WML 1.3 Developer’s Guide
# Contents

## About This Book
- Openwave SDK
- Audience
- Style and Typographical Conventions
- Related Documentation
- Technical Support and Other Resources

## 1 Introduction to WML
- Overview of WML Syntax
- Navigating Between Cards and Decks
- Handling User Input
- Formatting Text

## 2 Doing More with WML
- Using Contexts to Structure Your Service Interface
- Letting Users Bookmark Your Service
- Interacting with the Mobile Browser Cache
- Using Timers

## 3 Dynamic WML Services
- Creating Dynamic WML Services
- Tuning Performance with Digests
- Modifying the Mobile Browser Device Cache
- Initiating Voice Calls

## 4 Using Images
- Using Images in WML
- Using Images Efficiently
5 Sending Notifications 59
   Notification Content Types 60
   How the Mobile Access Gateway Delivers Notifications 62
   How WML Services Can Use Notifications 66
   Sending Notifications 67
   Combining Alerts and WML Content in Pull Notifications 67
   Debugging Notifications 68
   Notifications and Security 72

6 Security 75
   Overview of Mobile Access Gateway Platform Security 75
   Writing WML That Minimizes Security Risks 76
   Common Security Risks for WML Services 77

7 Interacting with Other Services 79
   Overview 79
   Issuing Requests to Registered Services 80
   Providing a Registered Service 89

8 User Interface Guidelines 93
   Testing Your User Interface 93
   Best Practices 93
   Graphical Browser Enhancements 95
   Navigation 95
   Forms 99
   Data Entry 101
   Function Keys 102
   Alerts 103
   Images 104

9 Internationalization 105
   Providing WML Content in Multiple Languages 105
   Handling Different Character Sets 106

Index 113
About This Book

This book provides information on developing WML 1.3 services and applications for mobile browsers accessing an Openwave Mobile Access Gateway. Topics covered include using browser cache, bookmarks, dynamic WML services, security, images, and interface guidelines.

Wireless Markup Language (WML) is an XML-based solution developed by the WAP Forum. It was designed to address the need for a standard markup language for small, narrowband, mobile devices. Browsers used in these mobile devices are WAP compliant if they can handle WML documents.

The following table shows which Openwave Mobile Browsers support which WML versions.

<table>
<thead>
<tr>
<th>WML Version</th>
<th>Mobile Browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>WML 1.3</td>
<td>Mobile Browser, WAP Edition 5.0 or later</td>
</tr>
<tr>
<td>WML 1.1</td>
<td>Mobile Browser 4.1</td>
</tr>
<tr>
<td></td>
<td>Mobile Browser 3.2</td>
</tr>
<tr>
<td>WML 1.0</td>
<td>Mobile Browser 4.0</td>
</tr>
</tbody>
</table>

**IMPORTANT** For a complete and current list of mobile browser devices, Openwave Mobile Browser releases, and Openwave Mobile Access Gateway releases, check the Openwave Developer web site at:

http://developer.openwave.com
**Openwave SDK**

The Openwave Software Development Kit (SDK) is a useful tool for developing, debugging, and maintaining your WML programs. Using the SDK, you can test your code from your local disk, through your own server, or through a Mobile Access Gateway. The tools in the SDK include an editor, an output window that lists transaction information, an HTTP window that displays source code, and the ability to view history, cookies, and variables.

In order to use the SDK, you must make certain that you download and install the correct release. SDK releases support the following browsers:

<table>
<thead>
<tr>
<th>SDK Version</th>
<th>Mobile Browser Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDK WAP Edition 5.0</td>
<td>Mobile Browser, WAP Edition 5.0 or later</td>
</tr>
<tr>
<td>SDK Release 4.1</td>
<td>Mobile Browser 4.1 or later</td>
</tr>
<tr>
<td>SDK Release 4.0</td>
<td>Mobile Browser 4.0 or later</td>
</tr>
<tr>
<td>SDK Release 3.2 for WML1.1</td>
<td>Mobile Browser 3.1 or later</td>
</tr>
</tbody>
</table>

**Audience**

This book is intended for developers who are creating wireless WML services for mobile browser devices that are accessing an Openwave Mobile Access Gateway.

To use this book profitably, you should be familiar with the following:

- HTML, because WML has roots in HTML and is similar in syntax
- XML, because WML is an XML-based language
- Dynamic content creation using Common Gateway Interface (CGI) protocol or active server pages (ASP)
- Programming languages such as C and Perl

**Style and Typographical Conventions**

This manual uses different fonts to represent the information you must enter:

- Text that appears like this identifies command names, path names, URLs, and specific text that you must enter.
- Text that appears like this identifies placeholders or variables that you should replace with values appropriate to your environment.
- Text that appears like this identifies default WML attribute values.
**Code Examples**

Omitted code is indicated with ellipses. For instance, the ellipses in the following example indicate that additional code exists in this WML card definition:

1. `<wml>`
2. `<card>`
3. `<p>`
4.  `Your card definition goes here.`
5.  `</p>`
6.  `...`
7.  `</card>`
8.  `</wml>`

Line numbers that appear in some code examples are for reference only and are not part of the actual code.

**Related Documentation**

**WML**

The *WML 1.3 Language Reference* is a companion book to this book. It provides detailed information on the WML elements and attributes. Use of this book is essential in order to make full use of the WML features and Openwave extensions.

**WMLScript**

WMLScript compliments WML by providing code that runs within the WML document while it is displayed on the mobile browser device.

The *WMLScript 1.2 Developer’s Guide* provides detailed instructions for implementing WMLScript programs.

The *WMLScript 1.2 Reference* provides detailed information about the WMLScript programming language.

**Openwave SDK**

The SDK is a useful environment for developing and debugging WML code.

The *Getting Started Guide* provides instructions for installing and getting started with the Openwave SDK and the basics of creating a wireless service. It also describes the operation of the browser user interface in detail.
HDML
HDML was the predecessor to WML and is supported by translation on the Mobile Access Gateway 5.0.
The HDML 3.0 Reference provides reference information on each HDML statement and option.
The HDML Developer’s Guide provides instructions for developers for implementing a HDML service.

Advanced Wireless Features
The Tools and API Reference provides reference information about the tools and APIs available with Openwave SDKs. These tools consist of Perl and C libraries that automate some of the processes associated with creating a wireless service.
The Push Library Developer’s Guide provides detailed reference information and programming information for developing services that push information and service to mobile browser device users.

Other Documentation
For a complete list of available documentation, see the Openwave Developer site at:
http://developer.openwave.com

Technical Support and Other Resources
The best resource for up-to-date information on developing wireless web services is the Openwave Developer site at:
http://developer.openwave.com
You can download tools and find a variety of useful resources, including Frequently Asked Questions, bug reporting, technical support, and an interactive developer forum.
This chapter introduces the Wireless Markup Language (WML) and demonstrates how to use it to build simple WML services. Because the examples in this chapter are relatively simple, you should read the user interface guidelines in Chapter 8 before implementing a complete WML service.

Overview of WML Syntax

WML is a markup language based on Extensible Markup Language (XML). XML, like HTML, is a Standardized Generalized Markup Language (SGML) variant.

The WAP Forum provides a formal Document Type Definition (DTD) for WML at:

http://www.wapforum.org

Two DTDs are referred to throughout this document:

http://www.wapforum.org/DTD/wml13.dtd
http://www.openwave.com/dtd/wml13.dtd

The Openwave DTD includes the Openwave WML 1.3 extensions. You should use this DTD if you plan to use WML 1.3 with Openwave extensions. Otherwise you should use the WAP Forum DTD, which refers to the official WAP Forum WML 1.3 specification.

The WML language defines elements and attributes that you use to specify the user interface components, called cards, that users see on their mobile browser device. Just as a web browser can navigate from one page to another, the Openwave Mobile Browser can navigate from one card to another.

A card can specify multiple user actions by including one or more of the following:

- Formatted text, including text, images, and links
- input elements that let the user enter a string of text
- select elements that let the user choose from a list of options
- fieldset elements that act as organizational containers for other elements
The smallest unit of WML that you can send to a mobile browser device is a deck—one or more cards that a user is likely to access together. When a mobile browser device receives a WML deck, it displays the content defined in the first card and allows the user to respond. Depending on the card definition, the user can respond by entering text or by choosing an option. Mobile browser devices with larger displays typically present each card as a single screen. Some smaller devices present each card as a collection of screens.

**NOTE** The Mobile Access Gateway supports a special construct called a *digest* that you can use to package multiple decks into a single HTTP response. For information about using this multipart MIME format, see Chapter 3, “Dynamic WML Services.”

The high-level syntax for defining a WML deck is as follows:

```xml
<wml>
  <head>
    access control and meta information
  </head>
  <template>
    deck-level event bindings
  </template>
  <card>
    card-level event bindings
  </card>
</wml>
```

The following table describes the elements in this syntax. Required elements are in bold type.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;wml&gt;</code></td>
<td>Required. The <code>&lt;wml&gt;</code> element specifies a deck.</td>
</tr>
<tr>
<td><code>&lt;head&gt;</code></td>
<td>The <code>&lt;head&gt;</code> element specifies information about the deck as a whole, including metadata and access control information. All WML decks are private by default—you must explicitly define a deck as public for cards or decks outside your domain to access it.</td>
</tr>
<tr>
<td><code>&lt;template&gt;</code></td>
<td>The <code>&lt;template&gt;</code> element defines deck-level event bindings—characteristics that apply to all cards in the deck. You can override these characteristics for a particular card by specifying the same event bindings within the <code>&lt;card&gt;</code> element.</td>
</tr>
</tbody>
</table>
| `<card>`  | Required. One or more `<card>` elements define the actions and user interface for the deck. A card can contain the following components:  
- Formatted text, which can include text, images, and links  
- `<input>` elements, which allow the user to enter a string of text  
- `<select>` elements, which allow the user to choose from a list of options  
- `<fieldset>` elements, which act as organizational containers for other elements |
Most WML elements have one or more attributes, some of which are optional. Use attributes to specify additional information about how the device should handle the element. Although the exact syntax of a WML statement depends on whether or not the element has content, attributes always appear in the element start tag:

```xml
<element a1="value1" a2="value2" ...> content </element>
```

You must enclose attribute values within single (') or double (") quotation marks and separate each attribute-value pair with white space (space, tab, newline, or carriage return character). White space is not allowed, however, between the attribute name, equal sign, and attribute value.

Like HTML, WML uses the printable 8-bit character set and converts one or more contiguous newlines, carriage returns, tabs, or spaces to a single space. Unlike HTML, WML is case-sensitive—all WML elements and attributes must be lowercase. The examples in this manual use newlines and tabs for readability, but this formatting is not required. In fact, the Mobile Access Gateway removes it before sending the WML to the phone. You can also include comments in your WML code using the following syntax:

```xml
<!-- This is the first line of my comment. -->
<!-- This is the second line of my comment. -->
```

**Navigating Between Cards and Decks**

Like conventional web sites, a WML service usually consists of a set of screens that present information with which users can interact. In WML, these screens are defined as a set of one or more cards. The actual correspondence between cards and screens depends on the device display size, as well as on how you define your cards. On larger devices, the Openwave Mobile Browser may present a card as a single screen. On smaller devices, the same card may appear as multiple screens.

The key to constructing an efficient and easy-to-use WML service is determining the best way to

- Group your information content into cards and decks
- Create mechanisms that let users navigate between these components.
Before discussing how to navigate between multiple cards, it is helpful to look at a simple synopsis of the `<card>` element:

```wml
card id="name">
  do type="type" label="label">
    go href="url"/>
  </do>

  p
  text
  </p>
</card>
```

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Specifies a name that lets you navigate to the card from other cards in the same deck.</td>
</tr>
<tr>
<td><code>&lt;do&gt;</code></td>
<td>Defines the action that the mobile browser device performs when the user presses the function key specified by the <code>type</code> attribute. The <code>label</code> attribute specifies a label to display for the function key. If you do not specify this option, the device uses a default label.</td>
</tr>
<tr>
<td><code>&lt;go&gt;</code></td>
<td>Defines the specific task to perform within the <code>&lt;do&gt;</code> action, in this case a <code>&lt;go&gt;</code> task that requests a particular URL.</td>
</tr>
<tr>
<td><code>text</code></td>
<td>Specifies the formatted text displayed on the mobile browser device.</td>
</tr>
</tbody>
</table>

**NOTE** For a complete synopsis of the `<card>` element and its attributes, see the [WML 1.3 Language Reference](#).

### Specifying URLs

As in HTML, navigation in WML occurs by specifying URLs. To navigate between multiple cards and decks, you simply specify the URL that you want to open.

To navigate to a particular card, specify a URL with the following syntax:

```wml
deck_URL#card_id
```

If you do not specify a card, the device automatically displays the first card in the deck.
For example, if the deck `services/mydeck.wml` has the following definition,

```wml
<wml>
  <card id="card1">
    ...
  </card>
  <card id="card2">
    ...
  </card>
  <card id="card3">
    ...
  </card>
</wml>
```

you would specify `services/mydeck.wml` (the URL of the deck) to navigate to the first card. To navigate to the third card, you would specify the following URL:

```
services/mydeck.wml#card3
```

To navigate from one card to another card in the same deck, you can also use a relative URL and simply reference it as `#card_id`.

**Specifying CGI Arguments**

Specifying CGI arguments within URLs deserves special attention because of two important differences between HTML and WML:

- You can use a leading question mark (?) to indicate the URL of the current deck. For example, specifying `URL=?foo=bar` reloads the current URL with the `foo` argument. Specifying `URL=?` reloads the current URL with no arguments.

- You must use the special character element `&amp;` to specify the ampersand (&) character when you use CGI arguments in URL strings.

**The History Stack**

The mobile browser device maintains a history of the cards the user visits. Each time the user navigates forward to a card, the mobile browser device pushes it onto the history stack. When the user invokes the `PREV` mechanism, the mobile browser device pops the current card off the history stack and returns to the previous card.

Mobile browser device history operation is similar to that of most web browsers, except that users cannot navigate forward through the history stack—the current card is always at the top of the stack.
For example, suppose that the deck just described defines the following cards.

**Figure 1-1. Card history stack**

Figure 1-1 illustrates how the history stack changes as the user navigates among the cards.

To view the history stack in the Openwave Mobile Browser Simulator, choose Info > History. The Openwave Mobile Browser Simulator displays the history stack from bottom to top in the Phone Information window—in other words, from least-recently visited to most-recently visited. So, for example, the history stack for the Menu card shown in Figure 1-1 would appear as follows:

http://foo.com/services/mydeck.wml
http://foo.com/services/mydeck.wml#card2
http://foo.com/services/mydeck.wml#card3
Defining Actions for Multiple Cards

As described earlier, the `<do>` element lets you associate an action (for example, going to a particular URL) with a mechanism for invoking it (for example, pressing the ACCEPT or OPTIONS function key). You can define multiple `<do>` statements for a card, each of which assigns a task to a specific key or user interface mechanism.

In some cases, you may want to assign the same task to a particular key for every card in a deck. One way to accomplish this is to specify identical `<do>` statements in each `<card>` statement. As shown in the following example, a more efficient way is to use the `<template>` element to define behavior that applies to the entire deck.

Listing 1-1. moon.wml

```wml
<wml>
  <template>
    <do type="options" label="Moon">
      <go href="moon.wml"/>
    </do>
  </template>

  <card>
    <do type="accept" label="Tues">
      <go href="#tues"/>
    </do>
    <p>
      Current temps <br/>
      Hi: 60 <br/>
      Lo: 28
    </p>
  </card>

  <card id="tues">
    <do type="accept" label="Wed">
      <go href="#wed"/>
    </do>
    <p>
      Tuesday temps <br/>
      Hi: 78 <br/>
      Lo: 36
    </p>
  </card>

  <card id="wed">
    <p>
      Wednesday temps <br/>
      Hi: 80 <br/>
      Lo: 40
    </p>
  </card>

</wml>
```
As illustrated in Figure 1-2, the <template> statement instructs the Mobile Browser to do the following for every card in the deck:

- Label the OPTIONS key with the word Moon.
- Display the first card in the moon.wml deck when the user presses the OPTIONS key.

The <do> statements apply only to the cards for which they are defined. The <do> statement for the first card instructs the Mobile Browser to do the following:

- Change the label of the ACCEPT key from OK (the default) to Tues
- Display a card named tues when the user presses the ACCEPT key

The <do> statement for the second card instructs the Mobile Browser to do the following:

- Change the label of the ACCEPT key from OK to Wed
- Display a card named wed when the user presses the ACCEPT key

The third card in the deck has no <do> statement because the default action for the ACCEPT key is to return to the previous card. When the user presses the ACCEPT key, the Mobile Browser automatically displays the previous card. For more information about the default actions for different keys, see the WML 1.3 Language Reference.
Overriding Deck-Level Actions

As shown in the previous example, you can define <do> statements at both the deck level (using the <template> element) and the card level. A deck-level <do> statement applies to all cards in the deck. A card-level <do> statement applies to the individual card only. Card-level definitions override (or shadow) deck-level definitions if both statements have the same name, or are of the same type if you do not specify a name.

The following example (illustrated in Figure 1-3) defines a deck-level <do> statement for the ACCEPT key. The first card in the deck specifies a <do> statement with the same type attribute, so the card-level binding overrides the deck-level binding for that card. The deck-level binding still applies to the second and third cards in the deck.

```
<wml>
  <template>
    <do type="accept" label="Pick">
      <prev/>
    </do>
  </template>

  <card id="card1">
    <do type="accept" label="Black">
      <go href="#card2"/>
    </do>
    <do type="options" label="White">
      <go href="#card3"/>
    </do>
  </card>

  <card id="card2">
    <p>
      You picked black.
    </p>
  </card>

  <card id="card3">
    <p>
      You picked white.
    </p>
  </card>
</wml>
```
Using Multiple Decks

The sample deck in Listing 1-1 on page 7 also illustrates that card navigation can span multiple decks. The card that displays the date of the next full moon could have been defined in the same deck as the other cards. Because of packet size limitations, however, it often makes sense to break cards into separate decks.

**IMPORTANT** The size limit for a deck after the Mobile Access Gateway compiles it is 2000 bytes. Because dynamic WML services typically generate responses of varying length, you should limit the size of your decks to 1200 bytes. To check the size of a compiled deck, view it in the Openwave SDK simulator and check the compiled file size in the Browser Output window.

As shown in line 4 of Listing 1-1, you can use a relative URL to specify another deck in the same directory. As mentioned earlier, you do not have to specify a card name when navigating to a different deck unless you want to go to a card other than the first card in the deck.

---

1. The compiled file size is usually less than the uncompiled file size. However, it may be larger if the deck contains text with many words and spaces because the Mobile Access Gateway tokenizes each word as a null-terminated string and each space as a byte code.
Defining Links

Anchored links provide another mechanism for letting users navigate between cards and decks. Unlike the more generic <do> element, in which you can bind a task to a variety of user interface mechanisms, users always invoke the task associated with an anchored link by pressing the ACCEPT key. As in HTML, WML links are anchored to text that the device displays in a different manner than surrounding text. WML links can also be anchored to the <img> element. The following is a synopsis of the <anchor> element, which defines an anchored link.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Specifies a label for the ACCEPT key (optional). When users scroll to the link, this label overrides any other label defined for that key in the current card or deck.</td>
</tr>
<tr>
<td>task</td>
<td>Specifies the task to execute, either &lt;go&gt;, &lt;prev&gt;, or &lt;refresh&gt;.</td>
</tr>
<tr>
<td>text</td>
<td>Specifies the text to which the link is anchored.</td>
</tr>
<tr>
<td>&lt;img&gt;</td>
<td>Specifies the image to which the link is anchored.</td>
</tr>
</tbody>
</table>

There is also a short form syntax for anchors. It uses the <a> tag instead of the <anchor> tag and can only be used to define (implied) <go> tasks that require a URL specification.

```wml
<a href="url" !-- required -->
title="label">
any valid combination of text, and br and img elements
</a>
```

The following WML defines a card with several links (shown in Figure 1-4).

```wml
<wml>
  <card>
    <do type="accept" label="More">
      <go href="dl.wml"/>
    </do>
    <p>
      Some links:<br/>
      <anchor title="Link1"><go href="#news"/>News</anchor><br/>
      <anchor title="Link2"><go href="#sports"/>Sports</anchor>
    </p>
  </card>

  <card id="news">
    <p>
      You chose News.
    </p>
  </card>

  <card id="sports">
    <p>
      You chose Sports.
    </p>
  </card>
</wml>
```
Handling User Input

WML supports two kinds of user input: entering text and selecting from a list. The following sections describe how to both prompt the user and handle the response.

Letting Users Enter Text

You use the `<input>` element to prompt users to enter a string of text or numbers. You can control the format. The following is a simplified synopsis of the `<input>` element:

```
<text>
<input name="variable" title="label"
    format="specifier" maxlength="n" emptyok="boolean"/>
```

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The prompt that the phone displays to the user.</td>
</tr>
<tr>
<td>name</td>
<td>Required. The name of the variable in which to store the text entry.</td>
</tr>
<tr>
<td>title</td>
<td>A brief label for the input item (applicable for style=&quot;SET&quot; only).</td>
</tr>
</tbody>
</table>
Introduction to WML
Handling User Input

**IMPORTANT** For a complete synopsis of the `<input>` element and its attributes, see the *WML 1.3 Language Reference*.

For example, the following deck defines two cards. The first card prompts the user to enter a name. Once the name is entered, the ACCEPT key navigates to the second card which displays it (see Figure 1-5).

```wml
<wml>
<card>
  <do type="accept">
    <go href="#disp"/>
  </do>
  <p>
    Enter your name: <br/>
    <input name="username"/>
  </p>
</card>
<card id="disp">
  <p>
    Hi $username!
  </p>
</card>
</wml>
```

**Figure 1-5. Card with user entry field**

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>A character format specifier for the user entry (see “Specifying Text Entry Formats” on page 14). If you omit this attribute, the Mobile Browser lets the user enter characters of any type.</td>
</tr>
<tr>
<td>maxlength</td>
<td>The maximum number of characters the user can enter. If you omit this attribute, the Mobile Browser imposes a limit of 256 characters.</td>
</tr>
<tr>
<td>emptyok</td>
<td>Specifies whether or not the field is optional (the default is FALSE). If the user does enter a value, the device uses the format specifier defined for the field.</td>
</tr>
</tbody>
</table>
The `name` attribute specifies a variable in which the phone stores the text that the user enters. To display the value in the variable at run time, prefix the variable name with a dollar sign ($). For more information about setting and using variables, see “Using Variables” on page 22.

### Specifying Text Entry Formats

You use the `format` attribute to specify the type and case of characters the user can enter. You can use a combination of the following special tags.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Anything except numbers (no symbols or alphanumeric characters)</td>
</tr>
<tr>
<td>a</td>
<td>Anything except numbers (no symbols or alphanumeric characters)</td>
</tr>
<tr>
<td>N</td>
<td>Any numeric or alphabetical character</td>
</tr>
<tr>
<td>X</td>
<td>Anything except numbers or special characters (can be changed to lower case)</td>
</tr>
<tr>
<td>x</td>
<td>Anything except numbers or special characters (can be changed to lower case)</td>
</tr>
<tr>
<td>M</td>
<td>Anything except numbers or special characters (can be changed to lower case) (default first character uppercase)</td>
</tr>
<tr>
<td>m</td>
<td>Anything except numbers or special characters (can be changed to lower case) (default first character lowercase)</td>
</tr>
</tbody>
</table>

For example, specifying `format="NAAA"` requires the user to enter a number followed by exactly three symbols or uppercase alphabetic characters. In contrast, specifying a single-digit number before the character tag limits the number of characters that users can enter without requiring a specific number. For example, specifying `format="N3A"` requires the user to enter a number followed by zero to three symbols or uppercase alphabetic characters.

To let users enter an unlimited number of characters of a particular type, specify an asterisk (*) before the character tag. For example, specifying `format="NN*M"` requires the user to enter two numbers followed by any number of symbols, numbers, or alphabetic characters.

**IMPORTANT** You can only use a number or an asterisk with the last character tag in a format specifier. Specifiers such as "4AN" and "*M2N" are not allowed.

The `M` and `m` format tags set the default capitalization when you precede them with a number or asterisk. The `M` specifier makes the first letter the user enters uppercase by default; the `m` specifier makes it lowercase by default. The user can override the default capitalization in both cases.
Allowing Users to Enter Nothing

If you do not specify the format attribute, or if you specify a format that allows zero or more characters (for example, *A, 5a, or 2N), the user can press ACCEPT and proceed to the next card at any point. However, if you specify a format that requires a minimum number of characters (for example, NAAN or mm*), the Mobile Browser hides the ACCEPT key label (either one that you have specified or the default) until the user enters the last character required by the format specifier. In this case, the user cannot proceed to the next card until he or she has entered all of the required characters.

**NOTE** If the user presses the ACCEPT key while the label is hidden, the Mobile Browser simply performs the default ACCEPT key behavior (invokes a prev task).

To make field entry optional rather than required, specify emptyok="TRUE" in your <input> statement. This attribute lets the user leave the field blank without removing the format requirements if he or she does enter a value.

Adding Automatic Characters

You can use the format attribute to insert uneditable characters in an entry field. This feature is useful for imposing a known format on user-entered data—for example, making sure that phone numbers use parentheses or that dates use slashes (/) instead of dashes. To add automatic characters, insert each character in the desired position in your format specifier, preceding each with a backslash (\). The Mobile Browser automatically inserts the character in the field as the user enters a value.

For example, specifying format="\(NN\)-" instructs the Mobile Browser to insert a left parenthesis before the user enters anything and a right parenthesis and a dash after the user enters two numbers. The following table summarizes this behavior.

<table>
<thead>
<tr>
<th>User action</th>
<th>Text that appears in entry field</th>
</tr>
</thead>
<tbody>
<tr>
<td>No action</td>
<td>(</td>
</tr>
<tr>
<td>User enters a 1</td>
<td>(1</td>
</tr>
<tr>
<td>User enters a 2</td>
<td>(12) -</td>
</tr>
<tr>
<td>User backs up (erases) one digit</td>
<td>1</td>
</tr>
</tbody>
</table>

For example, the following WML generates the card shown in Figure 1-6:

```xml
<wml>
  <card>
    <do type="accept">
      <go href="http://www.foo.com/jkh/ssinfo.cgi?SS=$ssnum"/>
    </do>
    <p>Enter SS#:
      <input name="ssnum" format="NNN-NN-NNNN"/>
    </p>
  </card>
</wml>
```
NOTE The Mobile Browser stores the entire field value, including automatic characters, in the variable specified by the name attribute. Therefore, if you use automatic characters in your format specifier, you should include them in any default value that you specify for the field.

Text Entry Modes

The Mobile Browser supports the following text entry modes: alpha, ALPHA, smart, SMART, NUM, and SYM. When you define an input field, the Mobile Browser automatically reserves the OPTIONS key for toggling between entry modes. If you specify the format attribute, the user can only toggle through the entry modes allowed by the format specifier; otherwise, the user can toggle through all of the entry modes. For more information on text entry modes, see the quick reference, Using the Openwave Mobile Browser Simulator.

For example, suppose that you want the user to enter a code consisting of a lowercase letter and a number. To do this, you could use the following WML:

```wml
<wml>
  <card>
    <do type="accept">
      <go href="#disp"/>
    </do>
    <p>
      Enter the code: <br/>
      <input name="code" format="aN"/>
    </p>
  </card>
  <card id="disp">
    <p>
      The code is $code. <br/>
    </p>
  </card>
</wml>
```

The interface for this deck is shown in Figure 1-7. Note that after the user enters the letter q, the phone automatically changes the entry mode to NUM. After the user enters the number 1 (the final required character), the Mobile Browser displays the OK label above the ACCEPT key.
Figure 1-7. Format specifier and text entry modes

User enters the letter q.

User enters the number 2.

User presses ACCEPT.

The code is q2.
Letting Users Select From a List

The `<select>` element causes the device to prompt the user to select one or more items from a specified list. The following is a simplified synopsis of the `<select>` element:

```
text
  <select title="label" name="variable" ivalue="default">
    <option value="value">content</option>
    <option value="value">content</option>
  ...
</select>
```

**Component** | **Description**
--- | ---
**text** | The prompt the phone displays to the user.
**title** | A brief label for the selected item.
**name** | The name of the variable in which to store the user selections.
**ivalue** | The default items selected if the `name` variable has no current value.
**<option>** | An individual choice item where `content` specifies the label for the item and, optionally, an `<onevent>` statement that associates a task with the item (see “Specifying a Task for a Selection Item” on page 19). The `value` attribute specifies the value that the device adds to the `name` variable when the user selects that item.

**IMPORTANT** For a complete synopsis of the `<select>` element and its attributes, see the *WML 1.3 Language Reference*.

For example, the following WML deck generates the display shown in Figure 1-8:

```
<wml>
  <card>
    <do type="accept">
      <go href="#displaychoice"/>
    </do>
  </card>
  <p>
    Choose a sign:
    <select name="choice" ivalue="2">
      <option value="Avoid making promises.">Capricorn</option>
      <option value="Great things lie ahead.">Pisces</option>
      <option value="New love interest awaits.">Aries</option>
    </select>
  </p>
  </card>
  <card id="displaychoice">
    <p>
      $choice
    </p>
  </card>
</wml>
```
Because the choice variable does not initially have a value, the mobile browser device sets the default selection to the item with the index value specified by ivalue (the second item in this case). When the user selects an option, the mobile browser device stores the value associated with that option to the choice variable.

**Specifying a Task for a Selection Item**

In some cases, you may want the act of choosing a selection item to trigger a particular action. You can accomplish this in WML by associating a task with an *intrinsic event* (or state transition) for the selection item. Specifying the `<onevent>` element within an `<option>` statement binds the specified task (<go>, <prev>, <noop>, or <refresh>) to that selection item.

If the task you want to define is a <go> task that simply navigates to another URL, you can use a shorthand version of the `<onevent>` statement. In this case, you can simply specify the onpick attribute in your `<option>` statement.
For example, suppose that you want to modify the previous example so that one of the options is to enter a sign not shown in the list. To do this, add an \(<\text{option}>\) statement that displays a card with a text entry field when the user selects it (see Figure 1-9).

\[
\begin{verbatim}
<wml>
  <card>
    <do type="accept">
      <go href="#displaychoice"/>
    </do>
    <p>
      Choose a sign:
      <select name="choice" ivalue="2">
        <option value="Avoid making promises.">Capricorn</option>
        <option value="Great things lie ahead.">Pisces</option>
        <option value="New love interest awaits.">Aries</option>
        <option onpick="#entersign">Other sign ...</option>
      </select>
    </p>
  </card>
  <card id="displaychoice">
    <p>
      \$choice
    </p>
  </card>

  <!-- All other signs get the same horoscope -->
  <card id="entersign">
    <do type="accept">
      <go href="#displaychoice">
        <setvar name="choice" value="Expect a visitor."/>
      </go>
    </do>
    <p>
      Enter your sign:
      <input name="sign"/>
    </p>
  </card>
</wml>
\end{verbatim}
\]

**NOTE** The next section explains how to use the \(<\text{setvar}>\) element to pass information when performing a \(<\text{go}>\) task.
As described earlier, the task in this example is a simple <go> statement. In this case, using the onpick attribute is the most efficient way to define this behavior. You could, however, have specified exactly the same behavior using the following full <onevent> syntax:

```
...  
  <option>
    <onevent type="onpick">  
      <go url="#entersign"/>
    </onevent>
    Other sign ...
  </option>
...
```

You must use the <onevent> syntax if you want to do either of the following:

- Specify a more complex <go> task—for example, one that sets variable values or that uses the sendreferer or accept-charset attribute or the postfield element.
- Bind a different task element—for example, <prev>, <noop>, or <refresh>.

**IMPORTANT** For more information about intrinsic events and the <onevent> element, see the *WML 1.3 Language Reference*. 
Using Variables

WML variables are named pieces of information that the mobile browser device substitutes at run time, so that you can write WML that dynamically changes content and navigation based on user input.

The previous section explained how to use variables to capture user entries or selections and then display their values in formatted text. The following sections describe other ways to use variables, including how to set them explicitly and substitute them into WML attributes.

Setting Variables Explicitly

You use the `<setvar>` element to pass a variable when you execute a `<go>`, `<prev>`, or `<refresh>` task. You must specify one `<setvar>` element for each variable that you want to set. The `<setvar>` element has the following form:

```
<setvar name="name" value="value"/>
```

**IMPORTANT** The case of variable names is significant. For example, `variable1`, `Variable1`, and `vaRiable1` are all different variables.

The following deck includes a card that sets the value of two variables when the user presses ACCEPT. A second card displays the variable values, as shown in Figure 1-10.

```wml
<wml>
  <card>
    <do type="accept">
      <go href="#viewvalues">
        <setvar name="var1" value="value 1"/>
        <setvar name="var2" value="value 2"/>
      </go>
    </do>
    <p>
      Press OK to view variable values.
    </p>
  </card>

  <card id="viewvalues">
  <p>
    $var1<br/>
    $var2
  </p>
  </card>
</wml>
```
Using Variables

Initializing Variables

To set or initialize a variable without requiring the user to do anything, bind the \textless refresh\textgreater task to the \texttt{onenterforward} intrinsic event.

The following example initializes the variables shown in Figure 1-11 without requiring the user to press the \texttt{ACCEPT} key:

\begin{verbatim}
<wml>
  <card>
    <onevent type="onenterforward">
      <refresh>
        <setvar name="var1" value="value 1"/>
        <setvar name="var2" value="value 2"/>
      </refresh>
    </onevent>
    <p>
      $var1
      $var2
    </p>
  </card>
</wml>
\end{verbatim}

When the user navigates forward to this deck, the mobile browser device automatically executes the \texttt{refresh} task bound to the \texttt{onenterforward} intrinsic event and initializes the variables, as shown in Figure 1-11.

Figure 1-11. Card that initializes variables using the onenterforward event
Referencing Variables

Much like UNIX shell variables, you can use variables in your WML code in formatted text, URLs, selection items, or default values. To substitute a variable value, reference the variable using the syntax $(myvar).

The Mobile Browser automatically applies URL escaping rules based on the context. However, if you want to override the normal escaping rules for variables in URL strings, you can use the following options to force a particular behavior.

<table>
<thead>
<tr>
<th>Variable reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(myvar:escape)</td>
<td>Forces <em>escaping</em> of symbolic characters</td>
</tr>
<tr>
<td>$(myvar:noescape)</td>
<td>Forces <em>no escaping</em> of symbolic characters</td>
</tr>
<tr>
<td>$(myvar:unescape)</td>
<td>Forces <em>removal of</em> symbolic character escaping</td>
</tr>
</tbody>
</table>

The Mobile Browser ignores these options if you specify them for variables in contexts other than URL strings.

**NOTE** To include a "\$" character in the following attributes, you need to write "\$$"

- `<do type="\$$">`
- `<onevent type="\$$">`
- `<access domain="\$$" path="\$$">`
- `<meta http-equiv="\$$" name="\$$" content="\$$" scheme="\$$">`
- `<go accept-charset="\$$">`
- `<input format="\$$">`
- `<table align="\$$">`

All other attributes are either “vdata” or have such types that "\$" is not legal. In “vdata” attributes you must also escape the "\$".

Returning Data to Your Application

There are two ways to return variable information to your application:

- Specify the method="*post*" and use postfield elements for the <go> task
- Use variables to set arguments in a URL string

It is strongly recommended that you use the first method because it allows the Mobile Access Gateway to perform any necessary data transcoding from the mobile browser device character set to a character set understood by your application. For more information on character sets and internationalization issues, see Chapter 9, “Internationalization.”
Using method="post" and postfield Elements

The following WML lets the user enter a stock symbol and then generates a URL request with the stock symbol specified in an argument (see Figure 1-12).

```wml
<wml>
	<card>
		<do type="accept">
			<go href="http://foo.com/stock.cgi" method="post">
				<postfield name="stock" value="$(stock)"/>
			</go>
		</do>
		<p>
			Enter stock:
			<input name="stock" format="4A"/>
		</p>
	</card>
</wml>
```

Figure 1-12. Returning user entry as postfield

Using Query String Arguments

Suppose that you want to implement the same interface but to return information using URL arguments instead of a `postfield` element. The following WML demonstrates how to do this:

```wml
<wml>
	<card>
		<do type="accept">
			<go href="http://foo.com/stock.cgi?stk=$(stock)"/>
		</do>
		<p>
			Enter stock:
			<input name="stock" format="4A"/>
		</p>
	</card>
</wml>
```
Figure 1-13. Returning a user entry as a URL argument

![Diagram of user entry and URL request]

NOTE As mentioned earlier, the maximum packet size for data that you send to a mobile browser device is 1492 bytes. The same limit also applies to data that you retrieve from the phone.

**Formatting Text**

The following sections describe how to use WML elements to format text displayed on the phone.

**Adding Line Breaks and Paragraphs**

To start a new line, use the `<br/>` element at the point where you want the new line to start.

The following is a synopsis of the `<p>` element, which defines a new paragraph:

```xml
<p align="alignment" mode="wrapmode"/>
```

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>align</td>
<td>Specifies line alignment (either left, right, or center). If you do not specify a value, left alignment is used by default.</td>
</tr>
<tr>
<td>mode</td>
<td>Specifies the text wrapping mode to use (either wrap or nowrap). The initial setting is wrap mode.</td>
</tr>
</tbody>
</table>

NOTE The WML compiler treats carriage return and line feed characters as white space (equivalent to a single space), so inserting those characters in your text has no effect.
Setting Line Wrap Mode

There are two ways in which mobile browser devices can handle text lines that exceed the display width.

- **Wrapping**: The phone spills extra text onto subsequent lines in the display.
- **Horizontal scrolling (also called Times Square scrolling)**: The phone displays the entire text line on one line and scrolls the line from left to right so the user can see the whole line.

The default text wrapping mode is wrap. If you do not specify the mode attribute, the Mobile Browser automatically wraps the text. To use horizontal scrolling instead, specify `<p mode="nowrap"/>` at the point where you want the new mode to start. Once you specify a mode, it applies to all subsequent lines until you reset it. Therefore, if you set the mode to horizontal scrolling and want to change back to text wrapping, you must explicitly specify `<p mode="wrap"/>`.

For example, the following deck displays the card in Figure 1-14:

```
<wml>
  <card>
    <p mode="nowrap">Line 1 scrolls horizontally.</p>
    <p>Line 2 continues to use nowrap until you reset it.</p>
  </card>
</wml>
```

**Figure 1-14. Card that uses horizontal scrolling**

When the Mobile Browser wraps a line of text, it attempts to wrap between words. To force the Mobile Browser to keep two words together, use a nonbreaking space (`&nbsp;`) between them rather than a regular space.

For example, the following deck displays the card shown in Figure 1-15:

```
<wml>
  <card>
    <p>
      The words Hello&nbsp;World must stay together.
    </p>
  </card>
</wml>
```

**Figure 1-15. Card that uses nonbreaking spaces to control word wrapping**
Specifying Text Alignment

The default text alignment mode is left. To specify a different alignment for a particular line, use `<p align="right"/>` or `<p align="center"/>` at the start of the line. Unlike the mode attribute, the align attribute applies only to the current line. You cannot center or right-align text associated with items in a `<select>` list.

Creating Tables

Use the `<table>`, `<td>`, and `<tr>` elements to create tables in WML. You can control the overall alignment of the table, but not the alignment of individual cells. WML tables are similar to HTML tables but with fewer capabilities. See the WML 1.3 Language Reference for syntax information.

Displaying Special Characters

WML reserves the `<`, `>`, `’`, `“`, and `&` characters. To display one of these characters in formatted text, you must specify one of the following character elements.

<table>
<thead>
<tr>
<th>Character</th>
<th>Character element</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td><code>&lt;</code> (less than)</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>&gt;</code> (greater than)</td>
</tr>
<tr>
<td><code>’</code></td>
<td><code>’</code> (apostrophe)</td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td><code>&quot;</code> (quote)</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td><code>&amp;</code> (ampersand)</td>
</tr>
<tr>
<td><code>\</code></td>
<td><code>\</code> (dollar sign)</td>
</tr>
<tr>
<td>\nbsp;</td>
<td>Nonbreaking space</td>
</tr>
<tr>
<td>\shy;</td>
<td>Soft hyphen</td>
</tr>
</tbody>
</table>

**IMPORTANT** The semicolon (`;`) is part of the character element and must be included. If you omit it, the WML compiler generates an error message.

For example, the following code displays the card shown in Figure 1-16:

```xml
<wml>
  <card>
    <p>
      &lt;$amp;P&gt;
    </p>
  </card>
</wml>
```
Figure 1-16. Card that displays special characters

<SP>

OK

IMPORTANT The character elements shown earlier apply to display text only. You do not need to escape these characters (except when using "$" in non-vdata attributes) when they occur as part of your WML code (in other words, when you specify elements and attributes, such as <card id="mycard">). Use normal URL escaping conventions when specifying URL values.
This chapter describes advanced WML features, including the following:

- Using contexts that help you organize and define your service interface
- Defining access control
- Letting users bookmark cards or decks in your service
- Interacting with the Openwave Mobile Browser cache
- Using timers

### Using Contexts to Structure Your Service Interface

Openwave extensions to WML provide nested contexts to help you structure your WML service. Specifically, contexts help you do the following:

- Organize your service user interface so that it is intuitive to the user
- Provide scoping for variables

Contexts correspond to tasks the user wants to carry out; each context consists of one or more cards. A WML service can provide multiple contexts to its users. For example, an email service could provide contexts such as “Create a New Message” and “View Inbox.” The “Create a New Message” context might include cards to choose an email address, enter a subject, and enter the message body.

Figure 2-1 provides a conceptual overview of services and contexts.
The WML examples in Chapter 1 demonstrate simple individual contexts, each of which consists of a single linear series of cards. However, many services need to provide more complex contexts, which contain subcontexts or child contexts. For example, suppose that you are implementing the inbox portion of an email service. The overall context that a user accomplishes with an email inbox is viewing mail. A nested context is reading an individual piece of mail. In some cases, contexts are nested several levels deep. For example, responding to a message could be a nested context within the viewing mail context.

When you design a WML service, you should use an approach similar to the “top-down” design approach popularized by structured programming languages. Start with general contexts and then divide them into progressively smaller child contexts. Continue until you have divided the child contexts into cards. Structuring your service this way yields a tremendous advantage. When you code it in WML, many of the trickier aspects of WML, such as navigation and variable scoping, virtually take care of themselves. If you use contexts correctly, your application user interface will be clean and intuitive.
Implementing Contexts in WML

Chapter 1 provides examples that use the `<go>` and `<prev>` task types to navigate from card to card within a context. To navigate between contexts, you use the following task types.

<table>
<thead>
<tr>
<th>Task type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;spawn&gt;</code></td>
<td>Requests a card, spawning a nested context (pushes the nested context onto the history stack)</td>
</tr>
<tr>
<td><code>&lt;exit&gt;</code></td>
<td>Returns to the context that invoked the current context (pops the current context off history stack)</td>
</tr>
<tr>
<td><code>&lt;throw&gt;</code></td>
<td>Returns to the first context that contains the appropriate <code>&lt;catch&gt;</code> element exception handler.</td>
</tr>
</tbody>
</table>

Important options that you can use with these task types are discussed later in this chapter.

**Important** Although the difference between the `<go>` and `<spawn>` elements appears subtle, it is important. When you request a card (URL) with `<go>`, you are including it in the current context. When you request a card with `<spawn>`, you are effectively starting a new child context.

To implement a context in WML, follow these general guidelines:

- To implement navigation within a context, use the `<go>` and `<prev>` elements.
- To implement navigation to a child context, use the `<spawn>` element to request the first card in the child context.
- To implement navigation from a child context back to the context that called it, use the `<exit>` or `<throw>` element.
Example

Suppose that you want to provide a simple news service that allows the user to get news headlines and stories. Figure 2-2 illustrates the contexts and tasks that the service provides.

Figure 2-2. Contexts and cards for a simple news service
Doing More with WML
Using Contexts to Structure Your Service Interface

Figure 2-3. Implementation of a news service “view story” context

<card title="Headlines">
  <do type="accept">
    <spawn href="$(cardname:noesc)">
      <!-- Catch all exceptions -->
      <catch/>
    </spawn>
  </do>
  <p mode="nowrap">
    Financial News
    <select name="cardname">
      <option value="#n1">Stock Market Up</option>
      <option value="#n2">Fed Raises Interest Rates</option>
      <option value="#n3">Labor Strike Looming</option>
    </select>
  </p>
</card>

<card id="n1" title="Story">
  <do type="accept" label="Skip">
    <go href="#n2"/>
  </do>
  <do type="options" label="Done">
    <exit/>
  </do>
  <p>Stock Market UP...</p>
</card>

<card id="n2" title="Story">
  <do type="accept" label="Skip">
    <go href="#n3"/>
  </do>
  <do type="options" label="Done">
    <exit/>
  </do>
  <p>Fed Raises Interest Rates...</p>
</card>

<card id="n3" title="Story">
  <do type="accept" label="Skip">
    <go href="#n1"/>
  </do>
  <do type="options" label="Done">
    <exit/>
  </do>
  <p>Labor Strike Looming...</p>
</card>
This simple application demonstrates how to implement a list of items, the details for which can be browsed one item after the other. News headlines are presented in a top-level menu (the parent context). After the user selects a headline, he or she can skip to view each news story in succession. When the user presses the Done soft key, the nested context is exited and all news story cards are removed from the history stack. This significantly simplifies navigation for the user, because there is no risk of revisiting any of the news story cards when the PREV key is pressed to leave the news application and return to the calling application (for example, a portal service).

There is no particular correspondence between contexts and WML decks. A deck can contain multiple contexts, and a context can span multiple decks.

**Contexts and the History Stack**

Chapter 1 describes how the phone maintains a history stack of the cards the user visits. That is a simplification of what the phone does; the phone actually maintains a history stack of visited contexts and the cards the user visits within those contexts. Each time the user navigates to a nested context, the phone pushes the current context onto the history stack. When the phone pushes a context onto the stack, it maintains the entire user interface state of the calling context, including its variable values and the cards the user has visited in it. When the user navigates from a nested context back to the context that called it, the phone pops the nested context off the stack, restoring the calling context (and its user interface state) to the top of the stack.

Figure 2-4 illustrates the state of the history stack at each stage of the interface described in the previous section.

**Figure 2-4. History stack for visit to the Openwave web site**

When the user returns to the top-level card, the item that he or she chooses is still chosen. This is because the phone has preserved the state of the context.

Of course, after the user has backed up and the phone has popped a context off the stack, the user interface state of the child context is lost.
Contexts and Variable Scoping

The scope of a WML variable is a context. A context can include many cards, and any card in a context can reference variables in any other card in the context. Until the user invokes a destination that uses the `<spawn>` element, or issues an `<exit/>` or `<throw/>` task to pop the current context, the variables remain in scope.

Handling Context Return Events

There are several ways in which a nested context can allow a user to return to a calling context:

- Use the `<exit>` element in the first card of the context
- Specify an exception with the `<throw>` task
- Specify a `<prev/>` task in the first card of the child context

By default, each of these returns the user to the card in the calling context that invoked the nested context, assuming that there is a `<catch>` element or an `onexit` attribute. In some cases, you might not want to return the user to the card in the calling context that invoked a nested context. For example, if the card provides a list that the nested context modifies, returning directly to it would be not be desirable, because the list would be obsolete.

Fortunately, you can change the default navigation from a nested context back to the context that called it. You can change the navigation by specifying the following WML elements and attributes:

- The `<spawn onexit="NextCard">` element and attribute specifies which card to go to if the child context is exited with the `<exit>` element.
- The `<throw>` element raises an exception, terminates the active child context, and returns to the appropriate `<catch>` element exception handler.
- The `<prev/>` element raises a specific exception named “prev,” terminates the active child context, and returns to the `<catch name="prev">` element exception handler.
Example

Processing an e-commerce or financial transaction is a good example of using return events to pass completion status from a child context back to a parent context. In the bookstore example application that follows, the parent context allows the user to select a book from the store catalog. When the user selects a book, the parent context calls a child context (buy.wml), which implements the purchase transaction (acquiring the user’s credit card information, committing the purchase to the server’s database, and so on). There are three possible events that can occur, depending on how the child context returns.

<table>
<thead>
<tr>
<th>Return event</th>
<th>How the child returns</th>
<th>How the parent context handles the event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit</td>
<td>&lt;exit/&gt;</td>
<td>&lt;spawn href=&quot;buy.wml&quot; onexit=&quot;#confirm&quot;&gt;</td>
</tr>
<tr>
<td>Normal completion</td>
<td></td>
<td>Display the card named “confirm,” which displays the results of the transaction to the user.</td>
</tr>
</tbody>
</table>

**Exception: Abort**

Application-defined exception

<table>
<thead>
<tr>
<th>Exception: Abort</th>
<th>&lt;throw name=&quot;abort&quot;/&gt;</th>
<th>&lt;catch name=&quot;abort&quot; onthrow=&quot;#error&quot;/&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>User presses the prev key from the first card in the child context</td>
<td>Display the card named “error,” which alerts the user that the transaction was cancelled.</td>
<td></td>
</tr>
</tbody>
</table>

**Exception: Prev**

Browser-defined exception

<table>
<thead>
<tr>
<th>Exception: Prev</th>
<th>User presses the prev key from the first card in the child context</th>
<th>&lt;catch name=&quot;prev&quot; onthrow=&quot;#error&quot;/&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display the card named “error,” which alerts the user that the transaction was cancelled.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The result of all three return events is that an implicit <go> task is executed, and the card named in the associated event handler (onexit or onthrow) is pushed onto the history stack of the parent context. When the user presses the accept key after viewing the “confirm” or “error” card, the card is popped from the history stack and the user is taken back to the main “catalog” card.

A key advantage to implementing the transaction as a child context is that the user’s credit card information is managed in a secure and temporary variable space. When the child context exits, the credit card variables are automatically cleared from browser memory, eliminating the risk of other applications reading the variables. In general, nested contexts should always be used to manage private user data.
Passing Information Between Contexts

WML provides several mechanisms for passing information between contexts:

- A context can use the `<setvar>` element to set variable values for a nested context.
- When a context spawns a child context, it can use the `<receive>` element to specify variables that accept return values from the child context; the child context uses the `<send>` element to return values to these variables.

The following sections describe in detail these methods of passing variable information.
Passing Variables to a Nested Context

The `<setvar>` element sets variables by name that can then be used in child contexts. The syntax for the `<setvar>` element is:

```xml
<setvar name="var1" value="val1"/>
<setvar name="var2" value="val2"/>
...
<setvar name="varN" value="valN"/>
```

The child context uses the variable names to reference variables that the calling context specifies in the `<setvar>` element; it must know the names of the variables in order to use them.

**IMPORTANT** When a child context changes the value of the variables passed to it, it does not affect any similarly named variables in the calling context.

Returning Values to a Calling Context

To accept return values from a child context, you specify the `<send>` element in the child context and the `<receive>` element in the calling context.

Each `<send>` element specifies a value to return to the parent context. When the child context exits, the phone uses the values in the `<send>` elements to set the corresponding `<receive>` variables.

The phone assigns the return values by position. That is, it assigns each value in a series of `<send>` elements to the variables that occupy the corresponding position in a series of `<receive>` elements. If the `<send>` element series doesn’t provide a corresponding value for a variable in the `<receive>` elements, the phone clears the variable so that it has no value.

The following table provides some examples of how the phone sets variables in a `<receive>` element series.

<table>
<thead>
<tr>
<th><code>&lt;receive&gt;</code> elements of calling context</th>
<th><code>&lt;send&gt;</code> elements of child context</th>
<th>Variable values after child context returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;receive name=&quot;var1&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;a&quot;/&gt;</code></td>
<td>var1: a</td>
</tr>
<tr>
<td><code>&lt;receive name=&quot;var2&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;b&quot;/&gt;</code></td>
<td>var2: b</td>
</tr>
<tr>
<td><code>&lt;receive name=&quot;var3&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;c&quot;/&gt;</code></td>
<td>var3: c</td>
</tr>
<tr>
<td><code>&lt;receive name=&quot;var1&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;a&quot;/&gt;</code></td>
<td>var1: a</td>
</tr>
<tr>
<td><code>&lt;receive name=&quot;var2&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;b&quot;/&gt;</code></td>
<td>var2: b</td>
</tr>
<tr>
<td><code>&lt;receive name=&quot;var3&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;c&quot;/&gt;</code></td>
<td>var3: (no value)</td>
</tr>
<tr>
<td><code>&lt;receive name=&quot;var1&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;a&quot;/&gt;</code></td>
<td>var1: a</td>
</tr>
<tr>
<td><code>&lt;receive name=&quot;var2&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;b&quot;/&gt;</code></td>
<td>var2: b</td>
</tr>
<tr>
<td><code>&lt;receive name=&quot;var3&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;a&quot;/&gt;</code></td>
<td>var3: (no value)</td>
</tr>
<tr>
<td><code>&lt;receive name=&quot;var1&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;c&quot;/&gt;</code></td>
<td>var1: c</td>
</tr>
<tr>
<td><code>&lt;receive name=&quot;var2&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;b&quot;/&gt;</code></td>
<td>var2: b</td>
</tr>
<tr>
<td><code>&lt;receive name=&quot;var3&quot;/&gt;</code></td>
<td><code>&lt;send value=&quot;a&quot;/&gt;</code></td>
<td>var3: a</td>
</tr>
</tbody>
</table>
Example: An E-Commerce Transaction

In a bookstore application, the parent context allows the user to select a book from the store catalog. When the user selects a book, the associated item code is passed to the child context, so that the value can subsequently be used to query the server-side database. The parent context passes the current value of the local variable named code. In the child context, a local variable named item is initialized with the same value. The variable named code is scoped in the parent context, and the variable named “item” is scoped in the child context:

```
<setvar name="item" value="${code}"/>
```

To improve the usability of the bookstore application, the code is enhanced so that the details of the user’s purchase are passed back to the parent context when the child context exits. The parent context then displays the transaction details in the Results card. When the child context exits, it sends the total cost of the purchase, along with the number of days it will take for the order to ship. The send values are generated dynamically by the buy.cgi script on the server side. Note that the card that executes the <exit> task does so transparently, using an onenterforward event. This allows the buy.cgi script to return to the browser and exit the child context without requiring the user to press any keys.

<table>
<thead>
<tr>
<th>&lt;receive&gt; elements of parent context</th>
<th>&lt;send&gt; elements of child context</th>
<th>Variable values after child context returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;receive name=&quot;cost&quot;/&gt;</td>
<td>&lt;send value=&quot;7.50&quot;/&gt;</td>
<td>cost: 7.50</td>
</tr>
<tr>
<td>&lt;receive name=&quot;ship&quot;/&gt;</td>
<td>&lt;send value=&quot;2-3&quot;/&gt;</td>
<td>ship: 2-3</td>
</tr>
</tbody>
</table>
Figure 2-6. Child context

<table>
<thead>
<tr>
<th>Shop Context</th>
<th>Buy Context</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;wml&gt;</code></td>
<td><code>&lt;card id=&quot;order4&quot; title=&quot;Order&quot;&gt;</code></td>
</tr>
<tr>
<td><code>&lt;card title=&quot;Order&quot;&gt;</code></td>
<td><code>&lt;do type=&quot;accept&quot; label=&quot;Yes&quot;&gt;</code></td>
</tr>
<tr>
<td><code>&lt;do type=&quot;accept&quot;&gt;</code></td>
<td><code>&lt;go href=&quot;buy.cgi&quot;&gt;</code></td>
</tr>
<tr>
<td><code>&lt;go href=&quot;#order2&quot;/&gt;</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>&lt;p&gt;</code></td>
<td><code>&lt;do type=&quot;options&quot; label=&quot;No&quot;&gt;</code></td>
</tr>
<tr>
<td>Credit Card:</td>
<td><code>&lt;throw name=&quot;abort&quot;/&gt;</code></td>
</tr>
<tr>
<td><code>&lt;input name=&quot;credit&quot; format=&quot;NNNNNNNNNN*N&quot;/&gt;</code></td>
<td><code>&lt;/do&gt;</code></td>
</tr>
<tr>
<td><code>&lt;p&gt;</code></td>
<td><code>&lt;p&gt;</code></td>
</tr>
<tr>
<td>Card Expires: [MM-YY]:</td>
<td>Proceed and submit this order?</td>
</tr>
<tr>
<td><code>&lt;input name=&quot;exp&quot; format=&quot;NN\-NN&quot;/&gt;</code></td>
<td><code>&lt;/p&gt;</code></td>
</tr>
<tr>
<td><code>&lt;p&gt;</code></td>
<td><code>&lt;onevent type=&quot;onenterforward&quot;&gt;</code></td>
</tr>
<tr>
<td><code>&lt;card&gt;</code></td>
<td><code>&lt;exit/&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;/onevent&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;/card&gt;</code></td>
</tr>
</tbody>
</table>
Defining Access Control

All WML decks are public by default. Whenever you access a URL, you gain access to any variables it uses, creating a potential security risk. To control this risk, you can define access control rules on a per-deck basis. To specify which decks (in other words, URLs) can access a particular deck, you must specify an `<access>` statement in the deck header. The following is a synopsis of the `<access>` element:

```xml
<wml>
  <head>
    <access domain="domain" path="path"/>
  </head>
...  
</wml>
```

### Using the Domain and Path Attributes

The default access control settings allow all decks (URLs) in the same domain to access your deck. You generally do not need to change the default settings unless:

- You want to restrict access to your deck to URLs in a particular domain or path.
- You want to navigate between decks in different domains.
- You want to set an entry point for your application (for example, home/index). In this case, you must make your deck bookmarkable (see “Letting Users Bookmark Your Service” on page 46).

When you specify values for these attributes, the Mobile Browser compares those values to the URL of the requesting deck. The `domain` and `path` attributes are inclusive, meaning that the requesting deck must match both values in order to access the deck.

**NOTE** You cannot specify multiple values for either the `domain` or the `path` attribute. You must specify `domain=".com"` if you want to grant access to more than one domain (for example, both `sun.com` and `microsoft.com`). Similarly, you must specify `path="/"` to grant access to both the `/foo` and `/bar` directories.
The following table summarizes the effects of specifying different values for the `domain` and `path` attributes for a deck located at http://my.com/deck.wml.

<table>
<thead>
<tr>
<th>DOMAIN</th>
<th>PATH</th>
<th>URLs that can access deck.wml</th>
<th>URLs that cannot access deck.wml</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>default</td>
<td>Examples</td>
<td>Examples</td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="http://my.com/dir/d3.wml">http://my.com/dir/d3.wml</a></td>
<td></td>
</tr>
<tr>
<td>foo.com</td>
<td>default</td>
<td>Examples</td>
<td>Examples</td>
</tr>
<tr>
<td><a href="http://www.foo.com">www.foo.com</a></td>
<td>default</td>
<td>Examples</td>
<td>Examples</td>
</tr>
<tr>
<td></td>
<td>/dir</td>
<td>Examples</td>
<td>Examples</td>
</tr>
<tr>
<td>foo.com</td>
<td>/dir</td>
<td>Examples</td>
<td>Examples</td>
</tr>
</tbody>
</table>

When you specify a `domain` value, the Mobile Browser performs a literal string match between the domain of the requesting deck and that value—it does not resolve names or aliases into canonical domain names. For example, if you specify `domain="204.163.167.193"` for your deck, a deck in the domain `devgate2.openwave.com` will not have access to your deck, even if `devgate2.openwave.com` is an alias for `204.163.167.193`. In this case, the Mobile Browser must have loaded the requesting deck from the URL `http://204.163.167.193/path` in order for it to access your deck.
Navigating to Other Decks

As illustrated in Figure 2-7, you use the default access control settings to navigate from your deck to any other URL in the same domain.

Figure 2-7. Navigating between decks in the same domain

If you try to navigate to a deck in a different domain, the Mobile Browser displays an error message (see Figure 2-8).

Figure 2-8. Error navigating between decks in different domains

NOTE Pressing the OPTIONS key displays the message Access Control Error.

In order for `deck1.wml` to access `deck2.wml` in this example, you must use the domain attribute to grant access to `mydomain.com` (as shown in Figure 2-9).
Letting Users Bookmark Your Service

Mobile Browser bookmarks are similar to conventional web browser bookmarks. When a user bookmarks a card, the Mobile Browser creates a bookmark that consists of two items:

- A label that identifies the bookmark. This label comes from the `title` attribute in the `<card>` definition. If you do not specify a card title, the Mobile Browser uses the first line of display text as the bookmark label.

- The URL to open when the user selects the bookmark.

All decks default to markable, `content="true"`. The following example illustrates how to set a deck so that it is not bookmarkable:

```xml
<wml>
  <head>
    <meta content="false" name="vnd.up.markable" forua="true"/>
  </head>

  <card title="Can’t Bookmark Me">
    ...
    ...
  </card>
</wml>
```

When a user bookmarks a card, the Mobile Browser automatically sets the bookmark URL to the URL for the deck. If you want to use a different URL, you can specify a `<meta>` element (of type `vnd.up.bookmark`) in the deck header using the following syntax:

```xml
<meta name="vnd.up.bookmark" content="url"/>
```

where `url` is the URL that you want to use for the bookmark.
IMPORTANT Because the UP.Apps bookmark application resides on the Mobile Access Gateway, you must use the Openwave Mobile Browser Simulator in Mobile Access Gateway mode to test your bookmarks. For more information on using the Openwave Mobile Browser Simulator, see the Tools and API Reference.

Interacting with the Mobile Browser Cache

Like conventional web browsers, the Mobile Browser has a memory cache that caches decks the user has visited most recently. If the user requests a cached deck, the browser can quickly redisplay a deck without requesting it again from the web server. When the cache exceeds the mobile browser device memory capacity, the device drops the least recently visited decks.

When a user navigates to a deck that exists in the cache, the Mobile Browser determines whether to retrieve the deck from cache or to request it again from the server by comparing the deck time to live (TTL) to the elapsed time since last requesting the deck. If the elapsed time is greater than the specified cache life, the Mobile Browser requests the deck from the server. Otherwise, it simply displays the cached deck.

The default Mobile Browser TTL is 30 days (or until memory is exhausted). If a deck contains time-sensitive information, you can specify a shorter TTL so that the device will reload the deck from the server more frequently. You can set the TTL for a deck by including a <meta> statement in the deck <head> element:

```html
<wml>
<head>
  <meta http-equiv="Cache-Control" content="max-age=time" forua="true" />
</head>
...
</wml>
```

where max-age specifies the time (in seconds) to cache the deck. For example, to drop a deck from the cache after 1 hour, you would specify max-age=3600 (3600 seconds). For extremely time-sensitive information, you can set max-age=0 to force the Mobile Browser to always retrieve the deck from the server. You should use this setting sparingly, because retrieving information from the server always takes longer (incurs a larger latency) than retrieving it from cache.

Using the Openwave SDK to View the Cache Status

Because the Openwave SDK emulates the caching behavior of real mobile browser devices, it can help you tune the performance of your application. You can view cache status in the Openwave SDK by opening the Browser Output window. As you navigate from URL to URL, the Browser Output displays messages indicating whether the URLs
are retrieved from cache or from the web server. For example, the Browser Output window might print the following:

- **Cache Miss**: http://www.somesite.com/index.wml
- **Net Request**: http://www.somesite.com/index.wml
- **HTTP GET**: http://www.somesite.com/index.wml
- **NETWORK REPLY FILE SIZE**: 2666 bytes
- **NETWORK REPLY HTTP STATUS**: 200
- **NETWORK CONTENT TYPE**: text/vnd.wap.wml
- **NETWORK FILE SIZE COMPILED**: 1027 bytes

The lines following the miss statement indicate the details of downloading the file. If the file is found in the cache, the Browser Output window displays the following and the download information is not displayed because no download is necessary:

- **Cache HIT**: http://www.somesite.com/index.wml

---

**Using Timers**

You use the `<timer>` element to invoke a task automatically after some period of user inactivity. Timers have a number of useful applications, including the following:

- Displaying a splash screen for 5 seconds before advancing to the next card
- Automatically updating a stock quote every 15 minutes
- Limiting the time that sensitive data is displayed on the screen

The following is a simplified synopsis of the `<timer>` element:

```xml
<timer default="value"/>
```

You must specify default values in units of 1/10 seconds; for example, a value of 100 equals 10 seconds. Specifying a value of 0 disables the timer.

The following example displays the text **Acme Inventory Listing** for 5 seconds. When the timer expires, the device opens the URL specified by the `ontimer` attribute, which in this case executes the `Inventory.cgi` script.

```xml
<wml>
  <card id="splash" ontimer="Inventory.cgi">
    <timer value="50"/>
    <p>
      Acme Inventory Listing
    </p>
  </card>
</wml>
```

**NOTE** For a complete synopsis of the `<timer>` element and its attributes, see the *WML 1.3 Language Reference*. 
This chapter describes how to create dynamic WML services. Specifically, it describes how to:

- Generate decks and digests dynamically
- Invalidate decks in the mobile browser device cache to force the mobile browser device to reload them
- Initiate voice calls using Wireless Telephony Application Interface (WTAI)
- Extract mobile browser device, user, and Mobile Access Gateway information from HTTP requests
- Request user preference information, such as the user’s time zone preference

Creating Dynamic WML Services

The examples so far have focused on using WML decks. Static decks are useful for providing menus and nonvolatile information. To generate content dynamically, however, you need to implement your service as a CGI or web server application. Doing so lets you not only deliver dynamic content but also broadcast alerts and modify the mobile browser device cache.

To create a dynamic WML service, you must do the following:

- Generate message entities (MIME types) acceptable to the UP.Link Server
- Use the appropriate HTTP post and response headers
- Identify your service to the UP.Link Server

The following sections discuss these topics in detail.
Valid Content Types

The following table lists the valid content types that your service can send to an UP.Link Server.

<table>
<thead>
<tr>
<th>Content</th>
<th>MIME type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WML</td>
<td>text/vnd.wap.wml</td>
<td>A WML deck</td>
</tr>
<tr>
<td>WMLScript</td>
<td>text/vnd.wap.wmlscript</td>
<td>A WMLScript deck</td>
</tr>
<tr>
<td>Alert</td>
<td>application/x-up-alert</td>
<td>A command that instructs the mobile browser device to beep or display a visual signal to notify the subscriber that information has been pushed to the phone</td>
</tr>
<tr>
<td>Cache operation</td>
<td>application/x-up-cacheop</td>
<td>A command that invalidates an individual deck or an entire service in the mobile browser device cache</td>
</tr>
<tr>
<td>Digest</td>
<td>multipart/mixed</td>
<td>A combination message containing at least one WML deck and zero or more content entities of any of the other types just listed</td>
</tr>
</tbody>
</table>

These content types all leverage the MIME message format. Each entity consists of a header and a body as specified in:

ftp://ftp.isi.edu/in-notes/rfc822.txt
ftp://ftp.isi.edu/in-notes/rfc1521.txt

The digest format, in particular, is simply a standard MIME multipart/mixed message.

**NOTE** You can optionally specify a character set when you use the WML and alert content types. See “Character Set Names” on page 111 for a list of valid character sets. The UPSDK utilities for generating message entities and digests do not set a character set by default, but you can override this setting.
For example, the following is a valid HTTP response containing a single uncompiled WML deck:

```
Content-type: text/vnd.wap.wml

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.3//EN"
 "http://www.wapforum.org/DTD/wml13.dtd">
<wml>
<card>
<p>
Hello, Openwave World!
</p>
</card>
</wml>
```

**IMPORTANT** The UP.Link Server requires that you include at least one blank line between the content type and the rest of your WML deck (as shown earlier).

**Tuning Performance with Digests**

The UP.Link platform supports a multipart MIME format, called a *digest*, which you can use to send one or more entities—that is, WML decks and the other content types listed in “Valid Content Types” on page 50—in a single message to the UP.Link Server. This enables you to optimize use of the wireless network and make your service’s user interface appear more responsive to the user.

If you know that the user will request multiple entities, you can send them all in a single response, instead of sending them individually as the user requests them. Because each HTTP request-response cycle involves a minimum time overhead regardless of the amount of data transmitted, sending a single response is normally much more efficient than sending multiple responses.

**IMPORTANT** Because dynamic WML services typically generate responses of varying length, you should limit the size of your digests to no more than 1200 bytes to allow adequate headroom. However, Openwave recommends that developers keep all decks to under 500 bytes to minimize the download-latency and improve the end-user experience. Content that exceeds 500 bytes (for example, email messages) should be segmented across multiple linked decks. Refer to the News example in the Openwave UP.SDK examples/wml directory. To view the size of a compiled digest, load it in the Openwave Mobile Browser Simulator, which displays the digest size in the Browser Output window.
UP.Link Digest Format

The digest follows the standard multipart MIME format. The following is a high-level view of the format:

```
Digest header
Boundary
Content entity
Boundary
Content entity
Boundary
...
```

**IMPORTANT** The order of the content entities is significant; the mobile browser device interprets the entities according to their order in the digest.

Each content entity must include the required `Content-type` header. To be referenced by another entity, a content entity must also include a `Content-location` header. This header must specify a URL relative to the digest URL.

For example, Figure 3-1 illustrates a simple digest containing two WML decks. When the user navigates from `deck1` to `deck2`, the mobile browser device does not need to issue another request to the UP.Link Server. It simply retrieves it from the digest, which is already in memory. This improves performance when the user navigates between the decks.

**Figure 3-1. A digest containing two decks**

```
Content-length: 402
Content-type: multipart/mixed;boundary="-0"
--0
Content-location: ?NS=deck1
Content-length: 108
Content-type: text/vnd.wap.wml

<wml>
<card>
<do type="accept">
<go href="?ns=deck2"/>
</do>
<p>
This is a deck
</p>
</card>
</wml>

--0
Content-location: ?ns=deck2
Content-length: 114
Content-type: text/vnd.wap.wml

<wml>
<card>
<do type="accept">
<go href="?ns=deck1"/>
</do>
<p>
This is another deck
</p>
</card>
</wml>
```
Modifying the Mobile Browser Device Cache

You can use a cache control `<meta>` tag to specify how long a mobile browser device should cache a deck, called the deck time to live (TTL) (see “Interacting with the Mobile Browser Cache” on page 47). Specifying a TTL, however, does not guarantee that the mobile browser device will reload a deck from the server—for example, if the user backs up to a cached deck, the mobile browser device does not reload it, even if its TTL has expired.

**NOTE** Specifying the `must-revalidate` option for the `<meta>` element’s `content` attribute forces the browser to revalidate the deck TTL when navigating to the deck in the backward direction (which it never does by default).

With the UP.Link platform, you can push cache operations to a mobile browser device to invalidate stale URLs in cache and force the device to reload information from the server. If the user is likely to navigate to a deck that you are invalidating, you can package the cache operation together with the replacement deck in a digest. When the user navigates to that URL, the replacement deck is already in cache. You can also invalidate all decks from your service with a single cache operation.

A common way to execute cache operations is to send them asynchronously. This is an efficient way to update timely information “behind the scenes,” without explicit user intervention. For information on this process, see “How WML Services Can Use Notifications” on page 66.

Initiating Voice Calls

In addition to specifying the WML language itself, the WAP Forum has also specified an application interface for initiating voice calls from a WML service. This interface, called the Wireless Telephony Application Interface, consists of a function library that resides on the mobile browser device. The WAP Forum provides a detailed WTAI specification at:

```
http://www.wapforum.org
```

The following example illustrates how to initiate a voice call in a WML deck by using a `<go>` task that specifies the WTAI interface in the URL:

```xml
<wml>
  <card>
    <do type="options" label="Call">
      <go href="wtai://wp/mc;6508171600"/>
    </do>
    <p>Please call Openwave</p>
  </card>
</wml>
```
When the user presses the OPTIONS key, the Mobile Browser switches the phone to voice mode and dials the number you specify in the URL (650-817-1600 in this example). In some cases, the phone may prompt the user for a dialing preference before dialing the number. When the user hangs up, the Mobile Browser returns to the same card that initiated the call.
With the Wireless Markup Language (WML), you can specify graphic images in formatted text. Using graphic images can improve the look and feel of your WML application. This chapter describes how to implement images in WML.

Not all mobile browser devices currently display images. However, you can send WML that contains images to devices that do not support them.

#### Using Images in WML

To include an image anywhere in formatted text, use the `<img>` element. The image appears wherever you place the `<img>` statement in the text.

The following shows a simplified synopsis for the `<img>` element:

```html
<img alt="text" src="url" localsrc="icon"/>
```

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt</td>
<td>Specifies the text to display if the device does not support images or cannot find the specified image.</td>
</tr>
<tr>
<td>src</td>
<td>The URL of the image to display. If you specify a valid icon for the localsrc attribute, the device ignores this attribute.</td>
</tr>
<tr>
<td>localsrc</td>
<td>The name of a known icon. If the device cannot find the icon in read-only memory (ROM), it attempts to retrieve it from the Mobile Access Gateway. If you specify a valid icon (see the WML 1.3 Language Reference for a list of icon names), the device ignores the src attribute.</td>
</tr>
</tbody>
</table>

**IMPORTANT** For a complete synopsis of the `<img>` element and its attributes, see the WML 1.3 Language Reference.
The following simple example uses an application-specific image:

```xml
<wml>
  <card id="OpenwaveHome">
    <p>
      <img alt="Openwave" src="../images/logo.bmp"/>
      Welcome!<br/>
    </p>
  </card>
</wml>
```

The Openwave Mobile Browser supports both the WMP format and 1-bit BMP image format. It is up to the WML service to ensure that images fit on the device display.

### Images in Static Decks

The following WML displays multiple local icons in a static deck (see Figure 4-1).

```xml
<wml>
  <card>
    <do type="accept">
      <go href="#outlook"/>
    </do>
    <p>
      Welcome to Llewellyn Weather!<br/>
      Click OK to see the 3-day outlook.
    </p>
  </card>

  <card id="outlook">
    <p>
      <img alt="Sunny" src="" localsrc="sun"/>
      <img alt="Partly Cloudy" src="" localsrc="partcloudy"/>
      <img alt="Rain" src="" localsrc="rain"/>
    </p>
  </card>
</wml>
```
Using Images Efficiently

Because images are specified by their own URLs, loading a deck that uses them requires multiple web server requests—one for the WML deck itself and additional requests for each image. Each image that you use increases the latency period between the time the user navigates to the deck and the time the Mobile Browser displays it. Packaging a deck and its associated image files into a digest significantly improves user response time because the device can retrieve all the files in a single network request.

The only disadvantage to this approach is that the Mobile Browser must refetch all images in the digest every time it has to reload a deck in the digest, regardless of whether the images already exist in cache.

**NOTE** Like all digests, those that include images must be less than 1500 bytes\(^1\) (UP.Link 3.x) or 2000 bytes (UP.Link 4.x or later) after compilation. Even if you do not package an image file in a digest, the Mobile Access Gateway performs this packaging automatically before passing it to the device. As a result, the Mobile Browser rejects image files that are larger than the maximum digest size.

---

1. Compilation normally decreases the size of a digest. However, it may *increase* the size if the digest contains text with many words and spaces, because the Mobile Access Gateway tokenizes each word as a null-terminated string and each space as a byte code.
The conventional web model is a synchronous model, in which the user iteratively requests information, which web servers supply. This model requires users to actively search for information that is important to them. Another Internet interaction model, the asynchronous model (also called the push model), has become increasingly popular. Servers that use the asynchronous model do not wait until a user requests information. Instead, they use a profile to determine which information is important to the user and asynchronously send the information to the user as soon as it becomes available.

For more information on push, see the Push Library Developer's Guide.

The Mobile Access Gateway platform fully supports this model of Internet interaction, allowing WML services to send information to Mobile Access Gateway subscribers asynchronously. The asynchronous messages that WML services send to Mobile Access Gateway subscribers are called notifications. You can use notifications for tasks such as the following:

- Alerting subscribers to security price changes or trade fulfillments
- Providing timely warnings of hazardous weather or traffic conditions
- Updating sports scores in which the subscriber is interested
- Ensuring that obsolete data from your service is removed from the mobile browser device cache

The rest of this chapter describes how to create WML services that send notifications. It also provides instructions for testing notifications with the Openwave Mobile Browser Simulator.

**NOTE** To ensure a consistent user interface across WML services, you should adhere to the user interface guidelines for notifications described in this chapter. For maximum usability, see also the complete list of UI guidelines in Chapter 8.

This chapter assumes a basic knowledge of Secure Sockets Layer (SSL), certificates (or digital IDs), and certificate authorities. If you are not familiar with these concepts, you can find background information at the following web sites:

- http://www.verisign.com
Notification Content Types

A WML service can send notifications containing one or more of the following entities:

- **Alert**: A command that instructs the mobile browser device to beep or display a visual signal to notify the subscriber that new information is available
- **Cache operation**: A command that removes an individual URL, or all the URLs for a service, from the mobile browser device cache
- **Decks**
- **Images**
- **Digests**

The deck, image, and digest content types are described in “Valid Content Types” on page 50. The following sections describe the alert and cache operation content types in detail.

**Alerts**

From the user’s perspective, the most obvious notification content type is an alert. An alert typically includes a short message (or title) and a URL that the user can request. When a WML service sends an alert, the mobile browser device can do any of the following, depending on the urgency specified:

- Beep, buzz, or flash a light, indicating that the alert has arrived (these features are not available on all mobile browser device models)
- Display a pop-up message card (shown in Figure 5-1) to inform the user that an alert has arrived and allow him or her to view it or skip it
- Display an alert icon, as shown in Figure 5-1

The user can configure the mobile browser device to enable or disable each of these indicators.

**Figure 5-1. Pop-up message from the UP.Mail service**

 Displays URL sent with the alert

 Skips alert and lets users continue what they were doing

**Message from "UP.Mail". View it now?**

**ACCEPT**

**Messages:**

1>12:11p Presentin

**OK**

**Menu**

WML 1.3 Developer’s Guide
In addition to providing the alert indication just described, the mobile browser device adds each alert it receives to its Inbox card. This card is normally accessible from the user’s Home menu, shown in Figure 5-2.

Users can navigate to the Inbox card at any time to view the alerts they have received most recently. When the user chooses the alert in the Inbox, the mobile browser device requests the URL provided by the alert.

Figure 5-2. Inbox card displaying UP.Mail and UP.Market alerts

NOTE Most devices running the Openwave Mobile Browser 5.0 reserve approximately 1500 bytes for storing nine alert titles and associated URLs in the Inbox. Therefore you should limit your alert information (title and URL) to a combined size of 160 bytes. You should also keep your alert title as short as possible so that users can read it without scrolling.

Cache Operations

Mobile browser device caching can greatly improve the performance perceived by the user. When a user visits a URL, the mobile browser device caches it. The next time the user requests the URL, the mobile browser device retrieves it directly from cache instead of re-requesting it from the Mobile Access Gateway. See “Interacting with the Mobile Browser Cache” on page 47 for more information.

You can control when a URL expires from the cache by including a `<meta>` element in the deck header that specifies the time to live (TTL). In some cases, however, you may want to explicitly remove a URL from the cache—for example, if it becomes obsolete before its TTL expires. To remove a URL from the cache, you must send a cache operation to the device. Cache operations can instruct the mobile browser device to remove individual URLs or all URLs for your service from the cache.

As described in “Modifying the Mobile Browser Device Cache” on page 53, you can include a cache operation in a digest along with WML content. To remove a URL asynchronously “behind the scenes” (in other words, without waiting for the user to request the digest), send a notification that contains a cache operation.
Sending Notifications
How the Mobile Access Gateway Delivers Notifications

How the Mobile Access Gateway Delivers Notifications

The Mobile Access Gateway platform supports several wireless network types, each of which operates differently. Packet-switched networks, such as CDPD, treat all data transmissions the same, regardless of their size. Circuit-switched networks, such as GSM, use a special method, Short Message Service, to deliver very short asynchronous transmissions. They also require the user to explicitly authorize larger data transmissions. Thus the Mobile Access Gateway cannot push large messages to mobile browser devices on circuit-switched networks asynchronously.

To support notifications consistently across different network types, the Mobile Access Gateway platform provides two logical delivery channels for notifications:

- The push channel: WML services can use the push channel to push notifications asynchronously to any mobile browser device over either packet-switched or circuit-switched networks. The Mobile Access Gateway attempts to deliver push channel notifications immediately. If the mobile browser device is not available, the Mobile Access Gateway periodically attempts to deliver the notification. You should include only alert and cache operation content types in push channel notifications.

- The pull channel: WML services normally use the pull channel to send notifications that contain less time-sensitive information. You can include any Mobile Access Gateway-supported content type in a pull-channel notification. You can’t count on the Mobile Access Gateway delivering a pull notification immediately.

On packet-switched networks, the Mobile Access Gateway attempts to deliver pull notifications as quickly as possible, with only mobile browser device availability, the network transport layer, and web server latency impeding delivery time. On circuit-switched networks, the Mobile Access Gateway never initiates contact to deliver a pull channel notification. The mobile browser device must open a circuit and initiate contact with the Mobile Access Gateway before the Mobile Access Gateway delivers the notification.

The following sections describes in detail how the Mobile Access Gateway handles push and pull channel notifications.
Push Channel Notifications

Figure 5-3 illustrates a simple push notification transaction.

Figure 5-3. Push channel notification

1 The WML service posts the notification to the Mobile Access Gateway. The notification specifies the subscriber ID of the mobile browser device to which the notification is directed, and a TTL, specifying how long the Mobile Access Gateway should attempt to deliver the notification.

2 The Mobile Access Gateway issues the notification to the mobile browser device. If the notification contains a cache operation, the mobile browser device removes the specified URL from its cache. If the notification contains an alert, the mobile browser device signals the user that an alert has arrived and adds it to the Inbox card (see Figure 5-2).

3 When the user chooses the alert, the mobile browser device requests the specified URL.

4 The Mobile Access Gateway relays the request to the WML service.

5 The service returns the content for the URL.

6 The Mobile Access Gateway relays the content to the mobile browser device.

NOTE The component of the WML service that initiates the notification (the notifier) and the component that provides the URL specified by the alert do not have to be integrated—in fact, they can be on different systems.

For detailed information on push, see the Push Library Developer’s Guide.
Pull Channel Notifications

As mentioned earlier, pull notification delivery depends on the type of wireless network the subscriber is on. The figures in this section show how the Mobile Access Gateway delivers pull notifications over circuit-switched and packet-switched networks.

Pull Notifications on Circuit-Switched Networks

Figure 5-4 illustrates a typical pull notification delivered over a circuit-switched network. The steps are described after the figure.

Figure 5-4. Pull notification over a circuit-switched network

1 The WML service posts a pull notification to the Mobile Access Gateway. The notification specifies the following: the subscriber ID of the mobile browser device to which the notification is directed; a time-to-live (TTL), specifying how long the Mobile Access Gateway should attempt to deliver the notification; and a URL that supplies the notification data.

2 At some later time, the user turns on the mobile browser device and opens a circuit to the network. The mobile browser device automatically requests any pending notifications from the Mobile Access Gateway.

3 The Mobile Access Gateway requests the specified URL from the WML service.

4 The service returns the data to the Mobile Access Gateway.

5 The Mobile Access Gateway relays the data (in the form of a compiled digest) to the mobile browser device. The digest can contain an alert entry, which instructs the mobile browser device to notify the user that new information has arrived.
Sending Notifications
How the Mobile Access Gateway Delivers Notifications

Pull Notifications on Packet-Switched Networks

Figure 5-5 illustrates a typical pull notification delivered over a packet-switched network.

Figure 5-5. Pull notification over a packet-switched network

1. The WML service posts a pull notification to the Mobile Access Gateway.
2. The Mobile Access Gateway requests the specified URL.
3. The service returns the data to the Mobile Access Gateway.
4. The Mobile Access Gateway relays the data (in the form of a compiled digest) to the mobile browser device. The digest can contain an alert entry, which instructs the mobile browser device to notify the user that new information has arrived.

How the Mobile Access Gateway Queues Notifications

When you send a notification, the Mobile Access Gateway adds it to the notification queue. How long the notification remains in the queue depends on the following:

- If the notification is a push notification, or a pull notifications for a device on a packet-switched network, the Mobile Access Gateway attempts to deliver it immediately. If the device is not available, the Mobile Access Gateway keeps the notification in queue and periodically reattempts delivery.
- If the notification is a pull notification for a device on a circuit-switched network, the Mobile Access Gateway queues it until the subscriber opens a circuit.
- If the notification TTL has expired, the Mobile Access Gateway removes it from the queue.
A queued notification that has not been delivered is called a *pending* notification. Your service can delete or request status for any pending notification that it has originated. You can also request status for completed notifications.

The Mobile Access Gateway and the mobile browser device identify notifications by their URLs and subscriber IDs. When the Mobile Access Gateway receives a new notification with a URL and a subscriber ID that match those of a pending notification, it replaces the pending notification with the new notification. This process is known as *coalescing*. It allows WML services to update out-of-date notifications before they are sent to a user’s mobile browser device. The mobile browser device also coalesces the notifications that it receives from the Mobile Access Gateway.

### How WML Services Can Use Notifications

The following table lists some common applications for notifications and recommendations on how to implement them.

<table>
<thead>
<tr>
<th>Application</th>
<th>Channel</th>
<th>Content types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inform subscribers of timely and volatile information (for example, changes in stock prices)</td>
<td>Push</td>
<td>Alert, Cache Operation (optional)</td>
<td>A push notification containing an alert that specifies a URL with the latest stock price. If the alert URL may already exist in the mobile browser device cache, you should include a Cache Operation in the notification to invalidate it.</td>
</tr>
<tr>
<td>Ensure that obsolete information, such as a changed directory entry, is removed from the cache</td>
<td>Push</td>
<td>Cache Operation</td>
<td>A push notification containing a cache operation that removes the specified URL from the cache. The next time the user navigates to the URL, the mobile browser device requests it from the web server, ensuring that the URL is up to date.</td>
</tr>
<tr>
<td>Update timely but nonvolatile information, such as sports scores, without interrupting users</td>
<td>Pull</td>
<td>Cache Operation, WML deck</td>
<td>A pull notification containing a WML deck and a cache operation that invalidates any older version of the deck. The next time the user navigates to the deck, it will already be updated so the user won’t need to request it again.</td>
</tr>
</tbody>
</table>

**NOTE** This table lists some common ways to use notifications, but you can use various permutations of content types and delivery channels to meet your specific needs.
Sending Notifications

To send notifications, follow these general steps, which are described in detail later in this chapter.

1. Get the subscriber IDs of the Openwave Mobile Access Gateway subscribers to whom you would like to send notifications.

   To send notifications to a subscriber, you must know the subscriber ID. To get the ID, you can check the HTTP headers for requests from the mobile browser device or ask the subscriber or Mobile Access Gateway administrator.

   If you have a mobile browser device or have registered your Openwave Mobile Browser Simulator on an Mobile Access Gateway, you can view your subscriber ID by choosing Settings > Advanced > Subscriber ID. Note that the Settings option may or may not be on the Home card for a particular Mobile Access Gateway. You can find it under Utilities on the default devgate2 Home card.

2. Make sure that your corporate firewall allows you to send notifications.

   Make sure that your corporate firewall allows you to send packets to the notification port you will use (TCP destination port 3356 for secure notifications or TCP destination port 4445 for nonsecure notifications).

3. Use one of the SDK tools and libraries to send notifications.

   See the Tools and API Reference.

4. Use the SDK tools and libraries to check the status of notifications you have sent.

   See the Tools and API Reference.

5. Make sure that your application complies with the user interface guidelines for notifications (see Notifications).

Combining Alerts and WML Content in Pull Notifications

Pull notifications that contain both an alert and WML content give users the perception of extremely responsive performance. Because your information is preloaded, the Mobile Browser displays it as soon as the user selects the alert without requiring another network exchange. Some possible disadvantages of this approach are the following:

- If the mobile browser device is on a circuit-switched network, the Mobile Access Gateway won’t deliver the notification until the subscriber turns the mobile browser device on and establishes a circuit.

- The information that the phone preloads and caches may become obsolete by the time the user selects the alert.

For this reason, you should use push notifications for highly volatile information such as stock quotes. Suppose that you want to implement a notification that informs the user of a timely sports headline (see Figure 5-6).
To create this interface, you use two components (or applications):

- One application sends a pull notification that specifies the URL of a digest, which the mobile browser device preloads.
- The other application provides the digest specified by the pull notification. The digest includes a single WML deck, which provides the content, and an alert.

## Debugging Notifications

Following are the general steps for debugging notifications:

1. Make sure that the expiration time (TTL) for the notification is long enough.
   
   The expiration time is specified in seconds. Specify a TTL of 0 to set the longest TTL allowed by the Mobile Access Gateway.

2. If you are sending secure notifications, make sure that the notification URLs match the URL specified by your certificate.

   The Common-Name of the certificate must match exactly (or be an immediate subdomain of) the domain that hosts the notification alert and pull URLs.

3. Check the HTTP response code returned by the Mobile Access Gateway.

   When you post a notification request, the Mobile Access Gateway generates a response with an HTTP code, indicating whether the Mobile Access Gateway accepted or rejected the notification.

   For a description of status codes returned by the Mobile Access Gateway, see “Checking Notification Status” on page 70.

4. Request notification status information from the Mobile Access Gateway.

   SDK notification utilities and APIs allow you to request information on the status of individual notifications—you can determine whether a notification has been delivered, expired, or deemed undeliverable.
5 Use the web interface to check notification status on the Mobile Access Gateway.

If you have login access to the Mobile Access Gateway, you can also use a web interface to check the status of notifications for subscribers you have provisioned. For more information, see “Checking Notification Status” on page 70.

Checking Return Codes

When the Mobile Access Gateway receives a notification request, it returns one of the HTTP status codes shown in the following table. The mechanism for obtaining the status code depends on the tool or API that you use to send notification requests.

**IMPORTANT** If a notification call returns a -1 error value, a general I/O failure has occurred. The most common cause of this error is a corporate firewall blocking the destination notification port.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>The notification status request was accepted by the Mobile Access Gateway.</td>
</tr>
</tbody>
</table>
| 204  | The notification addition request succeeded: The notification was added by the Mobile Access Gateway.  
The notification deletion request succeeded: The notification was deleted by the Mobile Access Gateway. |
| 400  | The notification failed because one of the following was missing or invalid:  
• Subscriber ID: The subscriber ID must be valid.  
• URL: The URL must be valid and accessible to the Mobile Access Gateway. If you are sending a secure notification, it must match the URL of your certificate. (For more information, see “Sending Secure Notifications with Invalid URLs” on page 74.)  
• Notification type: Should be either alert, prefetch, or cacheop.  
• TTL: Should specify an integer number of seconds.  
• Alert type: Required. Consult the documentation of the notification tool or library you are using to make sure that you have specified the correct format. See the Tools and API Reference for complete documentation of the notification tools and libraries.  
• Alert title: Required for alert notifications. Make sure that the title is shorter than 256 characters.  
The most frequent cause for the 400 status code is an invalid subscriber ID. Before you retry the notification request, make sure that all of the data just described is correct. |
| 403  | The notification request failed: Your application does not have permission to make requests to the specified Mobile Access Gateway.  
To determine requirements for issuing notifications on an Mobile Access Gateway, consult the Mobile Access Gateway administrator. |
Sending Notifications
Debugging Notifications

IMPORTANT The description of the 204 HTTP status code ("Empty Request") might seem to imply that your request to add or delete a notification has failed. However, it actually means the request has succeeded.

Checking Notification Status
If you have login access to the Mobile Access Gateway to which you are sending notifications, you can easily determine the disposition of individual notifications. For example, you can determine how long it has been since the Mobile Access Gateway received a notification and whether the notification has been successfully delivered.

If you register as an Openwave developer, you can log in to the Openwave Developer Mobile Access Gateway. To register, see the Developer web pages at:

http://developer.openwave.com

Connecting to a Mobile Access Gateway is useful for debugging notifications. If your application appears to be sending notifications successfully, but the mobile browser device is not receiving them, you can check the notification status on the Mobile Access Gateway.

To Check the Status of Pending or Recently Delivered Notifications

1 Use a web browser to log in to the provisioning pages of the Mobile Access Gateway.

To log in to the provisioning pages, you need a username and a password from the Mobile Access Gateway administrator.

2 Find the profile of the subscriber to whom you sent the notification.

With the Mobile Access Gateway provisioning pages, you can search for individual subscribers. For instructions on searching for subscribers, see the Tools and API Reference.

3 Click the Notifications link at the bottom of the Subscriber Profile page.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>404</td>
<td>The notification deletion or status request failed. The specified notification is not pending on the Mobile Access Gateway for any of the following reasons:</td>
</tr>
<tr>
<td></td>
<td>• The notification was already delivered.</td>
</tr>
<tr>
<td></td>
<td>• The notification has already expired.</td>
</tr>
<tr>
<td></td>
<td>• The notification was already deemed undeliverable.</td>
</tr>
<tr>
<td>503</td>
<td>The notification request failed: The request was valid and authorized. However, the Mobile Access Gateway was unable to accept the notification for delivery. Retry the request later.</td>
</tr>
</tbody>
</table>
The following table describes the fields in the Subscriber Notifications list.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>The notification type: push or pull</td>
</tr>
<tr>
<td>Service ID</td>
<td>For trusted services, such as UP.Mail, the IP address of the service</td>
</tr>
<tr>
<td></td>
<td>originating the notification</td>
</tr>
<tr>
<td>Date of Post</td>
<td>The date when the notification was posted to the Mobile Access Gateway</td>
</tr>
<tr>
<td>Time of Post</td>
<td>The time when the notification was posted to the Mobile Access Gateway</td>
</tr>
<tr>
<td>Date of Expiry</td>
<td>The date when the notification is due to expire</td>
</tr>
<tr>
<td>Time of Expiry</td>
<td>The time when the notification is due to expire</td>
</tr>
<tr>
<td>Status</td>
<td>The status of the notification. One of the following:</td>
</tr>
<tr>
<td></td>
<td>delivered: Downloaded to the mobile browser device</td>
</tr>
<tr>
<td></td>
<td>pending: Not yet downloaded to the mobile browser device</td>
</tr>
<tr>
<td></td>
<td>undeliverable: Mobile Access Gateway Messenger unable to deliver</td>
</tr>
<tr>
<td></td>
<td>expired: Mobile Access Gateway Messenger unable to deliver before expiration</td>
</tr>
</tbody>
</table>

**Figure 5-7. Subscriber Notifications page**

The following notifications are currently pending delivery or have completed in the Messenger for subscriber 8598592191-4976. To sort them, click on one of the highlighted columns. To view the notification profile, click on the type field.

<table>
<thead>
<tr>
<th>Type</th>
<th>Service ID</th>
<th>Date of Post</th>
<th>Time of Post</th>
<th>Date of Expiry</th>
<th>Time of Expiry</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>public HTTPsvc 204.163.162.183</td>
<td>12/04/98</td>
<td>15:32:34</td>
<td>12/04/98</td>
<td>16:32:36</td>
<td>delivered</td>
</tr>
<tr>
<td>PUSH</td>
<td>private HTTP svc 204.163.167.193</td>
<td>12/04/98</td>
<td>11:20:42</td>
<td>12/04/98</td>
<td>11:20:44</td>
<td>delivered</td>
</tr>
</tbody>
</table>

**IMPORTANT** By default, the Mobile Access Gateway maintains the status of a notification in its database for 48 hours after the notification expires, is delivered, or is deemed undeliverable. This period is configurable by the Mobile Access Gateway operator.
Notifications and Security

Because careless or malicious services could annoy Mobile Access Gateway subscribers with notifications, the Mobile Access Gateway provides the following controls on notifications:

- An access control list: The Mobile Access Gateway administrator can specify the domains from which the server will (or will not) accept notifications.
- A secure notification port: This port accepts notifications only from services with a server certificate from an approved certificate authority. The Mobile Access Gateway operator must enable this port after installation—the nonsecure notification port is enabled by default.

The certificate required to send notifications to the secure port (usually called a server certificate) can only be used to issue notifications from a specific domain. The Common-Name certificate stores the subdomain and domain names of the host sending the notifications and ensures that malicious services cannot misrepresent themselves and send spurious notifications. For more information, see Chapter 6, “Security”.

Sending Secure Notifications

To send secure notifications, you need to add the following two steps to the beginning of the general steps outlined in “Sending Notifications” on page 67.

If you want to send secure notifications, request a certificate from a certificate authority.

To send secure notifications, you must request and install a server certificate from a certificate authority approved by the Mobile Access Gateway provider.

For more information, see the Tools and API Reference.

Requesting Verisign Certificates

Verisign currently provides information and instructions for obtaining certificates at the following URL:


Before you request a real certificate, you should request a free test certificate, which you can use to prototype and test your application. Verisign provides information about test certificates at the following URL:

http://digitalid.verisign.com/test_server_ids.html

Test certificates are typically issued immediately on request. However, they are valid only for a short time (normally two weeks from when they are issued).
Determining Which Security Mode to Use

Using the SDK notification APIs, you can issue either secure or nonsecure notifications. When you issue a secure notification, the default mode is secure preferred. In this mode, the APIs first attempt to connect to the Mobile Access Gateway secure port. If the secure port is not available, they attempt to connect to the nonsecure port.

The following table lists notification modes and the types of connections that the APIs establish with different Mobile Access Gateway ports enabled.

<table>
<thead>
<tr>
<th>Notification mode</th>
<th>Nonsecure only</th>
<th>Secure only</th>
<th>Both</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonsecure only</td>
<td>Nonsecure connection</td>
<td>Connection fails</td>
<td>Nonsecure connection</td>
</tr>
<tr>
<td>Secure preferred</td>
<td>Nonsecure connection</td>
<td>Secure connection</td>
<td>Secure connection</td>
</tr>
<tr>
<td>Secure only</td>
<td>Connection fails</td>
<td>Secure connection</td>
<td>Secure connection</td>
</tr>
</tbody>
</table>

Secure preferred mode is the only mode that guarantees notification delivery. However, you should not use it for information that requires security. The following table summarizes the recommended uses for each notification mode.

<table>
<thead>
<tr>
<th>Notification mode</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonsecure only</td>
<td>Testing</td>
</tr>
<tr>
<td>Secure preferred</td>
<td>Applications for which it is important to work on every Mobile Access Gateway, regardless of security (most applications)</td>
</tr>
<tr>
<td>Secure only</td>
<td>Applications that require security</td>
</tr>
</tbody>
</table>

Notification Port Numbers

In general, you do not need to worry about notification port numbers because the SDK APIs issue notifications to the appropriate ports by default. The following table lists the default Mobile Access Gateway notification ports.

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4445</td>
<td>Internet Assigned Number Authority (IANA) assigned port for nonsecure notifications</td>
</tr>
<tr>
<td>3356</td>
<td>IANA-assigned port for secure notifications</td>
</tr>
</tbody>
</table>
Sending Secure Notifications with Invalid URLs

If you send a secure notification with an invalid URL for your certificate (for example, if the notification URL is http://mycompany.com but the Common-Name for your certificate is ntfn.www.mycompany.com), the following can happen:

- If the Mobile Access Gateway supports secure notifications, the notification fails with a 400 HTTP error code; no failover to the nonsecure port occurs. This is because a valid secure connection can be established, but the URL is invalid.
- If the Mobile Access Gateway does not support secure notifications and you have allowed failover, the notification fails to the Mobile Access Gateway nonsecure port. This is because URLs are not validated for nonsecure notifications.
Chapter 6 Security

This chapter describes security issues for WML services. It provides an overview of the security measures that the Mobile Access Gateway platform supports and describes how to set up WML services to minimize security risks to yourself and to the Mobile Access Gateway subscriber.

Overview of Mobile Access Gateway Platform Security

The Mobile Access Gateway platform extends standard web security practices and protocols to handheld devices. The following sections summarize how the Mobile Access Gateway platform supports secure communication.

Mobile Browser Device–Mobile Access Gateway Communication

Communication between the mobile browser device and the Mobile Access Gateway is generally very secure. The default communication protocol provides the following security features:

- **Authenticity**: Shared secret and session keys mutually authenticate the mobile browser device and Mobile Access Gateway.
- **Privacy**: The content of each message is encrypted using the RC5 cipher algorithm provided by RSA Data Security.
- **Integrity**: Both the mobile browser device and Mobile Access Gateway detect message corruption.

For more information about RSA or particular security algorithms, see:

http://www.rsa.com

Mobile Access Gateway administrators can choose to use an unencrypted communication protocol, which provides relatively little security. To determine whether an Mobile Access Gateway is using encrypted or unencrypted communication, consult the administrator.
Service-to-Mobile Access Gateway Communication

The security of communications between your service and an Mobile Access Gateway depends on the communication protocol you choose. The Mobile Access Gateway platform supports the following protocols.

<table>
<thead>
<tr>
<th>Protocol or security practice</th>
<th>Security level</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>None</td>
</tr>
<tr>
<td>HTTP with Basic Authentication</td>
<td>Relatively low</td>
</tr>
<tr>
<td>HTTPS with Basic Authentication</td>
<td>High</td>
</tr>
<tr>
<td>HTTPS with SSL Client Certificates</td>
<td>Very high</td>
</tr>
</tbody>
</table>

**IMPORTANT** Use the same security precautions (for example, HTTPS and Basic Authentication) in mobile browser device transactions as you would for the same transactions with a conventional web browser.

Writing WML That Minimizes Security Risks

The following WML attribute helps control security risks from malicious services.

<table>
<thead>
<tr>
<th>Element</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>sendreferer</td>
<td>Specifies whether the Openwave Mobile Browser should provide the URL of the current deck when requesting the specified URL. Setting sendreferer=&quot;true&quot; causes the device to specify the deck URL in the http_referer request header.</td>
</tr>
</tbody>
</table>

If you do not specify this attribute, the phone uses the default setting, which provides the highest degree of security. In general, the default setting provides the most security. However, there are some specific areas where you should be cautious. These are described in “Common Security Risks for WML Services” on the next page.

Checking the HTTP Referer Header

If your service provides URLs that perform sensitive operations, it should check the HTTP Referer header (the HTTP_REFERER environment variable set by the web server) to make sure that the requests it handles originate from friendly domains. The phone does not set the Referer header unless you specify sendreferer="true" in the <go> task that makes the request.

**IMPORTANT** Always check the Referer header when handling requests for sensitive information or operations.
Common Security Risks for WML Services

The most common type of security risk is presented by a Mobile Access Gateway subscriber voluntarily requesting a “malicious” service masquerading as a legitimate service. Such a malicious service might attempt to use the subscriber’s access to your WML service illicitly. There are several ways it might attempt to do this:

- Directly accessing a card in your service that conducts sensitive operations, such as banking transactions or stock trades
- Gaining access to variables in your service that provide confidential information, such as messages and financial information
- Clearing variables used by your service

Figure 6-1 shows a simple example of a security risk. A banking service allows the user to transfer funds. The user presses ACCEPT in a card that describes the proposed transaction; the card’s type="accept" action invokes the URL of a script that conducts the fund transfer.

The malicious service, depicted on the left, poses as a weather service. When the user presses OK to get the weather, the service requests the fund transfer URL from the banking service, using slightly different arguments. The user thinks that he or she is requesting today’s weather report but is actually transferring funds to the owner of the malicious service.
To avoid this risk, the banking service should make several changes:

- The card that requests the funds transfer should set sendreferer="true".
- The script that actually transfers funds (transfer.cgi) should check the URL specified by the REFERER header of the HTTP request to ensure that the request is coming from one of the banking service’s decks.
- The service should use HTTPS and require basic authentication, just as it would if the transaction were being conducted with a conventional web browser. Relying on the phone’s identity alone does not provide sufficient security.
Interacting with Other Services

Chapter 7 Interacting with Other Services

This chapter describes how to interact with other WML or HDML services, including:

- Calling registered services to perform tasks, such as sending email or adding entries to an address book or to-do list
- Providing a registered service to which other services can issue requests

**IMPORTANT** The Openwave Mobile Browser, WAP Edition, Release 5.0 supports contexts, or private subfunctions that always return to the calling function. As a result, you can call the Mobile Access Gateway applications directly from a WML deck.

Overview

The Mobile Access Gateway platform provides a Service Registry, which lets users designate services to perform specific types of tasks and coordinates interaction between those services. The user can register services in the following categories:

- **Email**: An email service that can send electronic mail.
- **AddrBook**: The user’s preferred address book service. The service should support at least the selection of email, phone, and fax addresses.
- **ToDo**: The user’s preferred to-do list service.

The Service Registry provides a generic URL for each registry category. For example, the generic URL for the email service is:

\[
\text{device:home/goto?svc=Email}
\]

After the user has registered a service, the Service Registry redirects requests for the generic URL to the URL of the registered service. For example, if a user registers a service at http://mymail.foo.com as an email service, the Service Registry redirects requests for `device:home/goto?svc=Email` to http://mymail.foo.com.

The Service Registry makes it easy for other services to issue requests to the user’s registered services. For example, if your service needs to send an email message on behalf of the user, you don’t need to know about http://mymail.foo.com—you can simply request `device:home/goto?svc=Email`, and the Service Registry automatically redirects your request to the correct URL.
Users can register more than one service for each category—for example, multiple email services. If a user has registered multiple services for a category, the device displays a list of those services on every request for that type of service, allowing the user to select the service to use for that request.

### Issuing Requests to Registered Services

Your service can call registered services to send mail, retrieve or add address book entries, or add to-do entries.

To request registered services, use the following URL:

```
device:home/goto?svc=category&url=url&err=flag
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>svc</td>
<td>The registry category for the service (Email, AddrBook, or ToDo).</td>
</tr>
<tr>
<td>url</td>
<td>(Optional) The URL to open if the device cannot redirect your request. The URL typically displays an error deck with the reason that the redirect failed. If you don’t specify this argument, redirection errors are handled as specified by the err argument.</td>
</tr>
<tr>
<td>err</td>
<td>(Optional) The action to take if an error occurs. If you specify T, the device simply executes an <code>&lt;exit&gt;</code> element. If you do not specify this element, the device displays an error card (equivalent to setting err=F), which returns using an <code>&lt;exit&gt;</code> element.</td>
</tr>
</tbody>
</table>

### Using the Registered Email Service

Service Registry-compliant email services such as UP.Mail provide interfaces that allow other services to send email messages on the user’s behalf. To send an email message using the user’s registered email service, specify an element as follows:

```
<spawn href="device:home/goto?svc=Email&SUB=sendMsg"
<setvar name="to" value="address"/>
</spawn>
```

where SUB specifies the UP.Mail subroutine to call. To send an email, it must be set to sendMsg (the email send subroutine).

A Service Registry-compliant email service displays the message that you specify, allowing the user to confirm the address and content. This ensures that a malicious service cannot send a message without the user’s knowledge.
For example, the following WML deck invokes the registered email service (UP.Mail in this case) to send a simple message to Developer Support at Openwave. The interface created by this code is similar to that shown in Figure 7-1.

```xml
<!-- Sendmail via Openwave Mail passing To, Subj, & MessageText variables -->
<wml>
  <card title="SendMail">
    <do type="accept" label="Send">
      <spawn href="device:home/goto?svc=Email&amp;SUB=sendMsg">
        <setvar name="TO" value="developer@openwave.com"/>
        <setvar name="SUBJ" value="Mail From ACME"/>
        <setvar name="MT" value="Mail from ACME Example Application"/>
      </spawn>
    </do>
    <p>Press Send to send email.</p>
  </card>
</wml>
```

**Figure 7-1. Sending email with UP.Mail**

[Diagram showing the process of sending email with UP.Mail]
Using the Registered Address Book Service

Service Registry-compliant address book services, such as UP.Organizer, provide interfaces that allow other services to do the following:

- Prompt the user to choose an address from the address book. The UP.Organizer service returns the chosen address to the <spawn> context.

- Add an address record to the address book. The UP.Organizer service confirms that the user wants to add the record and then adds it.

Selecting Addresses from a Registered Address Book

To prompt the user to select an address from a registered address book, specify a context as follows:

```xml
<spawn
href="device:home/goto?svc=AddrBook&amp;SUB=pickAddr&amp;GET=field">  
 <receive/>
 <receive name="field"/>
</spawn>
```

where SUB specifies the UP.Organizer subfunction to call. To request an address, it must be set to pickAddr (the address book selection subfunction).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Specifies the address book field that the address book returns to the first variable in the &lt;receive&gt; list, either fax, phone (prompts the user to choose between home and work), or email. If GET is left out or is set to something other than fax, phone, or email, the default setting is phone.</td>
</tr>
</tbody>
</table>

You must specify the <receive> element with variables for each value to be returned when you request the address book URL. If the address book cannot provide the value specified by the GET argument, it returns empty strings for all <receive> variables, and the device displays an error message.
For example, the following WML deck invokes the registered address book service (UP.Organizer in this case) to select an address and display it (see Figure 7-2).

```xml
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//OPENWAVE.COM//DTD WML 1.3//EN" "http://www.openwave.com/dtd/wml13.dtd">
<!-- Openwave Organizer address book lookup via email -->
<wml>
    <card title="PickAddr">
        <do type="accept" label="PICK">
            <spawn onexit="#DispName" href="device:home/goto?svc=AddrBook&SUB=pickAddr&GET=email">
                <receive/>
                <receive name="email"/>
                <receive name="first"/>
                <receive name="last"/>
            </spawn>
        </do>
    </card>
    <card title="DispName" id="DispName">
        <p>Press PICK to pick an address.</p>
    </card>
</wml>
```

**IMPORTANT** You must specify the `href` attribute on a single line. In other words, you cannot use line breaks in this value.
Adding Entries to a Registered Address Book

To add an entry to a registered address book, specify a WML context as follows:

```wml
<spawn href="device:home/goto?svc=AddrBook&amp;SUB=addAddr">
  <setvar name="LAST" value="lname"/>
  <setvar name="FIRST" value="fname"/>
  ...
</spawn>
```

where SUB specifies the UP.Organizer subroutine to call. To add an address, it must be set to addAddr (the address book insertion subfunction). You can specify any combination of one or more of the following address book fields for the name attribute.
## Interacting with Other Services

### Issuing Requests to Registered Services

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST</td>
<td>URL-escaped string specifying the first name</td>
</tr>
<tr>
<td>LAST</td>
<td>URL-escaped string specifying the last name</td>
</tr>
<tr>
<td>EMAIL</td>
<td>The Internet email address</td>
</tr>
<tr>
<td>FAX</td>
<td>URL-escaped string specifying the fax phone number (for example, a string in the format NNN–NNN–NNNN)</td>
</tr>
<tr>
<td>WORK</td>
<td>URL-escaped string specifying the work phone number (for example, a string in the format NNN–NNN–NNNN)</td>
</tr>
<tr>
<td>HOME</td>
<td>URL-escaped string specifying the home phone number (for example, a string in the format NNN–NNN–NNNN)</td>
</tr>
<tr>
<td>HOME2</td>
<td>URL-escaped string specifying the second home phone number (for example, a string in the format NNN–NNN–NNNN)</td>
</tr>
<tr>
<td>CELL</td>
<td>URL-escaped string specifying the cell phone number (for example, a string in the format NNN–NNN–NNNN)</td>
</tr>
<tr>
<td>PAGER</td>
<td>URL-escaped string specifying the pager phone number (for example, a string in the format NNN–NNN–NNNN)</td>
</tr>
<tr>
<td>PRIMARYNUMBER</td>
<td>A flag identifying the primary contact number, one of the following values: w: work (default) h: home c: cell p: pager a: alternate (home2)</td>
</tr>
<tr>
<td>COMPANY</td>
<td>URL-escaped string specifying the company name</td>
</tr>
<tr>
<td>COMPANYSTREET</td>
<td>URL-escaped string specifying the company street address</td>
</tr>
<tr>
<td>COMPANYCITY</td>
<td>URL-escaped string specifying the company city</td>
</tr>
<tr>
<td>COMPANYSTATE</td>
<td>URL-escaped string specifying the company state</td>
</tr>
<tr>
<td>COMPANYZIP</td>
<td>URL-escaped string specifying the company zip code (for example, a string in the format NNNNNN or NNNNN–NNNNN)</td>
</tr>
<tr>
<td>COMPANYCOUNTRY</td>
<td>URL-escaped string specifying the company country</td>
</tr>
<tr>
<td>HOMESTREET</td>
<td>URL-escaped string specifying the home street address</td>
</tr>
<tr>
<td>HOMECITY</td>
<td>URL-escaped string specifying the home city</td>
</tr>
<tr>
<td>HOMESTATE</td>
<td>URL-escaped text specifying the home state</td>
</tr>
<tr>
<td>HOMEZIP</td>
<td>URL-escaped string specifying the home zip code (for example, a string in the format NNNNN or NNNNN–NNNN)</td>
</tr>
<tr>
<td>HOMECOUNTRY</td>
<td>URL-escaped string specifying the home country</td>
</tr>
</tbody>
</table>
For example, suppose that you want to add an entry to the address book. You could use a WML deck similar to the following. Figure 7-3 depicts the resulting interface.

```xml
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//OPENWAVE.COM//DTD WML 1.3//EN" "http://www.openwave.com/DTD/wml13.dtd">
<!-- Openwave Organizer add address -->
<wml>
  <card title="AddAddr">
    <do type="accept" label="Add">
      <spawn href="device:home/goto?svc=AddrBook&amp;SUB=addAddr">
        <setvar name="LAST" value="Best"/>
        <setvar name="FIRST" value="Eric"/>
        <setvar name="EMAIL" value="eric@openwave.com"/>
      </spawn>
    </do>
    <p>Press Add to add address.</p>
  </card>
</wml>
```

**Figure 7-3. Adding an address to the UP Organizer address book**
Using the Registered To-Do Service

Service Registry-compliant to-do services, such as UP.Organizer, provide interfaces that allow other services to do the following:

- Add a call entry to the to-do list. The UP.Organizer service confirms that the user wants to add the call and then adds it.
- Add a task entry to the to-do list. The UP.Organizer service confirms that the user wants to add the task and then adds it.

Adding Call Entries to a Registered To-Do Service

To add a call to a registered to-do service, specify a WML context as follows:

```wml
<spawn href="device:home/goto?svc=ToDo&amp;SUB=addCall">
  <setvar name="TEXT" value="Eric+Best"/>
  <setvar name="PHONE" value="555-555-1212"/>
</spawn>
```

where SUB specifies the UP.Organizer subroutine to call. To add a call, it must be set to `addCall` (the to-do call insertion subroutine). You can optionally specify either of the following fields for the `name` attribute.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>The name of the person to call</td>
</tr>
<tr>
<td>PHONE</td>
<td>The phone number to call</td>
</tr>
</tbody>
</table>

**NOTE** If you do not specify the TEXT or PHONE argument, the device prompts the user to enter the appropriate values.

For example, suppose that you want to add a call to the to-do list. You could use a WML deck similar to the following. Figure 7-4 shows the resulting interface.

```wml
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//OPENWAVE.COM//DTD WML 1.3//EN" "http://www.openwave.com/dtd/wml13.dtd" >
<!-- UP.Organizer add Call ToDo entry -->
<wml>
  <card title="ToDoAdd">
    <do type="accept" label="Add">
      <spawn href="device:home/goto?svc=ToDo&amp;SUB=addCall">
        <setvar name="TEXT" value="Eric+Best"/>
        <setvar name="PHONE" value="555-555-1212"/>
      </spawn>
    </do>
    <p>
      Press Add to add Call ToDo entry.
    </p>
  </card>
</wml>
```
Adding Task Entries to a Registered To-Do Service

To add a task to a registered to-do service, specify a WML context as follows:

```wml
<spawn href="device:home/goto?svc=ToDo&amp;SUB=addToDo">
  <setvar name="TEXT" value="todotext"/>
</spawn>
```

where SUB specifies the UP.Organizer subroutine to call. To add a to-do item, it must be set to addToDo (the to-do task insertion subroutine).

For example, suppose that you want to add a task to the to-do list. You could use a WML deck similar to the following. Figure 7-5 shows the resulting interface.

```wml
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//OPENWAVE.COM//DTD WML 1.3//EN" "http://www.openwave.com/dtd/wml13.dtd" >
<!-- UP.Organizer add ToDo entry -->
<wml>
  <card title="ToDoAdd">
    <do type="accept" label="Add">
      <spawn href="device:home/goto?svc=ToDo&amp;SUB=addToDo">
        <setvar name="ACTION" value="Send Flowers"/>
      </spawn>
    </do>
    <p>Press Add to add ToDo entry.</p>
  </card>
</wml>
```
Providing a Registered Service

To let users register your email, address book, or to-do service in the Service Registry, display a card that lets the user request the following URL:

```
device:home/register?svc=svc&url=url&name=name
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>svc</td>
<td>The registry category for your service (Email, AddrBook, ToDo, or FaxHTML)</td>
</tr>
<tr>
<td>url</td>
<td>The URL for your service</td>
</tr>
<tr>
<td>name</td>
<td>The service name that the device displays to the user</td>
</tr>
</tbody>
</table>

For example, suppose that you want users to be able to register your service as an address book and your service has the following URL:

```
http://myaddr.foo.com/addrbook.cgi
```
You could use a WML deck similar to the following to register it:

```xml
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//OPENWAVE.COM//DTD WML 1.3//EN" "http://www.openwave.com/dtd/wml13.dtd" >
<!-- Add Registered Service -->
<wml>
  <card title="AddRegSvc">
    <do type="accept" label="Yes">
      <spawn onexit="#AddDone" href="device:home/register;svc=AddrBook&url=http://myaddr.foo.com/addrbook.cgi&name=MyAddr">
        MyAddr registered.
      </spawn>
    </do>
    <p>Register MyAddr as your address book?</p>
  </card>
  <card title="AddDone" id="AddDone">
    <p>MyAddr registered.</p>
  </card>
</wml>
```

**Figure 7-6. Enabling the user to register a service**

To properly implement a registered service, it must support the minimum functionality and interface for its service category (described in “Using the Registered Email Service” on page 80 for email services, “Using the Registered Address Book Service” on page 82 for address book services, and “Using the Registered To-Do Service” on page 87 for to-do services).
Security Issues for Registered Services

In addition to supporting the functionality and interface described in the previous sections, registered services should observe security precautions to protect users and themselves. Use the following guidelines when you implement a service that users can register:

- A registered email service should display any message that another service has asked it to send, allowing the user to confirm it. This ensures that a malicious service cannot send a message without the user’s knowledge.

- A registered address book service should display any address entry it is about to add at the request of another service.

- For secure information, the service should employ username-password and certificate authentication.
This chapter describes the user interface guidelines that you should follow when you create a WML service. Following these guidelines enhances usability and ensures that your user interface is consistent with those of other WML services.

**NOTE**  For additional design and implementation guidance, refer to the Openwave Developer web site at [http://developer.openwave.com](http://developer.openwave.com).

### Testing Your User Interface

You should test your WML services extensively to make sure that they are easy to use. It is important to test your services with the *least common denominator device*, the device with the smallest display—currently the Mitsubishi phone. For information about configuring the Openwave Mobile Browser Simulator to emulate a Mitsubishi phone, see the *Tools and API Reference*.

### Best Practices

You should always consider design philosophies when building an application for Openwave Mobile Browser, WAP Edition, Release 5.0 devices. The user’s experience with an application determines whether or how often he or she revisits the application. Keep in mind that most of these devices are used as a phone first, that users are concerned about airtime costs, and that they are looking for information rather than just browsing.

### Define the User

Identify who will use the application. Keep in mind that first-time users may not return to an application that is too difficult to use. You should build redundancy into your application to make it easier for regular and power users. Localize the application to the environment, using the correct terminology and date and time formats.
**Determine the Goal**
Create usage scenarios to tailor the application to the user. Identify the features that will be used most frequently and place those items at the top of the card, making them viewable on the display when the card is first downloaded.

**Make the Goal Easily Attainable**
Place the highest priority actions at the top. Create redundancy in the application so that an item can be selected from two menus. For example, a user may expect to be able to access restaurant information from menu items labeled Entertainment, Directories or Yellow Pages, or even Local Information.

**Make the Application Easy to Navigate**
Users can get lost in deep menus. Keep the navigation to three or four menu levels deep.

**Limit the Application to the Necessary Functionality**
Avoid extra activities that cause the application to have more menus than necessary. Seldom-used items should be buried in the application or removed entirely.

**Make the Application Consistent Throughout**
Whenever possible, reuse menu ordering, menu names, and softkey labels.

**Avoid Text Entry**
Whenever possible, use selection lists, multiple-selection lists, pop-up menus, and option buttons instead of text input. Create links to online address books and phone directories so that the user does not have to enter names, phone numbers, and email addresses.

**Prevent Unnecessary User Errors**
Provide descriptive card titles, softkey labels, and formatted text entry information. Create redundancy in the application so that users can find the same menu in different places. Replicate high-priority actions on a button as well as a softkey. Make search engines and date entries forgiving. For example, use formatted text fields or allow search engines to find partial matches, and make logins and passwords not case sensitive.
Graphical Browser Enhancements

The Openwave Mobile Browser, Release 5.0 offers improved support for the following graphical user interface features:

- Titles
- Forms
- Tables
- Multiple selection lists

The following features have been added to the browser to enhance the user’s experience:

- Buttons
- Text boxes
- Option buttons
- Pop-up menus
- Pop-up menu from the secondary softkey
- Horizontal rules

Many mobile browser devices have a text-based browser, so you must still write or maintain content for those browsers to ensure usable applications. An application that is written for the Openwave Mobile Browser, Release 4.x will still run on text-based browsers and on the Openwave Mobile Browser, Release 5.0. However, there are some additions and changes that you can make to enhance usability for the Openwave Mobile Browser, Release 5.0.

Navigation

To navigate Wireless Markup Language (WML) content, the user must move among cards in one or more decks. The cards can contain many different types of elements, including selection lists, multiple-selection lists, displayed information (such as an email message), input fields, buttons, option buttons, tables, and pop-up menus. To make applications run on multiple browsers, use the following general rules.

General Navigation

The Openwave Mobile Browser, Release 5.0 uses the primary softkey to select and deselect some items. Therefore it is important to provide a button or link at the end of the card to complete an action when the card consists of one or more input elements, pop-up menus, and option buttons. The primary softkey label is displayed only when the user scrolls through all elements (for example, top or bottom of the card). Therefore you should consider replicating the action of the softkey with a button or link. The default behavior for the primary softkey when no element is selected and no `<do type="accept">` task is assigned is to display Back mapped to the `<prev>` task.
Keep lists of menu items short, nine items or fewer, so that users can use the shortcut accelerator 1 through 9 keys (via the `<option onpick=href>` element) to quickly access the desired menu item. If more than nine items are needed, make the ninth item a More link and continue the list on another card. Links can also be assigned a key accelerator via the `accesskey` attribute. The items in the list should be sorted in a logical order, such as priority of use, alphabetical, date, or time, and not wrapped unless the item is used as a short subject header (for example, displaying the headlines for news articles). When displaying these headers, display each header on a card of its own followed by an anchor with the label Skip, so that the user can navigate to the next item. Map the `<do type="accept">` task to a View label so that the user can view the news item. While the user is viewing the headline, map the `<do type="accept">` task to a Skip label, so that the user can skip to the next headline.

The `<do type="accept">` label is displayed on the primary softkey, so you should define this softkey with the most likely chosen action. The `<do type="options">` label is displayed on the secondary softkey. When two or more options are assigned, the secondary softkey becomes a pop-up menu with the defined options. Therefore you should limit the number of softkey actions to two or fewer. This makes it easier for users to navigate because they don’t have to view items on a pop-up menu to see what additional functions are available.

By default, links are displayed sequentially in a menu list. Links can also be displayed side by side by adding a punctuation mark (for example, link1, link2) or other text (for example, link1 and link2) between the links. Optimum efficiency is achieved by embedding links in text.

To allow users to make a call from the browser, assign the action `href="wtai://wp/mc;phone number>"` and assign the label Call. It is possible to assign the call function to the Send key, the softkey, a link, and/or a button.

**Backward Navigation**

Backward navigation is important because users tend to use the Back key or softkey to back out of an application. Users are more likely to trust applications that have good backward navigation functions. Also, well-designed backward navigation lets users leave the application without forcing them to return to the home deck. In some cases, the design must prohibit navigation behind password-protected cards.

In general, when direct backward navigation is not suitable, you should map backward navigation to the next highest or most intuitive menu. Activities can assist with intuitive backward navigation in these situations. Activities create a start point that allows the application to return to a specified card when the user presses the Back key. The intermediate cards (those accessed before the return to the specified card) are removed from the history and the cache. For example, in purchase, banking, and login applications, you should not allow users to navigate back over the purchase because they won’t know if the transaction was committed or if going forward will request the purchase again. Similarly, if the user deletes an email message and the next email on the list is displayed automatically, when the user navigates backward, the inbox should be displayed rather than the deleted message. Empty the history for applications that require passwords or
secured data. This deletes variables in the handset. If that is not desirable, use
<onevent type="onenterbackward"> to catch the backward navigation.

<wml>
    <card id="first" title="Home Deck">
        <p>
            Welcome to our service. Please<br/>
            <do type="button" label="Continue">
                <go href="#bank"/>
            </do>
        </p>
    </card>

    <card id="bank" title="PCS Banking">
        <p>
            Welcome to your wireless bank!<br/>
            <select type="list">
                <option>
                    <onevent type="onpick">
                        <spawn href="#login">
                            <catch/>
                        </spawn>
                        <onevent>Login</option>
                </option>
                <option onpick="/services">Services</option>
                <option onpick="/about">About</option>
            </select>
        </p>
    </card>

    <card id="login" title="Bank Login">
        <do type="accept" label="Login">
            <exit/>
        </do>
        <p>
            Account Number:<br/>
            <input name="acctno" format="N*N" size="0"/>
        </p>
        <p>
            Password:<br/>
            <input name="passwd" size="0"/>
        </p>
        <p align="center">
            <do type="button" label="Login">
                <exit/>
            </do>
        </p>
    </card>
</wml>
Sometimes it’s convenient to create a second backward navigation when moving one step backward is less desirable than backing out to the top of the application. For example, if the user retrieves general information, such as checking flight status, the application can provide a way for the user to step back one card to change the flight number; purchase a ticket on that flight; and return to the top of the application. In this case, you can provide backward navigation via <do type="options"> if backward navigation should return the user to a higher menu or to the top menu of the application. Use the Done label to indicate that backward navigation takes the user back more than one step. The Done label appears on the secondary softkey.

```wml
<card id="flightstat" title="Flight Status">
  <do type="options" label="Done">
  <exit/>
  </do>
  <do type="accept" label="Buy">
    <go href="statrsrv.wml"/>
  </do>
  <p>
    $airline 104<br/>
    From $depart to $arrive<br/>
    Departs: $statdate at 2:20p<br/>
    Arrives: $statdate at 3:57p<br/>
    Flight is on time<br/>
    Seats available
  </p>
</card>
```

Defining an action of <noop/> type to a <do type="prev"> task disables the Back navigation key. This is not a good idea because many users rely on the Back key for general navigation, and if it is disabled the user would be required to return to the home deck to navigate to another application. Instead, bind the <prev> task to an intuitive place in the application, to a starting point in the application, or to the home deck.

When the user exits an entry field, it may be helpful to temporarily save the values of all or some of the variables in a cookie. This can reduce the amount of information the user must enter in the future. For example, it may be helpful to retain nonsecure information entered in an order form, such as the name and address, so that the user does not have to reenter it in the same browsing session.

If backing out of an application causes the user to loose data entered, you should provide a card asking the user to confirm that he or she wants to exit.
Forms

Forms can encompass one or more display or action types on a card or set of cards.

General Use

Long forms can be confusing; therefore you should create separate short forms of related data and display them in logical order. Use activities to create intuitive backward navigation that drops the cards out of the history when changing data between forms. However, retain the data so that the data already entered only needs to be modified.

After the final element in the form, use buttons to submit and/or cancel the data. Users often select the button without looking to see if there is additional information. Replicate the action for submitting the data on the `<do type="accept">` task. Buttons can also be used in forms to navigate to a list of existing text that can be inserted into an input field (for example, insert an email address into a To field). In this instance, the button is not used to submit data, but returns the user back to the form.

When submitting the date, in the final form of a series of forms, present the user with a final verification or confirmation card that recaps all entered or selected information. Allow the user to confirm or edit the information or to cancel out of the application. If the user chooses to cancel, present a card to inform and confirm that the data entered may be lost when exiting the application.

Using Graphical Elements

You can use the Openwave Mobile Browser, Release 5.0 graphical elements to allow users to set values, enter text, and press buttons to submit the data.

Never use `<select type="list">` with an `onpick` option to set the value of an item in a form. This can cause the browser to access a new card before the user can enter data or set values for other elements. Instead, use option buttons, pop-up menus, and multiple-selection lists to allow the user to set the value without leaving the card.
Use option buttons to allow the user to select a single option from a list. If no default item is specified, the first item on the selection list is automatically set as the default choice. In this case, if it is possible for the user select no item, make sure that the list has an option for 0 or None. Do not display more than four items in an option button list.

```xml
<card id="return" title="Return Flights">
  <do type="accept">
    <go href="reserveflt.wml"/>
  </do>
  <p>
    Return: $arrive to $depart on $rmonth $rday, $ryear.
  </p>
  <p mode="nowrap">
    <select name="rpickflt" title="Srchrtrn" type="radio">
      <option value="none">None</option>
      <option value="$rfltno1">$rtime1 $airline $rfltno1</option>
      <option value="$rfltno2">$rtime2 $airline $rfltno2</option>
      <option value="$rfltno3">$rtime3 $airline $rfltno3</option>
    </select>
  </p>
  <p mode="wrap" align="center">
    <do type="button" label="Reserve flights">
      <go href="reserveflt.wml"/>
    </do>
  </p>
</card>
```

Use a pop-up list to allow the user to select a single option from a list of five or more items. As with the option buttons, if it is possible for the user to select no item, make sure that the list has an option for 0 or None. Do not display more than nine items on the list. Because the entire list may not be visible to the user, define a logical order for the items.

```xml
<select type="popup" name="airline" title="Pick Airline">
  <option value="Any">Any</option>
  <option value="AA">American</option>
  <option value="AW">America West</option>
  <option value="AS">Alaska</option>
  <option value="CO">Continental</option>
  <option value="DL">Delta</option>
  <option value="NW">Northwest</option>
  <option value="SW">Southwest</option>
  <option value="TW">TWA</option>
  <option value="UA">United</option>
  <option value="US">USAir</option>
</select>
```
Use multiple selection lists to allow users to choose multiple items or to set the state of an item on or off, such as an alarm.

```html
<select multiple="true" name="alarm">
  <option value="on">Alarm</option>
</select>
```

### Data Entry

If the user must enter numeric or alphanumeric information, use the `input` element to elicit that information.

### General Use

Text entry on mobile phones is both difficult and time consuming. To avoid forcing the user to enter text, use selection lists, pop-up menus, and option buttons whenever possible. Prefill known data or anticipated information, such as today’s date for a travel application, or provide a button that allows users to access and insert data in the text field. Whenever possible, store the user’s entries and reuse them to avoid repetitive data entry.

When text entry is required, define the `maxlength` attribute to restrict the number of characters that can be entered. For general data input fields, restrict the length to 254 characters; for login and password fields, restrict the length to 32 characters. If an alphanumeric password is required, do not to mask it, because it’s easier for the user to hide the display from others than to type with masked characters.

Define an intuitive label for the `title` attribute and include descriptive text to inform the user what entry information is expected. For example, if the field requires a year in a specific format, the field title should indicate what is required; for example, YYYY.

```html
<br/>
Date: (DD/MM/YYYY)
<input format="NN/NN/NNNN" size="10" name="day" value="$date" />
<br/>
```

Set the default entry mode so that the user is not required to change the mode. For example, use `<type="phonenum">` so that the input mode is initially numeric but so that users can change to alphanumeric or symbol mode. Defining M*m defaults the input field to sentencelike formatting; M* sets the input to all capitals; and m* sets the input to lowercase.

Allow the user to easily submit the data by pressing a link or a button. Buttons are useful in forms, but they take up more space than links. Define the `<do type="accept">` element to replicate the action on the button or link.
Specify Formats

Formatted text fields can help users know what is required for their input. For example, you can include symbols (such as / : ; and -) in the formatting string to indicate the expected input and provide visual information to the user. It is also sometimes important to format a field by defining the number of digits to enter or forcing the user to enter text in a field by assigning the emptyok="false" attribute.

Defining the size attribute for a field can also provide information to the user. For example, if you define size="10", the text field displays exactly 10 characters, although this does not limit the length of the input. To define the size of the field to fill to the end of the line, define size="0". Do this only if the user does not need to view the entire text in the field—for example, a list of recipients in an email application.

```html
<do type="button" label="To:">
  <spawn href="emailaddr.wml">
    <receive name="moreto"/>
    <catch/>
  </spawn>
</do> <input size="0" name="newfro"/><br/>
```

In this case, make sure that a sufficient number of characters will fit on the line, particularly if the input field follows text. This is because the input element will render on the same line of the display as any text that precedes it. If you don’t know whether the input field will be large enough to accept enough characters, add <br/> after the displayed text and before the input field. To allow the field to grow as the user types, don’t define the size attribute. This is helpful for fields that can be of undetermined length, and prevents users from having to scroll through an empty text box.

Function Keys

The following sections provide guidelines for using function keys.

Assigning Tasks

**GUIDELINE: Assign the most frequently performed task to the ACCEPT function key.**

In general, the ACCEPT key should execute the action that the user is most likely to perform. Use the OPTIONS key for less common tasks or to display a menu of choices.
Assigning Labels

**GUIDELINE: Use standard function key labels.**

The following table indicates the recommended labels for common actions:

<table>
<thead>
<tr>
<th>Action</th>
<th>Function key</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display the next card for long display text</td>
<td>ACCEPT</td>
<td>More</td>
</tr>
<tr>
<td>Proceed to the next item in a list</td>
<td>ACCEPT</td>
<td>Next</td>
</tr>
<tr>
<td>Complete a task</td>
<td>ACCEPT</td>
<td>Done</td>
</tr>
<tr>
<td>Edit a specified value</td>
<td>ACCEPT</td>
<td>Edit</td>
</tr>
<tr>
<td>Execute a search</td>
<td>ACCEPT</td>
<td>Find</td>
</tr>
<tr>
<td>Answer a prompt positively</td>
<td>ACCEPT</td>
<td>Yes</td>
</tr>
<tr>
<td>Answer a prompt negatively</td>
<td>OPTIONS</td>
<td>No</td>
</tr>
<tr>
<td>Create a new item, record, or entry</td>
<td>ACCEPT or OPTIONS</td>
<td>New</td>
</tr>
<tr>
<td>Initiate a voice phone call</td>
<td>ACCEPT</td>
<td>Call</td>
</tr>
<tr>
<td>Originate an email message</td>
<td>ACCEPT or OPTIONS</td>
<td>Email</td>
</tr>
<tr>
<td>Provide a menu of tasks the user can perform</td>
<td>OPTIONS</td>
<td>Menu</td>
</tr>
<tr>
<td>Save a set of choices or preferences</td>
<td>OPTIONS</td>
<td>Save</td>
</tr>
</tbody>
</table>

If you create your own label, follow these guidelines:

- Capitalize only the first letter of the label.
- Use five characters or fewer, because some phone models truncate long labels.

Alerts

Alerts are currently supported on the Openwave Mobile Access Gateway. They add value to an application by notifying users of new information in their areas of interest.

The Openwave Mobile Browser, Release 5.0 supports WAP push according to the WAP 1.2.1 specifications. For information about push, refer to the Openwave Mobile Browser, *Push Library Developer’s Guide.*
An alert sent to the device should have a short title—15 characters or fewer. The browser uses this title to identify and sort the alert by the application that sent it. Therefore the same alert title should be sent whenever messages are delivered from an application. This is the title that appears in the inbox, not the message title. That is, only one alert title appears in the inbox, even if more than one message is sent to that box. Therefore you should use only one alert inbox slot and make sure that the same URL is used for all of that application’s alerts.

Set the message alert type (priority), but allow the user to override it. It is also possible to set the time to live (TTL), alert removal, delivery status, and security. Make sure that when an alert is sent the URL remains active for at least 24 hours, giving the user enough time to access the alert after it is delivered to the inbox.

### Images

Images provide information so that the user can quickly review a list of items or see a trend. They can be displayed as an icon inline with text or as a graphic in the card.

Graphics can be aligned according to the `<p>` element and can be embedded in a table. For example, a weather report can display a date along with an icon of the predicted weather. Similarly, an up or down arrow can precede a stock quote.

The following example allows an icon to be displayed on the same line as a menu item:

```xml
<option onpick="my_url"><img localsrc="envelope" src="" alt="" /> Email</option>
```

The example shows how to embed an image inline with a menu item. In this case, an envelope is displayed before the text Email.

Black-and-white images should be in `wbmp` format and color images should be in `png`, using the web-safe color palette to ensure that the color scheme of the image is respected. Always include descriptive alternative text for devices that do not support or that have trouble rendering the image. The WAP Forum does not currently define an animated image format, and there is no way to associate an area in an image to an action (there is no map function).

All Openwave Mobile Browser, WAP Edition, Release 5.0 devices have the Openwave icon set preloaded to enable faster rendering. Refer to the `localsrc` image table under the `<img>` element, in the WML reference guide section for the list of images. Be careful using images on cards with a timer element, because the timer may expire before the image is loaded. When delivering a deck that calls images, use a digest so that the image is displayed as soon as the card has finished loading. This loads the deck and image simultaneously. The maximum digest size must be less than the MAX PDU, which is device specific. The minimum size is 2984 bytes.
Internationalization

WML and the Mobile Access Gateway platform support a large variety of languages and character sets. This chapter describes how to make your service globally accessible.

Internationalization requires you to consider two principal issues.

- **Language handling:** The Mobile Access Gateway platform uses existing web standards for language handling. The mobile browser device indicates its language preferences to the Mobile Access Gateway, which relays this information to the WML service by setting the `Accept-Language` HTTP header. It is up to the WML service to check the header and to provide content localized for the specified language.

- **Character set handling:** The Mobile Access Gateway platform follows existing web standards for character set handling; it also removes most of the burden of character set handling from WML services. When the Mobile Access Gateway relays data between a mobile browser device and a WML service, it attempts to deliver the data in a character set that each one understands. If the mobile browser device and the WML service use different character sets, the Mobile Access Gateway transcodes the data, mapping it from one character set to the other. To enable the Mobile Access Gateway to transcode data correctly, your service must correctly set the appropriate HTTP headers and WML attributes.

**IMPORTANT** Correct character set handling is important for all applications. Although Mobile Access Gateway subscribers may use the same language as your service, different mobile browser devices may use different character sets. If you do not use the appropriate HTTP headers and WML attributes, some mobile browser devices, even in your own locale, may not be able to access your service.

**Providing WML Content in Multiple Languages**

When a mobile browser device generates a request, it indicates to the Mobile Access Gateway the languages in which it prefers to receive content. The Mobile Access Gateway relays this information by setting the `Accept-Language` header in the HTTP requests that it submits to WML services. Normally the web server that hosts the WML service converts this header to an environment variable named `HTTP_ACCEPT_LANGUAGE`. If you want your WML service to provide localized content, you should check this environment variable and provide the localized content accordingly.
The Accept-Language HTTP header and the HTTP_ACCEPT_LANGUAGE environment variable specify comma-separated lists of language-country identifiers (as defined in ISO 639) and two-character country codes (as defined in ISO 3166). For example, the identifier en-us specifies that the mobile browser device prefers content in U.S. English. You can find a list of these codes at the following URL:

http://www.unicode.org

The header lists language-country identifiers in order of preference. Therefore, if the HTTP_ACCEPT_LANGUAGE environment variable specifies a list of identifiers, you should attempt to provide content according to the first possible identifier in the list.

### Handling Different Character Sets

To ensure that the Mobile Access Gateway correctly transcodes data exchanged between your service and a mobile browser device, your service must do the following:

- Set the charset parameter in the HTTP header of each entity that contains text that will be displayed by the mobile browser device—that is, for each WML deck and each alert.
- Use the method="POST" attribute and postdata nested element in all tasks that return data.
- Use the accept-charset attribute whenever you pass user data to another application that supports a different character set than your own.

### Setting the charset Parameter

The charset parameter, an extension of the HTTP Content-type header, specifies the character set used in the content of a message entity. For example, an HTTP response containing a WML deck that uses the UTF-8 character set appears as follows:

```
Content-type: multipart/mixed;boundary="-0"
---0
Content-type: text/vnd.wap.wml; charset=utf-8

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.3//EN"
     "http://www.wapforum.org/DTD/wml13.dtd">
<wml>
<card>
<p>
    This is a utf-8 deck.
</p>
</card>
</wml>
---0--
```
INTERNATIONALIZATION

Handling Different Character Sets

IMPORTANT  If the character encoding is not UTF-8 or UTF-16, you must use the XML declaration. Otherwise, it is not a well-formed XML document. Since XML headers can be configured directly in the content, you can avoid changes to the web server or application scripts. For example:

```xml
<?xml encoding="Shift_JIS"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.3//EN"
 "http://www.wapforum.org/DTD/wml13.dtd">
<wml>
...
</wml>
```

Configuring Your Web Server

To set the charset parameter in static WML decks served by your web server, configure it as follows:

<table>
<thead>
<tr>
<th>Content</th>
<th>MIME type</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>WML</td>
<td>text/vnd.wap.wml; charset=charset_name</td>
<td>wml</td>
</tr>
</tbody>
</table>

where `charset_name` is the name of the character set you are using (for example, utf-8). For a list of supported character set names, see “Character Set Names” on page 111.

If you need to provide static decks that use different character sets, consider adding a unique filename extension to the decks that use each character set and configuring your web server to handle the different extensions. For example, if you want to serve decks that use UTF-8 and Shift-JIS, you could add wml-utf8 extensions to the UTF-8 decks and wml-sjis extensions to the Shift-JIS decks, and then configure your web server as follows:

<table>
<thead>
<tr>
<th>Content</th>
<th>MIME type</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>WML</td>
<td>text/vnd.wap.wml; charset=UTF-8</td>
<td>wml-utf8</td>
</tr>
<tr>
<td>WML</td>
<td>text/vnd.wap.wml; charset=Shift_JIS</td>
<td>wml-sjis</td>
</tr>
</tbody>
</table>

Using method="POST" and POSTDATA

Although the Mobile Access Gateway (4x and later) does transcode data passed in CGI query strings, it's a good idea to always use the method="POST" attribute and the nested `<postfield>` element when you pass user-entered data using the `<go>` task.
The following card demonstrates the correct way to return user-entered data:

```
<wml>
  <card>
    <p>
      <do type="accept">
        <go href="http://foo.com/news.cgi" method="POST"
            postfield="$(company)"/>
      </do>
      Enter company name:
      <input name="company"/>
    </p>
  </card>
</wml>
```

To specify multiple arguments for the POSTDATA attribute, delimit them with ampersands. For more information on the method attribute and `<postfield>` element, see the `<go>` element in the *WML 1.3 Language Reference*.

**Specifying Bookmarks**

The mobile browser device uses URLs to identify bookmarked cards or decks. Figure 9-1 illustrates the interface created by the deck in the previous section.

**Figure 9-1. HTTP request and response without `<postfield>` information**

```
<wml>
  <card>
    <p>
      ABC reported excellent earnings last quarter.
    </p>
  </card>
</wml>
```

HTTP POST:
http://foo.com/news.cgi
method=POST
<postfield name="ABC"
If users bookmark the card shown at the bottom of Figure 9-1, the bookmark uses the generic URL http://foo.com/news.cgi (without the user-entered <postfield>) by default. To make sure that users can return to the specific card for abc (for example), you should include a <meta> statement in the deck <head> that specifies the exact URL to use as the bookmark. For example, rather than returning the simple WML deck shown above, your application should return code like the following:

```xml
<wml>
<head>
  <meta name="vnd.up.markable" forua="true"/>
  <meta name="vnd.up.bookmark" content="http://foo.com/news.cgi?name=ABC"/>
</head>
<card title="ABC">
  ABC Company reported excellent earnings last quarter.
</card>
</wml>
```

**NOTE** Because bookmarks automatically use the HTTP GET method, your application must also handle user-entered data passed as CGI variables.

### Specifying Cache Information

Like bookmarks, the mobile browser device uses URLs to identify cards or decks in cache. Unless otherwise specified, the device cache uses the generic URL (without the user-entered <postfield>) by default. In the previous example, the URL http://foo.com/news.cgi represents the ABC card in the cache. If the user requests information for a different company (say, for example, DEF Company), that URL no longer references ABC in the cache but points to DEF instead.

To uniquely identify the ABC card (for example), you should include the <postfield> element as part of the URL in the original request. The <postfield> values are not transcoded or displayed, but they do uniquely identify the resulting deck. Your application should generate the request with code like the following:

```xml
<wml>
  <card>
    <do type="accept">
      <go href="http://foo.com/news.cgi?cache_id=$(company)" method="POST">
        <postfield name="$(company)"/>
      </go>
    </do>
    <p>Enter company name:
      <input name="company"/>
    </p>
  </card>
</wml>
```
This code causes the mobile browser device to cache the deck separately from other results generated using the same script, thereby allowing you to selectively remove it from the cache using a cache operation. To avoid confusing the URL request with a bookmark URL (see the previous section), you should specify a different query string argument (cache_id in this example) than for the <postfield> argument (name).

Using the accept-charset Attribute

When the Mobile Access Gateway transcodes “post” data for your application, it does so based on the HTTP headers that you specify. For example, if the HTTP header for a deck with an input field specifies the UTF-8 character set, the Mobile Access Gateway returns the user entry in UTF-8 characters.

If the HTTP deck header does not specify a character set, the Mobile Access Gateway uses the accept-charset attribute for the <go> task to determine the character set your application expects. You should set the accept-charset option whenever you pass user data to an application that supports a different character set than your own. You do not need to use this option when you pass data to your own application, because your content-type header already provides this information.

**IMPORTANT** If the accept-charset attribute and the HTTP header specify different character sets, the accept-charset attribute overrides the HTTP header.

For example, if the foo.bar service uses the UTF-8 character set, the WML deck in the previous section would be more correctly written as follows:

```wml
<wml>
  <card>
    <do type="accept">
      <go href="http://foo.com/news.cgi?cache_id=$(company)"
          method="POST" accept-charset="UTF-8">
        <postfield name="$(company)"/>
      </go>
    </do>
    <p>
      Enter company name:
      <input name="company"/>
    </p>
  </card>
</wml>
```

In this case, the Mobile Access Gateway transcodes the user-entered company value for the name argument but not for the cache_id argument, because it is passed as a CGI variable rather than as a <postfield> value.
Testing Your Service with Different Character Sets

You can simulate mobile browser devices that use a wide variety of character sets and languages when you run the Openwave Mobile Browser Simulator in Mobile Access Gateway mode. Because HTTP direct mode does not go through a Mobile Access Gateway, and therefore cannot perform transcoding, you should configure the Openwave Mobile Browser Simulator to use the same character set as your service when you use this mode. For more information about configuring the Openwave Mobile Browser Simulator for different locales and running the it in different modes, see the Tools and API Reference.

Character Set Names

The following table lists character set names that you can specify for the `charset` parameter in HTTP headers and the `accept-charset` attribute of `<go>` element. The Mobile Access Gateway supports a number of aliases for character set names, but you should use the names in the following list to ensure correct character set handling.

<table>
<thead>
<tr>
<th>Character set name</th>
<th>Character set name</th>
</tr>
</thead>
<tbody>
<tr>
<td>US-ASCII</td>
<td>Big5</td>
</tr>
<tr>
<td>ISO-8859-1</td>
<td>IBM037</td>
</tr>
<tr>
<td>ISO-8859-2</td>
<td>IBM273</td>
</tr>
<tr>
<td>ISO-8859-3</td>
<td>IBM280</td>
</tr>
<tr>
<td>ISO-8859-4</td>
<td>IBM285</td>
</tr>
<tr>
<td>ISO-8859-5</td>
<td>IBM297</td>
</tr>
<tr>
<td>ISO-8859-6</td>
<td>windows-1250</td>
</tr>
<tr>
<td>ISO-8859-7</td>
<td>windows-1251</td>
</tr>
<tr>
<td>ISO-8859-8</td>
<td>windows-1252</td>
</tr>
<tr>
<td>ISO-8859-9</td>
<td>windows-1253</td>
</tr>
<tr>
<td>latin6</td>
<td>windows-1254</td>
</tr>
<tr>
<td>SHIFT_JIS</td>
<td>windows-1255</td>
</tr>
<tr>
<td>EUC-JP</td>
<td>windows-1255</td>
</tr>
<tr>
<td>KS_C_5601-1987</td>
<td>windows-1256</td>
</tr>
<tr>
<td>UTF-8</td>
<td>windows-1257</td>
</tr>
<tr>
<td>GB2312</td>
<td></td>
</tr>
</tbody>
</table>
# Index

## Symbols

- $\$$ (dollar sign) 28
- & (ampersand) 28
- &apos; (apostrophe) 28
- &gt; (greater than) 28
- &lt; (less than) 28
- &nbsp; (non-breaking space) 28
- &quot; (quote) 28
- &shy; (soft hyphen) 28
- > 11

## A

- ACCEPT-charset attribute (go element) 21
- accept-charset attribute (go element) 110, 111
- access control
  - defining 43
  - error message 45
- Access Control Error 45
- access element 43
- actions, deck versus card level 9
- activities
  - tasks for 33
- address book services 82
  - adding addresses 84
  - selecting addresses 82
- alerts
  - in notifications 60
  - MIME type 50
- ampersand character (&), specifying in text 28
- anchor element 11
- apostrophe character (‘), specifying in text 28
- application/x-up-alert (MIME type) 50
- application/x-up-cacheop (MIME type) 50
- asynchronous communication 59
- attributes, syntax for 3

## B

- bookmarks 46
  - using CGI variables 108

## C

- cache, UP.Phone 47, 53, 109
- cache operations
  - in notifications 61
  - MIME type 50
- call entries, adding to To Do service 87
- calls (voice), initiating 53
- card element 2, 46
- cards
  - navigating between 3
  - using multiple 7
- case
  - in variable names 22
- certificates
  - and notifications 72
  - requesting from Verisign 72
- CGI variables
  - escaping ampersands 5
  - in bookmarks 108
  - returning query data 24
- character elements 5, 28
- character set
  - specifying for message entity 106
  - supported names 111
  - transcoding 24, 105, 107, 110, 111
- WML 3
- choice lists (see selection lists)
- communication
  - asynchronous 59
  - with other services 79
- content types (see message entities)
- context, implementing in WML 33
## Index

contexts
  - and history stack 36
  - and variable scoping 37
child 32
defined 79
definition 31
handling context return events 37
individual 32
parent 36
passing information between 39
passing variables 40
returning values 40
contexts, navigating between 33

data
  - passing user-entered information 107
  - returning to your application 24
decks
  - max-age (time to live) 53
  - navigating between 10
  - size limits 10
digests 2
  - format for 52
  - images 57
  - MIME type 50
  - using to tune performance 51
display text 26, 28
do element 4, 7, 12
dollar sign character ($), specifying in text 28
DOMAIN attribute (ACCESS element) 43
dynamic services, creating 49

E
email services 80
escaping, special characters 5, 28
Example
  - An e-commerce transaction 41
Extensible Markup Language 1
Extensions to WML 31
extensions to WML 31

F
fieldset element 2
format specifiers 14

G
go element 4, 19, 24, 53, 107, 110
greater than character (>)
H
  - specifying in text 28
head element 2
history stack 5
  - and contexts 36
HTTP_ACCEPT_LANGUAGE environment variable 105
HTTP status codes, for notifications 69
I
  - IDEFAULT attribute (SELECT element) 18
images
  - packaging multiple 57
  - supported formats 56
  - img element 55
INPUT element 12
input element 2
intrinsic events 19–21, 23, 48
  - onenterforward 23
  - onpick 19
K
  - KEY attribute (SELECT element) 18
  - latency
    - putting images in digests 57
    - retrieving decks from cache versus server 47
less than character (<), specifying in text 28
line breaks
  - in display text 26
  - in WML code 3
link 11
lists, for user selection 18

M
message entities
  - MIME types 50
  - specifying character set 106
meta element 53
method attribute (go element) 24, 107
MIME types (see message entities)
multipart/mixed (MIME type) 50

N
newline characters, in WML code 3
non-breaking space character, specifying in text 28
notifications
  applications for 66
  HTTP status codes 69
  pending 65
  pull channel 62, 64
  push channel 59, 62, 63
testing with UP.Simulator 70
types 62
  viewing status 70

O
onenterforward intrinsic event 23
onevent element 18, 19
onpick intrinsic event 19
ontimer attribute (card element) 48
Openwave extensions 31
option element 19

P
PATH attribute (ACCESS element) 43
pending notifications 70
phone calls, initiating 53
postfield attribute (go element) 21, 24
postfield element (go element) 107
postfieldelement (go element) 109
provisioning Web pages, UP.Link server 70
pull notifications 62, 64
push notifications 59, 62, 63

Q
query arguments, using variables 24
quote character ("), specifying in text 28

R
RC5 cipher algorithm 75
refresh element 23
registered services
  address book 82
  email 80
  issuing requests to 80
  providing 89
  security 91
  To Do 87
RSA Data Security 75

S
scrolling, text 27
security 43, 75
  common risks 77
  registered services 91
  RSA algorithms 75
SELECT element 18
select element 2
selection lists 18
  specifying task items 19
SENDREFERER attribute (go element) 76
sendreferer attribute (go element) 21
Service Registry, UP.Link server 79
services (see registered services)
  shadowing 9
  size limits, for decks or digests 10
  soft (discretionary) hyphen, specifying in text 28
  spaces, in WML code 3
special characters
  specifying in text 28
  specifying in URLs 5
syntax, overview of 1

T
tabs
  in display text 28
  in WML code 3
task entries, adding to To Do service 88
template element 2, 7
text
  alignment 28
  scrolling and wrapping 27
  using special characters 28
text/vnd.wap.wml (MIME type) 50
text/vnd.wap.wmlscript (MIME type) 50
text/x-wap.wml (MIME type) 107
text entry 12
  adding automatic characters 15
  format specifiers 14
  making optional 15
timer element 48
Time to Live (TTL)
  decks 53
  notifications 61, 63, 64
title attribute
  in card element 46
To Do services 87
  adding call entries 87
  adding task entries 88
transcoding 24, 105, 107, 110, 111
Index

U
UP.Link services
  interacting with  79
  Messenger  71
  provisioning Web pages  70
UP.Mail, issuing requests to  80
UP.Organizer
  adding addresses  84
  adding call entries  87
  adding task entries  88
  issuing requests to  82, 87
  selecting addresses  82
UP.Phone
  cache  47, 53, 109
  history stack  5
UP.Simulator
  testing different character sets  111
  testing notifications  70
URLs, specifying  4
user input, handling  12
user interface guidelines  93
using<User Type>exit  33
using<User Type>spawn  33
using<User Type>throw  33

V
variables
  as CGI arguments  24
  CGI arguments in bookmarks  108
  initializing  23
  naming  22
  passing to a child context  40
  referencing  24
  scoping  37
  setting explicitly  22
  setting from a child context  40
Verisign, requesting certificates from  72
voice calls, initiating  53

W
WAP Forum  1
white space, in WML code  3
WML
  character set  3
  generating content dynamically  49
  MIME type  50
  sending multiple decks  51
  syntax overview  1
wml.element  2
WMLScript
  MIME type  50
WML service, designing  32
  wrapping, text  27
WTAI (Wireless Telephony Application Interface)  53

X
XML  1