL "`/foo?x=1`":

```
<go href="/foo">
  <postfield name="x" value="1"/>
</go>
```

The following example will cause an HTTP POST to the URL "`/bar`" with a message entity containing "`w=12&y=test`":

```
<go href="/bar" method="post">
  <postfield name="w" value="12"/>
  <postfield name="y" value="test"/>
</go>
```

9.5.2 The Prev Element

```
<!ELEMENT prev (setvar)*>
<!ATTLIST prev
  %coreattrs;
  >
```

The `prev` element declares a `prev` task, indicating navigation to the previous URI on the history stack. A `prev` performs a "pop" operation on the history stack (see section 9.2).

Refer to section 12.5.2 for more information on the semantics of `prev`.

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)

9.5.3 The Refresh Element

```
<!ELEMENT refresh (setvar)*>
<!ATTLIST refresh
  %coreattrs;
  >
```

The `refresh` element declares a `refresh` task, indicating an update of the user agent context as specified by the `setvar` elements. User-visible side effects of the state changes (eg, a change in the screen display) occur during the processing of the `refresh` task. A refresh and its side effects must occur even if the elements have no `setvar` elements given that context may change by other means (eg, `timer`).

Refer to section 12.5.4 for more information on the semantics of `refresh`.

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)

9.5.4 The Noop Element

```
<!ELEMENT noop EMPTY>
<!ATTLIST noop
  %coreattrs;
  >
```

This `noop` element specifies that nothing should be done, ie, "no operation".

Refer to section 12.5.3 for more information on the semantics of `noop`.

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)

9.6 Card/Deck Task Shadowing

A variety of elements can be used to create an event binding for a card. These bindings may also be declared at the deck level:

- Card-level: the event-handling element may appear inside a `card` element and specify event-processing behaviour for that particular card.
- Deck-level: the event-handling element may appear inside a `template` 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`.

If a card-level element shadows a deck-level element and the card-level element specifies the noop task, the event binding for that event will be completely masked. In this situation, the card- and deck-level element will be ignored and no side effects will occur on delivery of the event. In this case, user agents should not expose the element to the user (eg, render a UI control). In effect, the noop removes the element from the card.

In the following example, a deck-level do element indicates that a prev task should execute on receipt of a particular user action. The first card inherits the do element specified in the template element and will display the do to the user. The second card shadows the deck-level do element with a noop. The user agent will not display the do element when displaying the second card. The third card shadows the deck-level do element, causing the user agent to display the alternative label and to perform the go task if the do is selected.

```
<wml>
  <template>
    <do type="options" name="do1" label="default">
      <prev/>
    </do>
  </template>

  <card id="first">
    <!-- deck-level do not shadowed.  The card exposes the
         deck-level do as part of the current card -->

    <!-- rest of card -->
    …
  </card>

  <card id="second">
    <!-- deck-level do is shadowed with noop.
         It is not exposed to the user -->
    <do type="options" name="do1">
      <noop/>
    </do>

    <!-- rest of card -->
    …
  </card>

  <card id="third">
    <!-- deck-level do is shadowed. It is replaced by a card-level do -->
    <do type="options" name="do1" label="options">
      <go href="/options"/>
    </do>

    <!-- rest of card -->
    …
  </card>
</wml>
```

9.7 The Do Element

```
<!ENTITY % task   "go | prev | noop | refresh">
<!ELEMENT do (%task;)>
<!ATTLIST do
  type         CDATA       #REQUIRED
  label        %vdata;     #IMPLIED
  name         NMTOKEN     #IMPLIED
  optional     %boolean;   "false"
  xml:lang     NMTOKEN     #IMPLIED
```

```
%coreattrs;
>
```

The do element provides a general mechanism for the user to act upon the current card, ie, 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 an individual type value, or on the mapping of 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 and may be located anywhere in the text flow. If the user agent intends to render the do element inline (ie, 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 template element, indicating a deck-level do element. A deck-level do element applies to all cards in the deck (ie, is equivalent to having specified the do within each card). For the purposes of inline rendering, the user agent must behave as if deck-level do elements are located 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 9.6 for more details). For a single card, the *active* do elements are defined as the do elements specified in the card, plus any do elements specified in the deck's template and not overridden in the card. All active do elements with a noop task must not be presented to the user. All do elements that are not active must not be presented to the user. All do elements with a task other than noop must be made accessible to the user in some manner. In other words, it must be possible for the user to activate these user interface items when viewing the card containing the active do elements. 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 the element should be mapped to a physical user interface construct. All types are reserved, except for those marked as experimental.

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, eg, Test* indicates any string starting with the word Test.

Table 1. 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 (eg, type="").

Type	***Description***
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 (eg, truncate the string). If an element can not be dynamically labelled, this attribute may be ignored.

To work well on a variety of user agents, labels should be 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 `card` element) and at the deck-level (in a `template` 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 `template` element. A `name` with an empty value is equivalent to an 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 defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

9.8 The Anchor Element

```
<!ELEMENT anchor ( #PCDATA | br | img | go | prev | refresh )*>
<!ATTLIST anchor
  title        %vdata;      #IMPLIED
  xml:lang     NMTOKEN      #IMPLIED
  %coreattrs;
  >
```

The `anchor` element specifies the head of a link. The tail of a link is specified as part of other elements (eg, a card name attribute). It is an error to nest anchored links.

Anchors may be present in any text flow, excluding the text in `option` elements (ie, anywhere formatted text is legal, except for `option` elements). Anchored links have an associated *task* that specifies the behaviour when the anchor is selected. It is an error to specify other than one task element (eg, `go`, `prev` or `refresh`) in an `anchor` element.

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

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

9.9 The A Element

```
<!ELEMENT a ( #PCDATA | br | img )*>
<!ATTLIST a
  href         %HREF;       #REQUIRED
  title        %vdata;      #IMPLIED
  xml:lang     NMTOKEN      #IMPLIED
  %coreattrs;
  >
```

The `a` element is a short form of the `anchor` element, and is bound to a go task without variables. For example, the following markup:

```
<anchor>follow me
  <go href="destination"/>
</anchor>
```

Is identical in behaviour and semantics to:

```
<a href="destination">follow me</a>
```

It is invalid to nest `a` elements. Authors are encouraged to use the `a` element instead of anchor where possible, to allow more efficient tokenisation.

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

9.10 Intrinsic Events

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

Table 2. WML Intrinsic Events

Event	***Element(s)***	***Description***
`ontimer`	`card`, `template`	The `ontimer` event occurs when a timer expires. Timers are specified using the `timer` element (see section 11.7).
`onenterforward`	`card`, `template`	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. The `onenterforward` intrinsic event may be specified at both the card and deck-level. Event bindings specified in the `template` element apply to all cards in the deck and may be overridden as specified in section 9.6.

Event	***Element(s)***	***Description***
onenterbackward	card, template	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. The onenterbackward intrinsic event may be specified at both the card and deck-level. Event bindings specified in the template element apply to all cards in the deck and may be overridden as specified in section 9.6.
onpick	option	The onpick 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 URI 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:

```
<card onenterforward="/url"> Hello </card>
```

This attribute value may only specify 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:

```
<card>
  <onevent type="onenterforward">
    <go href="/url"/>
  </onevent>
  <p>
  Hello
  </p>
</card>
```

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, eg, 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.

9.10.1 The Onevent Element

```
<!ENTITY % task   "go | prev | noop | refresh">
<!ELEMENT onevent (%task;)>
<!ATTLIST onevent
  type         CDATA       #REQUIRED
  %coreattrs;
  >
```

The onevent element binds a task to a particular intrinsic event for the immediately enclosing element, ie, specifying an onevent element inside an "XYZ" element associates an intrinsic event binding with 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

- `id` (see section 8.9)
- `class` (see section 8.9)

9.10.2 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 9.6. 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 `onenterforward` 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 backward navigation; and
- 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` element (see section 11.5). 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, and
- Reset implementation-specific state to a well-known value.

The `newcontext` is only performed as part of the `go` task. See section 12.5 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 string substitution operation. 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 and result in an error. 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 "`]`". 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   = lalpha | halpha
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 cannot be inferred from the surrounding context, eg, 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 3. 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 [RFC2396]. All lexically sensitive characters defined in WML must be escaped, including all `reserved` and `unsafe` URL characters, as specified by [RFC2396].

If no conversion is specified, the variable is substituted using the conversion format appropriate for the context. The `onenterbackward`, `onenterforward`, `href` and `src` 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 (eg, $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, eg, 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 `&#x24;`, 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, 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. When a variable is set and it is already defined in the browser context, the current value is updated.

The `setvar` element allows the author to set variable state as a side effect of navigation. `Setvar` may be specified in task elements, including `go`, `prev` and `refresh`. The `setvar` element specifies a variable name and value, for example:

```
<setvar name="location" value="$(X)"/>
```

The variable specified in the `name` attribute (eg, `location`) is set as a side effect of navigation. See the discussion of event handling (section 8.8 and section 12.5) for more information on the processing of the VAR element.

Input elements set the variable identified by the `name` attribute to any information entered by the user. For example, an `input` element assigns the entered text to the variable, and the `select` element assigns the value present in the `value` attribute of the chosen `option` element.

User input is written to variables when the user commits the input to the `input` or `select` element. Committing input is an MMI dependent concept, and the WML author must not rely on a particular user interface. For example, some implementations will update the variable with each character entered into an `input` element, and others will defer the variable update until the `input` element has lost focus. The user agent must update all variables prior to the execution of any task. The user agent may re-display the current card when variables are set, but the author must not assume that this action will occur.

10.3.5 Validation

Within the WML document, any string following a single dollar sign ('$') must be treated as a variable reference and validated. Each reference must use proper variable name syntax, according to section 10.3.1. Each reference must be placed either within a card's text (#PCDATA) or within %vdata or %HREF attribute values. The deck is in error if any variable reference uses invalid syntax or is placed in an invalid location.

Examples of invalid variable use:

```
<!-- bad variable syntax -->
Balance left is $10.00

<!-- bad placement (in the type attribute) -->
<do type="x-$(type)" label="$type">
```

10.4 Context Restrictions

User agents may provide users means to reference and navigate to resources independent of the current content. For example, user agents may provide bookmarks, a URL input dialog, and so forth. Whenever a user agent navigates to a resource that was not the result of an interaction with the content in the current context, the user agent must establish another context for that navigation. The user agent may terminate the current context before establishing another one for the new navigation attempt.

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 1.1//EN"
                     "http://www.wapforum.org/DTD/wml_1.1.xml">
```

It is an error to omit the prologue.

11.2 The WML Element

```
<!ELEMENT wml ( head?, template?, card+ )>
<!ATTLIST wml
  xml:lang        NMTOKEN        #IMPLIED
  %coreattrs;
  >
```

The `wml` element defines a deck and encloses all information and cards in the deck.

Attributes

`xml:lang=`*`nmtoken`*

The `xml:lang` attribute specifies the natural or formal language in which the document is written. See section 8.8 for more detail.

11.2.1 A WML Example

The following is a deck containing two cards, each represented by a `card` element (see section 11.5 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>
  <card>
    <p>
    <do type="accept">
      <go href="#card2"/>
    </do>
    Hello world!
    This is the first card...
    </p>
  </card>

  <card id="card2">
    <p>
    This is the second card.
    Goodbye.
```

```
    </p>
  </card>
</wml>
```

11.3 The Head Element

```
<!ELEMENT head ( access | meta )+>
<!ATTLIST head
  %coreattrs;
  >
```

The `head` element contains information relating to the deck as a whole, including meta-data and access control elements.

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)

11.3.1 The Access Element

```
<!ELEMENT access EMPTY>
<!ATTLIST access
  domain       CDATA       #IMPLIED
  path         CDATA       #IMPLIED
  %coreattrs;
  >
```

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. If a deck does not include an `access` element, access control is disabled. When access control is disabled, cards in any deck can access this deck.

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 URI 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 URI and the access path is prefix matched against the path portion of the referring URI.

`Domain` suffix matching is done using the entire element of each sub-domain and must match each element exactly (eg, `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 (eg, `/X/Y` matches path="`/X`" attribute, but does not match path="`/XZ`" attribute).

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 URIs. The user agent converts the relative path to an absolute path and then performs prefix matching against the `PATH` attribute.

For example, given the following access control attributes:

```
domain="wapforum.org"
path="/cbb"
```

The following referring URIs 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 URIs 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.

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)

11.3.2 The Meta Element

```
<!ELEMENT meta EMPTY>
<!ATTLIST meta
  http-equiv     CDATA      #IMPLIED
  name           CDATA      #IMPLIED
  forua          %boolean;  #IMPLIED
  content        CDATA      #REQUIRED
  scheme         CDATA      #IMPLIED
  %coreattrs;
  >
```

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.

It is an error for a `meta` element to contain more than one attribute specifying a property name, ie, more than one attribute from the following set: `name`, `http-equiv` and `user-agent`.

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 tokenised before it arrives at the user agent.

`forua=`*`boolean`*

This attribute specifies that the author intended the property to reach the user agent. In the case where the user agent supports the meta-data mechanism, and the property has its `forua` attribute set to `true`, the meta-data must be delivered to the user agent. The method of delivery may vary. For example, `http-equiv` meta-data may be delivered using HTTP or WSP headers. In the case where the user agent supports the meta-data mechanism, and the property has its `forua` attribute set to `false`, the element should be removed if the content is tokenised before it arrives at the user agent.

`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.

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)

11.4 The Template Element

```
<!ENTITY % navelmts "do | onevent">
<!ELEMENT template (%navelmts;)*>
<!ATTLIST template
  %cardev;
  %coreattrs;
  >
```

The `template` element declares a template for cards in the deck. Event bindings specified in the `template` element (eg, `do` or `onevent`) apply to all cards in the deck. Specifying an event binding in the `template` element is equivalent to specifying it in every card element. A card element may override the behaviour specified in the `template` element. In particular:

- `DO` elements specified in the `template` element may be overridden in individual cards if both elements have the same `NAME` attribute value. See section 9.6 for more information.
- Intrinsic event bindings specified in the `template` element may be overridden by the specification of an event binding in a card element. See section 9.9 for more information.

See section 11.5 for the definition of the card-level intrinsic events (the `cardev` entity).

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)
- `onenterforward` (see section 11.5.1)
- `onenterbackward` (see section 11.5.1)
- `ontimer` (see section 11.5.1)

11.5 The Card Element

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

11.5.1 Card Intrinsic Events

```
<!ENTITY % cardev
 "onenterforward  %HREF;          #IMPLIED
  onenterbackward %HREF;          #IMPLIED
  ontimer         %HREF;          #IMPLIED"
  >
```

The following attributes are available in the `card` and `template` elements.

Attributes

`onenterforward=`***HREF***

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

`onenterbackward=`***HREF***

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

`ontimer=`***HREF***

The `ontimer` event occurs when a `timer` expires.

11.5.2 The Card Element

```
<!ELEMENT card (onevent*, timer?, (do | p)*)>
<!ATTLIST card
  title           %vdata;       #IMPLIED
  newcontext      %boolean;     "false"
  ordered         %boolean;     "true"
  xml:lang        NMTOKEN       #IMPLIED
  %cardev;
  %coreattrs;
  >
```

The `card` 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 `card` 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 `card` 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 `card` 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. A card's id may be used as a fragment anchor. See section 5.2 for more information.

Attributes

`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 (eg, 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.

`ordered=`*`boolean`*

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

- ordered="true" - the card is naturally organised as a linear sequence of field elements, eg, 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 (eg, 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, `ordered` 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.

- ordered="false" - 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, unordered 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. A user agent may organise an unordered collection of elements in an ordered fashion.

The user agent may interpret the ordered attribute in a manner appropriate to its device capabilities (eg, 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 `card` with `ordered="false"` may use a softkey 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 screen 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 screen flip location:

- `fieldset` defines a logical boundary between fields.
- Fields (eg, `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. The `table` must be treated differently than `input` and `select`. The user agent must insert a line break before each `table` element, except when it is the first non-whitespace markup in a card. The user agent must insert a line break after each `table` element, except when it is the final element in a card.

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)
- `onenterforward` (see section 11.5.1)
- `onenterbackward` (see section 11.5.1)
- `ontimer` (see section 11.5.1)

11.5.2.1 A Card Example

The following is an example of a simple `card` 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>
  <template>
    <do type="accept" label="Exit">
      <prev/>
    </do>
  </template>
  <card>
  <p>
    Hello World!
  </p>
  </card>
</wml>
```

11.6 Control Elements

11.6.1 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 (ie, `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.

11.6.2 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.

11.6.2.1 The Select Element

```
<!ELEMENT select (optgroup|option)+>
<!ATTLIST select
  title        %vdata;          #IMPLIED
  name         NMTOKEN          #IMPLIED
  value        %vdata;          #IMPLIED
  iname        NMTOKEN          #IMPLIED
  ivalue       %vdata;          #IMPLIED
  multiple     %boolean;        "false"
  tabindex     %number;         #IMPLIED
  xml:lang     NMTOKEN          #IMPLIED
  %coreattrs;
  >
```

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.

`name=`*`nmtoken`*
`value=v`*`data`*

This `name` 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 `name` variable's value is used to pre-select options in the select list.

The `value` attribute indicates the default value of the variable named in the `name` attribute. When the element is displayed, and the variable named in the `name` attribute is not set, the `name` variable may be assigned the value specified in the `value` attribute, depending on the values defined in `iname` and `ivalue`. If the `name` variable already contains a value, the `value` attribute is ignored. Any application of the default value is done before the list is pre-selected with the value of the `name` 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 `name` variable is set with this result. In addition, the `value` attribute is interpreted as a semicolon-separated list of pre-selected options.

`iname=`*`nmtoken`*
`ivalue=`*`vdata`*

The `iname` 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 `ivalue` attribute indicates the default-selected `option` element. When the element is displayed, if the variable named in the `iname` attribute is not set, it is assigned the default-selected entry. If the variable already contains a value, the `ivalue` attribute is ignored. If the `iname` attribute is not specified, the `ivalue` 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 (eg, "1;2"). The `iname` variable is set with this result. In addition, the `ivalue` attribute is interpreted as a semicolon-separated list of pre-selected options (eg, "1;4").

`title=`*`vdata`*

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

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

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)
- `tabindex` (see section 11.6.1)

On entry into a card containing a `select` element, the user agent must select the initial `option` elements options in the following manner. Note that values are a semicolon delimited list of values when `multiple="true"`, but are otherwise treated as a single value (even if they contain semicolons). In addition, the default option index is an aggregate value (a list) when `multiple="true"` and is otherwise a single index.

The selection of initial `option` elements includes an operation named *validate*. This operates on a value, and determines if that value is a legal option index (or indices when `multiple="true"`). The operation consists of the following steps:

1. Remove all non-integer indices from the value.
2. Remove all out-of-range indices from the value, where out-of-range is defined as any index with a value greater than the number of options in the `select` or with a value less than one.
3. Remove duplicate indices

Note that an invalid index will result in an *empty* value.

The selection of the initial `option` elements consists of the following steps:

Step 1 - the *default option index* is determined using `iname` and `ivalue`:

- IF the `iname` attribute is specified AND names a variable that is set, THEN the default option index is the validated value of that variable.
- IF the default option index is empty AND the `ivalue` attribute is specified, THEN the default option index is the validated attribute value.
  - IF the default option index is empty, AND the `name` attribute is specified AND the `name` attribute names a variable that is set, THEN for each value in the `name` variable that is present as a value in the `select's option` elements, the index of the first `option` element containing that value is added to the default index if that index has not been previously added.
  - IF the default option index is empty AND the `value` attribute is specified THEN for each value in the `value` attribute that is present as a value in the `select's option` elements, the index of the first `option` element containing that value is added to the default index if that index has not been previously added.
  - IF the default option index is empty AND the `select` is a multi-choice, THEN the default option index is set to zero.
  - IF the default option index is empty AND the `select` is a single-choice, THEN the default option index is set to one.

Step 2 – initialise variables

- IF the `name` attribute is specified AND the `select` is a single-choice element, THEN the named variable is set with the value of the `value` attribute on the `option` element at the default option index.
- Else, IF the `name` attribute is specified and the `select` is a multiple-choice element, THEN for each index greater than zero, the value of the `value` attribute on the `option` element at the index is added to the `name` variable.
- IF the `iname` attribute is specified, THEN the named variable is set with the default option index.

Note: if the default option index is empty, the `name` and `iname` variables are unset.

Step 3 – pre-select option(s) specified by the default option index

- Deselect all options
- For each index greater than zero, select the option specified by the index.

When the user selects or deselects one or more `option` elements, the `name` and `iname` variables are updated with the option's value and index. The `name` is unset if all selected `option` elements contain an empty `value` attribute. However, in all cases, the user agent must not exhibit display side effects as a result of updating `name` and `iname` variables, except when there is an explicit refresh task (see section 9.4.3). The user agent must update `name` and `iname` variables (if specified) for each `select` element in the `card` before each and all task invocations according to steps 1 and 2 above.

Multiple choice selection lists result in a value that is a semicolon delimited list (e.g., `"dog;cat"`). This is not an ordered list and the user agent is free to construct the list in any order that is convenient. Authors must not rely on a particular value ordering. The user agent must ensure that the `iname` result contains no duplicate index values. The `name` result must contain duplicate values in the situation where multiple selected `option` elements have the same value. The `name` result must not contain empty values (e.g., `"cat;;dog"` is illegal).

way:

- If the `iname` attribute exists, the indices in the variable named by `iname` 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 `iname` attribute does not exist and the `name` attribute exists, the value of the variable specified by `name` is used to select options. If the variable specified by `name` 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 `name` is updated to the value of the option.

Both `name` and `iname`, or `value` and `ivalue` may be specified. `ivalue` takes precedence over `value` and `iname` takes precedence over `name`.

11.6.2.2 The Option Element

```
<!ELEMENT option (#PCDATA | onevent)*>
<!ATTLIST option
  value      %vdata;    #IMPLIED
  title      %vdata;    #IMPLIED
  onpick     %HREF;     #IMPLIED
  xml:lang   NMTOKEN    #IMPLIED
  %coreattrs;
  >
```

This element specifies a single choice option in a `select` element.

Attributes

`value=`*`vdata`*

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

The `value` attribute may contain variable references, which are evaluated before the `name` variable is set.

`title=`*`vdata`*

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

`onpick=`*`HREF`*

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

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

11.6.2.3 The Optgroup Element

```
<!ELEMENT optgroup (optgroup|option)+ >
<!ATTLIST optgroup
  title      %vdata;    #IMPLIED
  xml:lang   NMTOKEN    #IMPLIED
  %coreattrs;
  >
```

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.

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

11.6.2.4 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".

```
<wml>
  <card>
    <P>
    Please choose your favourite animal:
    <select name="X">
      <option value="D">Dog</option>
      <option value="C">Cat</option>
    </select>
    </p>
  </card>
</wml>
```

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.

```
<wml>
  <card>
    <p>
    Please choose your favourite animal:
    <select iname="I" ivalue="1">
      <option value="D">Dog</option>
      <option value="C">Cat</option>
    </select>
    </p>
  </card>
</wml>
```

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 "2;3". In addition, the "Dog" and "Cat" options would be pre-selected if the variable "I" had not been previously set.

```
<wml>
  <card>
    <p>
    Please choose <i>all</i> of your favourite animals:
    <select name="X" iname="I" ivalue="1;2" multiple="true">
      <option value="D">Dog</option>
      <option value="C">Cat</option>
      <option value="H">Horse</option>
    </select>
    </p>
  </card>
</wml>
```

In this example, a single choice select list is specified. The variable "F" would be set to the value of "S" if the user chooses the first option. The second option is always pre-selected, regardless of the value of the variable "F".

```
<wml>
  <card>
    <p>
    Please choose from the menu:
```

```
    <select name="F" ivalue="2">
      <option value="S">Sandwich</option>
      <option value="D">Drink</option>
    </select>
    </p>
  </card>
</wml>
```

In this example, the use of the `onpick` intrinsic event is demonstrated. If the user selects the second option, a `go` will be performed to the `/xyz` URL.

```
<wml>
  <card>
    <p>
    Select type of help:
    <select>
      <option onpick="/help.wml">Help</option>
      <option onpick="/morehelp.wml">More Help</option>
    </select>
    </p>
  </card>
</wml>
```

In this example, if the `name` variable is set to the value `"1;2"`, the third option will be pre-selected. This demonstrates that values containing semicolons are treated as a single value in a single-choice selection element.

```
<wml>
  <card>
    <p>
    Select one:
    <select name="K">
      <option value="1">One</option>
      <option value="2">Two</option>
      <option value="1;2">Both</option>
    </select>
    </p>
  </card>
</wml>
```

11.6.3 The Input Element

```
<!ELEMENT input EMPTY>
<!ATTLIST input
  name       NMTOKEN           #REQUIRED
  type       (text|password)   "text"
  value      %vdata;           #IMPLIED
  format     CDATA             #IMPLIED
  emptyok    %boolean;         "false"
  size       %number;          #IMPLIED
  maxlength  %number;          #IMPLIED
  tabindex   %number;          #IMPLIED
  title      %vdata;           #IMPLIED
  xml:lang   NMTOKEN           #IMPLIED
  %coreattrs;
  >
```

The `input` element specifies a text entry object. The user input is constrained by the optional `format` attribute. If a valid input mask is bound to an input object, the user agent must ensure that any value collected by the entry object conforms to the bound input mask. If the input collected does not conform to the input mask, the user agent must not commit that input and must notify the user that the input was rejected and allow the user to resubmit new input. The user agent must not initialise the input object with any value that does not conform to the bound input mask. In the

event that initialising data does not conform to the input mask, the user agent must behave as if there was no initialisation data.

Attributes

`name=nmtoken`
`value=vdata`

The `name` attribute specifies the name of the variable to set with the result of the user's text input. The `name` variable's value is used to pre-load the text entry object. If the `name` variable contains a value that does not conform to the input mask, the user agent must unset the variable and attempt to initialise the variable with the `value` attribute.

The `value` attribute indicates the default value of the variable named in the `name` attribute. When the element is displayed and the variable named in the `name` attribute is not set, the `name` variable is assigned the value specified in the `value` attribute. If the `name` variable already contains a value, the `value` attribute is ignored. If the `value` attribute specifies a value that does not conform to the input mask specified by the `format` attribute, the user agent must ignore the `value` attribute. In the case where no valid value can be established, the `name` variable is left unset.

`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 control. User agents should echo the input in a manner appropriate to the user agent and the input mask. If the submitted value conforms to an existing input mask, the user agent must store that input unaltered and in its entirety in the variable named in the `name` attribute. For example, the user agent must not trim the input by removing leading or trailing white space from the input. If the variable named by the `name` attribute is unset, the user agent should echo an empty string in an appropriate manner.
- `password` - a text entry control. Input of each character should be echoed in an obscured or illegible form in a manner appropriate to the user agent. For example, visual 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. Similar to a `text` type, if the submitted value conforms to an existing input mask, the user agent must store input unaltered and in its entirety in the variable named in the `name` attribute. User agents should not obscure non0formatting characters of the input mask. If the variable named by the `name` attribute is unset, the user agent should echo an empty string in an appropriate manner.

`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. An input mask is only valid when it contains only legal format codes. User agents must ignore invalid masks.

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 (ie, upper-case non-numeric character)

a entry of any lower-case alphabetic or punctuation character (ie, 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 choose 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 choose 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 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 (other than `*f` format code) and specifies what kind of characters can be entered. *Note: This format may only be specified once and must appear at the end of the format string*

\c display the next character, c, in the entry field; allows escaping of the format codes as well as introducing non-formatting characters so they can be displayed in the entry area. Escaped characters are considered part of the input's value, and must be preserved by the user agent. For example, the stored value of the input `"12345-123"` having a mask `"NNNNN\-3N"` is `"12345-123"` and not `"12345123"`. Similarly, if the value of the variable named by the `name` attribute is `"12345123"` and the mask is `"NNNNN\-3N"`, the user agent must unset the variable since it does not conform to the mask.

`emptyok=`*`boolean`*

The emptyok attribute indicates that this `input` element accepts empty input although a non-empty format string has been specified. Typically, the `emptyok` attribute is indicated for formatted entry fields that are optional. By default, `input` elements specifying a `format` require the user to input data matching the `format` specification.

`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

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)
- `tabindex` (see section 11.6.1)

11.6.3.1 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 name="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 name="NAME" type="text" value="Robert"/>
```

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

```
<card>
  <P>
  First name: <input type="text" name="first"/><br/>
```

```
    Last name: <input type="text" name="last"/><br/>
    Age: <input type="text" name="age" format="*N"/>
    </p>
  </card>
```

11.6.4 The Fieldset Element

```
<!ELEMENT fieldset (%fields; | do)* >
<!ATTLIST fieldset
  title           %vdata;      #IMPLIED
  xml:lang        NMTOKEN      #IMPLIED
  %coreattrs;
  >
```

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 11.5.2 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.

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

11.6.4.1 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>
  <card>
    <p>
    <do type="accept">
      <go href="/submit?f=$(fname)&amp;l=$(lname)&amp;s=$(sex)&amp;a=$(age)"/>
    </do>
    <fieldset title="Name">
      <p>First name: <input type="text" name="fname" maxlength="32"/>
      <br/>Last name: <input type="text" name="lname" maxlength="32"/>
      </p>
    </fieldset>
    <fieldset title="Info">
      <p>
      <select name="sex">
        <option value="F">Female</option>
        <option value="M">Male</option>
      </select>
      <br/>
      Age: <input type="text" name="age" format="*N"/>
      </p>
    </fieldset>
    </p>
  </card>
</wml>
```

11.7 The Timer Element

```
<!ELEMENT timer EMPTY>
<!ATTLIST timer
  name       NMTOKEN          #IMPLIED
  value      %vdata;          #REQUIRED
  %coreattrs;
  >
```

The `timer` element declares a card timer, which exposes a means of processing inactivity or idle time. The timer is initialised and started at card entry and is stopped when the card is exited. Card entry is any task or user action that results in the card being activated, for example, navigating into the card. Card exit is defined as the execution of any task (see sections 9.5 and 12.5). The value of a timer will decrement from the initial value, triggering the delivery of an `ontimer` intrinsic event on transition from a value of one to zero. If the user has not exited the card at the time of timer expiration, an `ontimer` intrinsic event is delivered to the card.

Timer resolution is implementation dependent. The interaction of the timer with the user agent's user interface and other time-based or asynchronous device functionality is implementation dependent. It is an error to have more than one `timer` element in a card.

The `timer` timeout value is specified in units of one-tenth (1/10) of a second. The author should not expect a particular timer resolution and should provide the user with another means to invoke a timer's task. If the value of the timeout is not a positive integral number, the user agent must ignore the `timer` element. A timeout value of zero (0) disables the timer.

Invoking a refresh task is considered an exit. The task stops the timer, commits it's value to the context, and updates the user agent accordingly. Completion of the refresh task is considered an entry to the card. At that time, the timer must resume.

Attributes

`name=`*`nmtoken`*

The `name` attribute specifies the name of the variable to be set with the value of the timer. The `name` variable's value is used to set the timeout period upon timer initialisation. The variable named by the `name` attribute will be set with the current timer value when the card is exited or when the timer expires. For example, if the timer expires, the `name` variable is set to a value of "`0`".

`value=`*`vdata`*

The `value` attribute indicates the default value of the variable named in the `name` attribute. When the timer is initialised and the variable named in the `name` attribute is not set, the `name` variable is assigned the value specified in the `value` attribute. If the `name` variable already contains a value, the `value` attribute is ignored. If the `name` attribute is not specified, the timeout is always initialised to the value specified in the `value` attribute.

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)

11.7.1 Timer Example

The following deck will display a text message for approximately 10 seconds and will then go to the URL `/next`.

```
<wml>
  <card ontimer="/next">
       <timer value="100"/>
       Hello World!
  </card>
</wml>
```

The same example could be implemented as:

```
<wml>
  <card>
    <onevent type="ontimer">
      <go href="/next"/>
    </onevent>
    <timer value="100"/>
    <p>
     Hello World!
    </p>
  </card>
</wml>
```

The following example illustrates how a timer can initialise and reuse a counter. Each time the card is entered, the timer is reset to value of the variable `t`. If `t` is not set, the timer is set to a value of 5 seconds.

```
<wml>
  <card ontimer="/next">
    <timer name="t" value="50"/>
    Hello World!
  </card>
</wml>
```

11.8 Text

This section defines the elements and constructs related to text.

11.8.1 White Space

WML white space and line break handling is based on [XML] and assumes the default XML white space handling rules for text. The WML user agent ignores all *insignificant* white space in elements and attribute values, as defined by the XML specification. White space immediately before and after an element is ignored. 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.

11.8.2 Emphasis

```
<!ELEMENT em     (%flow;)*>
<!ATTLIST em
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ELEMENT strong (%flow;)*>
<!ATTLIST strong
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ELEMENT i      (%flow;)*>
<!ATTLIST i
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ELEMENT b      (%flow;)*>
<!ATTLIST b
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ELEMENT u      (%flow;)*>
```

```
<!ATTLIST u
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ELEMENT big    (%flow;)*>
<!ATTLIST big
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ELEMENT small  (%flow;)*>
<!ATTLIST small
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
```

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. The `b`, `i` and `u` elements should not be used except where explicit control over text presentation is required.

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

11.8.3 Paragraphs

```
<!ENTITY % TAlign "(left|right|center)">
<!ENTITY % WrapMode "(wrap|nowrap)" >
<!ELEMENT p (%fields; | do)*>
<!ATTLIST p
  align       %TAlign;     "left"
  mode        %WrapMode;   #IMPLIED
  xml:lang    NMTOKEN      #IMPLIED
  %coreattrs;
  >
```

WML has two line-wrapping modes for visual user agents: breaking (or wrapping) and non-breaking (or non-wrapping). The treatment of a line too long to fit on the screen is specified by the current line-wrap mode. If `mode="wrap"` is specified, the line is word-wrapped onto multiple lines. In this case, line breaks should be inserted into a text flow as appropriate for presentation on an individual device. If `mode="nowrap"` is specified, the line is

not automatically wrapped. In this case, the user agent must provide a mechanism to view entire non-wrapped lines (eg, horizontal scrolling or some other user-agent-specific mechanism).

Any inter-word space is a legal line break point. 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 `p` element establishes both the line wrap and alignment parameters for a paragraph. If the text alignment is not specified, it defaults to `left`. If the line-wrap mode is not specified, it is identical to the line-wrap mode of the previous paragraph in the current card. Empty paragraphs (ie, an empty element or an element with only insignificant white space) should be considered as insignificant and ignored by visual user agents. Insignificant paragraphs do not impact line-wrap mode. If the first `p` element in a card does not specify a line-wrap or alignment mode, that mode defaults to the initial mode for the card. The user agent must insert a line break into the text flow between significant `p` elements.

Insignificant paragraphs may be removed before the document is delivered to the user agent.

Attributes

`align=`*`(left|right|center)`*

This attribute specifies the text alignment mode for the paragraph. 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.

`mode=`*`(wrap|nowrap)`*

This attribute specifies the line-wrap mode for the paragraph. `Wrap` specifies breaking text mode and `nowrap` specifies non-breaking text mode. If not explicitly specified, the line-wrap mode is identical to the line-wrap mode of the previous paragraph in the text flow of a card. The default mode for the first paragraph in a card is `wrap`.

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

11.8.4 The Br Element

```
<!ELEMENT br EMPTY>
<!ATTLIST br
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
```

The `br` element establishes the beginning of a new line. The user agent must break the current line and continue on the following line. User agents should do best effort to support the `br` element in tables (see section 11.8.7).

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

11.8.5 The Table Element

```
<!ENTITY % TAlign "(left|right|center)">
```

```
<!ELEMENT table (tr)+>
<!ATTLIST table
  title       %vdata;    #IMPLIED
  align       CDATA      #IMPLIED
  columns     %number;   #REQUIRED
  xml:lang    NMTOKEN    #IMPLIED
  %coreattrs;
  >
```

The `table` element is used together with the `tr` and `td` elements to create sets of aligned columns of text and images in a card. The `table` elements determine the structure of the columns. The elements separate content into columns, but do not specify column or intercolumn widths. The user agent should do its best effort to present the information of the table in a manner appropriate to the device.

The alignment of text and images within a column is specified by the `align` attribute. A column's contents can be centre aligned, left aligned or right aligned when it is displayed to the user by a visual user agent. The `align` attribute value is interpreted as a list of alignment designations, one for each column. Centre alignment is specified with the value "C", left alignment is specified with the value "L", and right alignment is specified with the value "R". The first designator in the list applies to the first column, the second designator to the second column, and so forth. If an alignment designation is omitted for one or all the columns in the table, the default alignment is applied where the designation is missing. For left-to-right languages, the default alignment is left alignment. For right-to-left languages, the default alignment is right alignment.

The number of columns for the row set must be specified by the `columns` attribute. The user agent must create a row set with exactly the number of columns specified by the `columns` attribute value. If the actual number of columns in a row is less than the value specified by the `columns` attribute, the row must be effectively padded with empty columns. The orientation of the table depends on the language. For left-to-right languages, the leftmost column is the first column in the table. Columns are added to the right side of a row to pad left-to-right tables. Columns are added to the left side of a row to pad right-to-left table.

If the actual number of columns in a row is greater than the value specified by the `columns` attribute, the extra columns of the row must be aggregated into the last column, such that the row contains exactly the number of columns specified. A single inter-word space must be inserted between two cells that are being aggregated.

Depending on the display characteristics, the user agent may create aligned columns for each table, or may use a single set of aligned columns for all table in a card. 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.

Attributes

`title=`*`vdata`*

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

`align=`*`cdata`*

This attribute specifies the layout of text and images within the columns of a row set. A column's contents can be centre aligned, left aligned or right aligned when it is displayed to the user. The attribute value is interpreted as a list of alignment designations, one for each column. Centre alignment is specified with the value "C", left alignment is specified with the value "L", and right alignment is specified with the value "R".

`columns=`*`number`*

This attribute specifies the number of columns for the row set. The user agent must create a row set with exactly the number of columns specified by the attribute value. It is an error to specify a value of zero ("0").

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

11.8.6 The Tr Element

```
<!ELEMENT tr (td)+>
<!ATTLIST tr
  %coreattrs;
  >
```

The `tr` element is used as a container to hold a single table row. Table rows may be empty (i.e., all cells are empty). Empty table rows are significant and must not be ignored.

Attributes defined elsewhere

- `id` (see section 8.9)
- `class` (see section 8.9)

11.8.7 The Td Element

```
<!ELEMENT td ( %text; | %layout; | img | anchor | a )*>
<!ATTLIST td
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
```

The `td` element is used as a container to hold a single table cell data within a table row. Table data cells may be empty. Empty cells are significant, and must not be ignored. The user agent should do a best effort to deal with multiple line data cells that may result from using images or line breaks.

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

11.8.8 Table Example

The following example contains a card with a single column group, containing two columns and three rows.

```
<wml>
   <card>
     <p>
     <table columns="2">
       <tr><td>One </td><td> Two </td></tr>
       <!-- row missing cells -->
       <tr><td>1</td></tr>
       <!-- row with too many cells -->
       <tr><td/><td> B </td><td>C<br>D</td></tr>
     </table>
     </p>
   </card>
</wml>
```

An acceptable layout for this card is:

```
  One    Two
  1
         B C
```

D

11.9 Images

```
<!ENTITY % IAlign "(top|middle|bottom)" >
<!ELEMENT img EMPTY>
<!ATTLIST img
  alt        %vdata;      #REQUIRED
  src        %HREF;       #REQUIRED
  localsrc   %vdata;      #IMPLIED
  vspace     %length;     "0"
  hspace     %length;     "0"
  align      %IAlign;     "bottom"
  height     %length;     #IMPLIED
  width      %length;     #IMPLIED
  xml:lang   NMTOKEN      #IMPLIED
  %coreattrs;
  >
```

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 (ie, the user agent does not support images, or the image contents can not be found).

`src=`*`HREF`*

This attribute specifies the URI for the image. If the browser supports images, it downloads the image from the specified URI 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 URI specified in the `src` attribute, ie, 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.

Attributes defined elsewhere

- `xml:lang` (see section 8.8)
- `id` (see section 8.9)
- `class` (see section 8.9)

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 private information to a secure service, an insecure or malicious service should not be able to retrieve that information from the user agent by other means.

A conforming WML user agent must implement deck-level access control, including the `access` element and the `sendreferer`, `domain` and `path` attributes.

A WML author should remove private or sensitive information from the browser context by clearing the variables containing this information.

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 (ie, 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 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. If this fails and the user agent has exhausted all memory, the user should be notified of the error, and the user agent should be reset to a predictable user state. For example, the browser may be terminated or the context may be cleared and the browser reset to a well-known state.

12.3 Error Handling

Conforming user agents must enforce error conditions defined in this specification and must not hide errors by attempting to infer author or origin server intent.

12.4 Unknown DTD

A WML deck encoded with an alternate DTD may include elements or attributes that are not recognised by certain user agents. In this situation, a user agent should render the deck as if the unrecognised tags and attributes were not present. Content contained in unrecognised elements should be rendered.

12.5 Reference Processing Behaviour - Inter-card Navigation

The following process describes the reference model for inter-card traversal in WML. All user agents must implement this process, or one that is indistinguishable from it.

12.5.1 The Go Task

The process of executing a `go` task comprises the following steps:

1. If the originating task contains `setvar` elements, the variable name and value in each `setvar` element is converted into a simple string by substituting all referenced variables. The resulting collection of variable names and values is stored in temporary memory for later processing. See section 10.3 for more information on variable substitution.
2. The target URI is identified and fetched by the user agent. The URI attribute value is converted into a simple string by substituting all referenced variables.
3. The access control parameters for the fetched deck are processed as specified in section 11.3.1.
4. The destination card is located using the fragment name specified in the URI.
    a) If no fragment name was specified as part of the URI, the first card in the deck is the destination card.
    b) 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.
    c) If the fragment name can not be associated with a specific card, the first card in the deck is the destination card.
5. The variable assignments resulting from the processing done in step #1 (the `setvar` element) are applied to the current browser context.
6. If the destination card contains a `newcontext` attribute, the current browser context is re-initialised as described in section 10.2.
7. The destination card is pushed onto the history stack.
8. If the destination card specifies an `onenterforward` intrinsic event binding, the task associated with the event binding is executed and processing stops. See section 9.9 for more information.
9. If the destination card contains a `timer` element, the timer is started as specified in section 11.7.
10. The destination card is displayed using the current variable state and processing stops.

12.5.2 The Prev Task

The process of executing a `prev` task comprises the following steps:

1. If the originating task contains `setvar` elements, the variable name and value in each `setvar` element is converted into a simple string by substituting all referenced variables. The resulting collection of variable names and values is stored in temporary memory for later processing. See section 10.3 for more information on variable substitution.
2. The target URI is identified and fetched by the user agent. The history stack is popped and the target URI is the top of the history stack. If there is no previous card in the history stack, processing stops.
3. 0The destination card is located using the fragment name specified in the URI.
    a) If no fragment name was specified as part of the URI, the first card in the deck is the destination card.
    b) 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.

4. The variable assignments resulting from the processing done in step #1 (the `setvar` element) are applied to the current browser context.
5. If the destination card specifies an `onenterbackward` intrinsic event binding, the task associated with the event binding is executed and processing stops. See section 9.9 for more information.
6. If the destination card contains a `timer` element, the timer is started as specified in section 11.7.
7. The destination card is displayed using the current variable state and processing stops.

12.5.3 The Noop Task

No processing is done for a `noop` task.

12.5.4 The Refresh Task

The process of executing a `refresh` task comprises the following steps:

1. For each `setvar` element, the variable name and value in each `setvar` element is converted into a simple string by substituting all referenced variables. See section 10.3 for more information on variable substitution.
2. The variable assignments resulting from the processing done in step #1 (the `setvar` element) are applied to the current browser context.
3. If the card contains a `timer` element, the timer is started as specified in section 11.7.
4. The current card is re-displayed using the current variable state and processing stops.

12.5.5 Task Execution Failure

If a task fails to fetch its target URI or the access control restrictions prevent a successful inter-card transition, the user agent must notify the user and take the following actions:

- The invoking card remains the current card.
- No changes are made to the browser context, including any pending variable assignments or `newcontext` processing.
- No intrinsic event bindings are executed.

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 1.1//EN
```

13.1.2 WML Media Type

Textual form:

```
application/vnd.wap.wml
```

Tokenised form:

```
application/vnd.wap.wml-wbxml
```

Ed: these types have been requested from IANA but are not yet registered.

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 1.1//EN"
                        "http://www.wapforum.org/DTD/wml_1.1.xml">
   <wml>
   ...
   </wml>
-->

<!ENTITY % length  "CDATA">    <!-- [0-9]+ for pixels or [0-9]+"%" for
                                   percentage length -->
<!ENTITY % vdata   "CDATA">    <!-- attribute value possibly containing
                                   variable references -->
<!ENTITY % HREF    "%vdata;">  <!-- URI, URL or URN designating a hypertext
                                   node. May contain variable references -->
<!ENTITY % boolean "(true|false)">
<!ENTITY % number  "NMTOKEN">  <!-- a number, with format [0-9]+ -->
<!ENTITY % coreattrs  "id    ID     #IMPLIED
                       class CDATA  #IMPLIED">

<!ENTITY % emph    "em | strong | b | i | u | big | small">
<!ENTITY % layout  "br">

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

<!-- flow covers "card-level" elements, such as text and images -->
<!ENTITY % flow     "%text; | %layout; | img | anchor | a | table">

<!-- Task types -->
<!ENTITY % task   "go | prev | noop | refresh">

<!-- Navigation and event elements -->
<!ENTITY % navelmts "do | onevent">

<!--================ Decks and Cards ================-->

<!ELEMENT wml ( head?, template?, card+ )>
<!ATTLIST wml
  xml:lang        NMTOKEN        #IMPLIED
  %coreattrs;
  >

<!-- card intrinsic events -->
<!ENTITY % cardev
 "onenterforward  %HREF;          #IMPLIED
  onenterbackward %HREF;          #IMPLIED
  ontimer         %HREF;          #IMPLIED"
  >

<!-- card field types -->
<!ENTITY % fields  "%flow; | input | select | fieldset">
```

```
<!ELEMENT card (onevent*, timer?, (do | p)*)>
<!ATTLIST card
  title          %vdata;       #IMPLIED
  newcontext     %boolean;     "false"
  ordered        %boolean;     "true"
  xml:lang       NMTOKEN       #IMPLIED
  %cardev;
  %coreattrs;
  >

<!--================ Event Bindings ================-->

<!ELEMENT do (%task;)>
<!ATTLIST do
  type         CDATA       #REQUIRED
  label        %vdata;     #IMPLIED
  name         NMTOKEN     #IMPLIED
  optional     %boolean;   "false"
  xml:lang     NMTOKEN     #IMPLIED
  %coreattrs;
  >

<!ELEMENT onevent (%task;)>
<!ATTLIST onevent
  type         CDATA       #REQUIRED
  %coreattrs;
  >

<!--================ Deck-level declarations ================-->

<!ELEMENT head ( access | meta )+>
<!ATTLIST head
  %coreattrs;
  >

<!ELEMENT template (%navelmts;)*>
<!ATTLIST template
  %cardev;
  %coreattrs;
  >

<!ELEMENT access EMPTY>
<!ATTLIST access
  domain       CDATA       #IMPLIED
  path         CDATA       #IMPLIED
  %coreattrs;
  >

<!ELEMENT meta EMPTY>
<!ATTLIST meta
  http-equiv     CDATA      #IMPLIED
  name           CDATA      #IMPLIED
  forua          %boolean;  #IMPLIED
  content        CDATA      #REQUIRED
  scheme         CDATA      #IMPLIED
  %coreattrs;
  >
```

```
<!--================ Tasks ================-->

<!ELEMENT go (postfield | setvar)*>
<!ATTLIST go
  href            %HREF;           #REQUIRED
  sendreferer     %boolean;       "false"
  method          (post|get)      "get"
  accept-charset  CDATA           #IMPLIED
  %coreattrs;
  >

<!ELEMENT prev (setvar)*>
<!ATTLIST prev
  %coreattrs;
  >

<!ELEMENT refresh (setvar)*>
<!ATTLIST refresh
  %coreattrs;
  >

<!ELEMENT noop EMPTY>
<!ATTLIST noop
  %coreattrs;
  >

<!--================ postfield ================-->

<!ELEMENT postfield EMPTY>
<!ATTLIST postfield
  name         %vdata;           #REQUIRED
  value        %vdata;           #REQUIRED
  %coreattrs;
  >

<!--================ variables ================-->

<!ELEMENT setvar EMPTY>
<!ATTLIST setvar
  name         %vdata;           #REQUIRED
  value        %vdata;           #REQUIRED
  %coreattrs;
  >

<!--================ Card Fields ================-->

<!ELEMENT select (optgroup|option)+>
<!ATTLIST select
  title        %vdata;           #IMPLIED
  name         NMTOKEN           #IMPLIED
  value        %vdata;           #IMPLIED
  iname        NMTOKEN           #IMPLIED
  ivalue       %vdata;           #IMPLIED
  multiple     %boolean;         "false"
  tabindex     %number;          #IMPLIED
  xml:lang     NMTOKEN           #IMPLIED
  %coreattrs;
  >
```

```
<!ELEMENT optgroup (optgroup|option)+ >
<!ATTLIST optgroup
  title      %vdata;    #IMPLIED
  xml:lang   NMTOKEN    #IMPLIED
  %coreattrs;
  >

<!ELEMENT option (#PCDATA | onevent)*>
<!ATTLIST option
  value      %vdata;    #IMPLIED
  title      %vdata;    #IMPLIED
  onpick     %HREF;     #IMPLIED
  xml:lang   NMTOKEN    #IMPLIED
  %coreattrs;
  >

<!ELEMENT input EMPTY>
<!ATTLIST input
  name       NMTOKEN           #REQUIRED
  type       (text|password)   "text"
  value      %vdata;           #IMPLIED
  format     CDATA             #IMPLIED
  emptyok    %boolean;         "false"
  size       %number;          #IMPLIED
  maxlength  %number;          #IMPLIED
  tabindex   %number;          #IMPLIED
  title      %vdata;           #IMPLIED
  xml:lang   NMTOKEN           #IMPLIED
  %coreattrs;
  >

<!ELEMENT fieldset (%fields; | do)* >
<!ATTLIST fieldset
  title          %vdata;      #IMPLIED
  xml:lang       NMTOKEN      #IMPLIED
  %coreattrs;
  >

<!ELEMENT timer EMPTY>
<!ATTLIST timer
  name       NMTOKEN           #IMPLIED
  value      %vdata;           #REQUIRED
  %coreattrs;
  >

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

<!ENTITY % IAlign "(top|middle|bottom)" >

<!ELEMENT img EMPTY>
<!ATTLIST img
  alt        %vdata;      #REQUIRED
  src        %HREF;       #REQUIRED
  localsrc   %vdata;      #IMPLIED
  vspace     %length;     "0"
  hspace     %length;     "0"
  align      %IAlign;     "bottom"
```

```
  height     %length;     #IMPLIED
  width      %length;     #IMPLIED
  xml:lang   NMTOKEN      #IMPLIED
  %coreattrs;
  >

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

<!ELEMENT anchor ( #PCDATA | br | img | go | prev | refresh )*>
<!ATTLIST anchor
  title        %vdata;      #IMPLIED
  xml:lang     NMTOKEN      #IMPLIED
  %coreattrs;
  >

<!ELEMENT a ( #PCDATA | br | img )*>
<!ATTLIST a
  href         %HREF;       #REQUIRED
  title        %vdata;      #IMPLIED
  xml:lang     NMTOKEN      #IMPLIED
  %coreattrs;
  >

<!--================ Tables ================-->

<!ELEMENT table (tr)+>
<!ATTLIST table
  title       %vdata;    #IMPLIED
  align       CDATA      #IMPLIED
  columns     %number;   #REQUIRED
  xml:lang    NMTOKEN    #IMPLIED
  %coreattrs;
  >

<!ELEMENT tr (td)+>
<!ATTLIST tr
  %coreattrs;
  >

<!ELEMENT td ( %text; | %layout; | img | anchor | a )*>
<!ATTLIST td
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >

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

<!ELEMENT em     (%flow;)*>
<!ATTLIST em
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >

<!ELEMENT strong (%flow;)*>
<!ATTLIST strong
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
```

```
<!ELEMENT b      (%flow;)*>
<!ATTLIST b
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >

<!ELEMENT i      (%flow;)*>
<!ATTLIST i
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >

<!ELEMENT u      (%flow;)*>
<!ATTLIST u
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >

<!ELEMENT big    (%flow;)*>
<!ATTLIST big
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >

<!ELEMENT small  (%flow;)*>
<!ATTLIST small
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >

<!ENTITY % TAlign "(left|right|center)">
<!ENTITY % WrapMode "(wrap|nowrap)" >
<!ELEMENT p (%fields; | do)*>
<!ATTLIST p
  align       %TAlign;     "left"
  mode        %WrapMode;   #IMPLIED
  xml:lang    NMTOKEN      #IMPLIED
  %coreattrs;
  >

<!ELEMENT br EMPTY>
<!ATTLIST br
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >

<!ENTITY quot  "&#34;">     <!-- quotation mark -->
<!ENTITY amp   "&#38;#38;"> <!-- ampersand -->
<!ENTITY apos  "&#39;">     <!-- apostrophe -->
<!ENTITY lt    "&#38;#60;"> <!-- less than -->
<!ENTITY gt    "&#62;">     <!-- greater than -->
<!ENTITY nbsp  " ">    <!-- non-breaking space -->
<!ENTITY shy   "&#173;">    <!-- soft hyphen (discretionary hyphen) -->
```

13.3 Reserved Words

WML reserves the use of several strings for future uses. These strings may not be used in any DTD or extension of WML. The following words are reserved:

```
style
```

14. A Compact Binary Representation of WML

WML may be encoded using a compact binary representation. This content format is based upon the WAP Binary XML Content Format [WBXML].

14.1 Extension Tokens

14.1.1 Global Extension Tokens

The [WBXML] global extension tokens are used to represent WML variables. 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 (eg, direct substitution, escaped substitution and unescaped substitution). The variable name is encoded in the current document character encoding and must be encoded as the specified in the source document (eg, variable names may not be shortened, mapped or otherwise changed). For example, the global extension token `EXT_I_0` represents an escaped variable substitution, with the variable name inline.

14.1.2 Tag Tokens

WML defines a set of single-byte tokens corresponding to the tags defined in the DTD. All of these tokens are defined within code page zero.

14.1.3 Attribute Tokens

WML defines a set of single-byte tokens corresponding to the attribute names and values defined in the DTD. All of these tokens are defined within code page zero.

14.2 Encoding Semantics

14.2.1 Encoding Variables

All valid variable references must be converted to variable reference tokens (eg, `EXT_I_0`). The encoder must validate that a variable reference uses proper syntax. The encoder should also validate that the placement of the variable reference within the WML deck is valid.

14.2.2 Document Validation

XML document validation (see [XML]) should occur during the process of tokenising a WML deck and must be based on the DOCTYPE declared in the WML deck. When validating the source text, the tokenisation process must accept any DOCTYPE or public identifier, if the document is identified as a WML media type (see section 13.1.2).

The tokenisation process should notify the user of any well-formedness or validity errors detected in the source deck.

14.2.2.1 Validate %length;

The WML tokenisation process should validate that attribute values defined as `%length;` contain either a `NMTOKEN` or a `NMTOKEN` followed by a percentage sign character. For example, the following attributes are legal:

```
vspace="100%"
hspace="123"
```

`%length;` data is encoded using normal attribute value encoding methods.

14.2.2.2 Validate %vdata;

The WML tokenisation process must validate the syntax of all variable references within attribute values defined as `%vdata;` contain variables and that other `CDATA` attribute values do not. Attribute values not defined in the DTD must be treated as `%vdata;` and validated accordingly.

14.3 Numeric Constants

14.3.1 WML Extension Token Assignment

The following global extension tokens are used in WML and occupy document-type-specific token slots in the global token range. As with all tokens in the global range, these codes must be reserved in every code page. All numbers are in hexadecimal.

Table 4. Global extension token assignments

Token Name	***Token***	***Description***
EXT_I_0	40	Variable substitution - escaped. Name of the variable is inline and follows the token as a `termstr`.
EXT_I_1	41	Variable substitution - unescaped. Name of the variable is inline and follows the token as a `termstr`.
EXT_I_2	42	Variable substitution - no transformation. Name of the variable is inline and follows the token as a `termstr`.
EXT_T_0	80	Variable substitution - escaped. Variable name encoded as a reference into the string table.
EXT_T_1	81	Variable substitution - unescaped. Variable name encoded as a reference into the string table.
EXT_T_2	82	Variable substitution - no transformation. Variable name encoded as a reference into the string table.
EXT_0	C0	Reserved for future use.
EXT_1	C1	Reserved for future use.
EXT_2	C2	Reserved for future use.

14.3.2 Tag Tokens

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

Table 5. Tag tokens

Tag Name	***Token***
a	1C
anchor	22
access	23
b	24
big	25
br	26
card	27

Tag Name	***Token***
do	28
em	29
fieldset	2A
go	2B
head	2C
i	2D
img	2E

Tag Name	***Token***
input	2F
meta	30
noop	31
p	20
postfield	21
prev	32
onevent	33
optgroup	34
option	35
refresh	36
select	37

Tag Name	***Token***
setvar	3E
small	38
strong	39
table	1F
td	1D
template	3B
timer	3C
tr	1E
u	3D
wml	3F

14.3.3 Attribute Start Tokens

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

Table 6. Attribute start tokens

Attribute Name	***Attribute Value Prefix***	***Token***
accept-charset		5
align		52
align	bottom	6
align	center	7
align	left	8
align	middle	9
align	right	A
align	top	B
alt		C
class		54
columns		53
content		D
content	application/vnd.wap.wmlc;charset=	5C
domain		F
emptyok	false	10
emptyok	true	11
format		12
forua	false	56

Attribute Name	***Attribute Value Prefix***	***Token***
forua	true	57
height		13
href		4A
href	http://	4B
href	https://	4C
hspace		14
http-equiv		5A
http-equiv	Content-Type	5B
http-equiv	Expires	5D
id		55
ivalue		15
iname		16
label		18
localsrc		19
maxlength		1A
method	get	1B
method	post	1C
mode	nowrap	1D
mode	wrap	1E

Attribute Name	***Attribute Value Prefix***	***Token***
multiple	false	1F
multiple	true	20
name		21
newcontext	false	22
newcontext	true	23
onenterbackward		25
onenterforward		26
onpick		24
ontimer		27
optional	false	28
optional	true	29
path		2A
scheme		2E
sendreferer	false	2F
sendreferer	true	30
size		31
src		32
src	http://	58
src	https://	59
ordered	true	33
ordered	false	34

Attribute Name	***Attribute Value Prefix***	***Token***
tabindex		35
title		36
type		37
type	accept	38
type	delete	39
type	help	3A
type	password	3B
type	onpick	3C
type	onenterbackward	3D
type	onenterforward	3E
type	ontimer	3F
type	options	45
type	prev	46
type	reset	47
type	text	48
type	vnd.	49
value		4D
vspace		4E
width		4F
xml:lang		50

14.3.4 Attribute Value Tokens

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

Table 7. Attribute value tokens

Attribute Value	***Token***
.com/	85
.edu/	86
.net/	87
.org/	88
accept	89
bottom	8A
clear	8B
delete	8C
help	8D

Attribute Value	***Token***
http://	8E
http://www.	8F
https://	90
https://www.	91
middle	93
nowrap	94
onenterbackward	96
onenterforward	97
onpick	95

Attribute Value	***Token***
ontimer	98
options	99
password	9A
reset	9B
text	9D

Attribute Value	***Token***
top	9E
unknown	9F
wrap	A0
Www.	A1

14.4 WML Encoding Examples

Refer to [WBXML] for additional examples.

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

```
<wml>
  <card id="abc" ordered="true">
    <p>
      <do type="accept">
        <go href="http://xyz.org/s"/>
      </do>
      X: $(X)<br/>
      Y: $(&#x59;)<br/>
      Enter name: <input type="text" name="N"/>
    </P>
  </card>
</wml>
```

Tokenised form (numbers in hexadecimal) 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:

```
01  04  6A  04  'X' 00  'Y' 00  7F  E7  21  03  'a' 'b' 'c' 00
33  01  20  E8  38  01  AB  4B  03  'x' 'y' 'z' 00  88  03
's' 00  01  03  ' ' 'X' ':' ' ' 00  82  00  26  03  ' ' 'Y'
':' ' ' 00  82  02  28  03  ' ' 'E' 'n' 't' 'e' 'r' ' ' 'n'
'a' 'm' 'e' ':' ' ' 00  AF  48  18  03  'N' 00  01  01  01
```

In an expanded and annotated form:

Table 8. Example tokenised deck

Token Stream	***Description***
01	WBXML Version number 1.1
04	WML 1.1 Public ID
6A	Charset=UTF-8 (MIBEnum 106)
04	String table length
'X', 00, 'Y', 00	String table
7F	`wml`, with content
E7	`card`, with content and attributes
55	`id=`
03	Inline string follows
'a', 'b', 'c', 00	string
33	`ordered="true"`
01	`END` (of `card` attribute list)
20	`p`
E8	`do`, with content and attributes
38	`type=accept`
01	`END` (of `do` attribute list)

Token Stream	***Description***
AB	go, with attributes
4B	href="http://"
03	Inline string follows
'x', 'y', 'z', 0	string
88	".org/"
03	Inline string follows
's', 0	string
01	END (of do element)
03	Inline string follows
' ', 'X', ':', ' ', 00	String
82	Direct variable reference (EXT_T_2)
00	Variable offset 0
26	br
03	Inline string follows
' ', 'Y', ':', ' ', 00	String
82	Direct variable reference (EXT_T_2)
02	Variable offset 2
26	br
03	Inline string follows
' ', 'E', 'n', 't', 'e', 'r', ' ', 'n', 'a', 'm', 'e', ':', ' ', 00	String
AF	input, with attributes
48	type="text"
21	name=
03	Inline string follows
'N', 00	String
01	END (of input attribute list)
01	END (of p element)
01	END (of card element)
01	END (of wml element)