WebSphere Voice Server for Multiplatforms

VoiceXML Programmer's Guide

WebSphere Voice Server for Multiplatforms

VoiceXML Programmer's Guide

Note

Before using this information and the product it supports, be sure to read the information in "Notices" on page 185

Sixth Edition (November 2005)

This edition applies to version 5 release 1 of IBM® WebSphere® Voice Server for Multiplatforms and to all subsequent releases and modifications until otherwise indicated in new editions. IBM may publish one or more new editions of this publication in a downloadable format after the program is generally available. To obtain the most recent edition of this publication, go to the Web site at
http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi

Contents

List of figures vii

List of tables ix

About this book xi
Who should read this book xi
Related publications xi
Specifications and standards. xi
Speech user-interface design xii
Server-side programming xii
Deployment information xiii
How this book is organized xiv
Document conventions and terminology . . xv
Making comments on this book xv

Chapter 1. Introduction to VoiceXML . . . 1
What are voice applications? 1
Why create voice applications? 2
What are typical types of voice applications? . 2
Queries 2
Transactions 3
What is VoiceXML? 3
What are the advantages of VoiceXML? . . . 4
How do you create and deploy a VoiceXML application? 5
How do users access the deployed application? 6

Chapter 2. Designing a SUI (SUI). 9
Introduction 9
The importance of SUI design 10
The bases of SUI design 10
The consumers of SUI design 10
e-Service and speech technology 10
Customer satisfaction with e-Service . . . 11
Service recovery. 12
SUI misconceptions 12
Fundamental SUI design 12
Major SUI objectives 12
The power of the SUI 13
Design methodology 13
Design Phase. 13
Prototype phase (“Wizard of Oz” testing) 16
Test phase. 18
Refinement phase 19
Getting started—high-level design decisions 19
Selecting an appropriate user interface . . 20
Deciding on the type and level of information 20
Choosing the barge-in style 21
Selecting recorded prompts or synthesized speech 24
Deciding whether to use audio formatting 27
Using simple or natural command grammars. 29
Adopting a terse or verbose prompt style 31
Allowing only speech input or speech plus DTMF 32
Adopting a consistent set of global navigation commands. 34
Deciding whether to use human agents in the deployed system 39
Choosing help mode or self-revealing help 40
Getting specific—low-level design decisions 43
Adopting a consistent “sound and feel”. . 43
Using consistent timing 44
Designing consistent dialogs 45
Creating introductions 46
Constructing appropriate menus and prompts 48
Designing and using grammars 59
Error recovery and confirming user input 64
Advanced user interface topics. 71
Issues in artificial personae 71
Controlling the “lost in space” problem . . 72
Managing audio lists 72

Chapter 3. VoiceXML language 75
Changes from VoiceXML 2.0 75
New elements and attributes 75
Speech Synthesis Markup Language (SSML). 76
The structure of a VoiceXML application . . 76
Forms and form items 76
Menus 77
Flow control 78
Subdialogs 79
Comments 79
A simple VoiceXML example 80
Static Content 81

Dynamic content 81
VoiceXML elements and attributes 82
Speech Synthesis Markup Language (SSML) 90
SSML elements and attributes 90
Built-in field types and grammars. 92
Recorded audio 95
Using prerecorded audio files 95
Recording spoken user input 95
Playing and storing recorded user input . 95
Recording user input during speech recognition 95
Document fetching and caching 96
Controlling fetch and cache behavior. . . 96
Preventing caching. 96
Events 96
Predefined events 97
Application-specific events 99
Recurring events 99
Variables and expressions 99
Using ECMAScript. 99
Declaring variables 100
Assigning and referencing variables. . . 102
Using shadow variables. 102
Grammars 104
Grammar syntax 105
Static grammars 107
Dynamic grammars 109
Grammar scope 109
Hierarchy of active grammars. 110
Mixed-initiative application and form-level grammars. 111
Specifying a sounds-like spelling in a Japanese, a Cantonese, or a Simplified Chinese grammar 114
Timeout properties 115
Incompletetimeout 115
Completetimeout 116
Example 116
Telephony functionality 117
Automatic Number Identification . . . 117
Dialed Number Identification Service . . 118
Call transfer. 119

Chapter 4. Hints, tips, and best practices 123
VoiceXML application structure 123
Deciding how to group dialogs 123
Deciding where to define grammars . . 124
Fetching and caching resources for improved performance 124
Invoking a State Table using Voice XML . . 125
Confidence-level processing 127
Using a proxy server. 127
Testing built-in field types 127
Sample code 129
Calling a Java application 129
Calling legacy telephony applications . . 133
Using *n*-best 134

Appendix A. CTTS caching 137

Appendix B. Canadian French 139
Built-in field types and grammars 139
Predefined events. 144
Built-in commands 145
Specifying character encoding. 145
Testing built-in field types 145
SSML elements and attributes. 146

Appendix C. German 149
Built-in field types and grammars 149
Predefined events. 153
Built-in commands 154
Specifying character encoding. 154
Testing built-in field types 155

Appendix D. Japanese 157
Built-in field types and grammars 157
Predefined events. 160
Built-in commands 160
Specifying character encoding. 160
Testing built-in field types 160
SSML elements and attributes. 162

Appendix E. Korean 163
Built-in field types and grammars 163
Predefined events. 167
Built-in commands 168
Specifying character encoding. 168
Testing built-in field types 168
SSML elements and attributes. 170

Appendix F. Simplified Chinese 171
Built-in field types and grammars 171
Predefined events. 174
Built-in commands 174
Specifying character encoding. 175
Testing built-in field types 175
SSML elements and attributes. 176

Appendix G. UK English 177
Built-in field types and grammars 177
Testing built-in field types 183

Notices 185
Trademarks 187
Accessibility 188
Other attributions 188

Glossary 189

Index 195

List of figures

1. Spoke-too-soon (STS) incident 66
2. Spoke-way-too-soon (SWTS) incident 66
3. Conference 120

List of tables

1. Sample script for "Wizard of Oz" testing 16
2. When to use a speech interface 20
3. Barge-in Enabled versus Disabled 21
4. Barge-in detection methods 22
5. Audio formatting 28
6. Simple versus natural command grammars 29
7. Prompt styles. 31
8. Mixed input modes. 33
9. Recommended list of global command types 35
10. Use of human agents 39
11. Comparison of help styles 40
12. Successful use of example 46
13. Unsuccessful use of example—switch to directed dialog 46
14. Recommended maximum number of menu items 49
15. Recognition errors when spelling 56
16. Grammar word/phrase length trade-offs 60
17. Vocabulary robustness and grammar complexity trade-offs 60
18. Number of active grammar trade-offs 61
19. Error-recovery techniques. 64
20. Summary of VoiceXML elements and attributes 82
21. Summary of SSML elements and attributes 90
22. Built-in types for US English 92
23. Properties for capturing audio during speech recognition 95
24. Predefined events and event handlers for US English 97
25. Variable scope 100
26. Examples of relational operators 102
27. Shadow variables 103
28. Speech Recognition Grammar limitations in WebSphere Voice Server . 104
29. Form grammar scope 110
30. Incompletetimeout. 116
31. Completetimeout 116
32. Sample input for US English built-in field types 128
33. Canadian French built-in types 139
34. Canadian French predefined events and event-handler messages 145
35. Canadian French built-in VoiceXML browser commands 145
36. Sample input for Canadian French built-in field types. 146
37. Limitations for Canadian French SSML elements 147
38. German built-in types 149
39. German predefined events and event-handler messages 153
40. German built-in VoiceXML browser commands 154
41. Sample input for German built-in field types 155
42. Japanese built-in types 157
43. Japanese predefined events and event-handler messages 160
44. Japanese built-in VoiceXML browser commands 160
45. Sample input for Japanese built-in field types 161
46. Limitations for Japanese SSML elements 162
47. Korean built-in field types 164
48. Korean predefined events and event-handler messages 168
49. Korean built-in VoiceXML browser commands 168
50. Sample input for Korean built-in field types 169
51. Limitations for Korean SSML elements 170
52. Simplified Chinese built-in types 171
53. Simplified Chinese predefined events and event-handler messages 174
54. Simplified Chinese built-in VoiceXML browser commands 175
55. Sample input for Simplified Chinese built-in field types. 175
56. Limitations for Simplified Chinese SSML elements 176
57. UK English built-in types 177
58. Sample input for UK English built-in field types 183

About this book

This book provides information about using VoiceXML Version 2.0 and 2.1 to design and develop voice applications. The resulting applications can then be deployed in a telephony environment using IBM® WebSphere® Voice Server to provide voice access to Web-based data using standard telephones.

Who should read this book

Read this book if you are:

- Someone who wants to find out about the advantages of using VoiceXML to deliver Web-based voice services
- An application developer interested in creating VoiceXML applications
- A content creator responsible for the creative aspects of VoiceXML applications

Related publications

Reference, design, and programming information for creating voice applications is available from a variety of sources, as represented by the documents listed in this section.

Note: Guidelines and publications cited in this book are for your information only and do not in any manner serve as an endorsement of those materials. You alone are responsible for determining the suitability and applicability of this information to your needs.

Specifications and standards

You may want to refer to the following sources for information about relevant specifications and standards:

- *Voice Extensible Markup Language (VoiceXML) Version 2.0* specification, published by W3C and available at **http://www.w3.org/TR/voicexml20/**
- *Speech Synthesis Markup Language Version 1.0* specification, published by W3C and available at **http://www.w3.org/TR/speech-synthesis/**
- *Speech Recognition Grammar Specification (SRGS) Version 1.0*, published by W3C and available at **http://www.w3.org/TR/speech-grammar/**
- *Semantic Interpretation for Speech Recognition W3C Working Draft 1 April 2003*, published by W3C and available at **http://www.w3.org/TR/semantic-interpretation/**
- *HTTP 1.1 Specification* available at **http://www.ietf.org/rfc/rfc2616.txt**

- *HTTP State Management Mechanism (Cookie Specification)* available at **http://www.w3.org/Protocols/rfc2109/rfc2109**
- *ECMA Standard 262: ECMAScript Language Specification, 3rd Edition*, published by ECMA at **http://www.ecma.ch/ecma1/stand/ECMA-262.htm**
- *The International Phonetic Alphabet*, published by the International Phonetic Association at **http://www2.arts.gla.ac.uk/IPA/ipachart.html**
- *The Unicode Standard Version 3.0*, published by The Unicode Consortium at **http://www.unicode.org**

Speech user-interface design

The speech user interface guidelines presented in this book are an evolving set of recommendations based on industry research and lessons learned in the process of developing our own VoiceXML and telephony applications. For more information, refer to speech industry literature and publications such as the following sources:

- *Audio System for Technical Readings (ASTeR)* by T. V. Raman, a Ph.D. Thesis published by Cornell University, May 1994.
- *Auditory User Interfaces—Towards The Speaking Computer* by T. V. Raman, published by Kluwer Academic Publishers, August 1997.
- "Directing the Dialog: The Art of IVR" by Myra Hambleton, published in *Speech Technology*, Feb/Mar 2000.
- *Handbook of Human-Computer Interaction* by Thomas K Landauer, Martin Helander, and Prasad V. Prabhu, published by Elsevier Science, Amsterdam, North-Holland, June 1997.
- *How to Build a Speech Recognition Application: A Style Guide for Telephony Dialogues* (Second Edition) by Bruce Balentine, David P. Morgan, and William S. Meisel, published by Enterprise Integration Group, San Ramon, CA, 2001.
- *Human Factors and Voice Interactive Systems* by Daryle Gardner-Bonneau, published by Kluwer Academic Publishers, Boston, MA, March 1999.

Server-side programming

Information about server-side programming is available from a number of sources, including the following:

- *Building Blocks for CGI Scripts in Perl* at **http://www.cc.ukans.edu/~acs/docs/other/cgi-with-perl.shtml**
- *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server* (IBM Redbook) SG24-5754-00
- *Developing an e-business Application for the IBM WebSphere Application Server* (IBM Redpiece) SG24-5423-00
- *JavaServer Pages (JSP)* at **http://java.sun.com/products/jsp**
- *Java Servlet* at **http://java.sun.com/products/servlet/**
- *Servlet/JSP/EJB Design and Implementation Guide* (IBM Redbook) SG24-5754-00

- *The ASP (Active Server Pages) Resource Index* at **http://www.aspin.com/index/default.asp**
- *The Front of IBM WebSphere Building e-business User Interfaces* (IBM Redbook) SG24-5488-00
- *WebSphere V4.0 Advanced Edition Handbook* (IBM Redbook) SG24-6176-00
- *WebSphere Version 4 Application Development Handbook* (IBM Redbook) SG24-6134-00

Deployment information

For information about deploying your voice applications, refer to the documentation provided with WebSphere Voice Server for Multiplatforms and WebSphere Voice Response for AIX®.

The following documentation is provided in softcopy only with the product, or can be downloaded from the IBM Publications Center:

- *IBM Text-to-Speech SSML Programming Guide* Version 6.7.3.0

WebSphere Voice Server for Multiplatforms

- *WebSphere Voice Server for Multiplatforms: Administrator's Guide*, G210-1561
- *WebSphere Voice Server for Multiplatforms: Application Development using State Tables*, G210-1562

WebSphere Voice Response for AIX

- *WebSphere Voice Response for AIX: General Information and Planning*, GC33-1840
- *WebSphere Voice Response for AIX: Installation*, GC33-1842
- *WebSphere Voice Response for AIX: User Interface Guide*, SC33-1841
- *WebSphere Voice Response for AIX: Configuring the System*, SC33-1843
- *WebSphere Voice Response for AIX: Managing and Monitoring the System*, SC33-1844
- *WebSphere Voice Response for AIX: Designing and Managing State Table Applications*, SC33-1845
- *WebSphere Voice Response for AIX: Application Development using State Tables*, SC33-1846
- *WebSphere Voice Response: Developing Java Applications*, GC34-6318
- *WebSphere Voice Response: Deploying and Managing VoiceXML and Java Applications*, GC34-6319
- *WebSphere Voice Response: Application Development using Java and VoiceXML*, GC34-6049
- *WebSphere Voice Response for AIX: Custom Servers*, SC33-1847
- *WebSphere Voice Response for AIX: 3270 Servers*, SC33-1848
- *WebSphere Voice Response for AIX: Problem Determination*, GC33-1849

- *WebSphere Voice Response for AIX: Fax using Brooktrout*, SC34-5982
- *WebSphere Voice Response for AIX: Cisco ICM Interface User's Guide*, SC34-5317
- *WebSphere Voice Response for AIX: Programming for the ADSI Feature*, SC34-5380
- *WebSphere Voice Response for AIX: Programming for the Signaling Interface*, SC33-1851

How this book is organized

Chapter 1, "Introduction to VoiceXML," on page 1 provides an introduction to voice applications and VoiceXML.

Chapter 2, "Designing a SUI (SUI)," on page 9 presents user-interface guidelines for developing voice applications.

Chapter 3, "VoiceXML language," on page 75 introduces basic concepts and constructs of VoiceXML. For complete syntax, please refer to the VoiceXML 2.0 specification at **http://www.w3.org/TR/voicexml20/**

Chapter 4, "Hints, tips, and best practices," on page 123 contains hints and tips for structuring and coding your VoiceXML applications. It includes examples of the following: use of the VoiceXML **<object>** element to reference Java code; use of the **<property>** element's **maxnbest** name to obtain *n*-best results; and, use of a precompiled grammar.

Appendix A, "CTTS caching," on page 137 describes audio caching for improving the performance of applications that use concatenative text-to-speech (CTTS) synthesis.

You might want to refer to the following appendixes for language-specific information:

- Appendix C, "German," on page 149 contains information that is specific to German.
- Appendix D, "Japanese," on page 157 contains information that is specific to Japanese.
- Appendix F, "Simplified Chinese," on page 171 contains information that is specific to Simplified Chinese.
- Appendix G, "UK English," on page 177 contains information that is specific to UK English.

"Notices" on page 185 contains notices and trademark information.

"Glossary" on page 189 defines key terminology used in this document.

Document conventions and terminology

The following conventions are used to present information in this document:

Italic	Used for emphasis, to indicate variable text, and for references to other documents.
Bold	Used for configuration parameters, file names, URLs, and user-interface controls such as command buttons and menus.
`Monospaced`	Used for sample code.
<text>	Used for editorial comments in scripts.

Making comments on this book

If you especially like or dislike anything about this book, feel free to send us your comments.

You can comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information that is in this book and to the way in which the information is presented. Speak to your IBM representative if you have suggestions about the product itself.

When you send us comments, you grant to IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

You can get your comments to us quickly by sending an e-mail to **idrcf@hursley.ibm.com**. Alternatively, you can mail your comments to:

User Technologies
IBM United Kingdom Laboratories,
Mail Point 095, Hursley Park,
Winchester, Hampshire, SO21 2JN, United Kingdom

Please ensure that you include the book title, order number, and edition date.

Chapter 1. Introduction to VoiceXML

This introduction addresses the following questions:

- "What are voice applications?"
- "Why create voice applications?" on page 2
- "What are typical types of voice applications?" on page 2
- "What is VoiceXML?" on page 3
- "What are the advantages of VoiceXML?" on page 4
- "How do you create and deploy a VoiceXML application?" on page 5
- "How do users access the deployed application?" on page 6

What are voice applications?

Voice applications are applications in which the input and/or output are through a spoken, rather than a graphical, user interface. The application files can reside on the local system, an intranet, or the Internet. Users can access the deployed applications anytime, anywhere, from any telephone.

"Voice-enabling the World Wide Web" does not simply mean:

- Using spoken commands to tell a visual browser to look up a specific Web address or go to a particular bookmark
- Having a visual browser throw away the graphics on a traditional visual Web page and read the rest of the information aloud
- Converting the bold or italics on a visual Web page to some kind of emphasized speech

Rather, voice applications provide an easy and novel way for users to surf or shop on the Internet—"browsing by voice." Users can interact with Web-based data (that is, data available via Web-style architecture such as servlets, ASPs, JSPs, Java Beans, CGI scripts, etc.) using speech rather than a keyboard and mouse.

The form that this spoken data takes is often not identical to the form it takes in a visual interface, due to the inherent differences between the interfaces. For this reason, transcoding—that is, using a tool to automatically convert HTML files to VoiceXML—may not be the most effective way to create voice applications.

Why create voice applications?

Until recently, the World Wide Web has relied exclusively on visual interfaces to deliver information and services to users via computers equipped with a monitor, keyboard, and pointing device. In doing so, a huge potential customer base has been ignored: people who (due to time, location, and/or cost constraints) do not have access to a computer.

Many of these people do, however, have access to a telephone. Providing "conversational access" (that is, spoken input and audio output over a telephone) to Web-based data will permit companies to reach this untapped market. Users benefit from the convenience of using the mobile Internet for self-service transactions, while companies enjoy the Web's relatively low transaction costs. And, unlike applications that rely on dual tone multi-frequency (DTMF) (telephone keypress) input, voice applications can be used in a hands-free or eyes-free environment, as well as by customers with rotary pulse telephone service or telephones in which the keypad is on the handset.

What are typical types of voice applications?

Voice applications will typically fall into one of the following categories:

- "Queries"
- "Transactions" on page 3

Queries

In this scenario, a customer calls into a system to retrieve information from a Web-based infrastructure.

The system guides the customer through a series of menus and forms by playing instructions, prompts, and menu choices using prerecorded audio files or synthesized speech.

The customer uses spoken commands or DTMF input to make menu selections and fill in form fields.

Based on the customer's input, the system locates the appropriate records in a back-end enterprise database. The system presents the desired information to the customer, either by playing back prerecorded audio files or by synthesizing speech based on the data retrieved from the database.

Examples of this type of self-service interaction include applications or voice portals providing weather reports, movie listings, stock quotes, health-care-provider listings, and customer service information (Web call centers).

Transactions

In this scenario, a customer calls into a system to execute specific transactions with a Web-based back-end database.

The system guides the customer to provide the data required for the transaction by playing instructions, prompts, and menu choices using prerecorded audio files or synthesized speech. The customer responds using spoken commands or DTMF input.

Based on the customer's input, the system conducts the transaction and updates the appropriate records in a back-end enterprise database. Typically the system also reports back to the customer, either by playing prerecorded audio files or by synthesizing speech based on the information in the database records.

Examples of this type of self-service interaction include applications or voice portals for employee benefits, employee timecard submission, financial transactions, travel reservations, calendar appointments, electronic relationship management (eRM), sales automation, and order management.

What is VoiceXML?

The Voice eXtensible Markup Language (VoiceXML) is an XML-based markup language for creating distributed voice applications, much as HTML is a markup language for creating distributed visual applications. It is an industry standard defined by the World Wide Web Consortium (W3C) at http://www.w3.org/TR/voicexml21/ .

The VoiceXML language enables Web developers to use a familiar markup style and Web server-side logic to deliver applications for use over telephone lines. The resulting VoiceXML applications can interact with existing back-end business data and logic.

Using VoiceXML, application developers can create Web-based voice applications that users can access by telephone or other pervasive devices.

VoiceXML supports dialogs that feature:

- Recognition of spoken input
- DTMF input
- Recording of spoken input
- Synthesized speech output ("text-to-speech")
- Pre-recorded digitized audio output
- Dialog flow control
- Scoping of input

- Automatic Number Identification (ANI)
- Dialed Number Identification Service (DNIS)
- Call transfer

What are the advantages of VoiceXML?

While you could certainly build voice applications without using a voice markup language and a speech browser (for example, by writing your applications directly to a speech API), using VoiceXML and a VoiceXML browser provide several important capabilities:

- VoiceXML is a markup language that makes building voice applications easier, in the same way that HTML simplifies building visual applications. VoiceXML also reduces the amount of speech expertise that developers need.
- VoiceXML applications can use the same existing back-end business logic as their visual counterparts, enabling voice solutions to be introduced to new markets quickly. Current and long-term development and maintenance costs are minimized by leveraging the Web design skills and infrastructures already present in the enterprise. Customers can benefit from a consistency of experience between voice and visual applications.
- VoiceXML implements a client/server paradigm, where a Web server provides VoiceXML documents that contain dialogs to be interpreted and presented to a user. The user's responses are submitted to the Web server, which responds by providing additional VoiceXML documents, as appropriate. VoiceXML allows you to request documents and submit data to server scripts using Universal Resource Identifiers (URIs). VoiceXML documents can be static, or they can be dynamically generated by CGI scripts, Java Beans, ASPs, JSPs, Java servlets, or other server-side logic.
- Unlike a proprietary Interactive Voice Response (IVR) system, VoiceXML provides an open application development environment that generates portable applications. This makes VoiceXML a cost-effective alternative for providing voice access services.
- Most installed IVR systems today accept input from the telephone keypad only. In contrast, VoiceXML is designed predominantly to accept spoken input, but it can also accept DTMF input, if desired. As a result, VoiceXML helps speed up customer interactions by providing a more natural interface that replaces the traditional, hierarchical IVR menu tree with a streamlined dialog using a flattened command structure.
- VoiceXML directly supports networked and Web-based applications, meaning that a user at one location can access information or an application provided by a server at another geographically or organizationally distant location. This capitalizes on the connectivity and commerce potential of the World Wide Web.

- Using a single VoiceXML browser to interpret streams of markup language originating from multiple locations provides the user with a seamless conversational experience across independent applications. For example, a voice portal application might allow a user to temporarily suspend an airline purchase transaction to interact with a banking application on a different server to check an account balance.
- VoiceXML supports local processing and validation of user input.
- VoiceXML supports playback of prerecorded audio files.
- VoiceXML supports recording of user input. The resulting audio can be played back locally or uploaded to the server for storage, processing, or playback at a later time.
- VoiceXML defines a set of events corresponding to such activities as a user request for help, the failure of a user to respond within a timeout period, and an unrecognized user response. A VoiceXML application can provide catch elements that respond appropriately to a given event for a particular context.
- VoiceXML supports context-specific and tapered help using a system of events and catch elements. Help can be tapered by specifying a count for each event handler, so that different event handlers are executed depending on the number of times that the event has occurred in the specified context. This can be used to provide increasingly more detailed messages each time the user asks for help. For more information, see "Choosing help mode or self-revealing help" on page 40.
- VoiceXML supports subdialogs, which are roughly the equivalent of function or method calls. Subdialogs can be used to provide a disambiguation or confirmation dialog, and to create reusable dialog components. For more information, see "Subdialogs" on page 79.

How do you create and deploy a VoiceXML application?

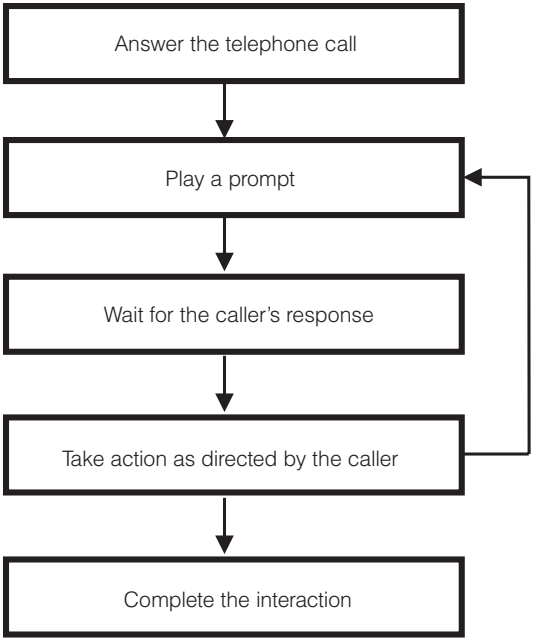
1. An application developer creates a voice application written in VoiceXML. You can write VoiceXML applications using a text editor but you might find it more convenient to use a graphical development environment that helps you create, manage and test VoiceXML files. The Voice Toolkit for WebSphere Studio (Voice Toolkit) supports the development of VoiceXML-based applications.

   VoiceXML pages can be static or may be generated dynamically from CGI scripts, Java Beans, ASPs, JSPs, Java servlets, or other server-side techniques.
2. (Optional) The developer publishes the VoiceXML application (VoiceXML documents, grammar files, any prerecorded audio files, and any server-side logic) to a Web server.

3. (Optional) The developer uses a desktop workstation and the Voice Toolkit to test the VoiceXML application running on the Web server or local disk, pointing the VoiceXML browser to the appropriate starting VoiceXML page.
4. A telephony expert configures the telephony infrastructure for WebSphere Voice Response for AIX. See *WebSphere Voice Response for AIX: Installation* and *WebSphere Voice Response for AIX: Configuring the System* for instructions.
5. The system administrator uses WebSphere Voice Response for AIX to configure, deploy, monitor, and manage a dedicated WebSphere Voice Server system. WebSphere Voice Response's telephone network connection provides the audio channels for the VoiceXML browser.
6. The developer uses a real telephone to test the VoiceXML application using WebSphere Voice Server.

How do users access the deployed application?

Once your voice applications are deployed, users simply dial the telephone number that you provide and are connected to the corresponding voice application. The figure below shows a flow chart of a typical call.



1. A user dials the telephone number you provide. WebSphere Voice Response answers the call and executes the application referenced by the dialed phone number.
2. WebSphere Voice Server plays a greeting to the caller and prompts the caller to indicate what information he or she wants.
   - The application can use prerecorded greetings and prompts, or the application can have the greeting or prompt synthesized from text using the text-to-speech engine.
   - If the application supports barge-in, the caller can interrupt the prompt if he or she already knows what to do.
3. The application waits for the caller's response for a set period of time.
   - The caller can respond either by speaking or by pressing one or more keys on a DTMF telephone keypad, depending on the types of responses expected by the application.
   - If the response does not match the criteria defined by the application (such as the specific word, phrase, or digits), the voice application can prompt the caller to enter the response again, using the same or different wording.
   - If the waiting period has elapsed and the caller has not responded, the application can prompt the caller again, using the same or different wording.
4. The application takes whatever action is appropriate to the caller's response. For example, the application might:
   - Update information in a database
   - Retrieve information from a database and speak it to the caller
   - Store or retrieve a voice message
   - Launch another application
   - Play a help message

   After taking action, the application prompts the caller with what to do next.
5. The caller or the application can terminate the call. For example:
   - The caller can terminate the interaction at any moment by hanging up. WebSphere Voice Response can detect if the caller hangs up and can then disconnect itself.
   - If the application permits it, the caller can use a command to indicate explicitly that the interaction is over (for example, by saying "Exit").
   - If the application has finished running, it can play a closing message and then disconnect.

Chapter 2. Designing a SUI (SUI)

This chapter covers the following topics:

- "Introduction"
- "The importance of SUI design" on page 10
- "Design methodology" on page 13
- "Getting started—high-level design decisions" on page 19
- "Getting specific—low-level design decisions" on page 43
- "Advanced user interface topics" on page 71

Introduction

The SUI guidelines presented here are just that: guidelines. In some cases, the requirements and objectives of particular VoiceXML applications may present valid reasons for overriding certain guidelines. Furthermore, these guidelines address the design points we have found to be the most important in producing speech/audio user interfaces, but are not as comprehensive as those found in a book dedicated to the topic. Finally, keep in mind that while the following guidelines can help you produce a usable application, they do not guarantee usability or user satisfaction; you should plan to conduct usability tests with your application. (See "Design methodology" on page 13.)

Note: The guidelines and referenced publications presented in this book are for your information only, and do not in any manner serve as an endorsement of those materials. You alone are responsible for determining the suitability and applicability of this information to your needs.

The goals of the guidelines presented in this chapter include:

- Helping you create standardized, well-behaved VoiceXML applications
- Reducing development time by teaching current best practices in SUI (SUI) design
- Increasing the usability of the SUI and reducing the end user's learning curve by promoting consistent computer output and predictable user input
- Decreasing the amount of rework required after prototype testing

The importance of SUI design

The bases of SUI design

Effective SUI design draws upon many disciplines. The key scientific disciplines are Psychology, Human-Computer Interaction, Human Factors, Linguistics and Communication Theory. The artistic disciplines of Auditory Design and Writing (especially the techniques of writing dialog) are also very important. Finally, for true craftsmanship in SUI design, there is no substitute for experience and codification of best practices (such as the information provided in this chapter).

The consumers of SUI design

A SUI has many consumers and must simultaneously satisfy many objectives. Among the consumers of SUIs are marketers, service providers, end users, and developers.

Marketers have the responsibility of selling speech applications to service providers. The primary objective of a SUI from a marketer's point of view is to appeal to the targeted service provider, thus helping to make the sale.

Service providers rely on the speech application to help them provide a service to end users. For service providers, the primary objectives of an SUI are to save money and maintain customer contact and satisfaction. To this end, they are also concerned with their corporate image and how their speech applications will fit into their overall branding strategies.

End users call the speech application for the purpose of obtaining a service from the service provider. End users want SUIs that are easy to use, allow efficient task completion, and provide a pleasant user experience.

Developers must write the code that creates the entire speech application, including the SUI. The primary objectives for developers creating SUIs are that the interface be technologically feasible, capable of completion given resource constraints, and require minimal development effort.

e-Service and speech technology

Speech applications are one way to provide customer self-service over electronic networks (the technologies collectively known as e-Service). e-Services must focus on meeting customer needs to increase market share and revenue. Technologies are enablers in e-Service that can enhance customer convenience and support, but applications require carefully designed user interfaces to manage customer expectations.

However, a key element of customer relationship building, human interaction, is absent from most e-Service facilities. Through careful design, speech technologies allow a more conversational, human interaction. Given the

limitations of current technologies, conversations with automated system are very different from conversations between humans. Nonetheless, excellent user interface designs working with current technologies can create the illusion of human-like interaction. To achieve this level of excellence in user interface design, it is important to consider users' interpersonal communication, cognitive, and social skills in the initial design, then to apply usability testing methods throughout the design cycle to tune the interface.

Good e-Service design requires a balance of business needs and end user needs. Some key business needs are:

- Provide information and services.
- Market products and services.
- Build customer loyalty and trust.
- Convey a positive business image to the customer.
- Reduce costs.

In contrast, key customer (end user) goals are:

- Obtain information or perform a task quickly and easily.
- Have a smooth and pleasant interaction.

These goals converge when designers apply knowledge of human conversational behavior to the user interface. Key elements to consider are:

- Organization and callflow
- Language and prompt style
- The system voice (especially loudness, pitch and pitch variation)
- Use of non-speech audio (for example, music and audio logos)
- Social expectation of the service provider role

Customer satisfaction with e-Service

According to recent market research, customers are satisfied with self-service technologies when they:

- Are better than other service alternatives by virtue of saving time, saving money, providing more customer control or avoiding service employees. (68%)
- Do their jobs and perform as intended. (21%)
- Solve an intensified need. (11%)

Customers become dissatisfied when:

- Technology fails. (43%)
- Applications have poor design. (36%)
- Service process fails. (17%)
- The customer makes a mistake. (4%)

Poor user interface design can be the root cause of customer perception of these apparently different problems.

Service recovery

A common factor that runs through customer dissatisfaction is the lack of service recovery in the face of system failure. To avoid customer dissatisfaction, it is important to:

- Use high quality, reliable telephony and speech technology.
- Design for communication breakdown by expecting misrecognitions to occur.
- Reinforce correct user responses by moving forward.
- Use a hierarchy of non-intrusive error messages that foster an impression of moving forward while repairing the communicative breakdown in the most appropriate way.
- Use a user centered design process of iterative evaluation and redesign to quickly identify and avoid user problems.

SUI misconceptions

These statements are *not* necessarily true:

- It is always easy and natural to use SUIs. After all, almost everyone can talk!
- Talking to a computer is just like talking to a person.
- Usable speech applications require the latest and greatest technologies.
- Barge-in is essential. Without barge-in, a SUI is unusable.
- Users hate beeps and other tones in SUIs because they aren't natural.

Fundamental SUI design

SUI design is *not* just reading a visual web page. You must decide:

- What to present.
- How much to present.
- How to present it.
- When to present it.

Effective SUI design is based on:

- Understanding customer profiles.
- Meeting realistic expectations.
- Following a design methodology that uses proven techniques.

Major SUI objectives

In their influential book on SUIs, Balentine and Morgan stated that the main enemy of the spoken user interface is time. The basis for this assertion is that speech has a temporary existence and listeners must remember what they have heard. If prompts in a speech application are too short, however, they

can be subject to multiple interpretations. A clear design objective, therefore, is to avoid making users hear more (or less) than they need to hear or to say more (or less) than they need to say.

It is also important to strive to make every interaction move the user forward (or at least create the illusion of moving forward). This is easier said than done because dialogs need careful crafting and usability evaluation. To work toward these objectives, use prompts that are succinct and sincere (modeled after the prompts provided by expert call center agents), provide self-revealing contextual help messages, and use a professional voice talent for recorded prompts.

The power of the SUI

A good SUI has a natural, human-like quality. There is no need to associate a function with a number (in contrast to touchtone user interfaces) when speech labels provide good functional descriptions. System prompts become much shorter and more natural, and it is possible to add options in the future without any need to change existing speech labels, even if the order of functions changes. Finally, there is no need for a user to move the phone away from his or her ear to find the button to press.

A Harris survey conducted in 2003 supports the claim of the effectiveness of speech applications with the following results:

- Speech is widely used and accepted (only 7% of respondents in the survey would avoid future use of speech systems).
- Consumers reported high satisfaction with speech experiences (61% highly satisfied with most recent speech interaction).
- Consumers feel that speech provides many advantages over other e-Service methods (90% of respondents preferred speech to touchtone systems).

Design methodology

Developing SUIs, like most development activities, involves an iterative 4-phase process:

- "Design Phase"
- "Prototype phase ("Wizard of Oz" testing)" on page 16
- "Test phase" on page 18
- "Refinement phase" on page 19

Because this process is iterative, you should attempt to keep the interface as fluid as possible for as long as possible.

Design Phase

In this phase, the goal is to define proposed functionality and create an initial design. This involves the following tasks:

- "Analyzing your users"
- "Analyzing user tasks"
- "Developing the conceptual design (vision clips)" on page 15
- "Making high-level decisions" on page 15
- "Making low-level decisions" on page 15
- "Defining the complete callflow" on page 15
- "Creating the initial dialog script" on page 16
- "Planning for expert users" on page 16

Analyzing your users

The first step in designing your VoiceXML applications should be to conduct user analysis to identify any user characteristics and requirements that might influence application design. For example:

- How frequently will your users use the system?
- What is their motivation for using the system?
- In what type of environment will your users use the system (quiet office, outdoors, noisy shopping mall)?
- What type of telephone connection will most of your users have (land-line, cordless, cellular)?
- Are many of your intended users non-native speakers of the language in which the application will be written?
- How comfortable are your users with automated ("self-service") applications?
- Will your application be personalized (based on ANI information or user login)?

Analyzing user tasks

After you have identified who your users are, the next step is to determine what tasks your application should support.

Consider:

- What are the most common tasks your users will perform? What tasks are less common?
- Are your users familiar with the tasks they will need to complete?
- Will your users be able to perform these tasks by other means (in person, using a visual Web interface, by calling a customer service representative, etc.)?
- Will your users have the option of transferring to a human operator?
- What words and phrases do your users typically use to describe the tasks and items in your proposed application?

For example, tasks in a banking application could include transferring money, obtaining current account balance, listing some number of most recent transactions, etc. You might allocate the functions as follows:

- Have the application locate, sort, and store account information, and make any routine decisions. For example, if the user attempts to transfer more money than is available, the application plays a warning message.
- Have the user confirm transactions and make any non-routine or elective decisions. For example, ask the user to confirm the amount of money and account number before the application submits a form that initiates a monetary transfer.

Developing the conceptual design (vision clips)

After identifying the tasks, the first design activity should be the conceptual development of the application. This is usually done by writing high-level scripts of proposed user-system interactions called *vision clips*. There is no attempt made to completely define the callflow. Rather, the focus is on the user-system interaction in key parts of important tasks. This design activity provides an inexpensive first step prior to any large investment of resource.

In a vision clip, designers create preliminary samples of conversations to promote discussions between the designer and the customer regarding the customer's task and user interface expectations. Ideally, these scripts are recorded so customers can review the sound and feel of the proposed interactions. Designers of vision clips should be very familiar with the capabilities of the speech technologies to avoid preparing vision clips that would be difficult or impossible to deploy as applications.

Making high-level decisions

The next step is to make high-level application decisions, such as selecting the appropriate user interface, barge-in style, prompt style, and help style. For details, see "Getting started—high-level design decisions" on page 19.

Making low-level decisions

After you have made the high-level decisions, you should proceed to the lower-level system decisions that address such issues as sound and feel, word choice, etc. "Getting specific—low-level design decisions" on page 43 provides information to help you make these decisions.

Defining the complete callflow

Next, you will want to outline the callflow that maps the interaction between your application and the user. For example:

- What questions do you need to ask the user?
- When the user answers, how should the application respond?

Your application interaction should have a logical progression that takes into account typical responses, unusual responses, and any error conditions that might occur.

Creating the initial dialog script

After you have defined the callflow, you should be ready to create an initial draft of the script for the dialog between the application and the user. The script should include all of the text that will be spoken by the application, as well as expected user responses.

Planning for expert users

As the final step in the Design phase, you may want to identify the potential for expert users and begin considering where you may be able to help them cut through some of the interface to quickly perform common tasks.

Prototype phase ("Wizard of Oz" testing)

The goal of this phase is to create a prototype of the application, leaving the design flexible enough to accommodate changes in prompts and dialog flow in subsequent phases of the design.

For the first iteration, you may want to use a technique known as "Wizard of Oz" testing. This technique can be used before you begin coding, as it requires only a prototype paper script and two people: one to play the role of the user, and a human "wizard" to play the role of the computer system. Here's how it works:

- The script should include the proposed introduction, prompts, list of global commands, and all planned self-revealing help. Consider Table 1 as an example of a script appropriate for "Wizard of Oz" testing:

Table 1. Sample script for "Wizard of Oz" testing

Title	Message types	Prompts and responses	System actions
Greeting	Intro message	Welcome to Phone Pay! You can say Repeat or Help at any time.	Go to Get Account Number.
Get Account Number	Prompt	Account number?	
	Help1	Your account number is on the upper-right portion of your bill. Speak only the numbers on the left side of the dash. You can ignore leading zeros.	

Table 1. Sample script for "Wizard of Oz" testing (continued)

Title	Message types	Prompts and responses	System actions
	Help2	At any time, you can say Help, Repeat, Go Back, Main Menu, Exit, or Transfer to Agent. To continue, say or enter your account number.	
	Caller response	<Says or enters numbers>	If input spoken, go to Confirm Account Number. If entered via DTMF, go to Get PIN Number.
Confirm Account Number	Prompt	Was that <number>?	
	Help1	Please say Yes, No, or Repeat.	
	Help2	At any time, you can say Help, Repeat, Go Back, Main Menu, Exit, or Transfer to Agent. To continue, say Yes or No.	
	Caller response	Yes	Go to Get PIN Number.
	Caller response	No	Go to Get Account Number.

- The two participants should be physically separated so that they cannot use visual cues to communicate; a partition will suffice, or you could use separate rooms and allow the people to communicate by telephone.
- The wizard must be very familiar with the script. The user should never see the script.
- The user telephones (or pretends to telephone) the wizard, who begins reading the script aloud. The user responds to the prompts, and the wizard continues the dialog based on the scripted response to the user's utterance.

"Wizard of Oz" testing helps you fix problems in the script and task flow before committing any of the design to code. However, it cannot detect other

types of usability problems, such as recognition or audio quality problems; addressing problems linked to these types of errors requires a working prototype.

Test phase

After you have incorporated the results of the "Wizard of Oz" testing, you will want to code and test a working prototype of the application. During this phase, be sure to analyze the behavior of both new and, if applicable, expert users.

Identifying recognition problems

As you proceed with the Test phase, note any *consistent* recognition problems.

The most common cause of recognition problems is acoustic confusability among the currently active phrases. For example, both Madison and Addison are US airports. Thus, these potential user inputs to a travel application are highly confusable:

```
User:       Flying from Madison
User:       Flying from Addison
```

Sometimes there is nothing you can do when this happens. Other times you can try to correct the problem by:

- Using a synonym for one of the terms. For example, if the system is confusing "no" and "new," you might be able to replace "new" with "recent," depending on the application's context.
- Adding a word to one or more of the choices. For the Madison/Addison airport confusion, you could make states optional in the grammar for most cities, but require the state for low-traffic airports that have acoustic confusability with higher-traffic airports.
- Plan for disambiguation by writing code that includes or accesses data about typical acoustic confusions. For example:

```
System:     Flying from?
User:       Los Angeles <not flagged as confusable>
System:     Flying to?
User:       Newark <flagged as confusable with New York>
System:     Newark, New Jersey or New York, New York?
User:       Newark, New Jersey
```

Identifying any user interface breakdowns

The Test phase is also where you will identify potential user interface breakdowns. Some factors you may want to analyze include:

- Percentage of users who did not successfully complete your test scenarios
- Percentage of users who transferred to a human operator, when this was not the desired outcome
- Points in the application where users experienced the most difficulty
- Unexpected user behaviors
- Effectiveness of error recovery mechanisms
- Time to complete typical transactions
- Self-reported level of user satisfaction

The first round of user testing typically reveals places where the system's response needs to be rephrased to improve usability. For this reason, system prompts and other messages should be left flexible for as long as possible, at least until after the first round of user testing.

Refinement phase

During this phase, you will update the user interface based on the results of testing the prototype. For example, you may revise prototype scripts, add tapered prompts and customizable expertise levels, create dialogs for inter- and intra-application interactions.

Finally, you will want to iterate the Design—Prototype—Test—Refine process. This includes, in the Test phase, users from previous rounds of testing and users new to the system. Ideally, the final usability test should be on a deployed system to allow evaluation of its accuracy, latency, barge-in characteristics and quality of speech output.

Getting started—high-level design decisions

Designing a SUI involves at least two levels of design decisions. First, you need to make certain high-level design decisions regarding system-level interface properties. Only then can you get down to the details of designing specific system prompts and dialogs. The high-level decisions you need to make include:

- "Selecting an appropriate user interface" on page 20
- "Deciding on the type and level of information" on page 20
- "Choosing the barge-in style" on page 21
- "Selecting recorded prompts or synthesized speech" on page 24
- "Deciding whether to use audio formatting" on page 27
- "Using simple or natural command grammars" on page 29
- "Adopting a terse or verbose prompt style" on page 31
- "Allowing only speech input or speech plus DTMF" on page 32
- "Adopting a consistent set of global navigation commands" on page 34

- "Deciding whether to use human agents in the deployed system" on page 39
- "Choosing help mode or self-revealing help" on page 40

There is no single correct answer; the appropriate decisions depend on the application, the users, and the users' environment(s). The remainder of this section presents the trade-offs associated with each of these decisions.

Selecting an appropriate user interface

The first decision that you must make is to select the appropriate user interface for your application. Not all applications are well-suited to a SUI; some work best with a visual interface, and others benefit from a multi-modal interface (that is, both a speech and a visual interface).

The characteristics in Table 2 can help you decide whether your application is suited to a SUI.

Table 2. When to use a speech interface

Consider using speech if:	Applications may not be suited to speech if:
Users are motivated to use the speech interface because it: • Saves them time or money • Is available 24 hours a day • Provides access to features not available through other means • Allows them to remain anonymous and avoid discussing sensitive subjects with a human	Users are not motivated to use the speech interface.
Users will not have access to a computer keyboard when they want to use the application.	The nature of the application requires a lot of graphics or other visuals (for example, maps or commerce applications for apparel).
Users want to use the application in a "hands-free" or "eyes-free" environment.	Users will be operating the application in an extremely noisy environment (due to simultaneous conversations, background noise, etc.)
Users are visually impaired or have limited use of their hands.	Users are hearing impaired, have difficulty speaking, or are in an environment that prohibits speech (for example, a courtroom).

Deciding on the type and level of information

To keep from overloading the user's short-term memory, information presented in a SUI must generally be more concise than information presented visually. It is common to present only the most essential information initially, then give users the opportunity to access detailed information.

For example, consider a banking application in which a user can request a list of recently cleared checks. In a visual interface, the application might return a table showing the check number, date cleared, payee name, and amount. A similar application with a speech interface might return only the check number and date cleared, and then permit the user to select a specific check number to hear the payee name and amount, if desired.

Choosing the barge-in style

Enabling barge-in allows the computer and the user to speak at the same time, permitting the user's speech to interrupt system prompts as the machine plays them.

On the surface, it might seem that enabling barge-in is always preferable to disabling barge-in. It is easy to imagine experienced users wanting to interrupt prompts (especially lengthy ones) when they know what to say. There are situations, however, in which a system for which barge-in has been disabled will be as easy or easier to use.

Table 3 compares implementations with barge-in enabled or disabled:

Table 3. Barge-in Enabled versus Disabled

Style	Advantages	Disadvantages
Barge-in enabled	Experienced users can interrupt system prompts to speed up the interaction. Users can say "Quiet" to stop the prompt. **Note:** For commands in languages other than US English, see the appropriate appendixes.	Inexperienced users may inadvertently interrupt the prompt before hearing enough to form an acceptable response. You can minimize this problem by keeping system prompts short, to lessen the user's need to barge in; if your prompts are long, you should try to present key information early in the prompt. When using hotword barge-in (see Table 4 on page 22), Lombard speech and the stuttering effect can be problematic. To minimize this problem, you should keep required user inputs very short. See "Controlling Lombard speech and the stuttering effect" on page 23 for more information.

Table 3. Barge-in Enabled versus Disabled (continued)

Style	Advantages	Disadvantages
Barge-in disabled	Guarantees that the entire prompt text plays. This may be especially useful for applications with lots of legal notices, advertisements, or other information that you want to make sure always gets presented to the user. Creates a "my turn-your turn" rhythm for the dialog.	Experienced users cannot interrupt prompts; however, if the prompts are short enough, users should not need to interrupt. Users may experience turn-taking errors. Keeping prompts short helps minimize this.

If enabling barge-in, you should play an initial prompt of 3 seconds or longer with barge-in disabled to give the system time to calibrate echo cancellation.

Comparing barge-in detection methods

To use barge-in effectively, it is important to understand how the system determines when to stop an interrupted prompt. For WebSphere Voice Server the default barge-in detection method is *speech*.

Table 4 compares the available barge-in detection methods.

Table 4. Barge-in detection methods

Barge-in detection method	Description	Advantages	Disadvantages
hotword	Audio output stops only after the system determines that the user has spoken a complete word or phrase that is valid in a currently active grammar.	Resistant to accidental interruptions, such as, those caused by coughing, muttering, or using the system in an environment with loud ambient conversation.	Increased incidence of Lombard speech and the "stuttering effect" (see next section); however, you can control this somewhat by making required user responses as short as possible. The time required to recognize spoken input can cause slower system response times.
speech	Audio output stops as soon as the speech recognition engine detects sound.	This behavior is more typical of conversation between two humans. Minimizes Lombard speech, the stuttering effect, and the distortion to the first syllable of user speech that often occurs when users barge in.	Susceptible to accidental interruption due to background noise, non-speech vocalizations, and speech not intended for the system.

Controlling Lombard speech and the stuttering effect

When speaking in noisy environments, people tend to exaggerate their speech or raise their voices so others can hear them over the noise. This distorted speech pattern is known as *Lombard speech* (named for the researcher E. Lombard, who in 1911 was the first to report such an effect), and it can occur even when the only noise is the voice of another participant in the conversation (for example, when one person tries to interrupt another, or, in the case of a voice application, when the user tries to barge-in while the computer is speaking).

The "stuttering effect" may occur when a prompt keeps playing for more than about 300 ms after the user begins speaking. Unless users have undergone training with the system, they may interpret the continued playing of the prompt as evidence that the system did not hear them. In response, some users may stop what they were saying and begin speaking again – causing a stuttering effect. This stuttering makes it virtually impossible for the system to match the utterance to anything in an active grammar, so the system generally treats the input as an "out-of-grammar" utterance, even if what the user intended to say was actually in one of the active grammars.

To control Lombard speech and the stuttering effect when using hotword barge-in detection, the prompt should stop within about 300 ms after the user begins talking. The average time required to produce a syllable of speech is about 150-200 ms, this means that the system design should promote short user responses (ideally no more than two or three syllables) when using hotword barge-in detection. You should also try to keep prompts as short as possible to minimize the likelihood that users will want to interrupt the prompt. If this is not possible, you should consider switching to speech barge-in detection, or in extreme cases consider disabling barge-in.

Weighing user and environmental characteristics

When deciding whether to use barge-in and which type of barge-in detection is most appropriate, you should consider how frequently users will use your application (expert users are more likely to barge in), and in what environment (quality of the telephone connection, general noise level, etc.).

In general, you should enable barge-in for deployed applications. However, if echo cancellation on your telephony equipment is not good enough, it might be necessary to disable barge-in.

Minimizing the need to barge in

Even when the system permits barge-in, many users do not like to interrupt the system. To minimize the user's need to barge in, you might consider placing short pauses (around 0.75 second) at logical points during and between prompts, such as at the end of a sentence or after each menu item. These brief pauses will give users the opportunity to begin talking without

actively interrupting the system. In systems for which barge-in has been disabled, you can simulate barge-in by enabling recognition during these pauses. Be sure not to produce a turn-taking tone at the end of these "recognition windows" because speech at these times is optional, not required.

Using audio formatting

If you need to temporarily disable barge-in (using **<prompt bargein="false">**), such as while the system reads legal notices or advertisements, you may want to use a unique background sound, tone, or prompt as an indicator. For guidance, see "Applying audio formatting" on page 28.

If you disable barge-in, consider playing a tone to signal the user when it is time to speak. The introductory message should explicitly tell users to speak only after this "turn-taking" tone.

Note: The use of tones to signal user input is somewhat controversial, with some designers avoiding tones based on a belief that tones are unnatural in speech and annoying to users. Others contend that effective computer speech interfaces need not perfectly mimic human conversation, and that a well-designed tone can promote clear and efficient turn-taking without annoyance. For guidance in creating an effective turn-taking tone, see "Designing audio tones" on page 27.

Wording prompts

For systems without barge-in, make prompts as concise as possible. If a prompt must be relatively long, place the key information toward the end of the prompt to discourage users from speaking before their turn.

You can do the same for systems with barge-in, assuming your prompts are relatively short; if the prompts are long, you may decide to move the key information to the beginning of the prompt so users know what input to provide if they interrupt the prompt.

Selecting recorded prompts or synthesized speech

Synthesized speech (*text-to-speech* or *TTS*) is useful as a placeholder during application development, or when the data to be spoken is "unbounded" (not known in advance), which makes it impossible to prerecord.

When deploying your applications, however, you should plan to use professionally recorded prompts whenever possible. Users expect commercial systems to use high-quality recorded speech, and only recorded speech can guarantee highly natural pronunciation and prosody.

Creating recorded prompts

The following guidelines will help you generate high-quality recorded prompts:

- Use professional voice talent, quality recording equipment, and a suitable recording environment.
- Maintain consistency in microphone placement and recording area.
- If prompts contain long numbers, or if many of the application's users are not native speakers of the language in which the application speaks, consider slowing down the speech or exaggerating natural pauses.
- If you are planning on disabling barge-in, aggressively trim recorded prompts to remove the beginning and ending silences. If you are planning on enabling barge-in or if another speech segment will follow immediately, trim the beginning aggressively, but leave silence at the end that is appropriate for the ending punctuation (500 ms for final punctuation, 250 ms for non-final punctuation). Otherwise, leave as little silence as possible.
- As a general rule, use only one voice. When using multiple voices, have a clear design goal (for example, a female voice for introduction and prompts, and a male voice for menu choices). For a consistent sound, you should record your own messages to handle the <error>, <cancel> and <help> events.
- If a voice segment will appear in phrases with different intonations, be sure to record that segment for each intonation. For example, suppose the system will seek confirmation of a telephone number using the phrase "Was that four three three <pause> five five six three?" The "three" that appears before the pause should have a slightly falling pitch, but the "three" that appears before the question mark should have a rising pitch. The "three" that appears between two other numbers should have a steady pitch. This suggests that it will be necessary to obtain one recording for each of the three intonations, to obtain the highest-quality speech output. Note that the development effort required for this might not be appropriate for every application.
- Be aware of the appropriate stress to use in each segment that you plan to record. If the appropriate stress point is not the last open-class item (which is either a noun or a verb) in the sentence, make a note of where the speaker should place the stress.
- If you are recording segments that the application will play sequentially (in other words, will splice), be sure to choose the splice points carefully. If possible, choose splice points at natural pause points. Avoid splice points that separate articles such as ″a,″ ″an,″ and ″the″ from the following word (or any other combination that speakers normally run together).
- If you intend to translate the application into other languages, plan ahead when defining the audio segments to record. You might need to seek assistance from a native speaker of the target language. In general, try to avoid defining audio segments to record that are isolated nouns because in

many languages the correct form for determiners (in English, “a”, “an” and “the”) depends on the following noun. You should be aware that there might be other contextual dependencies that are important in the target language. Some of the known issues are gender sensitivity, ordering of recorded segments and plurality. Good planning early in the definition of audio segments can prevent unnecessary rework during translation.

When using recorded prompts, you can improve system performance by prefetching and caching the audio files. See “Fetching and caching resources for improved performance” on page 124.

For **DTMF prompts** (for example, “For checking, press 1. For savings, press 2. To transfer to an agent, press 3.”) use the following timing guidelines:

- Use a 500 ms pause between items.
- Use a 250 ms pause before “press”.
- No detectable pause after “for”, “to” or “press”.

For **speech prompts** (for example, “Select checking, savings or transfer to agent.” or “To work with your checking account say checking.”) use the following timing guidelines:

- Use a 750 ms pause between items when there are more than 3 items. When there are only two or three items, do not introduce any exaggerated pauses. Speak the phrase as a normal sentence.
- Use a 250 ms pause before “say” or “select”.
- No detectable pause after “say” or “select”.

For a mixture of both **DTMF and speech prompts** use the following timing guidelines:

- Use a 300 to 500 ms pause after an informational message that precedes the presentation of a menu.
- For longer messages, use 250 ms for a comma type pause and 500 ms for a period type pause.

Using TTS prompts

Although recorded prompts are best for many applications, it is important to keep in mind that it is easier to maintain and modify an application that uses TTS prompts. For this reason, you should typically use TTS prompts during development.

When you are ready to deploy your application, use recorded prompts when possible. If part of a sentence requires production via TTS, it is generally better to generate that entire sentence with TTS to avoid the jarring juxtaposition of recorded and artificial speech. It is also possible to design sentences to position the dynamic content at the end, and to play the dynamic

content following a short pause to separate the dynamic TTS content from the static recorded content. For now, designers should be cautious in using this approach because it isn't clear whether people would generally prefer hearing all TTS or this type of combination of recorded and TTS output.

Handling unbounded data: If the information that the application needs to speak is unbounded, you will need to use TTS. Examples of unbounded information include:

- Telephone directories
- E-mail messages
- Frequently updated lists of employee or customer names, movie titles, or other proper nouns
- Up-to-the-minute news stories

Improving TTS output: You can improve the quality of synthesized speech output by using SSML to provide additional information in the input text. For example:

- You can improve the TTS engine's processing of numerical constructs by using the **<say-as>** element to specify the desired pronunciation.
- You can improve the TTS engine's processing of uncommon names by using the **<phoneme>** tag.
- For synthesized speech, a speed of 150-180 words per minute is generally appropriate for native speakers. You can use the **<prosody>** element to slow down the speed of TTS output on a prompt-by-prompt basis for long numbers, or if many of the application's users are not native speakers of the language in which the application speaks.
- You can further improve the prosody of the TTS output by using the **<break>** and **<emphasis>** elements.
- You can change the gender and age characteristics of the TTS voice by using the **<voice>** element. As with recorded prompts, however, it is generally a good idea to use a single voice throughout your application unless there is a clear design goal that requires multiple voices.

Deciding whether to use audio formatting

Audio formatting is a useful technique that increases the bandwidth of spoken communication by using speech and non-speech cues to overlay structural and contextual information on spoken output. Audio formatting is analogous to the well-understood notion of visual formatting, where designers use font changes, indenting, and other aspects of visual layout to cue the reader to the underlying structure of the information being presented.

Designing audio tones

A disadvantage to using audio formatting is that there are not yet standard sounds for specific purposes. If you plan to use audio formatting, you may

want to work with an audio designer (analogous to a graphic designer for graphical user interfaces) to establish a set of pleasing and easily discriminated sounds for these purposes.

When designing audio formatting, the tones should be kept short: typically no longer than 0.5–1.0 seconds, and even as short as 75 ms. Shorter tones are generally less obtrusive, so users are more likely to perceive them as useful, rather than distracting.

Applying audio formatting

You can use non-speech cues to indicate dialog state, exceptions to normal system behavior, and content formatting, as described in Table 5.

Table 5. Audio formatting

Purpose	Recommendations
Turn-taking tone (barge-in disabled only)	If you disable barge-in, you might want to use audio formatting to indicate when it is the user's turn to speak, as described in Table 3 on page 21. An effective turn-taking tone will generally have the following characteristics: • duration of 75–150 ms • pitch of 750–1250 kHz • not too loud • gentle on the ear (a complex wave rather than sinusoid)
Barge-in temporarily disabled	When barge-in is temporarily disabled (for example, when legal notices are read), you may want to play a unique background sound or use a special tone or prompt as an indicator. • For recorded audio prompts, you will need to prerecord the speech mixed with the background sound. • For synthesized (TTS) prompts, you can use an introductory tone or prompt when you disable barge-in, and another tone or prompt to let the user know when you have re-enabled barge-in.
Audio cue for bulleted list	Consider using a short sound snippet as an auditory icon.
Audio cue for emphasis (akin to visual bold and italics)	Consider using an auditory inflection technique, such as changing volume or pitch.
Audio cue for secure transactions	For secure transactions, you may want to play a unique background sound or use a special tone or prompt. See the recommendations for *Barge-in temporarily disabled* above.

Table 5. Audio formatting (continued)

Purpose	Recommendations
Audio cue for "system busy" (akin to visual hourglass)	You can use the **fetchaudio** attribute to play an audio file when the system is busy fetching documents. The audio file stops playing as soon as the document is retrieved. If you use a ticking tone for "system busy," use a fairly slow ticking rate (about 1-2 seconds between ticks). Avoid rates that are faster than 1 second per tick. Alternatively, consider playing music when the system is busy. **Note:** When users are asked to wait, research has been shown that they will follow the instruction for at least 7 seconds of silence.

Using simple or natural command grammars

The VoiceXML browser uses grammar-based speech recognition, as explained in "Grammars" on page 104.

Grammars can be very simple or extremely complex, as explained in Table 6. The appropriate type to use depends on your application and user characteristics.

Table 6. Simple versus natural command grammars

Type of grammar	Description	Advantages	Disadvantages
Simple Grammar	A grammar that includes basic words and phrases that a user might reasonably be expected to say in response to a directed prompt. Many applications will not require complex grammars to be effective, as long as the system prompts constrain likely user input to the valid responses specified by the grammar.	Easier to code and maintain. When used with properly worded prompts, can be as effective as much more complex grammars.	When taken to the extreme, may impose excessive restrictions on what users can say. For example, although you can code an application that requires users to respond to every prompt with a "YES" or a "NO," this results in a cumbersome interface for all but the simplest of applications.

Table 6. Simple versus natural command grammars (continued)

Type of grammar	Description	Advantages	Disadvantages
Natural Command Grammar	A very complex statistical grammar that approaches natural language understanding (NLU) in its lexical and syntactic flexibility.	Can enhance the ability of the system to recognize what users are saying. Can increase dialog efficiency. See "Using menu flattening (multiple tokens in a single user utterance)."	Time-consuming to build and more difficult to maintain. Uses more system resources, possibly impacting performance.

Designing simple grammars

Simple grammars do not attempt to cover all possible ways that a user could respond; rather, you word your prompts in a way that guides users to speak one of a reasonably-sized list of responses, and you code your grammars to accept those responses.

The number and types of responses you will want to support (and therefore, the size and complexity of your grammar) depend on your application and your users.

Evaluating the need for natural command grammars or natural language understanding (NLU)

New SUI designers often start out by assuming that unless a system has true, statistical NLU or NLU-like capability, it will not be usable. This assumption is simply not correct. Well-designed prompts can focus user input so that a fairly small grammar has an excellent chance of matching the user input.

NLU call routing

An emerging area in which NLU applications have seen considerable success is NLU call routing. Callers respond to prompts such as ″How may I help you?″ and the application uses a number of statistical models to interpret the caller's response and to route the call appropriately. This approach is especially effective if there are many potential destinations and no way to efficiently or clearly group them into categories. For more information on IBM's NLU application development, see [insert HREF to NLU plugin here].

Using menu flattening (multiple tokens in a single user utterance)

NLU applications also have the potential for substantial *menu flattening* because the system can parse the user input and extract multiple tokens (where a *token* is the smallest unit of meaningful linguistic input), rather than requiring the user to provide these tokens one at a time. For example, if a user says to a travel reservations application, "Tell me all the flights from

Miami to Atlanta for tomorrow before noon," there are at least six tokens: all (rather than one or two), flights (rather than bus trips or trains), Miami (departure point), Atlanta (destination), tomorrow (date), before noon (time).

To take full advantage of the increased efficiency that menu flattening provides, the system prompts must encourage users to provide input with multiple tokens. Therefore, appropriate prompts for a natural command or NLU application are quite different from appropriate prompts for simple grammar systems. One way to do this is to provide a nondirective prompt with self-revealing help messages that contain examples of valid multiple-token commands. For details, see "Managing nondirective prompts" on page 45.

Adopting a terse or verbose prompt style

As one part of developing a consistent "sound and feel" for the interface, you need to decide whether you will use terse or verbose prompts. Each has advantages and disadvantages, as described in Table 7:

Table 7. Prompt styles

Prompting style	Advantages	Disadvantages
Terse	Uses the fewest words possible; makes efficient use of time. Often leads user to respond tersely, producing well-regulated spoken responses that are easy to recognize.	If too terse, callers might misinterpret its meaning, especially if the intonation is incorrect. You must also take care to avoid sounding impolite.
Verbose	Sometimes perceived as less impersonal, more human.	Might lead to excessively long prompts, especially if you don't adhere to the guidelines on prompt length and wording. The perception that an application just keeps talking can cause significant user frustration. Might cause the user to ascribe an excessive level of intelligence to the system; if this proves inconsistent with the actual abilities of the system, the interface can fail rapidly. You can minimize this risk by telling users up front that they are speaking to a computer (for example, "Welcome to the automated banking system"). Verbose prompts and mistaken assumptions about system intelligence can also lead users to produce responses that are not in the active grammar.

Weighing demographic factors

Accepted standards for terminology, formality, and interaction style in spoken communication vary widely based on demographic factors such as age, culture, and socioeconomic status, as well as the subject matter or purpose of the conversation. Obviously, you will want to design your application to conform to these behavioral norms.

Using terse prompts

In most cases, the balance will tip in favor of terser, or more concise, rather than more verbose prompts. Keep in mind that "terse" does not equal "machine-like." In fact, most expert call center operators use terse prompting to capture the information they need when filling forms for customer orders and reservations. You can recover a human feeling through the selection of a good voice for recording (warm, sincere, professional) and providing short but natural variation in the prompts. For example:

```
System:     Super Club auto reservations. You can use this system to make, change or cancel
            reservations. What would you like to do?

User:       I'd like to make a reservation.
System:     For what city?
User:       Pittsburgh.
System:     And for what day?
User:       Thursday May the third.
System:     Your arriving airline flight?
User:       US Air 2329.
```

A particularly advantageous way to provide variation is through the use of discourse markers such as ″first,″ ″next,″ ″finally,″ and ″now.″ Other useful discourse markers include ″oh,″ ″otherwise,″ ″OK,″ and ″sorry.″ Each of these words has its own function in linguistic discourse. To achieve a natural sounding dialog, it is important that they not be overused.

Using verbose prompts

If you elect to use verbose prompts, you will want to give careful consideration to the following:

- Design your dialogs to accept multiple pieces of information in a single utterance, and to parse the input regardless of order. See "Mixed-initiative application and form-level grammars" on page 111.
- Ensure the robustness of the vocabulary used and determine which expressions you will support.
- Provide plenty of pauses to invite users to barge in.

Allowing only speech input or speech plus DTMF

Applications that mix speech and DTMF are called mixed-mode applications. Because speech applications and DTMF applications do not typically have the

same “sound and feel,” it can be tricky to mix the two. For this reason, it is generally better not to explicitly design mixed-mode applications, unless you are migrating from a legacy DTMF application to a speech-enabled version.

Your system prompts should generally not attempt to mention both speech and DTMF. The application interface will be simpler if your prompts are focused primarily on speech.

Choosing the architecture of mixed-mode application

If you must mix modes, one of your first design decisions must be to choose the fundamental architecture of the mix. There are four choices, as described in Table 8:

Table 8. Mixed input modes

Type of mix	Advantages	Disadvantages
Speech as the primary interface, with DTMF support when beneficial or required	Seamless mixed-mode user interface. Can code to move user immediately back to speech (bounce style) or stay in DTMF mode (sticky style). Makes best use of both technologies.	More time-consuming to develop than a speech-only interface.
Completely separate speech and DTMF interfaces. Users make the choice as their first interaction with the system.	Most straightforward approach.	Requires you to maintain two separate applications. Both applications require full functionality. Users can’t switch from one mode to the other.
A unified system that allows users to freely switch between speech and DTMF modes, perhaps with the use of a DTMF key or key sequence.	Almost as straightforward as separate interfaces. Users can switch between modes.	Users must deal with two different interfaces. Users might experience mode errors—thinking they are in one mode when they are in the other.
A single application with DTMF-style prompts	Users only experience one interface.	Speech is not used to its full advantage.

Deciding when to use DTMF

You should certainly use DTMF whenever your customer demands it, and it’s a good idea to continue to provide DTMF when you have an existing base of power DTMF users. You may want to consider using DTMF for confirmation of sensitive transactions that users would not want overheard. However, you

should keep in mind the difficulty this can pose for users of mobile and rotary pulse telephones, as well as telephones where the keypad is on the handset.

If your application supports both speech and DTMF modes, you may want to switch to DTMF mode automatically if the user is experiencing consistent speech recognition errors. With good error recovery techniques (see "Error recovery" on page 64), it is not generally necessary or desirable to switch to DTMF mode after a single recognition error.

You may also want to consider using a DTMF key sequence to allow the user to switch to DTMF mode; if severe recognition problems are what is causing the user to want to switch modes, providing only a spoken command for mode switching may prove ineffective. If you plan to use DTMF as a backup system for excessive recognition errors or excessive false stopping of prompts, be sure to disable speech for the duration of the call.

Structuring mixed-mode applications

When feasible, you may want to move any interactions requiring DTMF input to the beginning of the application. This is especially important for interactions involving secure information; once users start talking to the application, they might continue to do so even when explicitly instructed to press keys. In some cases, this could compromise the information.

Wording DTMF prompts

You should consider using one word to introduce prompts for a single DTMF digit or command sequence and a different word to prompt for a string of digits. For instance, you might use "Press" to indicate that the user should input a single DTMF digit or command, and "Enter" to indicate that the user should input a string of DTMF digits. Unless there is a reason to do otherwise, provide the functional description before the number to press. For example:

```
System:     To return to the previous menu, press 9.
```

```
System:     Please enter your user ID.
```

Adopting a consistent set of global navigation commands

Selecting the command list

The basic commands recommended for any speech application are listed in Table 9 on page 35; commands shown in **bold** in the table are the minimum recommended set.

Table 9. Recommended list of global command types

Global commands	Description
Go Back	Backs up to the previous prompt. See "Go back."
Exit	Exits the application. See "Exit" on page 36.
Help	Provides help. See "Help" on page 36. **Note:** For language versions other than US English, see the appropriate appendixes.
List Commands	Lists global commands. See "List commands" on page 37.
Transfer to Agent	Transfer to a human call center agent. See "Transfer to agent" on page 37.
Quiet/Cancel	Stops playback of the current prompt. See "Quiet/cancel" on page 37. **Note:** For language versions other than US English, see the appropriate appendixes.
Repeat	Repeats the last prompt. See "Repeat" on page 37.
Start Over	Returns to the beginning of the interaction. See "Start over" on page 37.
What Can I Say Now	Lists all available commands.

Constructing global commands

While the specific implementation of each command depends on the requirements of your application, the general function is described here.

Built-in commands: The VoiceXML browser includes simple grammars for Help and Quiet/Cancel. Depending on the grammar style (simple or natural command grammar) of the rest of your application, you might want to add alternative ways of saying these commands by creating additional, auxiliary global grammars.

Application-specific commands: You should define application-specific commands in grammars that are always active (by specifying them in the application root document with **<form scope="document">**), so users can access them throughout your application. However, there is a trade-off to having a large and robust set of global commands: the potential for misrecognition increases. You should therefore exercise due care when constructing these grammars.

Go back

This command lets users back up to the previous prompt.

In any reasonably complex application, users (especially first-time users) might need to explore the interface; while exploring, they might go down an unintended path, and will need a command to back up through the menu (dialog) structure. For example:

```
System:     Select Leave message, Camp, or Forward call

User        Forward call
System:     Forward to which 4-digit extension?
User        Go back.
System:     Select Leave message, Camp, or Forward call
```

Exit

This command lets users exit the system.

Most users will probably just hang up when they have finished using the system; some users, however, might feel more comfortable if they can say "Exit" and hear the system confirm task completion with a closing message.

When this command is used, the application can also alert the user if a task has been left incomplete, and can prompt the user to complete the task. For example:

```
User        Exit.

System:     You haven't specified an appointment date. Do you want to cancel this
            appointment?
User        Yes.
System:     Goodbye. Thank you for using our automated appointment system.
```

Note: If you find that users frequently trigger the Exit command accidentally due to acoustic confusability with other active commands, you may need to reword the other commands or less preferably, use a different command for the exit function. Also, it's a good idea to include confirmation for the exit action. For example:

```
User        Exit.

System:     Are you sure you want to end this call?
User        Yes.
System:     Goodbye. Thank you for using our automated appointment system.
```

The Exit command also provides an opportunity to engage users in post-usage satisfaction surveys.

It is common to include the word "Goodbye" as a synonym for "Exit."

Help

This command starts a help mode or causes the application to play the next prompt in a sequence of self-revealing help prompts. See "Choosing help mode or self-revealing help" on page 40 for more information.

List commands

If your application has a large number of global commands, consider offering users a "List Commands" command.

Transfer to agent

Note: This command is only applicable in a deployed (telephony) environment, not on a desktop system.

This command transfers the call to a human call center agent, when available. Typically, the Transfer to Agent grammar should also support the DTMF key 0 as a command for transferring to an agent. See "Deciding whether to use human agents in the deployed system" on page 39 for more information.

In some systems, the phrase "Operator" might be preferable to "Transfer to Agent".

Quiet/cancel

This command stops playback of the current prompt (unless you have disabled barge-in).

Repeat

This command repeats the last prompt. (Refer to the description of the **<reprompt>** element in VoiceXML 2.0 for implementation details.) For example:

```
System:     The conversion rate from Australian dollars to euros is 0.548 euros per dollar.
            Would you like to do another conversion?

User        Repeat.
System:     The conversion rate from Australian dollars to euros is 0.548 euros per dollar.
            Would you like to do another conversion?
```

Note that in this example, the application has provided some information, then asked a question. In cases like this, the Repeat command should repeat the information and the question. The most likely scenario is that the user needs to hear the information again, not that the user has failed to understand or remember the question.

Start over

This command lets users start over when they want to abandon the path they are on and return to the beginning of the interaction. For example:

```
System:     Which option?
            Phone
            Fax

User        Phone.
System:     Which phone action?
            Leave message
            Camp
            Forward call

User        Forward call.
System:     Forward to which 4-digit extension

User        Start over.
System:     Which option?
            Phone
            Fax
```

If your application contains multiple modules, it might not be clear at which point the system will restart. In this case, you might want to code a set of commands in global grammars to allow users to jump to the different modules or back to the main menu. For example, the commands LIBRARY, BANKING, CALENDAR, and MAIN MENU might be in global grammars for a voice portal application with library, banking, and calendar modules. If you have explicitly labeled the starting point of your application "Main Menu," then you should accept the command MAIN MENU as well as START OVER.

Using the global commands

Simply having these commands in the system is not sufficient to make them usable; you may need to let users know that they exist. You can accomplish this in three ways:

1. Make sure that the application's introductory message tells users about the two or three most important global commands.

   It is not always necessary or even desirable to list all of the global commands; remember that if the user accidentally or naturally speaks one that was not mentioned, it will still have the desired effect. First time users will rarely remember the commands, but this presentation can be helpful for repeat users. For example:

```
System:     Welcome to the automated monetary conversion system. The commands "Repeat" and
            "Start Over" are always available.
```

2. At some point in the sequence of help prompts, inform the user about the other global commands.

3. List key global commands at task terminal points, either as the last choices in a short menu or following a 1500 to 2500 ms pause after the last option in a longer menu. A user is most likely to want to navigate away at the end of a task so provide information about global commands the user might need when the user most needs it. For example:

```
System:     You have transferred one thousand dollars from
            savings to checking. The confirmation number is
            6 5 4 3 2 1. Select Repeat, Perform Another Transaction,
            Go Back, Main Menu, Exit or Transfer to Agent.
```

```
System:     You have received one new message from David Jones.
            Select Play Message, Reply, Reply to All or Delete Message.
             <2000 ms pause>
            At any time you can say Help, Repeat, Go Back, Main Menu,
            Exit or Transfer to Agent.
```

Deciding whether to use human agents in the deployed system

A key motivation for developing interactive speech systems is to reduce the cost of call centers by handling routine calls automatically. Depending on an application's purpose, your company will decide whether or not to make human agents available to users of the system; this decision has consequences for the design of a speech application, as shown in Table 10. If agents are available, you will probably want to include the Transfer to Agent command (and its synonym, the DTMF key 0) in the set of global commands, as explained in"Adopting a consistent set of global navigation commands" on page 34.

Table 10. Use of human agents

Application goal	Recommendation
High-quality customer service	If high-quality customer service is a main goal of the application, then it should probably include the ability to transfer calls to human agents, either automatically on the detection of user difficulty with the application (see "Choosing help mode or self-revealing help" on page 40 for more details) or at the user's request. While automated systems have improved greatly since their initial introduction, they are not always able to provide the customer support required for a uniformly high level of customer satisfaction.

Table 10. Use of human agents (continued)

Application goal	Recommendation
Reduced costs	If reduced cost is the overriding goal of the application, there may not be any human agents. In this case, you should probably still put the Transfer to Agent command in a global grammar, but the system response on hearing the command is to inform the user that no agents are available. This prevents the speech recognition engine from trying to match the user input "Transfer to Agent" with something else in the grammar, which could lead to a misrecognition. It also directly and unambiguously informs the user that the system does not have any human agents and provides an opportunity to tell callers about other e-Service options such as a website.

For example, controlling costs is likely to be the primary concern for an application that provides company employees with a telephony interface to a directory dialer of internal telephone numbers. In the following example, the system offers the user an alternative way to get the desired information.

```
User:       Transfer to Agent.

System:     There are no agents available for this system. Would you like to hear the Web
            address for our internal telephone directory?

User:       Yes.

System:     The Web address is www.yourco.directory.com.
            Please select Repeat, Go Back, Main Menu or Exit.
```

Choosing help mode or self-revealing help

Certainly, you will want to have the command Help (and possibly a set of alternative phrases that act as synonyms for help) in a global grammar. But what should you do when a user asks for help? You can either switch to a separate help mode, or engage in a technique known as *self-revealing help*. Table 11 compares these help styles:

Table 11. Comparison of help styles

Help mode	Self-revealing help
Help is provided through a separate dialog.	Help is integrated into each application dialog.

Table 11. Comparison of help styles (continued)

Help mode	Self-revealing help
You will need to save the state of your application dialog so that you can return to it when the user exits help mode, and you must tell the user how to exit help mode. Alternatively, you could use a "one-shot" help mode message that automatically returns to the application after presenting the help text.	Does not require explicit mode management.
You may want to use a *help mode indicator* (such as a background sound, a different voice, or introductory and terminating audio cues or messages) to signal when the user is in help mode, rather than in the regular application.	Audio formatting is not required because help is integrated into each application dialog.
Good for providing general help information at an application, module, form, or menu level.	Uses a sequence of prompts to provide progressively greater levels of context-sensitive assistance at each turn in a dialog.
Requires less help text, but generally provides less-specific information.	Requires you to compose multiple prompts for each turn in a dialog.
Must be triggered explicitly, by the user speaking something from the Help grammar.	Can be triggered: • Explicitly, by the user speaking something from the Help grammar • Implicitly, by the user speaking an out-of-grammar utterance (a **nomatch** event) • Implicitly, by the user failing to respond to a prompt before a timeout occurs (a **noinput** event)
Can transfer to a human agent explicitly, by the user speaking something from the Operator grammar.	Transfer to a human operator can be triggered: • Explicitly, by the user speaking something from the Operator grammar • Implicitly, after a specified number of invalid user responses

Because the introduction of modes into a system complicates the interface, we generally recommend the self-revealing help technique. Alternatively, the requirements of your application or users may lead you to adopt a strategy that incorporates both styles of help: a **<catch>** element in the application root document, plus context-sensitive self-revealing help where appropriate. Regardless of the help style you choose, you should use it consistently throughout your application.

Maximizing the benefits of self-revealing help

It is possible that a user might experience momentary confusion or distraction, which leads the user to explicitly request help. If the system has been

designed to be self-revealing, these explicit requests for help could receive the same treatment as a silence timeout or any out-of-grammar utterance, allowing you to reuse the same code and prompts. This is style of help is referred to as "implicit" because the system never enters an explicit help mode. The theory is that implicit help provides more of a sense of moving forward (resulting in a better, more satisfying user interface) and is simpler to code than explicit help.

Implementing self-revealing help

The following example assumes barge-in enabled:

```
System:                  Welcome to our automated directory dialer.
                         You can call any of our employees by speaking the employee's
                         name and location.
                         You can say Help or Exit at any time.

User (interrupting):     Help.
System:                  Please say the desired name and location.
User:                    [The user coughs. The system interprets this as a nomatch]
System:                  For example, to call Joe Smith in Kansas City,
                         say, "Joe Smith, Kansas City."

User:                    [Silence timeout]
System:                  At any time you can say Help, Repeat,
                         Go Back, Start Over or Exit.
                         To continue, please say the desired name and location.
User:                    Ed Black, Poughkeepsie.
```

Notice the self-revealing nature of the dialog, and the way it works whether triggered by a silence timeout, an utterance from the Help grammar, or any out-of-grammar utterance. In addition, if this application is able to detect when the user requests an unsupported name or city, it can respond appropriately (for example, "Sorry, but there's no office in that location").

Also note the pattern for the introduction, which has the general sequence Welcome-Purpose-MoreInfo, followed by an ExplicitPrompt. Usually, the explicit prompt at the end of the introduction is enough to guide the user to provide an appropriate response; however, if the user input after that explicit prompt is an out-of-grammar utterance, something from the Help grammar, or a silence timeout, then the system falls through a sequence of prompts designed to provide progressively greater assistance. The beauty of this system is that the user never hears the same prompt twice, so the system appears to be responsive to the user and always moving the dialog forward; this help style promotes efficient use by most users. If the user still fails to provide an appropriate response, the sequence of prompts ultimately "*bails out*" (typically, transfers to a human operator or apologizes and exits).

Bailing out

Deciding how many attempts to allow the user before bailing out (leaving the speech application) is a judgement call. You do not want to bail out after the first unsuccessful attempt, or you will lose users too quickly; conversely, you do not want to require too many attempts, or you run the risk of frustrating users who are experiencing serious problems with the system. For most applications, if users are experiencing serious problems with the system, it is better to get them out of the system quickly and, if possible, give them another means to access the desired information (say, by pointing them to a visual Web site, switching to a DTMF system or transferring them to an agent).

In general, two or three progressively more directed prompts should suffice to help users recover from **noinput** events, **nomatch** events, or explicit requests for help. For example:

```
System:     Transfer $500 from checking to savings?

User:       (noinput)
System:     Please say Yes, No, or Repeat.
User:       (nomatch or noinput)
System:     If you want to transfer $500 from checking to savings, say Yes.
            Otherwise, say No.
User:       (nomatch or noinput)
System:     At any time you can say Help. Repeat, Go Back, Start Over.
            Transfer to Agent, or Exit. <2000 ms pause> To authorize the transaction,
            please say Yes or No.
```

Getting specific—low-level design decisions

Once you have decided on the high-level properties of your system, it's time to consider the low-level issues, especially regarding specific interaction styles and prompts. This section addresses the following issues:

- "Adopting a consistent "sound and feel""
- "Using consistent timing" on page 44
- "Designing consistent dialogs" on page 45
- "Creating introductions" on page 46
- "Constructing appropriate menus and prompts" on page 48
- "Designing and using grammars" on page 59
- "Error recovery and confirming user input" on page 64

Adopting a consistent "sound and feel"

To promote a consistent sound and feel, you may want to adopt the following guidelines.

Designing prompts

When designing prompts, you should choose one prompt style and use it consistently throughout your application. (See "Adopting a terse or verbose prompt style" on page 31.) In general, you should try to use the present tense and active voice for all prompts. For example, use:

```
System:     Transfer how much?
```

rather than:

```
System:     Amount to be transferred?
```

Unless there is a clear design goal to the contrary, you should adopt a consistent voice for your application, whether you use recorded or synthesized prompts. (See "Creating recorded prompts" on page 25.)

Standardizing valid user responses

Users master applications more quickly when they can predict what responses are valid. For example, if you allow positive responses of "Yes," "Okay," and "True" to one yes/no question, you should allow the same responses for all yes/no questions in your application. You can accomplish this by using the built-in types or reusing your own grammars. (See "Reusing dialog components" on page 46.)

Using consistent timing

Consistent timing is important in developing a conversational rhythm for the dialog. Activities you may want to consider include:

- The amount of user "think time"
- The length of pauses between menu items
- The amount of time the system takes to respond to a user utterance

Setting the default timeout value

The default timeout value for the VoiceXML browser is 7 seconds. For applications designed for special populations (such as non-native speakers or older adults), you might want to increase this by specifying a larger value (around 10 seconds) by specifying a value for the **timeout** property in your application root document.

Managing processing time

Some users may interpret a long processing delay as an indication that the system did not hear or did not accept the most recent input. This can cause users to repeat what they just said, which can lead to misrecognitions and accidentally stopping a prompt that has just begun to play.

If you anticipate that processing a user's request may take an extended period of time, you should consider playing a prompt to inform the user of the delay, or at least to confirm that the system accepted the input. For example:

```
System:     Processing your request. Please wait.
```

Designing consistent dialogs

When users don't know what they can say at a given point in a dialog, the interaction between the user and the application can quickly break down. To help users avoid this "What can I say now?" dilemma, try adopting consistent sound and feel standards to create dialogs that behave consistently, both within your application and across your platform (if you support multiple applications).

Writing directive prompts

You should try to write prompts that clearly indicate to users what they can say. For example:

```
System:     Say Yes, No, or Repeat.
```

or:

```
System:     Travel to which US state?
```

or:

```
System:     Select E-mail, Voice mail, or Faxes
```

Managing nondirective prompts

When you use a complex natural command grammar (or an NLU application) that can accept multiple tokens, you should use a nondirective prompt for the initial interaction with the user. The silence timeout that follows a nondirective prompt should be shorter than the standard default timeout of 7 seconds--typically about 3 seconds--to serve the dual purpose of providing a sufficient window for the more expert user to compose and begin speaking a multiple-token command but not making the less expert user wait too long to get the first self-revealing help message. Recent research has indicated that the appropriate first (and possibly second) levels of self-revealing help for these types of prompts should include examples of valid multiple-token commands. Users are often able to take a multiple-token example and replace the sample information with their own information. Only if these fail to lead the user to produce a valid input should the system switch to a one-token-at-a-time directed dialog. For example:

Table 12. Successful use of example

System:	What are your travel needs?
User:	<nomatch or silent for 3 seconds>
System:	For example, you might say "I want to go from New York to Orlando on December 1."
User:	I want to go from Chicago to Los Angeles on March 15.

Table 13. Unsuccessful use of example—switch to directed dialog

System:	What are you travel needs?
User:	<nomatch or silent for 3 seconds>
System:	For example, you might say "I want to go from New York to Orlando on December 1."
User:	<nomatch or silent for 7 seconds>
System:	You might also try "Book a flight on Delta from New York to Orlando on December 1."
User:	<nomatch or silent for 7 seconds>
System:	Departure date?

For NLU call routing applications, the examples should be acoustically distinct commands that connect users to frequently-requested destinations.

Maintaining a consistent sound and feel

When an application behaves in a way that the user did not anticipate, the user can become confused and/or frustrated and is more likely to respond inappropriately to subsequent prompts. This, in turn, causes the interaction between the user and the application to break down. To help users avoid this "But why did the application say that?" dilemma, try adopting consistent sound and feel standards (see "Adopting a consistent "sound and feel"" on page 43) to create dialogs that maintain synchronization between the application's actual state and the user's mental model of the application's behavior.

Reusing dialog components

You can improve application consistency and decrease user learning curves by reusing dialogs or dialog components where possible. Components you might reuse include:

- Subdialogs for data collection, error recovery, and other common tasks
- Built-in field types
- Application-specific grammars

Creating introductions

The introductory message(s) that the user hears when starting the application can be very important. In general, introductions should contain the following information:

1. Welcome

2. Purpose (optional)
3. Global List (only two or three key commands)
4. Speak After Tone (if using turn-taking tones)
5. Initial Prompt

Welcome

The welcome prompt should be short and simple, but also make clear that the user is talking to a machine, not a person. For example:

```
System:     Welcome to the automated currency conversion system.
```

To speed up the interaction for expert users, you might want to provide a brief (for example, 1.5 second) recognition window without a tone after this welcome prompt. Expert users will know how to provide valid input; if a silence timeout occurs, you can presume that the user is a novice and proceed to play the remaining introductory prompts. Keep in mind, however, that if the application allows barge-in, the system needs about 3-5 seconds to set it up. For this reason, you should plan to suppress barge-in and play 3-5 seconds (12-20 syllables) of speech in the welcome message before pausing to invite the caller to barge in. It is very important to trim all silence from the end of the audio that plays while the system is setting up barge-in (specifically, calibrating echo cancellation). If you need a pause between the end of this audio file and the beginning of the next, put the pause at the beginning of the next audio file, which begins playing when the application enables barge-in. Otherwise, the silence at the end of the initial audio will invite users to barge-in, but will cuase usability problems because the application has not yet enabled barge-in.

Note: A similar strategy with much shorter (say, 0.75 second) recognition windows can be used at other logical pausing places during and between prompts, such as at the end of a sentence or after each menu item. These brief pauses will give users the opportunity to talk without feeling that they are “being rude” by interrupting the application.

Purpose

If necessary, state the purpose of the system. For example:

```
System:     This system provides exchange rates between American dollars and the
            major currencies of the world.
```

Note: You may not need to include a statement of purpose if the welcome prompt adequately conveys the system’s purpose. Don’t include a statement of purpose unless it’s necessary; it makes the introduction longer.

Global list

Next, you will probably want to tell the user what commands are always available. If you have a large global set of commands, you probably don't want to list all of them here – just the key ones. For example:

```
System:     The commands "Repeat" and "Start over" are always available.
```

Speak after tone

In general, applications in which barge-in has been disabled should use a tone to signal the user to speak. In such cases, you should include a prompt such as:

```
System:     Speak to the system after the tone.
```

Interrupt any time

If you enable barge-in, you might want to let the caller know that it's OK to interrupt the application. It's not absolutely necessary because most users will naturally interrupt the application when it pauses. For some users, however, it might be beneficial to mention this explicitly. For example:

```
System:     You can interrupt me at any time. When you hear the choice you want, just say it.
```

Initial prompt

Finally, present the user with the first prompt requiring user input. For example:

```
System:     Select Buy Currency or Sell Currency.
```

Constructing appropriate menus and prompts

A characteristic of many voice applications is that they have little or no external documentation. Often, these applications must support both novice and expert users. Part of the challenge of designing a good voice application is providing just enough information at just the right time. In general, you don't want to force users to hear more than they need to hear, and you don't want to require them to say more than they need to say. Adhering to the following guidelines can help you achieve this:

- "Limiting menu length" on page 49
- "Grouping menu items, prompts, and other information" on page 50
- "Controlling prompt length" on page 51
- "Avoiding DTMF-style prompts" on page 52
- "Using the right words" on page 53
- "Providing instructional information" on page 58

Limiting menu length

When designing menus, you follow the guidelines in Table 14.

Table 14. Recommended maximum number of menu items

Application	Maximum number of menu items
Barge-in enabled	12
Barge-in disabled	5
Any application in which the menu items are long phrases	3

For cases in which you cannot stay within these limits, see "Managing audio lists" on page 72.

A deep menu structure is one in which there are few choices available at any given level in the structure but there are many levels. A flat menu structure is one in which there are many choices available at any given level in the structure but there are few levels. In the flattest possible structure, there is only one level which contains all the choices. A terminal node in a menu structure is a choice that does not lead to any additional sets of choices.

The conditions that favor deep menu structures are:

- Barge-in disabled.
- Terminal nodes have approximately equal frequency of selection.
- Menu items require many words or long duration.
- Menu items fall into easy to label, clearly defined categories.
- There are dependencies among the menu options.

The conditions that favor the use of flat menu structures are:

- Barge-in enabled.
- Need to surface frequently-used terminal nodes.
- Menu items have few words per item or short duration.
- Difficult to develop clear categories for menu items.

For most applications, the conditions will favor the use of flat rather than deep menu structures.

Older guidelines for DTMF user interfaces (for example, Marics & Engelbeck), strongly advised against exceeding four options per menu. This guideline is inappropriate for SUIs because options in speech menus typically have far fewer words than DTMF options. Recent human factors research has also challenged the applicability of this guideline for DTMF menus.

Some practitioners have suggested 7 ±2 menu items for speech menus. This suggestion assumes that users are trying to memorize each option as they hear them, but task analysis of selection from auditory menus does not support this assumption. A user does not need to memorize all of the items in a speech menu; users only need to remember the one that is the current best match to the desired function. If the user hears an excellent match, then he or she can barge in to select it (self terminating search). Otherwise, the user continues to listen to options until hearing a better match (discarding the old and remembering the new) or there are no more options (exhaustive search).

Grouping menu items, prompts, and other information

There are a number of strategies that you should consider when deciding how to group information.

Separating introductory/instructive text from prompt text: When appropriate, you should consider separating any introductory or instructive text from the prompt text; this allows you to reprompt without repeating the introductory text. For example, you might create one audio file that says, "Welcome to the WebVoice demo" and a separate audio file that says, "Say one of the following options: Library, Banking, Calendar." The first time through the sequence, the application could play the files in succession. If the user returns to this main menu later in the application session, the application could play only the second audio file. For example, the first time the user hears:

```
System:     Welcome to the WebVoice demo. Select Library, Banking, or Calendar.
```

On the second and any subsequent times, the user hears only:

```
System:     Select Library, Banking, or Calendar.
```

Separating text for each menu item: If appropriate for your application design, you might even create separate audio files for each menu choice.

Ordering menu items: When presenting menu items, consider putting the most common choices first so that most users don't have to listen to the remaining options. For example:

```
System:     Select List Specials, Place an Order, Check Order Status
            or Get Mailing Address.
```

A possible exception to this guideline is when the most common choice is a more general case of another choice. In this example, "Other loans" is presented last, regardless of its relative frequency of use:

```
System:     Loan type?
             <3 second pause>
            Select Car, Personal or Other Loan.
```

Controlling prompt length

Applications are generally most usable when system prompts are as short as possible (minimizing users' need to interrupt prompts) and user responses are relatively short (minimizing the likelihood of *Lombard speech* and the *stuttering effect*. See "Controlling Lombard speech and the stuttering effect" on page 23.).

Note: Especially for applications using hotword (recognition) barge-in detection, try to keep required user responses to no more than two or three syllables. If this is not possible, you may want to consider using speech-based barge-in.

Effectively worded shorter prompts are generally better than longer prompts, due to the time-bound nature of speech interfaces and the limitations of users' short-term memory. A reasonable goal might be to strive for initial prompt lengths of no more than 3 seconds, and to try to keep the greeting and opening menu to less than 20 seconds. If prompt lengths must consistently exceed 3 seconds, the application should permit barge-in. For planning purposes, assume that each syllable in a prompt or message lasts 150-200 ms.

Do not overuse of "please", "thanks" and "sorry". You can use them, but don't use them automatically or thoughtlessly; only when they serve a clearly defined purpose.

If you can remove a word without changing the meaning, then consider removing it (while keeping in mind that a certain amount of structural variation in a group of prompts increases the naturalness of the dialog). Strive to use clear and unambiguous terms.

In general, if you have a choice between long and short words that mean the same thing, choose the short word. In most cases, the short word will be more common than the long word and users will hear and process it more quickly. For example, "use" is a better choice than "utilize."

Consolidate common words and phrases. For example, you could combine "Are you calling about buying a fax machine for your home?" and "Are you calling about buying a fax machine for your business?" into "Are you calling about buying a fax machine for your home or business?".

In general, use the active voice rather than the passive voice. People process the active voice faster and more accurately. This is partly because phrases using the active voice (for example, "Say the book's title") tend to be shorter

than those using the passive voice (for example, “The book’s title must be spoken now”). In some cases a sentence will sound best in passive voice, but you should use passive voice only if attempts to rewrite the sentence in active voice don’t sound natural.

Good prompts do not necessarily have good grammar. As in normal conversation many natural phrases do not abide by the rules of grammar.

Avoiding DTMF-style prompts

Prompts in an application with a DTMF interface typically take the form “For option, do action.” With a SUI, the option is the action. In general, you should avoid prompts that mimic DTMF-style prompts; these types of prompts are longer than they need to be for most types of menu selections. For example, use:

```
System:     Select Marketing, Finance, Human Resources, Accounting, or Research
```

rather than:

```
System:     For the Marketing department, say 1
            For Finance, say 2
            For Human Resources, say 3
            For Accounting, say 4
            For Research, say 5
```

or worse:

```
System:     For the Marketing department, say Marketing
            For Finance, say Finance
            For Human Resources, say Human Resources
            For Accounting, say Accounting
            For Research, say Research
```

If the menu items are difficult for a user to remember (for example, if they are long or contain unusual terms or acronyms), you might choose to mimic DTMF prompts. Also, this style can work well for first level help messages as it slows things down, giving users more time to process their options.

Choosing a complex alternative

It isn’t always possible to have a simple label for a choice. Consider the following prompt:

```
System:     Would you like to hear your account balances at the beginning of every
            call, or just at the beginning of the first call of the day?
```

Try to imagine how many different ways a user might respond to this question. One way to deal with this situation is to change the prompt to a yes or no question that has the form, "You can choose A or B. Would you like A'?" where A and B are complex choices and A' is a short version of A. For example:

```
System:     You can hear your account balances at the beginning of every call, or
            just at the beginning of the first call of the day. Would you like to hear
            them in every call?
```

Using the right words

There are many aspects to consider when deciding how to word your application's prompts and menus. The choices you make will have a significant impact on the types of responses your users provide, and therefore on what you will need to code in your grammars. Some of the issues you may need to address include the following:

- "Adopting user vocabulary"
- "Being concise"
- "Mixing menu choices and form data in a single prompt" on page 54
- "Avoiding synonyms in prompts" on page 55
- "Promoting valid user input" on page 55
- "Tips for voice spelling" on page 55

Adopting user vocabulary: Applications that require users to learn new commands are inherently more difficult to use. During the Design phase of your application development process, you will want to make note of the words and phrases that your users typically use to describe common tasks and items. See "Design Phase" on page 13. These are the words and phrases that you will want to incorporate into your prompts and grammars.

Being concise: Regardless of whether your general style is terse or verbose, try to phrase prompts in a way that conveys the maximum amount of information in the minimum amount of time. For example, use:

```
System:     Say the author's last name, followed optionally by the author's first name.
```

rather than:

```
System:     Say the author's last name. If you also know the author's first name, state the
            author's last name and first name.
```

Avoid lengthy lead-in phrases to a set of options. Begin your prompts with words like "Select", "Choose" or , in some cases, "Say". For example, use:

```
System:     Select Checking, Savings or Money Market.
```

rather than:

```
System:     Please make one of the following choices:
            Checking, Savings or Money Market.
```

Sometimes, the most effective way to prompt the user is to word the prompt as a question. For example, use:

```
System:     Savings or checking?
```

rather than:

```
System:     Please choose from the inquiry menu:
            Savings
            Checking
```

Question prompts are especially useful when the user can make a choice by repeating one of the options verbatim. You can also use question prompts to collect information, as long as the question restricts likely user input to something the application can understand. For example:

```
System:     Transfer how much?
```

Using pronouns: Use pronouns such as ″it″ and ″one″ to avoid stilted repetition of words. For example:

```
System:     You have four new messages. The first one is from... The second one is from...
            The third one is from... The last one is from...
```

rather than:

```
System:     You have four new messages. The first message is from... The second message is
            from... The third message is from... The last message is from...
```

Mixing menu choices and form data in a single prompt: When mixing menu choices and form filling in the same list, you will want to word the prompt to clearly indicate what the user can say. For example, use a prompt such as:

```
System:     Please state the author's last name, or say List Best Sellers.
```

Avoiding synonyms in prompts: You should avoid using synonyms in prompts; these might mislead the user regarding valid input. For example, use:

```
System:     To search the database, say Search by Author.
```

rather than:

```
System:     To query the database, say Search by Author.
```

because the latter might cause the user to think that "query" is a valid response.

Promoting valid user input: Whenever possible, the prompt text should guide the user to the proper word choice. For example:

```
System:     If this is correct, say Yes.
```

Try to phrase prompts in a way that minimizes the likelihood of the user inserting extraneous words in the response. For example, if the system cannot interpret dates embedded in sentences, use:

```
System:     Please state the year you were born.
```

or:

```
System:     Birth year?
```

rather than:

```
System:     When were you born?
```

because the latter is likely to elicit a response such as:

```
User:       I was born on November 3rd, 1954.
```

Tips for voice spelling: In general, if you can avoid voice spelling, you should. The letters of the alphabet are notoriously difficult for computers (and humans) to recognize.

If a user must perform voice spelling, then you can take advantage of the fact that some recognition errors are more likely than others. Table 15 shows the results of an experiment conducted to investigate patterns of recognition errors for the letters of the English alphabet. In the table, uppercase letters indicate substitution probabilities that exceeded 10%. Lowercase letters indicate substitutions that occurred during the study, but had substitution probabilities less than 10%.

For example, if a user rejects a returned **K** in a voice-spelling application, then the letter most likely to have actually been spoken is **A**. Fourteen of the letters in the table have only one substitution for which the probability of substitution exceeded 10%. Eleven of the letters don't have any substitutes for which the probability of substitution exceeded 10%, and six of those didn't have any substitutions at all. This means that whenever the system returns these letters (H, I, U, W, X, and Y), you can have very high confidence that the speaker actually said that letter. Only two letters (T and G) had two substitutes for which the probability of substitution exceeded 10%. Note that the substitution probabilities are not usually symmetrical (F/S and V/Z are exceptions). If the system returned a **D** but the user indicated that it was not correct, the letter the user said was most likely an **E**. If the system returned an **E** but the user indicated that it was not correct, the letter the user said was most likely a **V**.

Table 15. Recognition errors when spelling

Letter returned	**Most likely substituted for**
A	I
B	d
C	V t z
D	E b v
E	V
F	S
G	P T v
H	
I	
J	a
K	A
L	m
M	N
N	x
O	L r u

Table 15. Recognition errors when spelling (continued)

Letter returned	Most likely substituted for
P	d e
Q	U p t
R	I f y
S	F h j
T	D E g p v
U	
V	Z
W	
X	
Y	
Z	V

There are several ways that you can use this information to improve the user experience when voice spelling. If the information that the user is spelling is unbounded (such as a user's last name), then you can offer alternatives in the order of their likelihood. For example:

```
System:     Please spell your last name.
User:       L e w i s
System:     L e w i f?
User:       No.
System:     First letter was L as in Lucy?
User:       Yes.
System:     Then E as in elephant?
User:       Yes.
System:     Then W as in walrus?
User:       Yes.
System:     Then I as in Iris?
User:       Yes.
System:     Then F as in Frank?
User:       No.
System:     Was it S as in Sally?
User:       Yes.
System:     L e w i s?
User:       Yes.
```

You can also use n-best decoding to find likely substitutes. See “Refining confirmation and error correction with confidence levels and n-best lists” on page 70 for details.

If the information that the user is spelling is bounded (such as a part number, or a list of users’ last names in a database), then you can often detect and correct voice spelling recognition errors without involving the user. For example, suppose the user is ordering a part by its code:

```
System:     Part code?

User:       S X H U F
System:     <Returns F X H U F>
            <Checks part number database>
            <Doesn't find this number>
            <Checks the following possibilities:
            <F X H U S
            S X H U S
            S X H U F>
            <Only third one is in database>
System:     S as in Sam, X, H, U, F as in Frank?
User:       Yes.
```

In the example above, the system created the different possible part numbers by using the information from the table of substitutions. Because there were no likely substitutes for X, H, or U, the system left them alone, and systematically changed F to S and vice versa.

Providing instructional information

In certain situations, you may need to provide instructional information to the users.

“Feeding-forward” information as confirmation: Where applicable, you may want to word prompts in a way that “feeds the result forward” (that is, incorporates the user response) into the next prompt. For example:

```
System:     Say Phone or Fax

User:       Phone
System:     Which phone action?
            Leave message
            Camp
            Forward call
```

This technique provides feedback that the system correctly understood the response and also reinforces the user’s mental model of the dialog state. Using

this technique eliminates the need for cumbersome confirmation of every user input; however, you should still confirm user requests for actions that cannot be undone.

Recovering from errors: This section deals with error recovery only as it relates to prompt wording. See "Error recovery and confirming user input" on page 64 for additional information about error recovery.

To promote faster error recovery, prompts should focus on keeping the dialog moving rather than on any mistakes. For example, if you are using self-revealing help, you should avoid having the system say:

```
System:     Sorry, I don't understand what you said.
```

because there is no evidence that this helps the user understand what to do next. See "Implementing self-revealing help" on page 42 for guidance on how to keep the dialog moving forward with self-revealing help.

Similarly, it is best to avoid claiming that the user "said" a particular response, since the information you present is actually just a reflection of how the speech recognition performed. For example, use:

```
System:     Was that 113?
```

instead of:

```
System:     You said 113. This is not a valid quantity.
            Please restate your quantity.
```

Designing and using grammars

Designing good grammars is as much art as science. Iterative prototyping is crucial to grammar design. See "Design methodology" on page 13.

Since only words, phrases, and DTMF key sequences from active grammars are possible speech recognition candidates, what you choose to put in a grammar and when you choose to make each grammar active have a major impact on speech recognition accuracy. In general, you should only enable a grammar when it is appropriate for a user to say something matching that grammar. When appropriate, you should reuse grammars to promote application consistency.

Managing trade-offs

There are many trade-offs that you will want to consider in deciding what words and phrases to include in your grammars and when to make each grammar active. Some of the major trade-offs are:

- Word and Phrase Length
- Vocabulary Robustness and Grammar Complexity
- Number of Active Grammars

Word and phrase length: One of the first trade-offs you are likely to encounter is how long users responses should be. Table 16 compares the two schemes.

Table 16. Grammar word/phrase length trade-offs

Longer words and phrases	Shorter words and phrases
Multisyllabic words and phrases generally have greater recognition accuracy because there is greater differentiation among valid utterances. Individual word choice is still important in longer phrases of because the VoiceXML browser's ability to match a menu choice based on a user utterance of one or more significant words.	Shorter words and phrases are more likely to be misrecognized; when a grammar permits many short user utterances, it is important to minimize acoustic confusability by making them as acoustically distinct as possible. Monosyllabic words and short words with unstressed vowels are especially prone to be recognized as each other, even though they may look and sound different to a human ear.
Dialogs may be slower.	Dialogs progress faster: choices are read faster, and user responses tend to be shorter.
Users may have difficulty remembering long phrases.	Easier for users to remember.
For applications with hotword (recognition) barge-in detection, longer words and phrases may induce stuttering and Lombard effects. See"Choosing the barge-in style" on page 21.	

Vocabulary robustness and grammar complexity: A related issue is how robust and complex your grammars should be, as illustrated in Table 17.

Table 17. Vocabulary robustness and grammar complexity trade-offs

Robust grammar	Simple grammar
Inclusion of synonyms and alternative phrases gives users greater freedom of word choice; however, users may incorrectly assume that they can say virtually anything, leading to a large number of out-of-grammar errors.	Narrow list of valid utterances places more constraints on user input.
Grammar files are larger and load more slowly.	Grammar files are smaller and load more quickly.

Table 17. Vocabulary robustness and grammar complexity trade-offs (continued)

Robust grammar	Simple grammar
Increased chance of recognition errors.	Simple grammars generally have better recognition accuracy.

Number of active grammars: Finally, you will want to consider when each grammar should be active, as presented in Table 18.

Table 18. Number of active grammar trade-offs

More active grammars	Fewer active grammars
May improve usability, such as by allowing anytime access to items on main menu.	
Increased chance of recognition conflicts.	Less chance of misrecognitions due to recognition conflicts.
Performance can degrade.	Better performance.

Note: You can limit the active grammars to just the ones specified by the current form by using the **<field>** element's modal attribute.

Improving recognition accuracy

In general, you can improve recognition accuracy by:

- Simplifying your grammars to minimize the possibility of confusion between words.
- Presenting fewer choices.
- Having fewer active grammars.
- Ensuring that the grammar can accept user responses that mirror key phrases from the preceding prompt. For example, if the prompt is ″Are you a distributor or a retailer?″ the grammar should be able to accept phrases such as ″distributor,″ ″a retailer,″ ″I am a distributor,″ and ″I'm a retailer.″

Using Boolean and yes/no grammars

General strategy: For the first presentation of a prompt with an expected answer of Yes or No, we generally recommend using the built-in boolean grammar. This grammar provides more flexibility in accepting user input than does a simple Yes/No grammar. For example:

```
System:     Do you want more information? (boolean grammar active)
User:       Okay
```

Recovering from a noninput event: If the system returns a noinput event in response to the initial prompt, we recommend that you attempt to recover by switching to a simple Yes/No grammar and a prompt that clearly directs the user to say "Yes" or "No", as shown here:

```
System:     Do you want more information? (boolean grammar active)
User:       (no response)
System:     Please say Yes, No,  or Repeat. (Yes/No grammar active, Repeat always available)
User:       Yes.
```

Recovering from a nomatch event: If the system returns a nomatch event based on the user input for the initial prompt, we recommend that you switch to a Yes/No grammar and use a prompt that attempts to confirm whether the user intended to provide a positive or negative response. In their book, *How to Build a Speech Recognition Application: A Style Guide for Telephony Dialogues,* Bruce Balentine and David P. Morgan recommend the phrase, "Was that a Yes?" for this purpose. For example:

```
System:     Do you want more information? (boolean grammar active)
User:       (unintelligible response)
System:     Was that a Yes? (Yes/No grammar active)
User:       Yes.
```

This design minimizes disruptions to the dialog flow because the user's response to the subdialog prompt is the same as the intended response to the prompt that generated the out-of-grammar error.

When initial accuracy is paramount: If it is more important to get extremely high accuracy on the first presentation of the Yes/No question than it is to accept a broader range of user responses, you could write the initial prompt so that it explicitly directs the user to say Yes or No, and use a Yes/No grammar. For example:

```
System:     Please say Yes or No. Do you approve this transaction? (Yes/No grammar active)
User:       Yes.
```

Using the built-in phone grammar

Some people tend to pause at the logical grouping points when speaking telephone numbers, especially if they are having difficulty remembering the number. For example, when speaking a USA telephone number, a user might pause after the 3-digit area code, and again after the 3-digit exchange. If the pauses are long enough, they might inadvertently trigger endpoint detection before the user has finished speaking the 10-digit telephone number.

If your application requires users to enter telephone numbers, you will want to take special care to thoroughly test your telephone number collection dialogs; if users experience difficulties, you may want to employ some of the following techniques to ensure that the data is being captured correctly:

- Set the **completetimeout** and **incompletetimeout** properties to a higher than default value (for example, 1150 to 1500 ms), to allow users more time to pause while speaking the telephone number. If you adopt this strategy, you should also test that the latency until the next system prompt does not become too great.

  Note: For a description of the **incompletetimeout** and **completetimeout** properties, see ""Timeout properties" on page 115.
- If the number of digits collected on the initial attempt indicates that the user probably paused too long at one of the logical grouping points, you might try collecting just the remaining digits. For example, if the recognition engine returned only 6 digits, it may be that the user paused too long after speaking the area code plus exchange. In this case, you might prompt only for the final four digits of a 10-digit USA telephone number.
- Design your application to require a fixed number of digits (for example, always requiring the area code, even for local telephone numbers).
- Allow or encourage users to use DTMF when entering telephone numbers.
- Have the application read back telephone numbers for user verification.

Note: These guidelines apply to any long alphanumeric string with a predictable format.

Testing grammars

When testing your grammars, you should test words and phrases that are out of your grammars as well as words and phrases that are in. (The purpose of testing "out-of-grammar" words and phrases is to ensure that the speech recognition engine is rejecting these utterances; erroneously accepting these utterances could cause unintended dialog transitions to occur.)

If your grammar tools allow you to list (enumerate) commands that the grammar can recognize, you can examine these lists of commands for phrases that you do not want to include in the grammar. For example, if a list of commands contains the sentence ″Play the next next message,″ you can modify the grammar to prevent inappropriate duplication of words.

If your application has more than one grammar active concurrently, you should test each grammar separately, and then test them together.

To help identify if there are any words that are consistently misrecognized, you should test your grammar with a group of test subjects that is representative of the demographics and environments of your users. For

example, you might want to vary the ambient noise level, gender, age, accent, and level of fluency during desktop testing. When you are ready to deploy your application, you may want to perform additional testing while varying the type of telephone (standard, cordless, cellular, and speaker phone, etc.).

If you discover words or phrases that are consistently problematic, you might need to rephrase some entries or add multiple pronunciations.

Remember that testing your grammar is an iterative process. As you make changes, you should go back and retest to verify that all of the valid words and phrases can still be recognized.

Error recovery and confirming user input

Error recovery

Error recovery in speech applications is very important because speech is transient and speech applications rarely have written user documentation. Self revealing help can be used to address many error recovery situations (see 40). Additional strategies are shown in Table 19.

Table 19. Error-recovery techniques

Situation	Example	Recommended strategy	Sample
System determines that recognized user input is invalid.	Incorrect number of digits in a social security number.	State the problem (without blaming the user because the problem might be in the user's utterance or in the system's interpretation of that utterance) and reprompt. Consider directing the user to make the second entry with the keypad.	System: I didn't get nine digits. Please use your keypad to enter your social security number.

Table 19. Error-recovery techniques (continued)

Situation	Example	Recommended strategy	Sample
A recognition error occurs while the user is making choices along a menu path or completing items in a form.	User is paying a telephone bill.	Feed recognized input forward into next prompt. Be sure to include information about the Go Back command in first level help.	System: Pay how much? User: $43.15 System: Paying $53.50 with electronic check or credit card? User: No, that's not right. To change the previous entry, say Go Back. To continue select electronic check or credit card.
User did not hear all of the information presented.	User distracted during the presentation of a menu.	Include the Repeat command as an always active command in the introduction. Following a 1500 to 2500 ms pause, play the always active commands after primary menu prompts. This is especially useful at task terminal points. Provide the options again in the first level help. Repeat the set of always active options in the second level help, then repeat the options again.	System: Say Make Payment, Account Balance, Become new customer, make purchase or Customer Service. User: [the user was distracted and says nothing] System: <2000 ms pause> At any time you can say Help, Repeat, Go Back, Main Menu or Exit. User: Repeat.

Understanding spoke-too-soon and spoke-way-too-soon incidents

In applications that have disabled barge-in, the user might cause an error by speaking before hearing the tone that indicates the system is ready for recognition. If the user continues speaking over the tone and into the recognition timeframe. This is called a *spoke-too-soon* (STS) incident (illustrated in Figure 1 on page 66). Because a portion of the user utterance was spoken outside of the recognition timeframe, the input that the speech recognition engine actually received often does not match anything in the active grammars, so the speech recognition engine treats it as an out-of-grammar utterance (triggering a nomatch event).

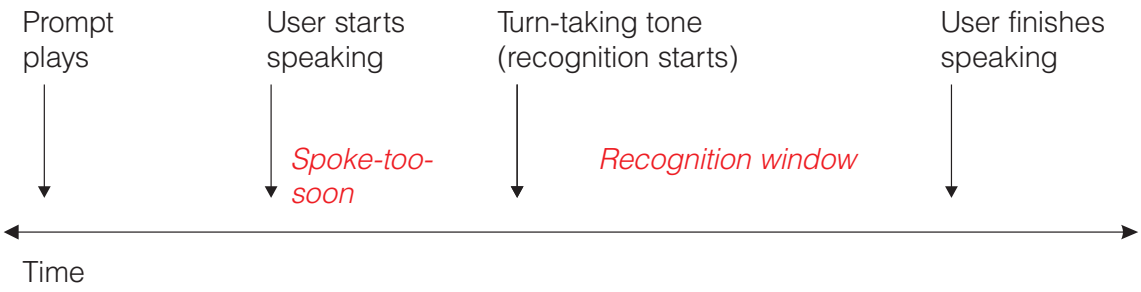


Figure 1. Spoke-too-soon (STS) incident

It is also possible for a user to finish speaking before the tone sounds; this is called a *spoke-way-too-soon* (SWTS) incident (illustrated in Figure 2). Because the entire user utterance occurs outside of the recognition timeframe, the speech recognition engine does not actually receive any input and the system will generally time out (triggering a noinput event) as it waits for the input that the user already gave.

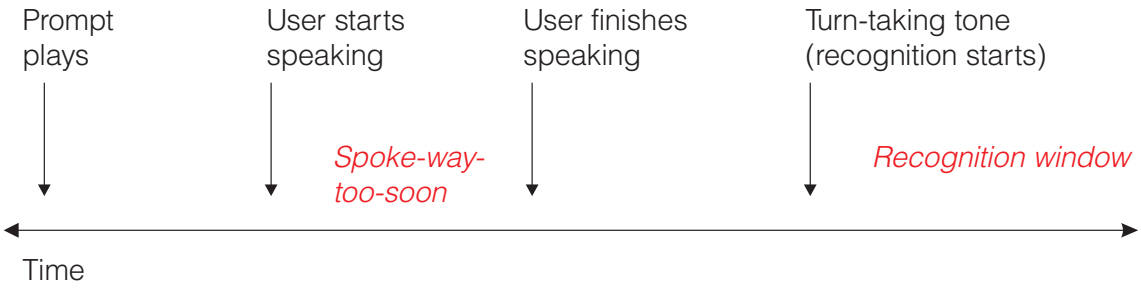


Figure 2. Spoke-way-too-soon (SWTS) incident

Minimizing STS and SWTS incidents: Spoke-too-soon and spoke-way-too-soon incidents can be a common source of speech recognition errors, especially in barge-in disabled applications with excessively long prompts. Here are some things you can do to minimize recognition errors associated with these incidents:

- Control the length of your system prompts to prevent them from being excessively long.
- Aggressively trim silence from the end of prerecorded audio files to prevent time lapses between the end of the prompt and the presentation of the tone.
- Place the "Please wait for the tone" prompt at the end of the introductory message, just before you request the initial input from the user.

Recovering from STS and SWTS incidents: These incidents can become more frequent as users become expert with a half-duplex system and rush to

provide input. A spoke-too-soon or spoke-way-too-soon incident almost never indicates a problem with the user understanding the system; therefore, the appropriate system response is to let the user get back to the application as quickly as possible, with a reminder to wait for the tone. For example:

```
System:     Do you want to do another transaction?
User:       (interrupting the tone) Yes.
System:     Please say yes or no.
User:       (speaking completely over the prompt, resulting in silence timeout)Yes.
System:     Remember to speak after the tone. Please say "yes" or "no."
```

Implications for self-revealing help when barge-in has been disabled:: If an application in which barge-in has been disabled returns a nomatch or noinput event, you can not be sure whether it was truly an out-of-grammar utterance or timeout, or whether it was caused by a spoke-too-soon or spoke-way-too-soon incident. Consequently, you should include early in the self-revealing help sequence a reminder to wait for the tone, along with whatever other help you want to provide for these return codes; this "wait for the tone" reminder is not necessary when the user said a phrase from the Help grammar.

Confirming user input

You should always ask for user confirmation if the system is about to perform an action it cannot undo. For example:

```
System:     Preparing to transfer $500 from checking to savings.
            Please say Transfer or Cancel.
```

For other data entry situations, there are a number of confirmation strategies available. These strategies fall into the general categories of immediate and delayed (or batch) confirmation.

Immediate confirmation: separate dialog turns: The simplest strategy is to provide a separate dialog turn for each item. The advantage of this approach is that there is no ambiguity and users enjoy a very high success rate. The disadvantage is that the additional dialog turns make the call longer and can lead to an interaction that feels sluggish. The more items there are, the more sluggish the interaction feels. For example:

```
System:     Which fund?
User:       Fidelity Magellan
System:     Was that Fidelity Magellan?
User:       yes
```

Immediate confirmation: feed forward: Another approach to immediate confirmation is to feed the data forward into the next prompt. The advantage of this approach is that it uses implicit confirmation to save a dialog step. The disadvantages are that the user action to disconfirm is not evident and it is trickier to develop this type of prompt when you are using recordings rather than TTS. For example:

```
System:     Which fund?
User:       Fidelity Magellan
System:     Buy how many shares of Fidelity Magellan?
User:       100 shares
```

Delayed (batch) confirmation: basic: The basic delayed confirmation strategy is to play all collected data in a single step, correct any incorrect items, then repeat the confirmation. The advantage of this approach is that if there are no errors, then all confirmation is done in a single step. The disadvantage is that if there are more than two errors, the correction process is very lengthy. The following example illustrates the resulting interaction if there are no errors:

```
System:     Which fund?
User:       Fidelity Magellan
System:     Buy how many shares?
User:       100
System:     Was that 100 shares of Fidelity Magellan?
```

However, if there are two errors, the resulting dialog is:

```
System:     Which fund?
User:       Fidelity Magellan
System:     Buy how many shares?
User:       100
System:     Was that 200 shares of Fidelity No Load?
User:       No
System:     Which item do you want to change?
User:       The number of shares.
System:     To how many shares?
User:       100
System:     Was that 100 shares of Fidelity No Load?
User:       No
System:     Which item do you want to change?
User:       The fund
System:     Change to which fund?
User:       Fidelity Magellan
System:     Was that 100 shares of Fidelity Magellan?
```

```
User:       Yes
```

The more items there are in the confirmation message, the more pronounced this problem becomes.

Delayed (batch) confirmation: improved: To achieve the advantages of delayed confirmation while eliminating the disadvantages of the basic strategy, there are several alternative delayed confirmation methods. The most important aspect of the improved methods is the elimination of the repeated playing of the full confirmation message. If there are no errors, the interaction is the same as that described for the basic delayed confirmation strategy. If there are two errors, the interaction could be:

```
System:     Which fund?
User:       Fidelity Magellan
System:     Buy how many shares?
User:       100
System:     Was that 200 shares of Fidelity No Load?
User:       No
System:     What do you want to change first?
User:       The number of shares.
System:     To how many shares?
User:       100
System:     Was that 100 shares?
User:       Yes
System:     Change anything else?
User:       Yes, the fund
System:     Change to which fund?
User:       Fidelity Magellan
System:     Was that Fidelity Magellan?
User:       Yes
```

If full confirmation (all data) is a customer requirement, do not engage in the immediate confirmation steps. Instead the interaction would be:

```
System:     Which fund?
User:       Fidelity Magellan
System:     Buy how many shares?
User:       100
System:     Was that 200 shares of Fidelity No Load?
User:       No
System:     What do you want to change first?
User:       The number of shares.
System:     To how many shares?
User:       100
System:     Change anything else?
```

```
User:       Yes, the fund.
System:     Change to which fund?
User:       Fidelity Magellan
System:     Was that 100 shares of Fidelity Magellan?
User:       Yes
```

Refining confirmation and error correction with confidence levels and n-best lists: It is possible to use information about confidence levels and n-best lists to refine confirmation and error correction strategies. Confidence levels are values produced by speech recognition engines in which utterances with close matches to recognized words get higher scores. n-best lists are the top *n* matches produced by a speech recognizer and ranked by confidence level. The match with the highest confidence level is the one that the engine returns to the application.

If you are using an immediate confirmation strategy with feed forward, do not put the feedback in the following prompt unless the recognition confidence for the item feeding forward is high. For example, if confidence is high:

```
System:     Which stock?
User:       Texaco (recognized with high confidence)
System:     How many shares of Texaco?
User:       500
System:     Sell 500 shares of Texaco?
User:       Yes
```

But if the recognition confidence is low:

```
System:     Which stock?
User:       Texaco (recognized as PepsiCo but with low confidence)
System:     How many shares?
User:       500
System:     What was that stock?
User:       Texaco (recognized this time with high confidence)
System:     Sell 500 shares of Texaco?
User:       Yes
```

Another refinement is to use n-best lists for disambiguation when the top candidates in the n-best list have close confidence scores. If you can apply back end logic to disambiguate the candidates then do so. If this isn't possible, provide a disambiguation dialog turn. For example:

```
System:     Do you want to buy Texaco?
User:       No.
```

```
System:     Buy PepsiCo?
User:       Yes
```

You can use the same approach for disambiguating homophones (such as Cisco and Sysco). Another refinement is to program the system so it will not make repeated recognition errors. One way to do this is the *use n-best* query instead of asking for repetition. This works well when top candidates in an n-best list have high recognition, but will not work well if the utterance was out of grammar.

To avoid repeated recognition errors:

- Reject a first choice that the user has already refused and present the second item in the n-best list instead.
- Use n-best or conditional probabilities to support correction for data entered with voice spelling (see "Tips for voice spelling" on page 55).
- If there's no match at the back end, create likely combinations and check to see if one or more of them matches.
- If no back end comparisons are possible, use lists to guide error recovery.

Advanced user interface topics

The guidelines presented in the previous sections of this chapter cover the fundamentals of creating a clean, usable SUI. You don't have to stop there, though, if you have the resources and motivation to create a more advanced user interface. The advanced user interface topics covered in this section are:

- "Issues in artificial personae"
- "Controlling the "lost in space" problem" on page 72
- "Managing audio lists" on page 72

Issues in artificial personae

Users tend to attribute human qualities to personified user interfaces. There are few guidelines for the design of effective personified user interfaces. In general, you should avoid personified user interfaces (applications in which the system has a name) unless there is a compelling reason to build one. When designing a persona, be user centered, not machine centered. No matter what you do, users will perceive your system as having a personality, so you should deliberately design your system to have a personality appropriate for the application. For applications that have self-service and call-routing functions, you will usually want a professional persona. You will also want the persona to be consistent with the client company's brand and other aspects of corporate image.

Key attributes of a professional persona are:

- Be efficient when communication is good.

- Be understanding and helpful when progress is slow.
- Always be courteous and polite.
- Assume that the user is busy.
- Never assign blame.
- Don't apologize too often.
- Try not to be too terse or verbose.

Controlling the "lost in space" problem

Users rarely get lost when using simple, straightforward applications. When an application contains deeply-nested menus, though, the user can get disoriented, causing the interaction between the user and the application to break down.

Minimizing the number of nested menus

To help users avoid this "lost in space" dilemma, you should try to minimize the number of nested menus required to reach any given dialog state. In speech interfaces (as in visual interfaces), users are less likely to get lost in a broad structure as opposed to a deep structure.

Using audio formatting

In addition, you may want to consider using audio formatting to help orient the user. For example, you might play different introductory or background tunes for different modules of the your application.

Providing landmarks

Another technique to consider is providing auditory landmarks at key places in the application. For example:

```
System:     You can make, change or cancel reservations. What would you like to do?
User:       I'd like to make a reservation.
System:     Making reservation (This is the landmark)
```

Managing audio lists

In general, audio menus should not contain more than a few items; recommended maximums are described in Table 14 on page 49. There might be times when you have no choice but to present a list of more than the recommended number of items. If so, a good way to manage the list may be by using a "speak to select" strategy or list-scanning commands.

Using a "speak to select" strategy

When presented with a running list of items, users (even first-time users) typically say one of a small number of phrases to interrupt the list when they hear the item they want, including utterances such as "OK", "Stop", or "Yes", or simply repeating the target item. For example:

```
System:     When you hear the name of the desired video, please repeat it:
            The Maltese Falcon
            Gone with the Wind
            The Nutty Professor
            The War of the Worlds

User:       Yes.
System:     Adding "The War of the Worlds" to your shopping cart.
```

Note: The prompt still suggests that the user repeat the desired movie name (rather than just saying "yes') because this strategy takes some of the time pressure off the user. The user can choose an item even after several additional selections have been read.

Using list-scanning commands

For very long lists, you may want to consider providing list-scanning commands that permit bi-directional navigation, such as BACKWARD, FORWARD, NEXT, PREVIOUS, TOP, and BOTTOM.

Unless you have previously identified that the user has experience with the system, you should plan to announce these list-scanning commands before presenting the list. You may also want to consider using special audio tones as feedback for the recognition some of these commands. For example:

```
User:       List the stocks in my portfolio.

System:     Your portfolio has fifty stocks. You can use the commands FORWARD, BACKWARD,
            NEXT, PREVIOUS, TOP and BOTTOM to control the playback of this list.
            General Motors
            IBM
            General Mills
            Hannaford Brothers
            Eastman Kodak
User:       Backward.
System:     (backward tone)
            Eastman Kodak
            Hannaford Brothers
            General Mills

User:       Forward.
System:     (forward tone)
            General Mills
            Hannaford Brothers
User:       Price.
System:     The current price for Hannaford Brothers is 25 7/8, up 1/8.
User:       Next.
System:     The current price for General Mills is 87 3/4, down 1/4.
User:       Bottom.
```

```
System:     (bottom tone)
            The current price for Wal-Mart Stores Inc. is 54, unchanged.
User:       Backward.
System:     (backward tone)
            Wal-Mart Stores Inc.
            Kellogg Company
            Monsanto Company
            (etc.)
```

Chapter 3. VoiceXML language

This chapter provides a brief introduction to basic VoiceXML concepts and constructs. It describes IBM's implementation of Version 2.1 of VoiceXML and provides an overview of the differences between VoiceXML Version 2.0 and Version 2.1. The information provided in this chapter is not a complete description of the functionality of the language. Refer to the VoiceXML 2.1 specification at **http://www.w3.org/TR/voicexml21/** for more information.

This chapter discusses the following topics:

- "Changes from VoiceXML 2.0"
- "The structure of a VoiceXML application" on page 76
- "A simple VoiceXML example" on page 80
- "VoiceXML elements and attributes" on page 82
- "Speech Synthesis Markup Language (SSML)" on page 90
- "Built-in field types and grammars" on page 92
- "Recorded audio" on page 95
- "Document fetching and caching" on page 96
- "Events" on page 96
- "Variables and expressions" on page 99
- "Grammars" on page 104
- "Timeout properties" on page 115
- "Telephony functionality" on page 117

Changes from VoiceXML 2.0

This section describes the changes added to VoiceXML in the 2.1 specification, as compared to VoiceXML 2.0. For a complete description of the differences between VoiceXML version 2.0 and version 2.1, refer to **http://www.w3.org/TR/voicexml21/** .

New elements and attributes

VoiceXML 2.1 introduces the following new elements and attributes:

- The element, **<data>** allows a VXML application to fetch XML data from a document server without transitioning to a new VXML document.
- The **<foreach>** element allows a VoiceXML application to iterate through an ECMAScript array and to execute the content contained within the **<foreach>** element for each item in the array.
- A second attribute, **nameexpr** has been added to the **<mark>** element.

- The **namelist** attribute has been added to the **<disconnect>** element.
- The **recordutterance** and **recordutterancetype** properties have been added, to be used with the **<property>** element.
- The **srcexpr** attribute has been added to the **<grammar>** element.
- The **srcexpr** attribute has been added to the **<script>** element.
- The **type** attribute has been added to the <transfer> element.

Various existing elements have changed in their implementation. Where this is the case it is indicated in Table 20 on page 82. You will need to refer to the W3C VoiceXML 2.1 specification for details.

Speech Synthesis Markup Language (SSML)

The Speech Synthesis Markup Language (SSML) is a new XML based markup language for handling text-to-speech (TTS) within a VoiceXML application. The new set of SSML elements has made the VoiceXML 1.0 elements dealing with TTS obsolete. This includes , <emp>, <pros> and <sayas> as well as IBM's extension elements <ibmlexicon>, <ibmvoice> and <word>. For a summary of the new SSML elements and attributes, see "Speech Synthesis Markup Language (SSML)" on page 90.

The structure of a VoiceXML application

A VoiceXML application is made up of one or more VoiceXML documents that together control the dialog with the caller. These documents typically consist of a single *root* document, and several *leaf* documents. Application specific information such as global commands, event handlers and global variables, are typically defined within the root document, while individual dialogs and local variables are placed in leaf documents. A leaf document indicates that it is part of a larger application by using the **application** attribute on the **<vxml>** element to specfy the URI of the application's root document.

VoiceXML documents are composed primarily of top-level elements called *dialogs*. There are two types of dialogs defined in the language: **<form>** and **<menu>**.

Forms and form items

Forms allow the user to provide voice or DTMF input by responding to one or more form items. The elements <field>, <subdialog>, <block>, <initial>, <record>, <transfer> and <object> are all form items.

Fields

Each field can contain one or more **<prompt>** elements that guide the user to provide the desired input. You can use the count attribute to vary the prompt based on the number of times that the prompt has been played.

Fields can also specify a **type** attribute or a **<grammar>** element to define the valid input values for the field, and any **<catch>** elements necessary to process the events that might occur. Fields may also contain **<filled>** elements, which specify code to execute when a value is assigned to a field. You can reset one or more form items using the **<clear>** element.

Subdialogs

The **<subdialog>** element is similar to a function call in a traditional programming language. It can be used to gather information and return it to the form. For more information, see "Subdialogs" on page 79.

Blocks

If your form requires prompts or computation that do not involve user input (for example, welcome information), you can use the **<block>** element. A **<block>** can also contain executable content such as **<goto>** or **<submit>**.

Types of forms

There are two types of form dialogs:

- *Machine-directed* forms — traditional forms where each field or other form item is executed once and in a sequential order, as directed by the system.
- *Mixed-initiative* forms — more robust forms in which the system or the user can direct the dialog. When coding mixed-initiative forms, you can use form-level grammars to allow the user to fill in multiple fields from a single utterance. You can use the **<initial>** element to prompt for form-wide information in a mixed-initiative dialog, before the user is prompted on a field-by-field basis. For more information, see "Mixed-initiative application and form-level grammars" on page 111.

Menus

A menu is essentially a simplified form with a single field. Menus present the user with a list of choices, and associate with each choice either a URI identifying a VoiceXML page or element to visit or an event which will occur if the user selects that choice. The grammar for a menu is constructed dynamically from the menu entries, which you specify using the **<choice>** element, text or an external grammar reference. You can use the **<menu>** element's **scope** attribute to control the scope of the grammar.

Note: The **<enumerate>** element instructs the VoiceXML browser to speak the text of each menu **<choice>** element when presenting the list of available selections to the user. If you want more control over the exact wording of the prompts (such as the ability to add words between menu items or to hide active entries in your menu), simply leave off the **<enumerate>** tag and specify your own prompts.

Menus can accept voice and/or DTMF input. If desired, you can implicitly assign DTMF key sequences to menu choices based on their position in the

list of choices by using the construct **<menu dtmf="true">**. The following example shows an example of a menu that accepts voice and/or DTMF input.

```
<?xml version="1.0" encoding="iso-8859-1"?>
   <vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

   <menu>

   <prompt>
      Welcome to the main menu.
      Please choose from the following choices.<enumerate/>
   </prompt>

   <choice dtmf="1" next="http://www.example.com/news.vxml">
      current news
   </choice>

   <choice dtmf="2" next="http://www.example.com/weather.vxml">
      current weather
   </choice>

   </menu>

</vxml>
```

Flow control

When the VoiceXML browser starts, it uses the URI you specify to request an initial VoiceXML document. Based on the interaction between the application and the user, the VoiceXML browser may jump to another dialog in the same VoiceXML document, or fetch and process a new VoiceXML document.

VoiceXML provides a number of ways for managing flow control. For example, the <**link**> element specifies a control common to all dialogs in the link's scope.

```
<link event="help">
```

```
<link next="http://www.yourbank.example/locations.vxml/">
```

The <**goto**> element directs the flow to another dialog or item in the same or different document.

```
<goto nextitem="field1"/>
```

The <**submit**> element is like the <**goto**> element but provides for submitting variables to the server.

```
<submit next="/servlet/start" namelist="family_name date_of_birth"/>
```

The <**choice**> element lists a menu choice.

```
<choice next="#option1">Account balance </choice>
```

If no transition is specified to a new dialog, the VoiceXML browser will exit when no more fields remain unprocessed in the current dialog.

Subdialogs

VoiceXML subdialogs are roughly the equivalent of function or method calls. A subdialog is an entire form that is executed, the result of which is used to fill one input item in the calling form. You can use subdialogs to provide a disambiguation or confirmation dialog (as described in "Error recovery" on page 64), as well as to create reusable dialog components for data collection and other common tasks.

Executing a **<subdialog>** element temporarily suspends the execution context of the calling dialog and starts a new execution context, passing in the parameters specified by the **<param>** element. When the subdialog has completed, it uses a **<return>** element to resume the calling dialog's execution context. The calling dialog accesses the results of the subdialog through the variable defined by the **<subdialog>** element's name attribute.

Comments

Information within a comment is ignored by the VoiceXML browser.

Single line comments in VoiceXML documents use the following syntax

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

Comments can also span multiple lines:

```
<!--start of multi-line comment
    more comments-->
```

A simple VoiceXML example

The following example illustrates the basic dialog capabilities of VoiceXML using a menu and a form:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

   <menu>

      <prompt>Welcome to the online banking super store.</prompt>
      <prompt>Please choose from the following choices.<enumerate/></prompt>

      <choice next="http://www.yourbank.example/locations.vxml">
         Branch Locations
      </choice>

      <choice next="http://www.yourbank.example/interest.vxml">
         Interest Rates
      </choice>

      <choice next="#invest">
         Investment Information
      </choice>

   </menu>


   <form id="invest">

      <field name="investment_amount" type="currency">
         <prompt>
            How much would you like to invest?
         </prompt>
      </field>

      <field name="zip_code">
         <grammar mode="dtmf" src="builtin:dtmf/digits"/>
         <prompt>Use your telephone keypad to enter the five-digit ZIP code
            of your location, followed by the # key.
         </prompt>
      </field>

      <filled>
         <submit next="http://www.yourbank.example/servlet/invest"
                     namelist="investment_amount zip_code" />
      </filled>

   </form>

</vxml>
```

Each menu or form field in a VoiceXML application must define a set of acceptable user responses. The menu uses **<choice>** elements to do this, while the **<field>** elements in the above example use the **type** attribute and the

<grammar> element to specify built-in grammars (here, **"currency"** and **"digits"**). You can also create your own application-specific grammars. (See "Grammars" on page 104.)

The resulting dialog can proceed in numerous ways, depending on the user's responses. Two possible interactions are described next.

Static Content

One sample interaction between the system and the user might sound like this:

```
System:     Welcome to the online banking super store.
            Please choose from the following choices.
            Branch Locations, Interest Rates, Investment Information.

User:       Branch Locations.
```

At this point, the system retrieves and interprets the VoiceXML document located at **http://www.yourbank.example/locations.vxml**. This new document specifies the next segment of the interaction between the system and the user, namely a dialog to locate the nearest bank branch.

This interaction uses the **<menu>** element, but not the **<form>** element, from the sample VoiceXML document. It illustrates a level of capability similar to that provided for visual applications by a set of static HTML hyperlinks. For dynamic distributed services, you need to also use the **<form>** element.

Dynamic content

This interaction uses both the **<menu>** and the **<form>** from the above VoiceXML example:

```
System:     Welcome to the online banking super store.
            Please choose from the following choices.
            Branch Locations, Interest Rates, Investment Information.

User:       Investment Information.
System:     How much would you like to invest?
User:       (says the amount)
System:     Use your telephone keypad to enter the five-digit ZIP code of your location,
            followed by the # key.
User:       (presses telephone keys corresponding to the ZIP code, and terminates data entry
            using the # key)
```

At this point, the system has collected the two needed fields **investment_amount** and **zip_code**, so it executes the **<filled**> element, which contains a **<submit>** element that causes the collected information to be

submitted to a remote server for processing. The remote server processes the transaction and then returns a VoiceXML page telling the user what happened.

VoiceXML elements and attributes

VoiceXML is an XML-based application, meaning that it is defined as a set of XML tags or elements. Table 20 lists the VoiceXML elements and provides implementation details specific to the VoiceXML browser in the WebSphere Voice Server products. For additional information on grammar support in WebSphere Voice Server, see "Grammars" on page 104.

Table 20. Summary of VoiceXML elements and attributes

Element	Description	Implementation details
<assign>	Assigns a value to a variable.	Supported as documented in VoiceXML 2.0.
<audio>	Plays an audio file within a prompt or synthesizes speech from a text string if the audio file cannot be found.	Supported as documented in VoiceXML 2.0; You can specify an audio file using a URI or you can use the **expr** attribute to play audio recorded using the **<record>** element. The supported audio formats are: • an 8KHz 8-bit mu-law **.au** file • an 8KHz 8-bit mu-law **.wav** file • an 8KHz 8-bit a-law **.au** file • an 8KHz 8-bit a-law **.wav** file • an 8KHz 16-bit linear **.au** file • an 8KHz 16-bit linear **.wav** file Use an 8-bit format whenever possible; 16-bit linear files take twice as much storage and require twice as long to download. WebSphere Voice Server for WebSphere Voice Response uses the WebSphere Voice Response system for audio playback.
<block>	Specifies a form item containing non-interactive executable content.	Supported as documented in VoiceXML 2.0.
<break>	Inserts a pause in TTS output.	VoiceXML 2.0 inherits this element from SSML. For more information on programming for TTS, see "Speech Synthesis Markup Language (SSML)" on page 90.
<catch>	Catches an event.	Supported as documented in VoiceXML 2.0.
<choice>	Specifies a menu item.	Supported as documented in VoiceXML 2.0.

Table 20. Summary of VoiceXML elements and attributes (continued)

Element	Description	Implementation details
<clear>	Clears one or more form item variables.	Supported as documented in VoiceXML 2.0.
<data>	Allows a VoiceXML application to fetch arbitrary XML data from a document server without transitioning to a new VoiceXML document.	Supported as documented in VoiceXML 2.1.
<disconnect>	Causes the VoiceXML browser to disconnect from a user telephone session.	VoiceXML 2.1 added the **namelist** attribute to this element. Otherwise, supported as documented in VoiceXML 2.0.
** **	Identifies the type of text for TTS output.	This element is not part of VoiceXML 2.0 and is not supported by WebSphere Voice Server. For more information on programming for TTS, see "Speech Synthesis Markup Language (SSML)" on page 90.
<dtmf>	Specifies a DTMF grammar.	This element is not part of VoiceXML 2.0 and is not supported by WebSphere Voice Server. You can, however, now specify a DTMF grammar using the **<grammar>** element with the **mode="dtmf"** attribute. **Note:** DTMF grammars used with WebSphere Voice Response do not support the use of hotword barge-in. This means that if a prompt is played with **bargeintype="hotword"** and a DTMF grammar is active, then the prompt will stop as soon as any DTMF key is detected and the prompt will be played as though it was set to **bargeintype="speech"**. This occurs regardless of whether or not any speech grammars are also active.
<else>	Conditional statement used with the **<if>** element.	Supported as documented in VoiceXML 2.0.
<elseif>	Conditional statement used with the **<if>** element.	Supported as documented in VoiceXML 2.0.
<emp>	Specifies the emphasis for TTS output.	This element has been replaced by the SSML element **<emphasis>**. For more information, see "Speech Synthesis Markup Language (SSML)" on page 90.

Table 20. Summary of VoiceXML elements and attributes (continued)

Element	Description	Implementation details
<enumerate>	Shorthand construct that causes the VoiceXML browser to speak the text of each **<choice>** element when presenting the list of menu selections to the user.	When using the **<enumerate>** element to play menu choices with the TTS engine, you do not have to add punctuation to control the length of the pauses between **<choice>** elements. The VoiceXML browser will automatically add the appropriate pauses and intonations when speaking the prompts.
<error>	Catches an error event.	Supported as documented in VoiceXML 2.0.
<exit>	Exits a VoiceXML browser session.	As described in VoiceXML 2.0, the **<exit>** element returns control to the interpreter, which determines what action to take. The **namelist** attribute is ignored. WebSphere Voice Response supports the use of the **expr** attribute to associate data with the call.
<field>	Defines an input field in a form.	Supported as documented in VoiceXML 2.0.
<filled>	Specifies an action to execute when a field is filled.	Supported as documented in VoiceXML 2.0.
<foreach>	Allows a VoiceXML application to iterate through an ECMAScript array and to execute the content contained within the element for each item in the array.	Supported as documented in VoiceXML 2.1.
<form>	Specifies a dialog for presenting and gathering information.	Supported as documented in VoiceXML 2.0. The VoiceXML browser supports mixed initiative dialogs using SISR. See "Mixed-initiative application and form-level grammars" on page 111 for details.
<goto>	Specifies a transition to another dialog or document.	Supported as documented in VoiceXML 2.0.
<grammar>	Defines a speech recognition grammar.	Voice XML 2.1 added the **srcexpr** attribute to this element. Supported values for the **type** attribute are: **Grammar** — **Type** SRGF ABNF — **application/srgs** SRGF XML — **application/srgs+xml** See "Grammars" on page 104 for additional information on grammar support in WebSphere Voice Server.
<help>	Catches a help event.	Supported as documented in VoiceXML 2.0.
<ibmlexicon>	A container for one or more **<word>** elements.	This is not supported by WebSphere Voice Server.

Table 20. Summary of VoiceXML elements and attributes (continued)

Element	Description	Implementation details
<ibmvoice>	Controls the TTS engine's synthesized voice on a per-prompt basis.	This is not supported by WebSphere Voice Server. For more information on programming for TTS, see "Speech Synthesis Markup Language (SSML)" on page 90.
<if>	Defines a conditional statement.	Supported as documented in VoiceXML 2.0.
<initial>	Prompts for form-wide information in a mixed-initiative form.	Supported as documented in VoiceXML 2.0.
<link>	Specifies a transition to a new document or throws an event, when activated.	Supported as documented in VoiceXML 2.0.
<log>	Generates a debug message.	Supported as documented in VoiceXML 2.0. For details about where data is logged or how data is accessed, refer to the *Administrator's Guide*.
<mark>	Places a marker into the text/tag sequence.	Supported as documented in VoiceXML 2.1.
<menu>	Specifies a dialog for selecting from of a list of choices.	Supported as documented in VoiceXML 2.0.
<meta>	Specifies meta data about the document.	The **http-equiv** attribute is supported for the values **date, expires** and **lastModified**. However, **<meta>** declarations in Speech Recognition Grammars are ignored. See Table 28 on page 104 for more information. The VoiceXML browser ignores the **name** attribute and any **content** specified with it. You can use these attributes to identify and assign values to the properties of a document, as defined by the HTML 4.0 specification (**http://www.w3.org/TR/REC-html40/**) and the HTTP 1.1 specification (**http://www.ietf.org/rfc/rfc2616.txt**).
<metadata>	Defines metadata information using a metadata schema	This is new for VoiceXML 2.0 but it is ignored by the VoiceXML browser if it is CDATA; otherwise an error is generated.
<noinput>	Catches a **noinput** event.	Supported as documented in VoiceXML 2.0.
<nomatch>	Catches a **nomatch** event.	Supported as documented in VoiceXML 2.0.

Table 20. Summary of VoiceXML elements and attributes (continued)

Element	Description	Implementation details
<object>	Specifies platform-specific objects.	For WebSphere Voice Server, some attributes are supported in specific ways: **classid** The following URI is supported: **method://<fully qualified java classname>/<java method name>** **codetype** The value `javacode` is supported for the **codetype** attribute. **archive** The archives can be arbitrary Jar files containing Java classes referenced by the **classid** attribute. The archives must be specified in a fully qualified URL format. If archives are not supplied, the system class loader (local class path) is used to resolve **classid** references. Fetching and caching attributes are not supported. All Ecmascript integers are passed to Java objects as **java.lang.Double**. If the VoiceXML application is to pass an integer value to a Java object, this must be done through a method which accepts that value as a **java.lang.Double** parameter object. A sample program that uses the **<object>** element is in "Calling a Java application" on page 129.
<option>	Specifies a field option.	Supported as documented in VoiceXML 2.0.
<param>	Specifies a parameter in an object or subdialog.	Supported as documented in VoiceXML 2.0.
<prompt>	Plays TTS or recorded audio output.	Supported as documented in VoiceXML 2.0.

Table 20. Summary of VoiceXML elements and attributes (continued)

Element	Description	Implementation details
<property>	Controls aspects of the behavior of the implementation platform.	The attributes: • **objectfetchhint** • **objectmaxage** • **objectmaxstale** are ignored by the VoiceXML browser. The names: • **sensitivity** • **speedvsaccuracy** are supported in this release. The names: • **com.ibm.speech.asr.saveaudio** • **com.ibm.speech.asr.saveaudiotype** • **com.ibm.speech.asr.endpointed** are supported in addition to those documented in VoiceXML 2.0. See "Recording user input during speech recognition" on page 95 for more information. In addition to the names listed above, VoiceXML 2.1 adds the **recordutterance** and **recordutterancetype** names to this element. If the value for **recordutterance** is set to "true" then three shadow variables may be used. For more details, see the VoiceXML 2.1 specification. Use the **recordutterancetype** property name to specify media formats for utterance recordings. For more details, see the VoiceXML 2.1 specification. The **confidencelevel** property specifies a threshold for determining whether recognition results, or scores, should be accepted by the VoiceXML application. When scores are above the confidence-level threshold, the VoiceXML browser considers the recognized words acceptable; the appropriate handlers are called, and the array of the shadow variable **application.lastresult$** is filled with the scores up to the **maxnbest** value. For a **<field>**, for example, the **<filled>** handler would be called. If no scores are above the confidence level, appropriate **<nomatch>** handlers are executed. **maxnbest** returns a maximum of 100 *n*-best results to the application. A sample program using **maxnbest** is shown in "Using *n*-best" on page 134.)

Table 20. Summary of VoiceXML elements and attributes (continued)

Element	Description	Implementation details
<pros>	Controls the prosody of TTS output.	This element has been replaced by the SSML element **<prosody>**. For more information, see "Speech Synthesis Markup Language (SSML)" on page 90.
<record>	Records spoken user input.	Any grammars active during **<record>** are ignored. The timeout attribute specified for a prompt is ignored during **<record>**; no noinput event is generated. If you are using the Voice Toolkit, finalsilence is ignored. This means that a recording will not terminate on an extended period of silence. The **type** attribute may take the following values: **audio/basic** Creates a .au file of 8kHz, 8-bit µ-law encoding **audio/x-alaw-basic** Creates a .au file of 8kHz, 8-bit a-law encoding **audio/x-wav** Creates a Microsoft wav file of 8kHz, 16-bit, linear PCM encoding The **beep** attribute defaults to false (a beep is not played). The **maxtime** attribute defaults to 60 seconds. The **final silence** attribute is set to a default of 12 seconds by WebSphere Voice Response. This value can only be changed from WebSphere Voice Response. Other attributes are not user-definable. Only the **duration**, **size** and **termchar** shadow variables have been implemented for **<record>**. **Size** contains the internal size of the raw audio data (8-bit headerless, µ-law or a-law encoding).
<reprompt>	Causes the form interpretation algorithm to queue and play a prompt when entering a form after an event.	Supported as documented in VoiceXML 2.0.
<return>	Returns from a subdialog.	Supported as documented in VoiceXML 2.0.
<sayas>	Controls pronunciation of words or phrases in TTS output.	This element has been replaced by the SSML element **<say-as>**. For more information, see "Speech Synthesis Markup Language (SSML)" on page 90.
<script>	Specifies ECMAScript code.	VoiceXML 2.1 adds the **srcexpr** attribute to this element. Otherwise, supported as documented in VoiceXML 2.0.

Table 20. Summary of VoiceXML elements and attributes (continued)

Element	Description	Implementation details
<subdialog>	Invokes a new dialog as a subdialog of the current one, in a new execution context.	Supported as documented in VoiceXML 2.0.
<submit>	Submits a list of variables to the document server.	Supported as documented in VoiceXML 2.0. To submit the results of a **<record>** element, you must use **enctype="multipart/form-data"** as shown in "Playing and storing recorded user input" on page 95.
<throw>	Throws an event.	Supported as documented in VoiceXML 2.0.
<transfer>	Connects the telephone caller to a third party.	VoiceXML 2.1 adds the **type** attribute to this element. The following attributes are not supported in this release: • **aaiexpr** • **connecttimeout** • **maxtime** • **transferaudio** The attribute **aai** is ignored by the VoiceXML browser. The attribute **bridge="true"** is supported only if WebSphere Voice Response is configured with a suitable telephony service. See *WebSphere Voice Response: Deploying and Managing VoiceXML and Java Applications* for more information. For bridged transfer (when **bridge="true"**), a conference call is set up to include the person or application specified by the **dest=attribute**.
<value>	Embeds a variable in a prompt.	Supported as documented in VoiceXML 2.0.
<var>	Declares a variable.	Refer to Table 25 on page 100 for information about variable scope and "Using shadow variables" on page 102 for information about shadow variables.
<vxml>	Top-level container for all other VoiceXML elements in a document.	All VoiceXML documents must specify either **version="2.0"** or **version="2.1"**, and **xmlns="http://www.w3.org/2001/vxml"**; if these attributes are missing, the VoiceXML browser will throw an **error.badfetch** event. The **xml:lang** attribute defaults to the language specified by the locale of the JVM in which the VoiceXML browser runs.

Table 20. Summary of VoiceXML elements and attributes (continued)

Element	Description	Implementation details
<word>	Specifies: • A pronunciation override, if the default TTS pronunciation for a word is not acceptable, or • An alternative pronunciation, if the speech recognition engine is having difficulty recognizing a word	This is not supported by WebSphere Voice Server. For more information on programming for TTS, see "Speech Synthesis Markup Language (SSML)."

Check for language specific limitations and considerations in the language appendixes.

Speech Synthesis Markup Language (SSML)

The Speech Synthesis Markup Language (SSML) is an XML based markup language for handling TTS within a VoiceXML application. VoiceXML 2.0 inherits a set of SSML 1.0 elements, as described in more detail in the VoiceXML 2.0 specification.

Table 21 summarises the SSML elements available in VoiceXML 2.0 and the WebSphere Voice Server implementation.

SSML elements and attributes

Table 21. Summary of SSML elements and attributes

Element	Description	Implementation details
<audio>	Plays an audio file within a prompt or synthesizes speech from a text string if the audio file cannot be found.	Supported as documented in VoiceXML 2.0, except that the **<audio>** element cannot be contained within other SSML elements, such as in:**<emphasis level="strong"> <audio .../> </emphasis>**
<break>	Inserts a pause in TTS output.	Supported as documented in VoiceXML 2.0.
<desc>	Describes the content of an audio source. This element is ignored by the VoiceXML browser.	Supported as documented in VoiceXML 2.0.
<emphasis>	Specifies the emphasis for TTS output.	Supported as documented in VoiceXML 2.0.

Table 21. Summary of SSML elements and attributes (continued)

<lexicon>	References external pronunciation definitions.	This element is not supported in WebSphere Voice Server.
<mark>	Inserts a reference point in a document.	This element is not supported in VoiceXML 2.0, but is supported in VoiceXML 2.1 as documented.
<metadata>	Specifies general information about the document.	This element is ignored by the VoiceXML browser.
<paragraph>	Specifies text structure in the absence of an end of sentence punctuation character.	Supported as documented in VoiceXML 2.0.
<phoneme>	Specifies the pronunciation and phonology for a TTS output.	Supported as documented in VoiceXML 2.0. WebSphere Voice Server supports the IPA and IBM alphabets. The IBM alphabet used in SSML refers to the phonology used by IBM TTS. The following example shows the US English phonetic pronunciation of "tomato" using the IBM TTS phonetic alphabet: `<phoneme alphabet="ibm" ph=".0tx.1me.0Fo"> tomato </phoneme>` For more information on IBM SPRs, see the *IBM Text-To-Speech SSML Programming Guide*.
<prosody>	Controls the pitch, range, rate and volume of TTS output.	The **contour** and **duration** attributes are not supported.
<say-as>	Specifies the type of text. For example, date, telephone number or currency.	The following attribute values are not supported: • **interpret-as="letters"** with **details="strict"** • **details="dictate"** • **interpret-as="words"** **Note:** The **details="punctuation"** attribute value is not supported in Simplified Chinese.**<say-as>** can be used with built-in grammar types.

Table 21. Summary of SSML elements and attributes (continued)

<sentence>	Specifies text structure in the absence of an end of sentence punctuation character.	Supported as documented in VoiceXML 2.0.
<sub>	Specifies substitute text for TTS output in place of a given input string.	Supported as documented in VoiceXML 2.0.
<voice>	Specifies the speaking voice used for TTS ouput.	Supported as documented in VoiceXML 2.0.

Language specific limitations and considerations are given in the language appendixes.

Built-in field types and grammars

The VoiceXML browser supports the complete set of built-in field types that are defined in VoiceXML 2.1. These built-in field types specify the built-in grammar to load (which determines valid spoken and DTMF input) and how the TTS engine will pronounce the value in a prompt, as documented in Table 22. Because these grammars provide support for many possible input values, developers may wish to write custom versions to constrain user input and to increase accuracy. Additional examples of the types of input you might specify when using the built-in field types are shown in Table 32 on page 128.

For information about writing your own application-specific grammars, see "Grammars" on page 104.

Table 22 shows the implementation details for US English only. Refer to the appropriate appendix for implementation details in another language.

Table 22. Built-in types for US English

Type	Implementation details
boolean	Users can say positive responses such as **yes**, **okay**, **sure**, and **true**, or negative responses such as **no**, **false**, and **negative**. Users can also provide DTMF input: **1** is yes, and **2** is no. The return value sent is a boolean true or false. If the field name is subsequently used in a **value** element within a prompt, the TTS engine will speak either **yes** or **no**.

Table 22. Built-in types for US English (continued)

Type	Implementation details
currency	Users can say US currency values in dollars and cents from 0 to $999,999,999.99 including common constructs such as "a buck fifty" or "nine ninety nine." Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a decimal point), and must terminate DTMF entry using the # key. The return value sent is a string in the format *UUUddddddddd.cc*, where *UUU* is a currency indicator or null. The only supported currency type is USD for U.S. dollars. If the field name is subsequently used in a **value** attribute within a prompt, then the TTS engine will speak the currency value. The currency is expressed as a floating point number, followed by the currency type (such as U.S. dollars). For example, the TTS engine speaks "a buck fifty" as **one point five zero U.S. dollars** and "nine ninety nine" as **nine point nine nine**.
date	Users can say a date using months, days, and years, as well as the words **yesterday**, **today**, and **tomorrow**. Common constructs such as "March 3rd, 2000" or "the third of March 2000" are supported. Users can also provide DTMF input in the form *yyyymmdd*. **Note:** The date grammar does not perform leap year calculations. **February 29** is accepted as a valid date regardless of the year. If desired, your application or servlet can perform the required calculations. The return value sent is a string in the format *yyyymmdd*, with the VoiceXML browser returning a **?** in any positions omitted in the spoken input. If the value is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:date**, then it is spoken as a date appropriate to the current language.
digits	Users can say numeric integer values as individual digits (0 through 9). For example, a user could say 124356 as "one two four three five six." Users can also provide DTMF input using the numbers 0 through 9, and must terminate DTMF entry using the # key. The return value sent is a string of one or more digits. If the result is subsequently used in **<say-as>** with the **interpret-as** value **vxml:digits**, it will be spoken as a sequence of digits appropriate to the current language. In the above example, the TTS engine speaks 124356 as **one two four three five six**. **Note:** Use this type instead of the number type if you require very high recognition accuracy for your numeric input.

Table 22. Built-in types for US English (continued)

Type	Implementation details
number	Users can say natural numbers (that is, positive and negative integers, 0, and decimals) from –999,999,999 to 999,999,999 as well as the words **point** or **dot** (to indicate a decimal point) and **minus** or **negative** (to indicate a negative number). Numbers can be spoken individually or in groups. For example, a user could say 123456 as "**one hundred twenty-three thousand four hundred and fifty-six**" or as "**one twenty-three four fifty-six**." Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a decimal point), and must terminate DTMF entry using the # key. Only positive numbers can be entered using DTMF. The return value sent is a string of one or more digits, 0 through 9, with a decimal point and a + or - sign as applicable. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:number**, then it is spoken as a number appropriate to the current language. In the above example, the TTS engine speaks 123456 as **one hundred twenty-three thousand four hundred fifty-six**. **Note:** Use this type to provide flexibility when collecting numbers; users can speak long numbers as natural numbers, single digits, or groups of digits. If you want to force the TTS engine to speak the number back in a particular way, use **<say-as interpret-as="vxml:***type***">** where *type* is the built-in type you want to specify. For example, use **<say-as interpret-as="vxml:digit">** to speak the number back as a string of digits.
phone	Users can say a telephone number, including the optional word **extension**. Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to represent the word "extension"), and must terminate DTMF entry using the # key. The return value sent is a string of digits without hyphens, and includes an **x** if an extension was specified. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:phone**, then it is spoken as a phone number appropriate to the current language. **Note:** For tips on minimizing recognition errors that are due to user pauses during input, see "Using the built-in phone grammar" on page 62.
time	Users can say a time of day using hours and minutes in either 12- or 24-hour format, as well as the word **now**. Users can also provide DTMF input using the numbers 0 through 9. The return value sent is a string in the format *hhmmx*, where *x* is **a** for AM, **p** for PM, **h** for 24-hour format, or **?** if unspecified or ambiguous; for DTMF input, the return value will always be **h** or **?**, since there is no mechanism for specifying AM or PM. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:time**, then it is spoken as a time appropriate to the current language.

Recorded audio

VoiceXML supports the use of recorded audio files for output. The VoiceXML browser plays an audio file when the corresponding URI (**<audio src="*URI of the audio prompt*">**) is encountered in a VoiceXML document.

Using prerecorded audio files

Prerecorded audio files must be:

- an 8KHz 8-bit mu-law **.au** file
- an 8KHz 8-bit mu-law **.wav** file
- an 8KHz 8-bit a-law **.au** file
- an 8KHz 8-bit a-law **.wav** file
- an 8KHz 16-bit linear **.au** file
- an 8KHz 16-bit linear **.wav** file

Note: 16-bit linear files will take twice as much storage and download time. For better performance, use 8-bit files. However, if you use 8-bit files, make sure they are compatible with the telephony environment you use. For example, if you are in North America, use µ-law recordings. If you use the wrong encoding, the audio file will need to be converted which may cause a degradation in the audio quality.

Recording spoken user input

You can use the VoiceXML **<record>** element to capture spoken input; the recording ends either when the user presses any DTMF key, or when the time you specified in the **maxtime** attribute is reached.

Playing and storing recorded user input

You can play the recorded input back to the user immediately (using an **<audio>** element with the expr attribute), or submit it to the server (using **<submit next="***URI***" method="post" enctype="multipart/form-data"/>**) to be saved as an audio file.

Recording user input during speech recognition

You can use the **<property>** element to capture spoken user input for speech recognition as an audio file as well as a text string. You can use the IBM **<property>** extensions shown in Table 23 to set audio recording at the application, document, dialog, form or menu level. Additionally, VoiceXML 2.1 includes the properties **recordutterance** and **recordutterancetype** which perform some of the same functions as the settings listed in the table. For more information on these new settings for the **<property>** element, see the VoiceXML 2.1 specification.

Table 23. Properties for capturing audio during speech recognition

Name	Description	Implementation

Table 23. Properties for capturing audio during speech recognition (continued)

com.ibm.speech.asr.saveaudio	Specifies whether or not to save spoken input as an audio file.	Used with **value="true"** (to capture the audio) or **value="false"** .
com.ibm.speech.asr.saveaudiotype	Specifies the format in which to save the audio file.	Used with **value="***type***"**. Where *type* is a media type as documented in appendix E of the VoiceXML 2.1 specification.
com.ibm.speech.asr.endpointed	Specifies whether or not to remove leading and trailing silence from the audio file.	Used with **value="true"** (to remove silence) or **value="false"** .

The audio file is captured in the shadow variable *x***$.asraudio** and stored in the application variable **application.lastresult$[i].asraudio**.

Document fetching and caching

The VoiceXML browser uses caching to improve performance when fetching documents and other resources (such as audio files, grammars, and scripts); see "Fetching and caching resources for improved performance" on page 124.

Controlling fetch and cache behavior

The document fetching and caching attributes are supported as described in the VoiceXML 2.1 specification. These attributes are available with various elements, including **<audio>**, **<choice>**, **<goto>**, **<grammar>**, **<prompt>**, **<property>**, **<script>**, **<subdialog>**, and **<submit>**.

Preventing caching

You can use the **<meta>** element to prevent caching of a VoiceXML document by specifying **http-equiv="expires"** and **content="*date*"**. For example:

```
<meta http-equiv="expires" content="Tue, 25 Nov 2003 14:00:00 GMT">
```

In this example, the **<meta>** element specifies that the document is invalid after the date specified in the **content** attribute. After that date, the browser should load the new document, not the cached document.

Events

The VoiceXML browser throws an event when it encounters a **<throw>** element or when certain specified conditions occur; these may be normal error conditions (such as a recognition error) or abnormal error conditions (such as an invalid page reference). Events are caught by **<catch>** elements that can be specified within other VoiceXML elements in which the event can occur, or inherited from higher-level elements.

Predefined events

The VoiceXML browser supports the complete set of predefined events documented in VoiceXML 2.0, as described in Table 24:

Table 24. Predefined events and event handlers for US English

Event	Description	Implementation details
cancel	The VoiceXML browser throws this event when the user says a valid word or phrase from the Quiet/Cancel grammar.	The default event handler stops the playback of the current audio prompt.
error.badfetch	The VoiceXML browser throws this event when it detects a problem, such as an unknown audio type, an invalid URI, or the expiration of a **fetchtimeout**, that prevents the browser from jumping to the next URI.	The default event handler plays the message "Sorry, must exit due to processing error" and then exits.
error.badfetch.http.<responsecode>	The VoiceXML browser uses this event to return an http response code in the event of a fetch failure.	The default event handler plays the message "Sorry, must exit due to processing error" and then exits.
error.badfetch.protocol.<responsecode>	The VoiceXML browser uses this event to return a protocol specific response code in the event of a fetch failure.	The default event handler plays the message "Sorry, must exit due to processing error" and then exits.
error.semantic	The VoiceXML browser throws this event when it detects an error within the content of a VoiceXML document (such as a reference to a non-existent application root document) or when a critical error (such as an error communicating with the speech recognition engine) occurs during operation.	The default event handler plays the message "Sorry, must exit due to processing error" and then exits.
error.unsupported.*element*	The VoiceXML browser throws this event when it encounters a reference to an unsupported element.	For example, if a document contains a **<transcribe>** element, the VoiceXML browser will throw an **error.unsupported.transcribe** event. The default event handler plays the message "Sorry, must exit due to processing error" and then exits.
error.unsupported.format.*element*	The VoiceXML browser throws this event when it encounters a reference to an unsupported resource format, such as an unsupported MIME type, grammar format, or audio file format.	The default event handler plays the message "Sorry, must exit due to processing error" and then exits.
exit	The VoiceXML browser throws this event when it processes a **<throw event="exit">**.	The VoiceXML browser performs some cleanup and then exits.
help	The VoiceXML browser throws this event when the user says a valid word or phrase from the Help grammar.	The default event handler plays the message "Sorry, no help available" and then re-prompts the user. For guidelines on choosing a scheme for your own help messages, see "Choosing help mode or self-revealing help" on page 40.

Table 24. Predefined events and event handlers for US English (continued)

Event	Description	Implementation details
noinput	The VoiceXML browser throws this event when the **timeout** interval, as defined by a timeout attribute on the current prompt, the **timeout** property, or the **vxml.timeout** Java system property, is exceeded.	The default event handler re-prompts the user. If you adhere to the guidelines for self-revealing help, this event can use the same messages as the help event. See "Implementing self-revealing help" on page 42 for details.
nomatch	The VoiceXML browser throws this event when the user says something that is not in any of the active grammars.	The default event handler plays the message, "Sorry, I didn't understand" and then re-prompts the user. If you adhere to the guidelines for self-revealing help, this event can use the same messages as the **help** event. See "Implementing self-revealing help" on page 42 for details. If the user pauses after uttering a partial response, and the silence period exceeds the duration specified by the **incompletetimeout** property, the VoiceXML browser's response depends on whether the user's partial utterance matches a valid utterance from any active grammar: • If the partial user utterance is itself a valid utterance, the VoiceXML browser returns a result. For example, if the built-in **phone** grammar is active and the user pauses after uttering 7 digits of a 10-digit telephone number, the VoiceXML browser will return the 7-digit number. • If the partial user utterance is not itself a valid utterance (for example, if the user pauses in the middle of a word), the VoiceXML browser throws a **nomatch** event.
connection.disconnect.hangup	The VoiceXML browser throws this event when the user hangs up the telephone or when a **<disconnect>** element is encountered.	Supported as documented in VoiceXML 2.0. The default event handler exits; however, you can override this handler if application-specific cleanup is desired.
connection.disconnect.transfer	The VoiceXML browser throws this event when the user has been unconditionally transferred to another line (by the **<transfer>** element) and will not return.	Supported as documented in VoiceXML 2.0. The **<transfer>** event is thrown after the transfer is initiated and before the interpreter exits. The default event handler exits; however, you can override this handler if application-specific cleanup is desired.

Note: UK English uses the same default event handlers. For language versions other than US English, see the appropriate appendixes.

If desired, you can override the predefined event handlers by specifying your own **<catch>** element to handle the events. For example, to go to *field1* after catching the event *error.badfetch*:

```
<catch event="error.badfetch">
   Caught bad fetch.
   <goto nextitem="field1"/>
</catch>
```

Application-specific events

In addition, you can define application specific errors, events and handlers. Error names should be of the form: **error**.*packagename*.*errorname*, where *packagename* follows the Java convention of starting with the company's reversed Internet domain name with the package name appended to it (for example, **com.ibm.mypackage**).

Recurring events

If desired, you can use the **<catch>** element's **count** attribute to vary the message based on the number of times that an event has occurred. This mechanism is also used for self-revealing help and tapered prompts.

For example:

```
<prompt> Name and location?</prompt>
<noinput count="1">
   <prompt>Please state the name and location of the employee you wish to call.</prompt>
</noinput>
<noinput count="2">
   <prompt>For example, to call Joe Smith in Dallas, say, "Joe Smith in Dallas"</prompt>
</noinput>
```

Variables and expressions

You can specify an executable ECMAScript enclosed within a **<script>** element or in an external file identified by the **src** attribute.

Using ECMAScript

The scripting language of VoiceXML is ECMAScript, an industry-standard programming language for performing computations in Web applications. For information about the ECMAScript, refer to the ECMAScript Language Specification, available at http://www.ecma.ch/ecma1/stand/ECMA-262.htm.

ECMAScript is also used for semantic interpretation in mixed-initiative forms. See "Mixed-initiative application and form-level grammars" on page 111 for details.

Declaring variables

You can declare variables using the **<var>** element or the **name** attribute of various form items. Table 25 lists the possible variable scopes:

Table 25. Variable scope

Scope	Description	Implementation details
anonymous	Used by the **<block>**, **<filled>**, and **<catch>** elements for variables declared in those elements.	Supported as documented in VoiceXML 2.1.
application	Variables are visible to the application root document and any other loaded application documents that use that root document.	Supported as documented in VoiceXML 2.1. The following application variables have been added to hold the audio recorded during speech recognition: • **application.lastresult$.asraudio** • **application.lastresult$[i].asraudio** See "Recording user input during speech recognition" on page 95 for more information.
dialog	Variables are visible only within the current **<form>** or **<menu>** element.	Supported as documented in VoiceXML 2.1.
document	Variables are visible from any dialog within the current document.	Supported as documented in VoiceXML 2.1.

Table 25. Variable scope (continued)

Scope	Description	Implementation details

session	Variables that are declared and set by the interpreter context.	The following session variables are fully supported:
		• **session.connection.aai** (contains application to application information passed during connection setup)
		• **session.connection.local.uri** (provides Dialed Number Identification Service)
		• **session.connection.originator** (directly references either the local or remote property)
		• **session.connection.remote.uri** (provides Automatic Number Identification)
		• **session.ibm.application_data** (contains the application data passed to the VoiceXML document via the VoiceXML Browser bean)
		• **session.ibm.line_address** (contains the line address of the current call)
		• **session.ibm.call_key** (contains the CallPath call key)
		
		The following session variables are not fully supported but can be referenced without causing an error:
		• **session.connection.protocol.name** (always contains the value "WVR_Generic_Protocol", not the name of the actual underlying telephony protocol)
		• **session.connection.protocol.version** (contains the installed version of WVR_Generic_Protocol, not the version of the actual underlying telephony protocol)
		• **session.connection.redirect** (always contains the ECMAScript value "undefined")
		
		II Digit (Information Indicator Digit) and User to User Information functionality are not available.
		
		To ensure that your application is portable, you should check if session variables are defined before attempting to use them:
		`<if cond="session.x==undefined">`
		`<exit>`
		`<else>`
		`ok to access the session variable`
		`<endif>`

Assigning and referencing variables

After declaring a variable, you can assign it a value using the **<assign>** element, and you can reference variables in **cond** and **expr** attributes of various elements. Because VoiceXML is an XML-based application, you must use escape sequences to specify relational operators (such as greater than, less than, etc.). Table 26 lists the escape sequences.

Table 26. Examples of relational operators

Relational operator	Escape sequence
<	&lt;
>	&gt;
<=	<=
>=	>=
&	&amp;
&&	&&

Using shadow variables

The result of executing a field item is stored in the field's **name** attribute; however, there may be occasions in which you need other types of information about the execution, such as the string of words recognized by the speech recognition engine. Shadow variables provide this type of information. Table 27 on page 103 documents the VoiceXML browser's support for shadow variables.

Table 27. Shadow variables

Element	Shadow variable	Description	Implementation details
<field>	*x*__$.asraudio__	An audio file of the words that were spoken by the caller while recognition was taking place.	This is an IBM extension to the VoiceXML 2.0 specification. See "Recording user input during speech recognition" on page 95 for more information.
	*x*__$.confidence__	The recognition confidence level from the speech recognition engine.	Supported as documented in VoiceXML 2.0.
	*x*__$.inputmode__	The input mode for a given utterance: voice or DTMF.	Supported as documented in VoiceXML 2.0.
	*x*__$.interpretation__	An ECMAscript variable containing the interpretation.	Supported as documented in VoiceXML 2.0.
	*x*__$.recording__	The variable that stores a reference to the recording, or undefined if no audio is collected.	Supported as documented in VoiceXML 2.1.
	*x*__$.recordingduration__	The duration of the recording in milliseconds, or undefined if no audio is collected.	Supported as documented in VoiceXML 2.1.
	*x*__$.recordingsize__	The size of the recording in bytes, or undefined if no audio is collected.	Supported as documented in VoiceXML 2.1.
	*x*__$.utterance__	A string that represents the actual words that were recognized by the speech recognition engine.	The actual value of the shadow variable is dependent on how the grammar was written.
<record>	*x*__$.confidence__	The confidence level for the utterance if the recording is terminated by speech recognition input. **Note:** Terminating **<record>** using grammars is not supported, so the **$.confidence** value should never be set.	Supported as documented in VoiceXML 2.0.
	*x*__$.duration__	The duration of the recording.	Supported as documented in VoiceXML 2.0.
	*x*__$.maxtime__	Indicates whether the recording was terminated because the maxtime duration was reached.	Supported as documented in VoiceXML 2.0.
	*x*__$.size__	The size of the recording, in bytes.	Supported as documented in VoiceXML 2.0.
	*x*__$.termchar__	The DTMF key used to terminate the recording.	Supported as documented in VoiceXML 2.0.

Table 27. Shadow variables (continued)

Element	Shadow variable	Description	Implementation details
<transfer>	*x***$.duration**	The duration of a successful call.	Supported as documented in VoiceXML 2.0.
	*x***$.inputmode**	The input mode of the terminating command: voice or DTMF.	Supported as documented in VoiceXML 2.0.
	*x***$.utterance**	A string that represents the words recognized by the speech recognition engine or the DTMF input used to terminate transfer.	Supported as documented in VoiceXML 2.0.

Grammars

A grammar is an enumeration, in compact form, of the set of *utterances*—words and phrases—that constitute the acceptable user response to a given prompt. The VoiceXML Browser currently supports only Speech Recognition Grammar Specification (SRGS) XML format and ABNF form grammars. See SRGS Version 1.0 at **http://www.w3.org/TR/speech-grammar/** for more information. The limitations shown in Table 28 apply.

Table 28. Speech Recognition Grammar limitations in WebSphere Voice Server

Limitation	Relevant section in specification
An exception is thrown if a grammar contains an external reference by URI.	2.2.2
The special rulename **GARBAGE** defaults to the rulename **NULL**.	2.2.3
An exception is thrown if a grammar contains a reference to an N-gram document.	2.2.4
Language declarations and attachments are ignored; the current speech recognition language is used.	2.7
References to external pronunciation lexicon documents are ignored.	4.10
<meta>and **http-equiv** declarations are ignored.	4.11

Grammars written in both SRGS XML format and ABNF also allow for the specification of semantic return values using the W3C Semantic Interpretation for Speech Recognition (SISR) 1.0 specification. For more information see **http://www.w3.org/TR/semantic-interpretation/**.

When you write your application, you can use the flexible, but generic, built-in grammars and create one or more of your own. Whether you use a

built-in grammar or your own customized grammar, you must decide when each grammar should be active. The speech recognition engine uses only the active grammars to define what it listens for in the incoming speech. This section discusses the following topics:

- "Grammar syntax"
- "Static grammars" on page 107
- "Dynamic grammars" on page 109
- "Grammar scope" on page 109
- "Hierarchy of active grammars" on page 110
- "Mixed-initiative application and form-level grammars" on page 111

Grammar syntax

A complete discussion of grammar syntax is beyond the scope of this Programmer's Guide. The information provided in this section is merely to acquaint you with the basics of writing grammars. The VoiceXML browser supports both SRGS XML format and SRGS ABNF. The examples in this section are in the XML format.

Grammar header

An SRGS XML format grammar consists of a grammar header and a grammar body. The grammar header declares the various attributes of the grammar, including the version of the grammar, its mode and its root rule. When included within a VoiceXML 2.1 document, a grammar header might look like this:

```
<grammar version="1.0" type="application/srgs+xml" root="startHere" mode="dtmf">
```

A standalone grammar header would include additional information, including the XML namespace:

```
<grammar version="1.0" xmlns="http://www.w3.org/2001/06/grammar"
mode="voice" root="citystate" tag-format="semantics/1.0">
```

For external grammars, the **mode** attribute must be specified on the grammar tag within the VoiceXML document as well as in the header of the external grammar.

Grammar body

The grammar body consists of one or more *rules* that define the valid set of utterances. The syntax for grammar rules is:

```
<rule id="rulename" scope="rule-scope">
   rule contents
</rule>
```

where:

rulename
is the name the rule will be referenced by from other grammars and VoiceXML applications.

scope
is either *public* or *private*. Publicly scoped rules can be referenced from any grammar or VoiceXML application; privately scoped rules are only accessible within the grammar file in which they are declared.

rule contents
are the various items that make up the rule itself. For example:

```
<rule id="direction" scope="public">
   <one-of>
      <item>left</item>
      <item>right</item>
   </one-of>
</rule>
```

identifies a rule that, when activated, will let someone say either "left" or "right".

Comments in grammars

Comments within a grammar are the same as they are anywhere in an XML document. They start with the characters`<!--` and end with `-->`. For example:

```
<!-- this is a comment -->
```

External and inline grammars

Grammars can either be defined *inline*, within a VoiceXML document, or the VoiceXML document can reference an *external* grammar file. When specified inline, the **<grammar>** tag must identify the root rule to activate, using the **root** attribute. When an external grammar is referenced from within a VoiceXML document, the document does not need to specify the root rule to activate. If none is specified, the root rule identified in the header of the grammar itself is activated. You can also activate a specific rule within the external grammar from the VoiceXML document, by appending a *fragment identifier* to the URI of the external grammar. For example:

```
<grammar src="citystate.grxml#citylist"/>
```

In this example, the rule named citylist in the grammar citysytate.grxml would be activated.

DTMF grammars

The following example defines an inline DTMF grammar that enables the user to make a selection by pressing the numbers 1 through 4, the asterisk or the pound sign.

```
<grammar version="1.0" xmlns="http://www.w3.org/2001/06/grammar"
mode="dtmf" root="digits" tag-format="semantics/1.0">
     <rule id="digits">
          <one-of>
               <item>1</item>
               <item>2</item>
               <item>3</item>
               <item>4</item>
               <item>*</item>
               <item>#</item>
           </one-of>
     </rule>
 </grammar>
```

If you are using external grammars, make sure you include the **mode="dtmf"** attribute in the <grammar> element.

Note: DTMF grammars used with WebSphere Voice Response do not support the use of hotword barge-in. This means that if a prompt is played with **bargeintype="hotword"** and a DTMF grammar is active, then the prompt will stop as soon as any DTMF key is detected and the prompt will be played as though it was set to **bargeintype="speech"**. This occurs regardless of whether or not any speech grammars are also active.

Static grammars

The following example illustrates the use of grammars in a Web-based voice application for a restaurant:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

 <form>

  <field name="drink">

    <prompt>What would you like to drink?</prompt>

    <grammar mode="voice" version="1.0" root="drinks">
      <rule id="drinks">
        <one-of>
          <item> coffee </item>
          <item> tea </item>
```

```
          <item> orange juice </item>
          <item> milk </item>
          <item> nothing </item>
        </one-of>
      </rule>
    </grammar>

  </field>


  <field name="sandwich">

    <prompt>What type of sandwich would you like?</prompt>

    <grammar src="sandwich.grxml"/>

  </field>

  <filled>
    <submit next="/servlet/order"/>
  </filled>

 </form>

</vxml>
```

In this example, the first grammar (for drinks) consists of a single rule, specified inline. In contrast, the second grammar (for sandwiches) is contained in an external grammar file, shown below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<grammar version="1.0" xmlns="http://www.w3.org/2001/06/grammar"
  xml:lang="en-US" mode="voice" root="sandwich" tag-format="semantics/1.0">

 <rule id="sandwich">
   <ruleref uri="#ingredient"/>

   <item repeat="0-">
     <item repeat="0-1"> and </item>
     <ruleref uri="#ingredient"/>
   </item>

   <item>
     <item> on </item>
     <ruleref uri="#bread"/>
   </item>
 </rule>

 <rule id="ingredient" scope="private">
   <one-of>
     <item> ham </item>
     <item> turkey </item>
     <item>
      <item> swiss </item>
```

```
      <item repeat="0-1"> cheese </item>
     </item>
   </one-of>
 </rule>

 <rule id="bread" scope="private">
   <one-of>
     <item> rye </item>
     <item> white </item>
     <item>
      <item repeat="0-1"> whole </item>
      <item> wheat </item>
     </item>
   </one-of>
 </rule>

</grammar>
```

Here, the ingredient and bread rules are private, meaning that they can only be accessed by other rules in this grammar file. The sandwich rule is public, meaning that it can be accessed by specifying this grammar file or by referencing the fully-qualified name of the sandwich rule.

A typical dialog might sound like this:

```
System:    What would you like to drink?

User:      Coffee.
System:    What type of sandwich would you like?
User:      Turkey and swiss on rye.
```

At this point, the system has collected the two fields needed to complete the order, so it executes the **<filled>** element, which contains a **<submit>** that causes the collected information to be submitted to a remote server for processing.

Dynamic grammars

In the restaurant example, the contents of the inline and the external grammars were static. However, it is possible to create grammars dynamically by using information from a back-end database located on an enterprise server. You can use the same types of server-side logic to build dynamic VoiceXML as you would use to build dynamic HTML: CGI scripts, Java Beans, servlets, ASPs, JSPs, etc.

Grammar scope

Form grammars can specify one of the scopes shown in Table 29 on page 110.

Table 29. Form grammar scope

Scope	Description	Implementation details
dialog	The grammar is accessible only within the current **<form>** or **<menu>** element.	Supported as documented in VoiceXML 2.0.
document	The grammar is accessible from any dialog within the current document. If the document is the application root document, the grammar is accessible to all documents that use that root document.	Supported as documented in VoiceXML 2.0. You can improve performance by using a combination of fetching and caching to prime the cache with all referenced grammar files before your application starts, as described in "Fetching and caching resources for improved performance" on page 124.

Field, link, and menu choice grammars cannot specify a scope:

- Field grammars are accessible only within the specified field.
- Link grammars are accessible only within the element containing the link.
- Grammars in menu choices are accessible only within the specified menu choice.

Hierarchy of active grammars

When the VoiceXML browser receives a recognized word or phrase from the speech recognition engine, the browser searches the active grammars, in the following order, looking for a match:

1. Grammars for the current field, including grammars contained in links in that field.
2. Grammars for the current form, including grammars contained in links in that form.
3. Grammars contained in links in the current document, and grammars for menus and other forms in the current document that have document scope.
4. Grammars contained in links in the application root document, and grammars for menus and other forms in the application root document that have document scope.
5. Grammars for the VoiceXML browser itself (that is, universal commands).

Disabling active grammars

You can temporarily disable active grammars, including the VoiceXML browser's grammars for built-in commands, by using the **modal** attribute on various form items. When **modal** is set to **true**, all grammars are temporarily disabled except the grammar for the current form item.

Resolving ambiguities

Ambiguities can arise when a user's utterance matches more than one of the following:

- Phrases in one or more active grammars
- Items in a menu or form
- Links within a single document

You should exercise care to avoid using the same word or phrase in multiple grammars that could be active concurrently; the VoiceXML browser resolves any ambiguities by using the first matching value.

Understanding how disambiguation works is especially important when multiple phrases in concurrently active grammars contain the same key words.

Mixed-initiative application and form-level grammars

In a machine-directed application, the computer controls all interactions by sequentially executing each form item a single time.

However, VoiceXML also supports mixed-initiative applications in which either the system or the user can direct the conversation. This is done using a combination of form level grammars to control user input and **<initial>** elements to specify prompts.

Form-level grammars allow a greater flexibility and more natural responses than field-level grammars because the user can fill in the fields in the form in any order and can fill more than one field as a result of a single utterance. For example, the following city/state grammar:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<grammar version="1.0" xmlns="http://www.w3.org/2001/06/grammar"
  xml:lang="en-US" mode="voice" root="citystate" tag-format="semantics/1.0">

<rule id="citystate">
   <one-of>

      <item>
         <ruleref uri="#cityfirst"/>
            <tag>
             $.city = $cityfirst.city;
             if ( typeof ( $cityfirst.state ) != "undefined" )
                $.state = $cityfirst.state;
            </tag>
      </item>

      <item>
         <ruleref uri="#statefirst"/>
            <tag>
             $.state = $statefirst.state;
```

```
            if ( typeof ( $statefirst.city ) != "undefined" )
               $.city = $statefirst.city;
            </tag>
      </item>

   </one-of>
</rule>

<rule id="cityfirst">
   <item>
      <ruleref uri="#citylist"/> <tag>$.city = $citylist;</tag>
   </item>

   <item repeat="0-1">
      <ruleref uri="#statelist"/> <tag>$.state = $statelist;</tag>
   </item>
</rule>

<rule id="statefirst">
   <item>
      <ruleref uri="#statelist"/><tag>$.state = $statelist;</tag>
   </item>

   <item repeat="0-1">
      <ruleref uri="#citylist"/> <tag>$.city = $citylist;</tag>
    </item>
</rule>

<rule id="citylist" scope="public">
   <one-of>
      <item>San Francisco</item>
      <item>New York</item>
      <item>Los Angeles</item>
      <item>Boca Raton</item>
   </one-of>
</rule>

<rule id="statelist" scope="public">
   <one-of>
      <item>California</item>
      <item>New York</item>
      <item>Florida</item>
   </one-of>
</rule>

</grammar>
```

can be used with this VoiceXML document:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

 <form id="getinfo">
  <grammar src="citystate.grxml"/>
```

```
  <initial name="start">
   <prompt>City and state please.</prompt>

   <noinput><reprompt/></noinput>

   <nomatch count="2">
    <prompt>
     Sorry, I am having trouble understanding you.  Let's try one piece of
     information at a time.
    </prompt>
    <reprompt/>
    <assign name="start" expr="true"/>
   </nomatch>
  </initial>

  <field name="city">
   <prompt>
    Please say the name of a city.
   </prompt>
   <grammar src="citystate.grxml#citylist"/>
   <filled>
    <prompt>
     The city is <value expr="city"/>
    </prompt>
   </filled>
  </field>

  <field name="state">
   <prompt>
    Please say the name of a state.
   </prompt>
   <grammar src="citystate.grxml#statelist"/>
   <filled>
    <prompt>
     The state is <value expr="state"/>
    </prompt>
   </filled>
  </field>

  <filled>
   <prompt>
    Thank you.  The city and state are <value expr="city"/>,
    <value expr="state"/>.
   </prompt>
  </filled>

 </form>

</vxml>
```

Specifying a sounds-like spelling in a Japanese, a Cantonese, or a Simplified Chinese grammar

This section applies only to VoiceXML applications that use Japanese, Cantonese, or Simplified Chinese.

The *Spelling~Soundslike* notation is only supported for the Japanese and Simplified Chinese languages in WebSphere Voice Server. It is not supported for Cantonese.

A single word may have several pronunciations. For example, the Japanese word 大和 can be pronounced as "Yamato" and "Daiwa." Subsequently, when the TTS engine receives this word for synthesis, it cannot accurately determine which pronunciation to use.

To solve this problem, you can specify a sounds-like spelling in a grammar for a word so that the TTS engine can return the word's correct pronunciation to the user. The format of the notation is *Spelling~Soundslike*. For example, if you want to pass the sounds-like pronunciation "drive" for the word "Dr", specify the following in the grammar:

```
<name> = Blvd | Ave | Dr~Drive
```

When the speech recognition engine receives the utterance "Drive", it will return the word "Dr~Drive" as the recognition result for the utterance. The VoiceXML browser saves "Dr~Drive" in the `utterance` shadow variable. (See Table 27 on page 103.) If the recognition result is used for a prompt, "Dr~Drive" is passed to the TTS engine, which uses the sounds-like information, if necessary.

Note: While the example below is shown in U.S. English, this technique applies only to Japanese, Cantonese, and Simplified Chinese.

A VoiceXML document like this:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

 <form id="address_form">

  <field name="my_address">

   <grammar version="1.0" type="application/srgs">

     #ABNF 1.0;

     language en;
     mode voice;

     root $street;
```

```
    $street = Main Boulevard | Main Avenue | Main Dr~Drive;
   ]]>
  </grammar>

  <prompt> Where do you live? </prompt>

  <filled>
   <prompt>
    You live along <value expr="my_address$.utterance"/>.
   </prompt>
  </filled>

 </field>
</form>
</vxml>
```

results in the following dialog:

```
System:     Where do you live?
User:       Main drive
System:     You live along Main drive.
```

Notes:

1. In Japanese, the *Soundslike* pronunciations must be given using Hiragana characters.
2. In Simplified Chinese, Pin Yin needs to contain a white space in a sounds-like spelling. To distinguish a white space in a sounds-like spelling from the white space of a delimiter, use an underscore. For example:

   Spelling~Wen1_Yan1
3. In Cantonese, Yue Pin (LSHK) needs to contain a white space in a sounds-like spelling. To distinguish a white space in a sounds-like spelling from the white space of a delimiter, use an underscore. For example:

   Spelling~Biu2_Daan1_Kap1

Timeout properties

The information in this section is intended as a clarification of the information in the VoiceXML 2.1 specification.

Incompletetimeout

The **incompletetimeout** property specifies how long to wait after a user has spoken a partial utterance in the scenarios shown in Table 30 on page 116. The default is **1000 ms** (1.0 second). The maximum value allowable is **10000 ms**; the minimum value is **100 ms**.

Table 30. Incompletetimeout

Scenario	Outcome
The partial utterance is not a complete match for any entry in any active grammar.	When the **incompletetimeout** period expires, the speech recognition engine will reject the partial utterance and return a **nomatch** event.
The partial utterance is a complete match for an entry in an active grammar; however, the user could speak additional words and match additional entries in this or other active grammars.	When the **incompletetimeout** period expires, the speech recognition engine will consider the utterance complete and will return the matching entry from the active grammar.

Note: An inappropriately long **incompletetimeout** value will slow down the dialog, while an **incompletetimeout** value that is too short may not give the user enough time to complete an utterance, especially one that tends to be spoken in segments with intervening pauses (such as a telephone number).

Completetimeout

The **completetimeout** property specifies how long to wait after the user has spoken an utterance in the scenario shown in Table 31. The default is **500 ms**. The maximum value allowable is **5000 ms**; the minimum value is **100 ms**. The value of the **completetimeout** property cannot be greater than the value of the **incompletetimeout** property. (For example, if you set the **completetimeout** property to a value greater than the **incompletetimeout** property, the **incompletetimeout** property will be raised to match it.)

Table 31. Completetimeout

Scenario	Outcome
The utterance is a complete and terminal match for an entry in an active grammar. There are no additional words that the user could speak and still match an entry in an active grammar.	When the **completetimeout** period expires, the speech recognition engine will return the matching entry from the active grammar. If more than one entry matches, the disambiguation scheme described in “Resolving ambiguities” on page 111 applies.

Example

This example shows the timeout properties being used to determine how long the system should wait after the user has made a response. If the user gives the response “San Francisco” or “New York City”, the completetimeout value is used and the system waits 500 ms before continuing. If, however, the user responds with “New York”, it is unclear whether the user will add the word “City”. In this case, the incompletetimeout value of 2 seconds is used to allow the user extra time to complete the utterance if necessary.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

 <property name="incompletetimeout" value="2s"/>
 <property name="completetimeout" value="500ms"/>

 <form>

  <field name="city">

   <prompt> What city please? </prompt>

   <grammar version="1.0" root="where">
    <rule id="where">
     <one-of>
      <item> San Francisco </item>
      <item> New York </item>
      <item> New York City </item>
     </one-of>
    </rule>
   </grammar>

   <filled>
    <prompt>
     <value expr="city"/>, here we come!
    </prompt>
   </filled>

  </field>
 </form>
</vxml>
```

Telephony functionality

The VoiceXML browser supports the following telephony features for use in a connection environment:

- "Automatic Number Identification"
- "Dialed Number Identification Service" on page 118
- "Call transfer" on page 119

Automatic Number Identification

If your central office and telephone switches provide ANI information, your VoiceXML application can use the **session.connection.remote.uri** variable to access the caller's telephone number. You can use the ANI information to personalize the caller's experience or to log when various customers are most likely to call.

Note: If the ANI information is not received from the central office, the session.connection.remote.uri will have a value of "?". Your application should verify that this value is defined before attempting to use it.

The following example shows an example of a VoiceXML application for a pay-per-use Customer Support line. For each incoming call, the VoiceXML application uses the ANI information to identify which customer to charge for the call.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

 <form id="ChargeCallers">

  <block>

   <!-- Obtaining the phone number from which the user phoned. -->
   <var name="ChargeCallToNum" expr="session.connection.remote.uri"/>
   <var name="TransferDestNum" expr="'tel:+18005551212'"/>

   <prompt>
    The phone number
    <say-as interpret-as="vxml:phone"><value expr="ChargeCallToNum"/></say-as>
    will be charged for this call to our support line.
   </prompt>

   <!-- Submit callers phone number and destination to servlet -->
   <submit namelist="ChargeCallToNum TransferDestNum"
           next="http://SupportLine.com/servlet/ChargeUserForCall"/>
  </block>

 </form>

</vxml>
```

Dialed Number Identification Service

If your central office and telephone switches provide DNIS information, your VoiceXML application can use the **session.connection.local.uri** variable to access the telephone number that the caller dialed.

Note: The WebSphere Voice Server platforms do not enforce any rules or validation for DNIS information; they just provide the raw data received from the central office to the VoiceXML applications.

You can use the DNIS information in many ways, such as to track responses to advertising. You can list different telephone numbers in different ads, and then track the response rates based on which number the user dialed.

Note: DNIS numbers are site-dependent because your location determines which users need to dial a country code, area code, city code, long distance access code, etc. to reach your application.

In the following example, a company wants to track which users are calling in response to an advertisement on the Web site and which in response to a

product catalog. The company used different telephone numbers in the two ads: (800) 555-9999 in the Web ad, and (800) 555-9990 in the print ad. The VoiceXML application uses the DNIS information to determine which ad the user is responding to and increments a counter on a server that is performing the data mining.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

 <var name="AddOneCaller" expr="1"/>

 <form id="RouteCallers">

  <block>

   <!-- Obtain phone number the user dialed. -->
   <var name="NumDialed" expr="session.connection.local.uri"/>

   <if cond="NumDialed == '8005559999'">
    <goto next="#UserFromWebsite"/>
   <elseif cond="NumDialed == '8005559990'"/>
    <goto next="#UserFromCatalog"/>
   </if>

  </block>

 </form>

 <form id="UserFromWebsite">

  <block>
   <!--Add one to the count of WebUsers-->
   <submit namelist="AddOneCaller"
           next="http://SupportLine.com/servlet/TrackWebUsers"/>
  </block>
 </form>

 <form id="UserFromCatalog">
  <block>
   <!-- Add one to the count of CatalogUsers -->
   <submit namelist="AddOneCaller"
           next="http://SupportLine.com/servlet/TrackCatalogUsers"/>
  </block>

 </form>

</vxml>
```

Call transfer

Typically, you use the call transfer feature to transfer a call to:

- A human operator
- Another VoiceXML application

For blind transfer (when **bridge="false"**), the VoiceXML browser simply initiates the call transfer request and then disconnects from the call; the system cannot verify whether the telephone number for the call transfer has been correctly configured in the telephony system. In addition, no information is returned on the status of the transfer, so there is no way to know whether the transfer was successful, the line was busy, the telephone number was invalid, or the call was dropped. The WebSphere Voice Response connection environment provides additional call transfer capabilities. For more information, refer to the WebSphere Voice Response for AIX documentation.

For bridged transfer (when **bridge="true"**), a conference call is set up to include the person or application specified by the **dest=attribute**, known as the "Callee."

The WebSphere Voice Response platform is not able to create a conference between itself, the Caller, and the Callee. WebSphere Voice Response must be configured with a suitable telephony service. See *WebSphere Voice Response: Deploying and Managing VoiceXML and Java Applications* for more information.

The telephony service allows WebSphere Voice Response to request that the conference be created by the telephony system to which WebSphere Voice Response is attached, as shown in the following figure.

Figure 3. Conference

This method of implementing a bridged transfer does not provide the same functionality as VoiceXML 2.1 because WebSphere Voice Response does not have control over the voice connections to the Caller and Callee separately. The differences are the following.

- Either the Caller or the Callee may terminate the bridge by DTMF, or by a speech grammar match, because WebSphere Voice Response cannot distinguish the source of the match.
- It is not possible for the Caller to terminate the bridge during the period of time where the telephony system is attempting to create the conference. Typically the Caller will hear "music on hold" that is generated by the Telephony system during this time period. Thus it is not possible for WebSphere Voice Response to support the **transferaudio=attribute**.
- WebSphere Voice Response is not able to terminate the conference even if the bridge is terminated using a grammar match, and a script execution

continues for the **<transfer>** element. It is necessary for either the Caller, Callee, or WebSphere Voice Response to disconnect in order to terminate the conference.

Note: What the user will hear if the transfer is unsuccessful depends on your central office and your telephony configuration. Even a standard setup on the telephony system may not provide consistent behavior from all central office service providers; depending on the information sent by the central office, a user might hear a busy signal, silence, or no answer.

One interesting way that you might use call transfer is to route calls to different language versions of the same application, based on the caller's DTMF response to an introductory, prerecorded audio prompt. For example:

System:	For English, press 1.	(Call will be transferred to the English application.)
	Pour français, appuyez sur 2.	(Call will be transferred to the French application.)

Note: You should not use TTS prompts or spoken input in this initial prompt, since these require the application to be running in the desired language version of the VoiceXML browser.

The two potential drawbacks of this design are:

- You will not know if the transfer was unsuccessful.
- You lose the ANI and DNIS information from the original call. (The ANI information for the transferred call is the IP address of the Voice Server that initiated the transfer. The DNIS information is the telephone number to which the call was transferred.)

Chapter 4. Hints, tips, and best practices

This chapter contains hints and tips for structuring, coding, and testing your VoiceXML applications. The topics discussed are:

- "VoiceXML application structure"
- "Confidence-level processing" on page 127
- "Using a proxy server" on page 127
- "Sample code" on page 129

VoiceXML application structure

When deciding how best to structure your VoiceXML applications, you may want to consider the issues discussed in this section.

Deciding how to group dialogs

Unlike an HTML page, which is a renderable unit, a VoiceXML document may contain several interactable units such as forms, menus, etc. While you could go to extremes and either write your entire VoiceXML application in a single VoiceXML document or have a separate document for each menu or form, most VoiceXML applications will consist of dialogs and supporting code grouped together into a number of files. For example:

- An application root document for defining global variables, global grammars and global **<menu>** and **<link>** elements.
- One or more VoiceXML documents containing the application dialogs or subdialogs.
- Grammar and prerecorded audio files, as required by the application dialogs.
- ECMAScript files, if any.
- Server-side logic for accessing information in back-end enterprise databases.

There are no firm rules for deciding how to group dialogs; however, careful consideration of the following issues can help you decide:

- Logical grouping of menus or forms
- Resources they require
- Functions they perform
- Expected frequency of use
- Number of pages you want the VoiceXML browser to request from the Web application server

For example, you might decide to use a separate VoiceXML document for a form or menu that is executed infrequently and contains a large grammar or references large grammar or audio files. This design will allow the large files to be downloaded only when needed.

Similarly, you might decompose a complex dialog sequence into a series of subdialogs. Subdialogs are also useful for creating reusable components.

Deciding where to define grammars

Grammars for VoiceXML applications can be defined in an external file or inline, as explained in "External and inline grammars" on page 106. The only difference is the level of interaction between the VoiceXML browser and Web application server, and the resulting application performance. You will want to weigh such factors as:

- Grammar size and its effect on document access time
- The importance of instantaneous response and the corresponding need to load the grammars up front

For example, you might decide to define a grammar as an external file if the grammar is large and is in a menu or form that is unlikely to be executed. This design will minimize document access time.

Conversely, you might decide to define a grammar inline if you want it to be instantaneously ready, rather than having to download it from the Web application server, when the user accesses the menu or form. Alternatively, you might achieve similar results by fetching and caching the grammars when the application loads, as explained next.

Fetching and caching resources for improved performance

You can improve performance by using a combination of fetching and caching to prime the cache with referenced audio, script and grammar files before you application starts. To do this, create a VoiceXML application root document that contains a form that will never be visited. In this form, specify the audio, script and grammar tags that will be referenced by the application. the form should reference the appropriate fetching properties described by the VoiceXML specification, for example **<property name="grammarfetchhint" value="prefetch"/>**. It is important to note that the default behavior is to prefetch all resources except for documents. If certain resources can not be cached or expire immediately, applications should disable caching for these resources (using property or attribute names) to improve system performance.

Here is a two-document application illustrating the preloading of grammar for customer inquiries on laptop computers:

Application root document (preload.vxml)

```
<?xml version="1.0" encoding="iso-8859-1"?>

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

 <form id="loadgrammar">
  <grammar src="laptops.grxml" type="application/srgs+xml"
           fetchhint="prefetch" scope="document"/>
 </form>

</vxml>
```

Main document (main.vxml)

```
<?xml version="1.0" encoding="iso-8859-1"?>

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US"
      application="preload.vxml">

 <form id="greeting">

  <block>
   <prompt>
    Welcome to Laptops at Your Fingertips. At any time, you can say
    help or stop.
    <break time="large"/>
   </prompt>
   <goto next="getModel.vxml"/>
  </block>

 </form>
</vxml>
```

When the first document (main.vxml) in the Voice application is loaded, the application root document (preload.vxml) is also loaded. The VoiceXML browser will load the referenced grammar file (laptops.grxml) and store them in the cache.

Note: The techniques shown above should only be used for a small number of resources that are critical to the responsiveness of the application. The number of prefetched audio files (or similar resources) should be limited to no more than 20–30, fewer if the files are large.

Invoking a State Table using Voice XML

To invoke a WebSphere Voice Response for AIX state table, you must use the **invokeStateTable()** method. Only string parameters may be passed to the state table using this method, so if your state table requires numeric parameters, you will need to invoke it from another state table with the capacity to accept strings from the Voice XML application using the **invokeStateTable()** method.

For example, the state table you want to invoke, **Sample**, retrieves values from a database. It has three parameters:

- Database name, in this example is a constant.
- Customer number, a variable which will have been obtained from the caller.
- Account balance, retrieved from the database and returned by the state table. This can then be spoken to the caller.

First set up a string array to hold the parameters the state table requires. The Customer number has been obtained from the customer and stored in the variable *customer_id.* The array must contain exactly the same number fields as the number of parameters required by the state table, any fields not used should be initialized with null values.

```
<var name="parmArray" expr="new Array('TestDB', customer_id, '')"/>
<form id="state_table">
```

Where 'TestDB' is the database name, customer_id is the customer number, and the third field initialized as null will hold the returned account balance.

The invokeStateTable method returns results in shadow variables of the <object> element:

```
vxml2.completionCode;
vxml2.completionCodeText;
vxml2.returnCode;
vxml2.parms.length;  // number of returned parameters
vxml2.parms[0];
vxml2.parms[1];
```

A non-existent return parameter, such as parms[2] when parms.length==1, has the value "undefined".

Example:

```
<object name="vxml2" classid="method://com.ibm.wvr.vxml2.NativeAppSupport/invokeStateTable"
 codetype="javacode-ext">
     <param name="setName" value="Sample"/>
     <param name="setEntryPoint" value="start"/>
     <param name="setParms" expr="parmArray"/>
  </object>
```

- **setName** is set to the name of the state table
- **setEntryPoint** is the state table entry point
- **setParms** takes the string array which contains the parameters that the state table requires.

To read back the account balance to the customer from the above example, type the following within a `prompt` tag.

```
<prompt>
   your account balance is <value expr="vxml2.parms[2]"/>
</prompt>
```

To assign values returned in the **invokeStateTable** call to local VoiceXML variables.

```
<block>
   <var name="customer_balance" expr="vxml2.parms[2]"/>
</block>
```

Confidence-level processing

Basic information on usage is provided in VoiceXML 2.0.

Tune the confidence levels for each active context during a prompt. Set the **confidencelevel** property value below that at which results would be rejected. Consider using shadow variables to verify the confidence level for each recognition event.

Using a proxy server

To access your application files using a proxy server, add the following Java command line to the node definition in your **default.cff** file:

```
JavaCommand=java -Dhttp.proxyHost=your.proxy.host.com -Dhttp.proxyPort=portnumber
```

For unmanaged browsers, add:

```
-Dhttp.proxyHost=your.proxy.host.com -Dhttp.proxyPort=portnumber
```

to the Java invocation line in the VoiceXML 2.0 script. You might want to make a copy of the script instead of editing the original.

Testing built-in field types

Table 32 on page 128 provides examples of the types of input you might specify when testing an application that uses the built-in field types.

For language versions other than US English, see the appropriate appendixes.

Table 32. Sample input for US English built-in field types

Built-in field type	Sample input
boolean	yes true positive right ok sure affirmative check yep no false negative wrong not nope
currency	three twenty five sixteen dollars and fifty seven cents ten dollars nine million two hundred thousand dollars
date	May fifth March December thirty first two thousand July first in the year of nineteen ninety nine yesterday tomorrow today
digits	0 1 2 3 4 5 6 7 8 9

Table 32. Sample input for US English built-in field types (continued)

Built-in field type	Sample input
number	ten million five hundred thousand fifty three negative one point five positive one point five point fifty three one oh oh point five three one hundred oh seven point seven dot six
phone	nine nine four nine four one two three eight eight nine zero four nine four five two three eight eight one nine zero four nine four five two three eight eight nine one four nine four five two three eight eight extension sixty three fifty one one eight hundred five five five one two one two one three eight five one three two six ten
time	one o'clock one oh five three fifteen seven thirty half past eight oh four hundred hours sixteen fifty twelve noon midnight now

Some of the built-in grammars are flexible, but generic. They might, therefore, not meet the needs of your application. Before you use a particular built-in grammar, check its's capabilities against your application's needs.

Sample code

Calling a Java application

The application **clock.vxml** prompts the user to ask for the time of day or today's date. The <**object**> element calls the applet **clock.java**, which returns the time or date. The grammar **clock.grxml** supports the application.

clock.grxml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<grammar version="1.0" xmlns="http://www.w3.org/2001/06/grammar"
  xml:lang="en-US" mode="voice" root="clock" tag-format="semantics/1.0">
```

```
 <rule id="clock">
  <one-of>

   <item>
    <item> what </item>
    <ruleref uri="#mode"/>
    <item repeat="0-1"> is it </item>
    <item repeat="0-1"> please </item>
    <tag> $.mode = $mode; </tag>
   </item>

   <item>
    <item> what is todays </item>
    <ruleref uri="#mode"/>
    <tag> $.mode = $mode; </tag>
   </item>

  </one-of>
 </rule>

 <rule id="mode">
  <one-of>
   <item> month </item>
   <item> date </item>
   <item> day </item>
   <item> time </item>
  </one-of>
 </rule>
</grammar>
```

clock.vxml

```
<?xml version="1.0" encoding="iso-8859-1"?>

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

 <form id="clock">

  <grammar src="clock.grxml" type="application/srgs+xml"/>

  <block>
   <prompt> Welcome to the Clock Application </prompt>
  </block>

  <initial name = "start">

   <prompt> What would you like to know? </prompt>
   <help> Please say what time is it or what is todays date. </help>

   <noinput count="1">
    <reprompt/>
   </noinput>

   <noinput count="2">
    <reprompt/>
    <assign name="start" expr="true"/>
```

```
   </noinput>

  </initial>


  <field name = "mode">
   <prompt> What would you like to know? </prompt>
  </field>


  <object name="clock"
          archive="http://www.example.com/java/Clock.jar"
          classid="method://Clock/getCurrent"
          codetype="javacode">
   <param name="setLocale" value="en_US"/>
   <param name="setMode" expr="mode"/>

   <filled>
    <prompt>
     The current <value expr="mode"/> is <value expr="clock"/>.
    </prompt>
   </filled>
  </object>

 </form>
</vxml>
```

clock.java

```
// package com.ibm.speech.test;
import java.util.Date;
import java.util.Locale;
import java.text.SimpleDateFormat;
import java.util.HashMap;
import java.util.StringTokenizer;


/**
 * Class Clock represents a clock maintaining both time of day and
 * current date. Applications can call setMode to determine the mode
 * to return the clock value in. The mode can be DAY, DATE, MONTH, TIME.
 * The default mode is DATE. After setting the mode the application can
 * call getCurrent to obtain the current clock value for that mode.
 * If the application doesn't call setLocale the default local is used.
 *
 * @author bhm
 * @version 1.0
 * @see java.util.Calendar
 * @see java.util.Date
 */

public class Clock {

 public Clock () {

   mode = MDATE;
   locale = Locale.getDefault ();

 } // <init>
```

```
/**
 * Method to set the current clock mode. The clock mode determines the
 * format value return by getCurrent.
 * @param mode One of MDATE, MMONT, MDAY, MTIME
 */

public void setMode (String mode) {

 if (FORMAT_TABLE.containsKey (mode))
   this.mode = mode;

} // setMode ()


/**
 * Method to set the locale for the clock presentation value.
 * @param locName The locale name to set as the current locale.
 */

public void setLocale (String locName) {
 String lang = null;
 String country = null;
 StringTokenizer st = new StringTokenizer (locName, "_");

 if (st.hasMoreTokens ())
  lang = st.nextToken ();

 if (st.hasMoreTokens ()) country = st.nextToken ();

 this.locale = new Locale (lang, country);
} // setLocale ()


/**
 * Method to return the current clock value in the specified mode.
 * @return The current clock value in the specified mode.
 */

public String getCurrent () {
 return new SimpleDateFormat ((String) FORMAT_TABLE.get (mode), locale) .format (new Date ());
} // getCurrent ()


public static final String MTIME = "time";
public static final String MDATE = "date";
public static final String MDAY = "day";
public static final String MMONT = "month";

private static final HashMap FORMAT_TABLE = new HashMap (4);

static {
  FORMAT_TABLE.put (MTIME, "K:mm a"); // time
  FORMAT_TABLE.put (MDATE, "MMMM d, yyyy"); // date
  FORMAT_TABLE.put (MDAY, "EEEE"); // day
  FORMAT_TABLE.put (MMONT, "MMMM"); // month
} // <static init>

private String mode; // the clocks mode private
Locale locale; // the clocks locale

} // Clock
```

Calling legacy telephony applications

Java applications

This application uses the **<object>** element with the VRBE extensions to call a telephony application written in Java.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd"
   version="2.0">

<form id="Javaapp">
   <object name="vrbe1" classid="method://com.ibm.wvr.vxml2.NativeAppSupport/invokeJavaApplication"
        type="javacode-ext">
      <param name="setApplicationToInvoke" value="app2" />
      <param name="setApplicationData" expr="new java.util.Vector()" />
      <param name="setWaitForReturn" expr="true" />
      <filled>
         <log>
          Data returned from application is: <value expr="vrbe1.applicationData" />
          Completion code is: <value expr="vrbe1.completionCode" />
          Completion text is: <value expr="vrbe1.completionCodeText" />
          </log>
       </filled>
    </object>

    <block>
        <log>The output from the object is <value expr="vrbe1.applicationData"/></log>
        <if cond="(vrbe1.applicationData == 'Whatever')">
        <log>Application data received from Java class so invocation passed</log>
        <else/>
        <log>Application data not received, so invocation failed</log>
        </if>
    </block>

</form>
</vxml>
```

State table applications

This application uses the **<object>** element with the VRBE extensions to call a state table application.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd"
   version="2.0">
 <var name="parmArray" expr="new Array('TestDB', customer_id, '')"/><form id="state_table">

    <object name="vrbe1" classid="method://com.ibm.wvr.vxml2.NativeAppSupport/invokeStateTable"
         codetype="javacode-ext">
   <param name="setName" value="Welcome" />
   <param name="setEntryPoint" value="begin" />
<param name="setParms" expr="parmArray" />
   <filled>
    <prompt>
     completion code text is
     <value expr="vrbe1.completionCodeText" />
     completion code is
     <value expr="vrbe1.completionCode" />
     return code is
     <value expr="vrbe1.returnCode" />
```

```
    number of parms is
    <value expr="vrbe1.parms.length" />
    parm 1 is
    <value expr="vrbe1.parms[0]" />
    parm 2 is
    <value expr="vrbe1.parms[1]" />
   </prompt>
  </filled>
 </object>

    <block>
    <log>The output from the object is <value expr="vrbe1.applicationData"/></log>
       <if cond="(vrbe1.applicationData == 'Whatever')">
       <log>Application data received from Java class so invocation passed</log>
       <else/>
       <log>Application data not received, so invocation failed</log>
       </if>
    </block>

 </form>
</vxml>
```

Using *n*-best

The application **nbest.vxml** prompts the user to say a 6-digit number. Five *n*-best results are returned.

The following grammar, **id.grxml**, supports the application **nbest.vxml** shown below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<grammar version="1.0" xmlns="http://www.w3.org/2001/06/grammar"
  xml:lang="en-US" mode="voice" root="id" tag-format="semantics/1.0">

 <rule id="id">
  <item repeat="6">
   <ruleref uri="#digit"/>
  </item>
 </rule>

 <rule id="digit">
  <one-of>
   <item>0</item>
   <item>1</item>
   <item>2</item>
   <item>3</item>
   <item>4</item>
   <item>5</item>
   <item>6</item>
   <item>7</item>
   <item>8</item>
   <item>9</item>
  </one-of>
 </rule>

</grammar>
```

nbest.vxml

```
<?xml version="1.0" encoding="iso-8859-1"?>

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

 <property name="maxnbest" value="5"/>

 <form>

  <field name="id">
   <grammar type="application/srgs+xml" src="id.grxml"/>

   <prompt> Please say your 6 digit identification number.</prompt>

   <filled>

    <prompt>
     There are <value expr="application.lastresult$.length"/> results.
    </prompt>

    <if cond="application.lastresult$.length >= 1">
      <prompt>
       <value expr="application.lastresult$[0].utterance"/> has score
       <value expr="application.lastresult$[0].confidence"/>.
      </prompt>
    </if>

    <if cond="application.lastresult$.length >= 2">
      <prompt>
       <value expr="application.lastresult$[1].utterance"/> has score
       <value expr="application.lastresult$[1].confidence"/>.
      </prompt>
    </if>

    <if cond="application.lastresult$.length >= 3">
      <prompt>
       <value expr="application.lastresult$[2].utterance"/> has score
       <value expr="application.lastresult$[2].confidence"/>.
      </prompt>
    </if>

    <if cond="application.lastresult$.length >= 4">
      <prompt>
       <value expr="application.lastresult$[3].utterance"/> has score
       <value expr="application.lastresult$[3].confidence"/>.
      </prompt>
    </if>

    <if cond="application.lastresult$.length >= 5">
      <prompt>
       <value expr="application.lastresult$[4].utterance"/> has score
       <value expr="application.lastresult$[4].confidence"/>.
      </prompt>
    </if>
```

```
    </filled>
   </field>
  </form>
 </vxml>
```

Appendix A. CTTS caching

An audio-caching function is provided by default for all supported CTTS voices in all supported languages except Simplified Chinese. The function caches 20 MB of shared memory. Using this function for repetitive text-to-speech synthesis can speed up synthesis processing significantly. However, audio caching might not improve processor performance or memory use for WebSphere Voice Server. Consult your IBM representative for more information.

Appendix B. Canadian French

This appendix contains information that is specific to Canadian French. If you are developing Canadian French voice applications, use the information in this appendix instead of the equivalent US English information in the corresponding sections of this book.

- "Built-in field types and grammars"
- "Predefined events" on page 144
- "Built-in commands" on page 145
- "Specifying character encoding" on page 145
- "Testing built-in field types" on page 145
- "SSML elements and attributes" on page 146

Built-in field types and grammars

Table 33 shows the built-in field types and grammars for Canadian French.

Table 33. Canadian French built-in types. Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
boolean	Users can say positive responses such as **oui**, **oui oui**, **ok**, **tout à fait**, **oui bien sûr**, **d'accord**, **vrai**, **affirmatif**, **oui c'est ça**, **juste**, and **c'est juste**, or negative responses such as **non**, **non non**, **faux**, **négatif**, **pas du tout**, and **pas question**. Variations on "oui" (**ouais**) may also be used. Users can also provide DTMF input **1** is yes, and **2** is no. **Note:** The IBM TTS engine is unable to synthesize the return value for this language.

Table 33. Canadian French built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
currency	Users can say currency values as a combination of a dollar component, such as "**vingt-sept dollars**" and a cent component, such as "**cinquante cents**." Users can also say one of these components without the other. If both components are present, they can be either not separated (as in "**vingt dollars soixante-dix**"), or separated by the word "**et**" (as in "**trente dollars et quarante cents**"). A dollar component may be in any of the formats: "**zéro dollar**," "**1 dollar**," "m **dollars**" (where 2 <= m <= 999,999,999), and "m" (where 1 <= m <= 999,999,999). A cent component may be in any of the formats: "**zéro cent**" and "n **cent**" (where 1 <= n <= 99). If both components are present, the cent component can also have the format "m" (where 0 <= n <= 99). Note that if users just say an integer value "m" (where 1 <= m <= 999,999,999), this will be interpreted as a number of dollars. Note also that users can also speak two numerical values, where the first is between 0 and 999,999,999 and the second is between 0 and 99. This will be interpreted as a number of dollars followed by a number of cents. Also, "**euro**," "**euros**," "**cent**," and "**cents**" may be used as currency types in place of "**dollar**," "**dollars**, " "**cent**," and "**cents**" in the above description. Similarly, "**dollar US**," "**dollars US**," "**cent**," and "**cents**" may be used. Note that both "**euro**" and "**euros**" can be used as the plural of "**euro**." Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a decimal point), and must terminate DTMF entry using the # key. The return value sent is a string in the format *UUUddddddddd.cc* where *UUU* is a currency indicator (CAD, EUR, USD). **Note:** The IBM TTS engine is unable to synthesize the return value for this language.

Table 33. Canadian French built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
date	Users can say a date using days, months and years, as well as the words **hier**, **aujourd'hui**, and **demain**. The year must be between 1900 and 2099. Users can also specify the year as a 4-digit year, such as "**en deux mille deux**" (2002) or a 2-digit year (such as "**cinquante-cinq**"). A 2-digit year will be interpreted as being between 1911 and 2010. Users can also say the day and month without specifying a year. This will be interpreted as a date in the current year. The "days" component can be a spoken as a number (between 1 and 31) or the ordinal number "**premier**" as alternative for the first day of the month. The "months" component is either the name of a month or a number between 1 and 12 that specifies the month. When their value is less or equal to 9, both "days" and "months" (in numeric form) can optionally be preceded by "0" (pronounced **zéro**). Most of the dates users can say can be spoken in the form: "**[le]** d **mars deux mille un**" (where d represents an integer number of days 1<=d<=31). For example: "**trois mars deux mille un**" (March 3, 2001). If users say only a month or a month and a year, the following alternatives apply: • When the name of the month begins with a consonant, as in **janvier**, **février**, **mars**, **mai**, **juin**, **juillet**, **septembre**, **novembre**, and **décembre**, the date can be spoken according to the scheme "**[en | au mois de] mars deux mille un**." For example: "**en mars deux mille un**" (March 2001). • When the month name begins with a vowel as in **avril**, **août**, and **octobre**, the scheme is "**[en | au mois d'] avril deux mille un**." For example "**au mois d'avril deux mille un**" (April 2001). • The "months" component can be spoken as a number m (such as 1<=m<=12) according to the following construct "**[le]** d m **deux mille un**," where 1<=d<=31. For example, "**dix octobre deux mille un**" (October 10, 2001). Any of these constructs can be optionally preceded by the day of the week and its proper optional article (**[le] lundi**, **[le] mardi**, **[le] mercredi**, **[le] jeudi**, **[le] vendredi**, **[le] samedi**, **[le] dimanche**), such as "**jeudi trois mars deux mille un**" (Thursday, March 3, 2001). However, the day and its article will be ignored. Thus, the specified date will be accepted whether it falls on the specified day. Users can also provide DTMF input in the form *yyyymmdd*. **Note:** The date grammar does not perform leap year calculations. February 29 is accepted as a valid date regardless of the year. If desired, your application or servlet can perform the required calculations. The return value sent is a string in the format *yyyymmdd*, with the VoiceXML browser returning a **?** in any positions omitted in spoken input. **Note:** The IBM TTS engine is unable to synthesize the return value for this language.

Table 33. Canadian French built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
digits	Users can say non-negative integer values as strings of individual digits (0 through 9). For example, a user could say "**123456**" as "**un deux trois quatre cinq six**." Users can say "**zéro**," "**un**," "**deux**," "**trois**," "**quatre**," "**cinq**," "**six**," "**sept**," "**huit**," and "**neuf**." Users can also provide DTMF input using the numbers 0 through 9, and must terminate DTMF entry using the # key. The return value sent is a string of one or more digits. If the result is subsequently used in **<say-as>** with the **interpret-as** value **vxml:digits**, it will be spoken as a sequence of digits appropriate to the current language. For example, the TTS engine speaks "123456" as "**un deux trois quatre cinq six**."
number	Users can say natural numbers (that is, positive and negative integers, 0, and decimals) from 0 to 999,999,999.9999. Users can say the words **point** and **virgule** to indicate a decimal point, **plus** to indicate a positive number, and **moins** to indicate a negative number. Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a comma), and must terminate DTMF entry using the # key. Only positive numbers can be entered using DTMF. The return value sent is a string of one or more digits, 0 through 9 , with a decimal point and a - sign as applicable. **Note:** The IBM TTS engine is unable to synthesize the return value for this language.

Table 33. Canadian French built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
phone	Users can say a telephone number, including the optional word **extension** or **poste** followed by extra numbers or digits. A phone number is composed of seven digits. The first three digits are uttered each in turn and the remaining ones are uttered digit by digit or double digits by double digits. For example, "714-3274" can be said as either "**sept un quatre trois deux sept quatre**" or "**sept un quatre trente-deux soixante-quatorze**." Users can also say an area code before a phone number either digit by digit or as a number. For example, "(866) 714-3274" can be said as either "**huit six six sept un quatre trois deux sept quatre**" or "**huit cent soixante-six sept un quatre trente-deux soixante-quatorze**." Users can also say a set of four digits before a phone number; this identifies a toll-free number. For example, "1 800 714-3274" can be said as either "**un huit zéro zéro sept un quatre trois deux sept quatre**" or "**un huit cents sept un quatre trois deux sept quatre**." Both national and international numbers can be given with an extension, which is replaced in the value field of the grammar with **x**. For example, a national phone number with an extension can be said either as "**514 911 11 99 extension 33 53 32**" or as "**514 911 11 99 poste 33 53 32**." Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to represent the word "extension"), and must terminate DTMF entry using the # key. The return value sent is a string of digits which includes an **x** if an extension was specified. The area code, if provided, is surrounded by brackets, and a dash is included after the third digit of the phone number. However, the IBM TTS engine is unable to synthesize the return value for this language. **Note:** For tips on minimizing recognition errors that are due to user pauses during input, see "Using the built-in phone grammar" on page 62.

Table 33. Canadian French built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
time	Users can say a time of day using hours and minutes in either 12- or 24-hour format, as well as the word **maintenant**. Times in 12-hour format can be stated in any of the following formats: n **heures** [**et**] m [**minutes**] **le matin à** n **heures** [**et**] m [**minutes**] **l'après-midi à** n **heures** [**et**] m [**minutes**] **le soir à** n **heures** [**et**] m [**minutes**] n **heures** [**et**] m [**minutes**] **du matin** n **heures** [**et**] m [**minutes**] **de l'après-midi** n **heures** [**et**] m [**minutes**] **du soir** n **heures et quart** n **heures moins quart** n **heures moins** m [**minutes**] n **heures** [**et demi** | **trente**] **midi et quart** **midi moins quart** **midi et demie** **minuit et quart** **minuit moins quart** **minuit et demie** where 1 <= m <= 59 and 0 <= n <= 12 Times in 24-hour format can be stated in the following formats : n **heures** [**et**] m [**minutes**] **du matin** where 1 <= m <= 59 and 0 <= n <= 23 Users can precede any of these times with "**vers**," "**il est**," "**il est exactement**," "**à partir de**," "**avant**," or "**après**." This will be ignored, and the exact time will be accepted. Users can optionally follow any of these times with "**précises**" or "**précisément**." Users can also provide DTMF input using the numbers 0 through 9. The return value sent is a string in the format *hhmmx*, where *x* is **a** for a.m., **p** for p.m., **h** for 24 hour format or **?** if unspecified or ambiguous. For DTMF input, the return value will always be **h** or **?,** since there is no mechanism for specifying a.m. or p.m. **Note:** The IBM TTS engine is unable to synthesize the return value for this language.

Predefined events

Table 34 on page 145 shows the Canadian French default event-handler messages for the error, help, and nomatch events.

Table 34. Canadian French predefined events and event-handler messages

Event	Default event-handler message
error.badfetch **error.noauthorization** **error.semantic** **error.unsupported.***element*	Désolé, une erreur s'est produite en traitant votre demande
help	Désolé, aucune aide supplémentaire n'est disponible pour cette application
nomatch	Désolé, je n'ai pas compris ce que vous avez dit

Built-in commands

Table 35 shows the built-in VoiceXML browser commands for Canadian French:

Table 35. Canadian French built-in VoiceXML browser commands

Grammar name	Valid user utterances	VoiceXML browser response
Quiet/Cancel	tranquille | arrête | arrête-toi | pause | annuler	When barge-in is enabled, stops the current spoken output and waits for further instructions from user.
Help	aide | je suis perdu | quoi	Plays the help event-handler message. See Table 24 on page 97.

Specifying character encoding

The default character encoding for XML documents is utf8, which has 7-bit ASCII as a proper subset. Character codes in VoiceXML documents that are greater than 127 (such as special characters è or ä) will therefore be interpreted as the first byte of a multi-byte utf8 sequence. You can force the XML parser to use another codepage by specifying the desired encoding as the very first thing in the file:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Testing built-in field types

Table 36 on page 146 provides examples of the types of input you might specify when testing a Canadian French voice application that uses the built-in field types.

Table 36. Sample input for Canadian French built-in field types

Built-in field type	Sample input
boolean	oui, ok, tout à fait, oui bien sûr, d'accord, vrai, affirmatif, oui c'est ça, c'est juste
currency	neuf virgule soixante dollars trois mille dollars neuf dollars quatre-vingt-dix cents un million de piastres
date	le trois mars en novembre huit août deux mille treize hier aujourd'hui demain
digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
number	dix millions cinq cent mille cinquante-trois moins deux virgule cinq plus deux virgule cinq
phone	quatre cent cinquante-trois un zéro un vingt-deux un zéro huit un neuf sept trois huit quatre un un deux un huit cents sept trois huit trente douze poste trente-trois douze quatre un huit cinq un deux trente quinze extension treize
time	deux heures sept heures quinze neuf heures sept du matin six heures moins quart minuit maintenant

SSML elements and attributes

Table 37 on page 147 shows the notes about SSML elements and attributes in Canadian French.

Table 37. Limitations for Canadian French SSML elements

Element	Implementation details
<phoneme>	The IPA alphabet is not supported. The IBM alphabet is supported. The IBM alphabet used in SSML refers to the phonology used by IBM TTS. The following example shows the US English phonetic pronunciation of “tomato” using the IBM TTS phonetic alphabet: `<phoneme alphabet="ibm" ph=".0tx.1me.0Fo"> tomato </phoneme>` For more information on IBM SPRs, see the *IBM Text-To-Speech SSML Programming Guide*.
<prosody>	The **pitch**, **range** and **rate** attributes are not supported.
<say-as>	The **interpret-as** attribute is only supported with the value ″**vxml:digits**″. The **digits** and **letters**attributes are also supported

Appendix C. German

This appendix contains information that is specific to German. If you are developing German voice applications, use the information in this appendix instead of the equivalent US English information in the corresponding sections of this book.

- "Built-in field types and grammars"
- "Predefined events" on page 153
- "Built-in commands" on page 154
- "Specifying character encoding" on page 154
- "Testing built-in field types" on page 155

Built-in field types and grammars

Table 38 shows the built-in field types and grammars for German.

Table 38. German built-in types. Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
boolean	Users can say one of the positive responses **ja**, **jawohl**, **stimmt**, **klar**, [**das ist**] **richtig**, **positiv**, **OK**, **natürlich**, **sicher**, or **auf jeden Fall**, or one of the negative responses **nein**, **falsch**, **negativ**, **auf** [**gar**] **keinen Fall**. Variations on "**ja**" (**joh**) and "**nein**" (**nee**, **nö**) may also be used. Users can also provide DTMF input: **1** is yes, and **2** is no. The return value sent is a boolean "true" for a positive response or "false" for a negative response. If the field name is subsequently used in a **value** attribute within a prompt, the TTS engine will speak "**ja**" or "**nein**."

Table 38. German built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
currency	Users can say currency values as a combination of a Euro component and a Cent component, such as "**siebenundzwanzig Euro**" and "**fünfzig Cent**." Users can also say one of these components without the other. If both components are present, they can be either not separated or separated by the word "**und**," as in "**zwanzig Euro siebzig**" and "**zwanzig Euro und siebzig Cent**." A Euro component may be in any of the formats: "**null Euro**," "**ein Euro**," "**m Euro(s)**" (where 2 <= m <= 999,999,999) and "m" (where 1 <= m <= 999,999,999). A Cent component may be in any of the formats: "**null Cent**," "**ein Cent**," "**n Cent(s)**" (where 2 <= n <= 99). If both components are present, the Cent component can also have the format "n" (where 0 <= n <= 99). Note that if users just say an integer value "m" (where 1 <= m <= 999,999,999), this will be interpreted as a number of Euro. Note also that users can also speak two numerical values, where the first is between 0 and 999,999,999 and the second is between 0 and 99. This will be interpreted as a number of Euros followed by a number of Cents. Also, "**Franken**" and "**Rappen**" may be used as currency types in place of "**Euro**," "**Euros**," "**Cent**," and "**Cents**" in the above specification. Similarly, "**Dollar**," "**Dollars**," "**Cent**," and "**Cents**" may be used. Note that both "**Euro**" and "**Euros**" can be used as the plural of "**Euro**." Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a decimal point), and must terminate DTMF entry using the # key. The return value sent is a string in the format *UUUddddddddd.cc* where *UUU* is a currency indicator (CHF, EUR, USD). If the field name is subsequently used in a **value** attribute within a prompt, the IBM TTS engine will speak the currency value. For example, the IBM TTS engine speaks "**EUR9,30**" as "**neun Komma drei null Euro**." **Note:** The above example only applies to currency, time, and number built-in grammars. For boolean, date, phone, and digits built-in grammars, the example is only correct if the field value is subsequently used in a **<say-as>** element with an **interpret-as** attribute value.

Table 38. German built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
date	Users can say a date using months, days and years, as well as the words **gestern**, **heute**, and **morgen**. The year must be between 1900 and 2099. Users can speak years from 1901 to 1999 as "**neunzehn** <1- or 2-digit number>" or "**neunzehn hundert [und]** <1- or 2-digit number>,"such as "**neunzehn vierundsiebzig**" and "**neunzehn hundert [und] vierundsiebzig**." Users can speak years from 2001 to 2099 as "**zweitausend [und]** <1- or 2-digit number>" such as "**zweitausend [und] vier**" and "**zweitausend [und] fünfundachtzig**." Uses can also specify a 2-digit year (such as "**einundneunzig**"). A 2-digit year will be interpreted as being between 1911 and 2010. The following common constructs for dates are supported: "**dritter März zweitausend [und] eins**," "**der dritte März zweitausend [und] eins**," and "(**am** | **den**) **dritten März zweitausend [und] eins**." Users can say "**Januar**" or "**Jänner**" for "**January**" and "**Juni**" or "**Juno**" for "**Juni**." Users can also say the day and month without specifying a year. This will be interpreted as a date in the current year. Any of these constructs can be preceded by a day of the week, such as "**Donnerstag dritter März zweitausend [und] eins**," "**Donnerstag der dritte März zweitausen**d **[und] eins**," and "**[am] Donnerstag den dritten März zweitausend [und] eins**." However, the day will be ignored. Thus the specified date will be accepted, whether or not it falls on the specified day. Users can also provide DTMF input in the form *yyyymmdd*. **Note:** The date grammar does not perform leap year calculations. February 29th is accepted as a valid date regardless of the year. If desired, your application or servlet can perform the required calculations. The return value sent is a string in the format *yyyymmdd*, with the VoiceXML browser returning a **?** in any positions omitted in spoken input. If the value is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:date**, then it is spoken as a date appropriate to this language.
digits	Users can say non-negative integer values as strings of individual digits (0 through 9). For example, a user could say "**eins zwo drei vier fünf sechs**." Users can say ″2″ by saying "**zwei**" or "**zwo**," and "5" by saying "**fünf**" or "**fünef**." Users can also provide DTMF input using the numbers 0 through 9, and must terminate DTMF entry using the # key. The return value sent is a string of one or more digits. If the result is subsequently used in **<say-as>** with the **interpret-as** value **vxml:digits**, it will be spoken as a sequence of digits appropriate to the current language. For example, the TTS engine speaks "123456" as "**eins zwei drei vier fünf sechs**."

Table 38. German built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
number	Users can say natural numbers (that is, positive and negative integers, 0, and decimals) from -999,999,999.9999 to 999,999,999.9999. Users can say the word **Komma** (to indicate a decimal point) and **plus** or **minus** (to indicate a positive or negative number). Users can say "2" by saying "**zwei**" or "**zwo**," "5" by saying "**fünf**" or "**fünef**," and "50" by saying "**fünfzig**" or "**fuffzig**." Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a comma), and must terminate DTMF entry using the # key. Only positive numbers can be entered using DTMF. The return value sent is a string of one or more digits, 0 through 9 , with a decimal point and a + or - sign as applicable. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:type**, where type is the type number you want to specify, then it is spoken as that type number appropriate to this language. For example, the TTS engine speaks &odq;123456&cdq; as &odq;<ph style="bold"> einhundert dreiundzwanzig tausend vierhundert f&ue;nfundsechzig</ph>.&cdq; Use **<say-as interpret-as="vxml:digit">** to have the number said as a string of digits.
phone	Users can say a telephone number, including the optional words **Durchwahl**, **Klappe**, or **Nebenstelle** to indicate an extension. In addition to digits (1 to 9), users can use double digits (such as "49") and the phrases "**zwei mal [die]**...," and "**drei mal [die]**..." before a digit. Users can also use the word "**hundert**, which is interpreted as "00." Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to represent the word "extension"), and must terminate DTMF entry using the # key. The return value sent is a string of digits which includes an **x** if an extension was specified. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:phone**, then it is spoken as a phone number appropriate to this language. **Note:** For tips on minimizing recognition errors that are due to user pauses during input, see "Using the built-in phone grammar" on page 62.

Table 38. German built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
time	Users can say a time of day using hours and minutes in either 12- or 24-hour format, as well as the words and phrases "**jetzt**," "**[jetzt] sofort**," and "**[jetzt] gleich**." Times can be stated in any of the following formats: **[morgens | am Morgen | vormittags | am Vormittag | mittags | nachmittags | am Nachmittag | abends | am Abend | nachts | in der Nacht | in der Früh]** ... ... **[eine Minute (nach | vor)]** n **[Uhr]** ... ... **[um]** [m (**nach | vor**)] n **[Uhr]** ... ... **[um]** [m **Minuten** (**nach | vor**)] n **[Uhr]** ... ... **[um] [viertel (nach | vor)]** n ... ... **[um] [viertel | dreiviertel]** n ... ... **[um] [halb]** n ... ... **[um]** N **Uhr** (M) ... ... **[morgens | am Morgen | vormittags | am Vormittag | mittags | nachmittags | am Nachmittag | abends | am Abend | nachts | in der Nacht | in der Früh | früh]** **[um] Mitternacht** **[am] Mittag** **12 Uhr Mittag** where 1 <= m <= 20, 1 <= M <= 59,1 <= n <= 12, and 0 <= N <= 24 Users can also provide DTMF input using the numbers 0 through 9. The return value sent is a string in the format *hhmmx*, where *x* is **a** for AM, **p** for PM, **h** for 24 hour format, or **?** if unspecified or ambiguous. For DTMF input, the return value will always be **h** or **?**, since there is no mechanism for specifying AM or PM. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:time**, then it is spoken as a time appropriate to this language.

Predefined events

Table 39 shows the German default event-handler messages for the error, help, and nomatch events.

Table 39. German predefined events and event-handler messages

Event	Default event-handler message
error.badfetch **error.noauthorization** **error.semantic** **error.unsupported.element**	Verarbeitungsfehler - Das Programm wird beendet
help	Keine weitere Hilfe verfügbar
nomatch	Entschuldigung, ich habe Sie nicht verstanden

Built-in commands

Table 40 shows the built-in VoiceXML browser commands for German:

Table 40. German built-in VoiceXML browser commands

Grammar Name	Valid User Utterances	VoiceXML Browser Response
Quiet/Cancel	[bitte] Ruhe [bitte] | sei [bitte] still | abbrechen	When barge-in is enabled, stops the current spoken output and waits for further instructions from user.
Help	Hilfe | zu Hilfe | wie bitte	Plays the help event-handler message. See Table 24 on page 97.

Specifying character encoding

The default character encoding for XML documents is utf8, which has 7-bit ASCII as a proper subset. Character codes in VoiceXML documents that are greater than 127 (such as special characters è or ä) will therefore be interpreted as the first byte of a multi-byte utf8 sequence. You can force the XML parser to use another codepage by specifying the desired encoding as the very first thing in the file:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Testing built-in field types

Table 41 provides examples of the types of input you might specify when testing a German voice application that uses the built-in field types.

Table 41. Sample input for German built-in field types

Built-in field type	Sample input
boolean	ja, ok, natürlich, richtig, positiv, sicher, nein, falsch, negativ, auf keinen Fall
currency	siebenundzwanzig neunzig drei Euro fünfzig sieben Franken und neunundneunzig Rappen zehn Millionen Euro
date	fünfter Mai März der siebzehnte Dezember neunundneunzig gestern heute morgen
digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, zwo
number	zehn Millionen fünf hundert tausend und dreiundfünfzig minus eins komma fünf plus eins komma fünf
phone	sieben drei fünf acht vier neun null null vier neun siebenundachtzig sechsundvierzig drei drei sechzehn vierzig sieben zweimal die null dreizehn Durchwahl zwölf Nebenstelle acht sieben fünf drei null null achthundert vier vier sechs vier
time	ein Uhr ein Uhr sieben halb drei viertel vor zehn Uhr abends zehn vor sechs jetzt

Appendix D. Japanese

This appendix contains information that is specific to Japanese. If you are developing Japanese voice applications, use the information in this appendix instead of the equivalent US English information in the corresponding sections of this book.

- "Built-in field types and grammars"
- "Predefined events" on page 160
- "Built-in commands" on page 160
- "Specifying character encoding" on page 160
- "Testing built-in field types" on page 160
- "SSML elements and attributes" on page 162

Built-in field types and grammars

Table 42 shows the built-in field types and grammars for Japanese.

Table 42. Japanese built-in types

Element	Implementation details
boolean	Users can say positive responses such as はい , **ok**, **and** 了解 or negative responses such as いいえ and 却下 . Users can also provide DTMF input: **1** is yes, and **2** is no. The return value sent is a boolean true or false. If the field name is subsequently used in a **value** attribute within a prompt, the TTS engine will speak はい or いいえ .

Table 42. Japanese built-in types (continued)

Element	Implementation details
currency	Users can say Japanese currency values in “yen” from 0 to 999,999,999,999. Users can also provide DTMF input using the numbers 0 through 9 and must terminate DTMF entry using the # key. The return value sent is a string in the format *UUUddddddddd.cc*, where *UUU* is a currency indicator; currently the only supported currency type is JPY (for Japanese yen.) If the field name is subsequently used in a **value** attribute within a prompt, the TTS engine will speak the currency value. In DTMF input’s case, only input digits are returned.
date	Users can say a date using months, days, and years, as well as the words 昨日 , 今日 , and 明日 . Common constructs such as “ 2000年3月3日 ” are supported. Users can also provide DTMF input in the form *yyyymmdd*. **Note:** The date grammar does not perform leap year calculations. 2月29日 is accepted as a valid date regardless of the year. If desired, your application or servlet can perform the required calculations. The return value sent is a string in the format *yyyymmdd*, with the VoiceXML browser returning a **?** in any positions omitted in spoken input. If the value is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:date**, then it is spoken as a date appropriate to this language.
digits	Users can say numeric integer values as individual digits (0 through 9). For example, a user could say 123456 as “1 2 3 4 5 6.” Users can also provide DTMF input using the numbers 0 through 9, and must terminate DTMF entry using the # key. The return value sent is a string of one or more digits. If the result is subsequently used in **<say-as>** with the **interpret-as** value **vxml:digits**, it will be spoken as a sequence of digits appropriate to the current language. In the above example, the TTS engine speaks 123456 as **1 2 3 4 5 6**. **Note:** Use this type instead of the number type if you require very high recognition accuracy for your numeric input.

Table 42. Japanese built-in types (continued)

Element	Implementation details
number	Users can say natural numbers (that is, positive and negative integers and decimals) from 0 to 999,999,999,999 as well as the words てん (to indicate a decimal point) and マイナス (to indicate a negative number). Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a decimal point), and must terminate DTMF entry using the # key. **Note:** Only positive numbers can be entered using DTMF. The return value sent is a string of one or more digits, 0 through 9, with a decimal point and a + or - sign as applicable. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:type**, where type is the type number you want to specify, then it is spoken as that type number appropriate to this language. The TTS engine speaks 123456 as 十二万三千四百五十六 . Use **<say-as interpret-as="vxml:digit">** to have the number said as a string of digits.
phone	Users can say a telephone number, in 10 or 11 digits, including the first digit "0". Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to represent the "extension"), and must terminate DTMF entry using the # key. The return value sent is a string of digits without hyphens. The return value sent includes an *x* if an extension was specified in DTMF input's case. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:phone**, then it is spoken as a phone number appropriate to this language. **Note:** For tips on minimizing recognition errors that are due to user pauses during input, see "Using the built-in phone grammar" on page 62.
time	Users can say a time of day using hours and minutes in either 12- or 24-hour format, as well as the word 現在の時刻 . Users can also provide DTMF input using the numbers 0 through 9. The return value sent is a string in the format *hhmmx*, where *x* is **a** for a.m., **p** for p.m., **h** for 24 hour format, or **?** if unspecified or ambiguous; for DTMF input, the return value will always be **h** or **?**, since there is no mechanism for specifying a.m. or p.m. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:time**, then it is spoken as a time appropriate to this language.

Predefined events

Table 43 shows the Japanese default event-handler messages for the error, help, and nomatch events.

Table 43. Japanese predefined events and event-handler messages

Event	Default event-handler message
error.badfetch **error.noauthorization** **error.semantic** **error.unsupported.***element*	エラーがあったので終了します。
help	ヘルプは用意されていません。
nomatch	すみません。理解できません。

Built-in commands

Table 44 shows the built-in VoiceXML browser commands for Japanese:

Table 44. Japanese built-in VoiceXML browser commands

Grammar name	Valid user utterances	VoiceXML browser response
Quiet/Cancel	静かに | キャンセル	When barge-in is enabled, stops the current spoken output and waits for further instructions from user.
Help	ヘルプ	Plays the help event-handler message. See Table 24 on page 97.

Specifying character encoding

The default character encoding for XML documents is utf8, which has 7-bit ASCII as a proper subset. You can force the XML parser to use another codepage by specifying the desired encoding as the very first thing in the file:

```
<?xml version="1.0" encoding="Shift_JIS"?>
```

Testing built-in field types

Table 45 on page 161 provides examples of the types of input you might specify when testing a Japanese voice application that uses the built-in field types.

Table 45. Sample input for Japanese built-in field types

Built-In Field Type	Sample Input
boolean	はい , OK, 了解 , いいえ , 却下
currency	三千百二十五円 九百二十万円
date	2月29日 2000年12月31日 昨日 今日 明日
digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
number	150万 マイナス1.5 プラス1.5 100.53
phone	0123456789 09012345678
time	1時 午前3時15分 午後7時半 現在の時刻

SSML elements and attributes

Table 46 shows the notes about VoiceXML elements and attributes for Japanese.

Table 46. Limitations for Japanese SSML elements

Element	Implementation details
<emphasis>	This element is not supported.
<prosody>	The **pitch**, **range** and **rate** attributes are not supported.

Appendix E. Korean

This appendix contains information that is specific to Korean. If you are developing Korean voice applications, use the information in this appendix instead of the equivalent US English information in the corresponding sections of this book. If you are using a text-to-speech facility provided by another vendor, you should refer to the vendor's documentation for details on their implementation of VoiceXML 2.0 and SSML. There may be differences that you should be aware of as you develop your XML documents.

- "Built-in field types and grammars"
- "Predefined events" on page 167
- "Built-in commands" on page 168
- "Specifying character encoding" on page 168
- "Testing built-in field types" on page 168
- "SSML elements and attributes" on page 170

Built-in field types and grammars

Table 47 on page 164 shows the built-in field types and grammars for Korean.

Table 47. Korean built-in field types

Element	Implementation details
boolean	Users can say positive responses such as 예 , 그렇습니다 and 맞습니다 or negative responses such as 아니오 and 아닙니다 . Users can also provide DTMF input: **1** is yes, and **2** is no. The return value sent is a boolean true or false. If the field name is subsequently used in a **value** attribute within a prompt, the IBM TTS engine will speak 예 or 아니오 .
currency	Users can say Korean currency values in “won” from 0 to 999,999,999. Users can also provide DTMF input using the numbers 0 through 9 and must terminate DTMF entry using the # key. The return value sent is a string in the format *UUUddddddddd*, where *UUU* is a currency indicator; currently the only supported currency type is WON (for Korean won). If the field name is subsequently used in a **value** attribute within a prompt, the IBM TTS engine will speak the currency value.

Table 47. Korean built-in field types (continued)

Element	Implementation details
date	Users can say a date using months, days, and years, or special dates such as 어제 , 오늘 and 내일 . Common constructs such as 2002 년 10 월 9 일 are currently supported. The grammar performs a validation check. For example, an illegal date such as 2 월 30 일 will not be accepted. Users can also provide DTMF input in the form *yyyymmdd*. If the value is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:date**, then it is spoken as a date appropriate to this language.
digits	Users can say numeric integer values as individual digits (0 through 9). For example, a user could say 12345 as 일 둘 삼 사 다섯 . Users can also provide DTMF input using the numbers 0 through 9, and must terminate DTMF entry using the # key. The return value sent is a string of one or more digits. If the result is subsequently used in **<say-as>** with the **interpret-as** value **vxml:digits**, it will be spoken as a sequence of digits appropriate to the current language. In the above example, the IBM TTS engine speaks 12345 as 일 이 삼 사 오 .

Table 47. Korean built-in field types (continued)

Element	Implementation details
number	Users can say natural numbers (that is, positive and negative integers and decimals) from 0 to 999,999,999 as well as the words 점 (to indicate a decimal point) and 마이너스 (to indicate a negative number). Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a decimal point), and must terminate DTMF entry using the # key. **Note:** Only positive numbers can be entered using DTMF. The return value sent is a string of one or more digits, 0 through 9, with a decimal point and a + or - sign as applicable. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:type**, where type is the type number you want to specify, then it is spoken as that type number appropriate to this language. The IBM TTS engine speaks 12345 as 만이천 삼백사십 오 . Use **<say-as interpret-as="vxml:digit">** to have the number said as a string of digits.
phone	Users can say a landline telephone number, including the region number, such as 공 이 삼 칠 팔 일 에 구천 이십 번 or a cellular phone number such as 공 일 육 삼 팔 이 국에 오 팔 오 이 . Among the numbers or digits, the separation words such as 예 and 국에 can exist. Users can also provide DTMF input using the numbers 0 through 9 and must terminate DTMF entry using the # key. The return value sent is a string of digits without hyphens. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:phone**, then it is spoken as a phone number appropriate to this language.

Table 47. Korean built-in field types (continued)

Element	Implementation details
time	Users can say a time of day using hours and minutes in either 12 or 24-hour format, as well as the words 지금 , 정오 or 자정 . Quantifiers such as 한 or 약 can be added, and tailing words such as 경 , 경에 or 에 can also be added. Users can also provide DTMF input using the numbers 0 through 9. The return value sent is a string in the format *hhmmx*, where *x* is **a** for a.m., **p** for p.m., **h** for 24-hour format, or **?** if unspecified or ambiguous; for DTMF input, the return value will always be **h** or **?**, since there is no mechanism for specifying a.m. or p.m. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:time**, then it is spoken as a time appropriate to this language.

Predefined events

Table 48 on page 168 shows the Korean default event-handler messages for the error, help, and nomatch events.

Table 48. Korean predefined events and event-handler messages

Event	Default event-handler message
error.badfetch **error.noauthorization** **error.semantic** **error.unsupported.***element*	죄송합니다, 문제가 발생하여 종료합니다.
help	죄송합니다, 도움말이 없습니다.
nomatch	죄송합니다, 알아듣지 못했습니다.

Built-in commands

Table 49 shows the built-in VoiceXML browser commands for Korean:

Table 49. Korean built-in VoiceXML browser commands. Square brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Grammar name	Valid user utterances	VoiceXML browser response
Quiet/Cancel	그만 | 스탑 | 스톱 | 취소 | 취소해 [주세요] | 멈춰 [주세요] | 요	When barge-in is enabled, stops the current spoken output and waits for further instructions from user.
Help	도움말 | 도와줘 | 도와주세요 | 도우미 | 사용법 [안내] | 안내	Plays the help event-handler message. See Table 24 on page 97.

Specifying character encoding

The default character encoding for XML documents is utf8, which has 7-bit ASCII as a proper subset. You can force the XML parser to use another codepage by specifying the desired encoding as the very first thing in the file:

```
<?xml version="1.0" encoding="euc-KR"?>
```

Testing built-in field types

Table 50 on page 169 provides examples of the types of input you might specify when testing a Korean voice application that uses the built-in field types.

Table 50. Sample input for Korean built-in field types

Built-in field type	Sample input
boolean	예, 네, 그렇습니다, 그래요, 맞습니다, 맞아, 맞아요, 맞는데요, 아니오, 아니요, 아닌데요, 아닙니다, 아니, 아니예요, 틀립니다, 틀려요, 틀린데요, 틀렸어요
currency	만 팔천 이백 구십원 삼억 팔십만 삼백원 이천 팔백 구십 사원 삼만원 이억원
date	천 구백 구십 오년 칠월 팔일 이천 십년 팔월 이십 사일 이천년 일월 일일
digits	일 이 삼 사 오 하나 둘 삼 사 오 팔 하나 넷 구 구 팔 칠 오 사 일 구
number	마이너스 팔십 오 점 구 사 일 플러스 영 점 이 사 음수 팔천 구백 이십 사 양수 구만 팔 점 오 공 이
phone	공 이 삼 칠 팔 이 국에 오 육 팔 사 번 공 삼 삼 삼 팔 일 에 사천 백 오십 사 공 일 육 오백 팔십 사 에 팔 사 오 하나 번 공 일 일 구 삼 이 사 오천 번

Table 50. Sample input for Korean built-in field types (continued)

Built-in field type	Sample input
time	지금 밤 열시 십오분 오전 다섯시 팔분 자정 정오 한 오후 세시 이십삼시 삼십분 세시 반 쯤에

SSML elements and attributes

Table 51 shows the limitations on the use of SSML elements and attributes for Korean.

Table 51. Limitations for Korean SSML elements

Element	Implementation details
<emphasis>	This element is not supported.
<prosody>	The **pitch**, **range** and **rate** attributes are not supported.

Appendix F. Simplified Chinese

This appendix contains information that is specific to Simplified Chinese. If you are developing Simplified-Chinese voice applications, use the information in this appendix instead of the equivalent US English information in the corresponding sections of this book.

- "Built-in field types and grammars"
- "Predefined events" on page 174
- "Built-in commands" on page 174
- "Specifying character encoding" on page 175
- "Testing built-in field types" on page 175
- "SSML elements and attributes" on page 176

Built-in field types and grammars

Table 52 shows the built-in field types and grammars for Simplified Chinese.

Table 52. Simplified Chinese built-in types

Element	Implementation details
boolean	Users can say positive responses such as 是，好，可以 , or negative responses such as 否，不，不行 . Users can also provide DTMF input: **1** is yes, and **2** is no. The return value sent is a boolean "true" for a positive response or "false" for a negative response.
currency	Users can say currency values from 0 to 999,999,999.99 including common constructs such as 七元九角 . Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a decimal point), and can terminate DTMF entry using the # key. The return value sent is a string in the format *UUUddddddddd.cc*, where UUU is a currency indicator; in Simplified Chinese version, "RMB" will be used. If the field name is subsequently used in a value attribute within a prompt, the TTS engine will speak the currency value.

Table 52. Simplified Chinese built-in types (continued)

Element	Implementation details
date	Users can say a date using years, months and days, as well as the words 昨天 , 今天 , and 明天 . Common constructs such as 二零零一年三月十四日 or 一千九百九十九年十二月三十日 are supported. Users can also provide DTMF input in the from *yyyymmdd*. **Note:** The date grammar does not perform leap year calculations; 二月二十九日 is accepted as a valid date regardless of the year. If desired, your application or servlet can perform the required calculations. The return value sent is a string in the format *yyyymmdd*, with the VoiceXML browser returning a **?** in any positions omitted in spoken input. If the value is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:date**, then it is spoken as a date appropriate to this language. For example, the TTS engine speaks 二零零零年一月一日 .
digits	Users can say numeric integer values as individual digits (0 through 9). For example, a user could say 123456 as 一二三四五六 . Users can also provide DTMF input using the numbers 0 through 9, and must terminate DTMF entry using the # key. The return value sent is a string of one or more digits. If the result is subsequently used in **<say-as>** with the **interpret-as** value **vxml:digits**, it will be spoken as a sequence of digits appropriate to the current language. In the above example, the TTS engine speaks 123456 as 一二三四五六 . **Note:** Use this type instead of the number type if you require very high recognition accuracy for your numeric input.

Table 52. Simplified Chinese built-in types (continued)

Element	Implementation details
number	User can say natural numbers (that is, positive and negative integers and decimals) from 0 through 999,999,999,999 as well as the word 点 (to indicate a dot) and 负 (to indicate a negative number). User can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a dot), and must terminate DTMF entry using the # key. **Note:** Only positive numbers can be entered using DTMF. The return value sent is a string of one or more digits, 0 through 9, with a decimal point and a + or - sign as applicable .If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:type**, where type is the type number you want to specify, then it is spoken as that type number appropriate to this language. In the above example, the TTS engine speaks 123456 as 十二万三千四百五十六 . Use **<say-as interpret-as="vxml:digit">** to have the number said as a string of digits.
phone	Users can say a telephone number including the optional word 转 or 分机 . Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to represent the word "extension"), and must terminated DTMF entry using the # key. The return value sent is a string of digits which includes an **x** if an extension was specified. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:phone**, then it is spoken as a phone number appropriate to this language. **Note:** For tips on minimizing recognition errors that are due to user parses during input, see "Using the built-in phone grammar" on page 62.

Table 52. Simplified Chinese built-in types (continued)

Element	Implementation details
time	Users can say a time of day using hours and minutes in either 12- or 24-hour format, as well as the word **当前时间** . Users can also provide DTMF input using the numbers 0 through 9. The return value sent is a string in the format *hhmmx*, where x is **a** for a.m, **p** for p.m, **h** for 24 hour format, or **?** if unspecified or ambiguous; for DTMF input, the return value will always be **h** or **?**, since there is no mechanism for specifying a.m.or p.m. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:time**, then it is spoken as a time appropriate to this language. For example, the TTS engine speaks **二十二点三十五分** .

Predefined events

Table 53 shows the Simplified Chinese default event-handler messages for the error, help, and nomatch events.

Table 53. Simplified Chinese predefined events and event-handler messages

Event	Default event-handler message
error.badfetch **error.noauthorization** **error.semantic** **error.unsupported.** *element*	**对不起，由于发生错误，系统将退出** and then exit.
help	**对不起，没有合适的帮助** and then reprompts the user. For guidelines on choosing a scheme for your own help messages, see “Choosing help mode or self-revealing help” on page 40.
nomatch	**对不起，我不理解** and then reprompts the use; if you adhere to the guidelines for self-revealing help, this event can use the same messages as the help event. See “Implementing self-revealing help” on page 42.

Built-in commands

Table 54 on page 175 shows the built-in VoiceXML browser commands for Simplified Chinese:

Table 54. Simplified Chinese built-in VoiceXML browser commands

Grammar name	Valid user utterances	VoiceXML browser response
Quiet/Cancel	[请]保持安静 | [请]取消	When barge-in is enabled, stops the current spoken output and waits for further instructions from user.
Help	[请]帮助	Plays the help event-handler message.

Specifying character encoding

The default character encoding for XML documents is utf8, which has 7-bit ASCII as a proper subset. Character codes in VoiceXML documents that are greater than 127 will therefore be interpreted as the first byte of a multi-byte utf8 sequence. You can force the XML parser to use another codepage by specifying the desired encoding as the very first thing in the file:

```
<?xml version="1.0" encoding="GBK"?>
```

Testing built-in field types

Table 55 provides examples of the types of input you might specify when testing a Simplified Chinese voice application that uses the built-in field types.

Table 55. Sample input for Simplified Chinese built-in field types

Built-in field type	Sample input
boolean	是，对，正确，好的，是的 不，否，错，不是，错误，不对
currency	三十六元，六千零四元八角五分，七毛，三分， 五块七，三十四点九二元
date	二零零一年五月十日，两千零一年七月二日 一九九零年十一月，二月十九日，十二月一号，三十日 昨天，今天，明天
digits	零，一，二，三，四，五，六，七，八，九
number	三万六千零五十五，负二十一，正九千 六十七万七千零九点一二，零点九九

Table 55. Sample input for Simplified Chinese built-in field types (continued)

Built-in field type	Sample input
phone	一一四，九五九二八，六二九八六六七七 六二九八六六七七转六六六 零一零六二九八六六七七 一三八零零一零零五零零
time	六点十二分，上午十点，二十二时四十七分 差一刻六点，下午五点半 当前时间

SSML elements and attributes

Table 56 shows the notes about SSML elements and attributes in Simplified Chinese.

Table 56. Limitations for Simplified Chinese SSML elements

Element	Implementation details
<emphasis>	This element is not supported.
<phoneme>	The IPA alphabet is not supported. The IBM alphabet is supported. The IBM alphabet used in SSML refers to the phonology used by IBM TTS. The following example shows the Simplified Chinese phonetic pronunciation of “ 计算机 ” using the IBM TTS phonetic alphabet: <phoneme alphabet=″ibm″ ph=″+.ji4.suan4.ji1″> 计算机 </phoneme> For more information on IBM SPRs, see the *IBM Text-To-Speech SSML Programming Guide*.
<prosody>	The **pitch**, **range** and **rate** attributes are not supported.
<say-as>	The **details=″punctuation″** attribute value is not supported.

Appendix G. UK English

This appendix contains information that is specific to UK English. If you are developing voice applications in UK English, use the information in this appendix instead of the equivalent US English information in the corresponding sections of this book.

- "Built-in field types and grammars"
- "Testing built-in field types" on page 183

Built-in field types and grammars

Table 57 shows the built-in field types and grammars for UK English.

Table 57. UK English built-in types. Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
boolean	Users can say one of the positive responses **yes**, **true**, **positive**, **right**, **OK**, **sure**, **affirmative**, **check**, and **correct**, or one of the negative responses **no**, **false**, **incorrect**, **negative**, **not**, and **wrong**. Variations on "yes" (**yep** and **yeah**) and "no" (**nope**) may also be used. Users can also provide DTMF input: **1** is yes, and **2** is no. The return value sent is a boolean "true" for a positive response or "false" for a negative response. If the field name is subsequently used in a **value** attribute within a prompt, the TTS engine will speak **yes** or **no**.

Table 57. UK English built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
currency	Users can say currency values as a combination of a pounds component, such as "**twenty seven pounds**" and a pence component, such as "**fifty pence**." Users can also say one of these components without the other. If both components are present, they can be either not separated (as in "**twenty pounds seventy**"), or separated by the word "**and**" (as in "**thirty pounds and forty pence**"). A pounds component may be in any of the formats: "**no pounds**," "**zero pounds**," "**nought pounds**," "**one pound**," "m **pounds**" (where 2 <= m <= 999,999,999) and "m" (where 1 <= m <= 999,999,999). A pence component may be in any of the formats: "**no pence**," "**nought pence**," "**zero pence**," "**a penny**," "**one penny**," "n **pence**" (where 1 <= n <= 99). If both components are present, the pence component can also have the format "n" (where 0 <= n <= 99). Note that if users just say an integer value "m" (where 1 <= m <= 999,999,999), this will be interpreted as a number of pounds. Note also that users can also speak two numerical values, where the first is between 0 and 999,999,999 and the second is between 0 and 99. This will be interpreted as a number of pounds followed by a number of pence. Also, "**euro**," "**euros**," "**cent**," and "**cents**" may be used as currency types in place of "**pound**," "**pounds**," "**penny**," and "**pence**" in the above description. Similarly, "**dollar**," "**dollars**," "**cent**," and "**cents**" may be used. Note that "**pound**" can be used instead of "**pounds**" (as in "**five pound ten**") and that both "**euro**" and "**euros**" can be used as the plural of "**euro**." Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a decimal point), and must terminate DTMF entry using the # key. The return value sent is a string in the format *UUUddddddddd.cc* where *UUU* is a currency indicator (GBP, EUR, USD). If the field name is subsequently used in a **value** attribute within a prompt, the TTS engine will speak the currency value. For example, the TTS engine speaks "GBP9.30" as "**nine point three zero pounds**."

Table 57. UK English built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
date	Users can say a date using days, months and years, as well as the words **yesterday**, **today**, and **tomorrow**. The year must be between 1900 and 2099. Users can speak years from 1901 to 1999 as "**nineteen** <2-digit number>" (such as "**nineteen oh seven**"). Users can speak years from 2001 to 2099 as "**two thousand and** <1-digit number>" (such as "**two thousand and four**"), "**two thousand and** <2-digit number>" (such as "**two thousand and eighty five**"), or as "**twenty** <2-digit number>" (such as "**twenty forty six**"). (Here, 2-digit numbers can have "oh" as the first digit). The following common constructs for dates are supported: "**March the third two thousand and one**," "**March third two thousand and one**," "**the third of March two thousand and one**," and "**third of March two thousand and one**." Users can also specify a 2-digit year (such as "**fifty five**"). A 2-digit year will be interpreted as being between 1911 and 2010. Users can also say the day and month without specifying a year. This will be interpreted as a date in the current year. Any of these constructs can be preceded by a day of the week, such as "**Thursday the Third of March two thousand and one**." However, the day will be ignored. Thus the specified date will be accepted, whether or not it falls on the specified day. Users can also provide DTMF input in the form *yyyymmdd*. **Note:** The date grammar does not perform leap year calculations. February 29th is accepted as a valid date regardless of the year. If desired, your application or servlet can perform the required calculations. The return value sent is a string in the format *yyyymmdd*, with the VoiceXML browser returning a ? in any positions omitted in spoken input. If the value is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:date**, then it is spoken as a date appropriate to this language. For example, the TTS engine speaks "20030511" as "**the eleventh of May two thousand and three**."

Table 57. UK English built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
digits	Users can say non-negative integer values as strings of individual digits (0 through 9). For example, a user could say "**one two three four five six**." Users can say "0" by saying "**nought**," "**zero**," or "**oh**." Users can also say "**one**," "**two**," "**three**," "**four**," "**five**," "**six**," "**seven**," "**eight**," and "**nine**." Users can also provide DTMF input using the numbers 0 through 9, and must terminate DTMF entry using the # key. The return value sent is a string of one or more digits. If the result is subsequently used in **<say-as>** with the **interpret-as** value **vxml:digits**, it will be spoken as a sequence of digits appropriate to the current language. For example, the TTS engine speaks "123456" as "**one two three four five six**."
number	Users can say natural numbers (that is, positive and negative integers, 0, and decimals) from -999,999,999.9999 to 999,999,999.9999. Users can say the words **point** and **dot** to indicate a decimal point, **minus** to indicate a negative number, and **plus** to indicate a positive number, which is the default. Users can say "0" by saying "**nought**," "**zero**," or "**oh**." Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a comma), and must terminate DTMF entry using the # key. Only positive numbers can be entered using DTMF. The return value sent is a string of one or more digits, 0 through 9, with a decimal point and a + or - sign as applicable. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:type**, where type is the type number you want to specify, then it is spoken as that type number appropriate to this language. For example, the TTS engine speaks "123456" as "**one hundred and twenty three thousand four hundred and fifty six**." Use **<say-as interpret-as="vxml:digit">** to have the number said as a string of digits.

Table 57. UK English built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
phone	Users can say a telephone number, including the optional word **extension**. In addition to digits (1 to 9), users can use double digits (such as "49") and the words "**double**," "**treble**," and "**triple**" (before a digit). Users can also use the words "**hundred**" and "**thousand**." "**hundred**" will be interpreted as "00" and "**thousand**" will be interpreted as "000." Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to represent the word "extension"), and must terminate DTMF entry using the # key. The return value sent is a string of digits which includes an **x** if an extension was specified. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:phone**, then it is spoken as a phone number appropriate to this language. **Note:** For tips on minimizing recognition errors that are due to user pauses during input, see "Using the built-in phone grammar" on page 62.

Table 57. UK English built-in types (continued). Brackets "[]" around a keyword mean that the keyword is optional. The vertical-bar symbol "|" indicates a choice between two or more keywords.

Element	Implementation details
time	Users can say a time of day using hours and minutes in either 12- or 24-hour format, as well as the word **now**. Times can be stated in any of the following formats: **one** [**minute**] **past** n [**o'clock**] [**in the morning** | **in the afternoon** | **in the evening** | **at night**] **one** [**minute**] **to** n [**o'clock**] [**am** | **pm** | **in the morning** | **in the afternoon** | **in the evening** | **at night**] m [**minutes**] **past** n [**o'clock**] [**am** | **pm** | **in the morning** | **in the afternoon** | **in the evening** | **at night**] m [**minutes**] **to** n [**o'clock**] [**am** | **pm** | **in the morning** | **in the afternoon** | **in the evening** | **at night**] [a] **quarter past** n [**o'clock**] [**am** | **pm** | **in the morning** | **in the afternoon** | **in the evening** | **at night**] **half past** n [**o'clock**] [**am** | **pm** | **in the morning** | **in the afternoon** | **in the evening** | **at night**] [**a**] **quarter to** n [**o'clock**] [**am** | **pm** | **in the morning** | **in the afternoon** | **in the evening** | **at night**] [**twelve**] **midnight** [**twelve**] **noon** **one** [**minute**] **past noon** | **midnight** **one** [**minute**] **to noon** | **midnight** m [**minutes**] **past noon** | **midnight** m [**minutes**] **to noon** | **midnight** [**a**] **quarter past noon** | **midnight** [**a**] **quarter to noon** | **midnight** **zero** | **oh** p | r [**hundred**] **hours** **zero** | **oh** p | r q **hours** **now** where 1 <= m <= 30, 1 <= n <= 12, 1 <=p <=9, 10 <= r <= 23 and 1<= q <= 59. Note that when m or m is "0" or is a 2-digit number beginning with "0" (such as "**11:07 am**"), the "0" should be pronounced "**oh**" or "**zero**." Also, users can precede any of these times with "**about**," "**approximately**," "**around**," "**at**," "**exactly**," "**nearly**," "**just after**," or "**just before**." This will be ignored, and the exact time will be accepted. Users can also provide DTMF input using the numbers 0 through 9. The return value sent is a string in the format *hhmmx*, where *x* is **a** for AM, **p** for PM, **h** for 24 hour format or **?** if unspecified or ambiguous. For DTMF input, the return value will always be **h** or **?**, since there is no mechanism for specifying AM or PM. If the field is subsequently spoken in **<say-as>** with the **interpret-as** value **vxml:time**, then it is spoken as a time appropriate to this language.

Testing built-in field types

Table 58 provides examples of the types of input you might specify when testing a UK English voice application that uses the built-in field types.

Table 58. Sample input for UK English built-in field types

Built-in field type	Sample input
boolean	yes, true, positive, right, ok, sure, affirmative, check, yep, correct, no, false, negative, wrong, not, nope, incorrect
currency	three twenty five sixteen pounds and fifty seven pence ten euros nine million two hundred thousand pounds
date	may fifth march the thirty first of december two thousand yesterday today tomorrow
digits	zero, oh, nought, one, two, three, four , five, six, seven, eight, nine
number	ten million five hundred thousand and fifty three minus one point five plus one point five point seven
phone	seven three five eight four nine oh four nine six two seven oh six five hundred oh nine one four nine four five eight oh six two extension seven two three extension six one six three
time	one o'clock five past one three fifteen seven thirty half past eight oh four hundred hours sixteen fifty twelve noon midnight

Notices

This information was developed for products and services offered in the United States.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information about the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department T01B
3039 Cornwallis Road
Research Triangle Park, NC 27709-2195
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

The products offered by IBM in this publication are platforms or middleware that can provide the capability for you to add, design, or create applications to take advantage of the products. Notwithstanding the terms of any other agreements You have with International Business Machines Corporation, or any of its related or affiliated companies, IBM shall not be liable to You for any and all claims of patent infringement, including inducement or contributory infringement, or any claims for indemnification for such claims brought against the products or based on the combination, use or operation of the products with software or hardware. You also should be aware that it may be necessary for You to obtain a patent license from one or more third parties, including, but not limited to, Ronald A. Katz or Ronald A. Katz Technology Licensing, L.P. (commonly referred to as RAKTL), before using certain applications designed and built to run on the IBM products listed in this publication.

COPYRIGHT LICENSE

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

AIX, IBM and WebSphere are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Dialogic and Intel are registered trademarks of Intel Corporation in the United States, other countries, or both.

Cisco is a registered trademark of Cisco Systems, Inc., in the United States, other countries, or both.

Sun, Java and Java-based marks are registered trademarks of Sun Microsystems, Inc., in the United States, other countries, or both.

For more information on CallPath products please contact Genesys Telecommunications Laboratories, Inc. On the World Wide Web, go to the CallPath Framework part of the Genesys Web site (http://www.genesyslabs.com).

Other company, product and service names may be trademarks or service marks of others.

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. These are the major accessibility features in WebSphere Voice Server:

- You can use screen-reader software and a digital speech synthesizer to hear the HTML version of the WebSphere Voice Server VoiceXML Programmer's Guide.
- You can operate many features using the keyboard instead of the mouse.

Other attributions

Some of the artwork in this publication incorporates clip art from CorelDRAW Version 9.0.

Glossary

This glossary defines terms and abbreviations used in this publication. If you do not find the term you are looking for here, refer to The IBM Dictionary of Computing, SC20-1699, New York: McGraw-Hill, copyright 1994 by International Business Machines Corporation. Copies may be purchased from McGraw-Hill or in bookstores.

active grammar. A speech grammar that the speech recognition engine is currently listening for. One or more grammars can be active at any time, and the content of the active grammar(s) defines the user utterances that are valid in a given context.

A-law. The compression and expansion algorithm used primarily in Europe when converting from analog to digital speech data.

ANI. Automatic Number Identification. A service offered by commercial telephone networks, which provides the directory billing number associated with a calling party. This is the originating telephone number of the incoming call, which can be used for call set-up or passed by the switch to the Voice Server, which can then use it to retrieve data from business databases. Often used as a synonym for calling number.

application. A set of related VoiceXML documents that share the same application root document.

application root document. A document that is loaded when any documents in its application are loaded, and unloaded whenever the dialog transitions to a document in a different application. The application root document may contain grammars that can be active for the duration of the application, and variables that can be accessed by any document in the application.

ASP. Microsoft Active Server Pages. One of many server-side mechanisms for generating dynamic Web content by transmitting data between an HTTP server and an external program. ASPs can be written in various scripting languages, including VBScript (based on Microsoft Visual Basic), JScript (based on JavaScript), and PerlScript (based on Perl).

bail out. The termination of a sequence of self-revealing help prompts, if the user repeatedly fails to provide an appropriate response. This is generally a transfer to a human operator (if available), or an exit.

barge-in. A feature of full-duplex environments that allows the user to interrupt computer speech output (audio file and text-to-speech). See also "full-duplex."

CGI. Common Gateway Interface. One of many server-side mechanisms for generating dynamic Web content by transmitting data between an HTTP server and an external program. CGI scripts are typically written in Perl, although they can be written in other programming languages.

called number. The number dialed by callers to reach the voice application, or the number dialed when making a call. Often used as a synonym for DNIS.

calling number. The number from which a call is made. Often used as a synonym for ANI

continuous speech recognition. The WebSphere Voice Server supports "continuous speech recognition," in which users can speak a string of words at a natural pace, without the need to pause after each word. Contrast with "discrete speech recognition."

cookie. Information that a Web server stores on a user's computer when the user browses a particular Web site. This information helps the

Web server track such things as user preferences and data that the user may submit while browsing the site. For example, a cookie may include information about the purchases that the user makes (if the Web site is a shopping site). The use of cookies enables a Web site to become more interactive with its users, especially on future visits.

cut-thru word recognition. See "barge-in."

data prompts. Prompts where the user must supply information to fill in the field of a form. Contrast with "verbatim prompts."

dialog. The main building block for interaction between the user and the application. VoiceXML supports two types of dialogs: "form" and "menu."

discrete speech recognition. Users must pause briefly after speaking each word, to allow the system to process and recognize the input. Contrast with "continuous speech recognition."

DNIS. Dialed Number Identification Service. A service supplied by the public telephone network to identify the number actually dialed. For example, calls placed to two or more 1-800 numbers will arrive at the same call center switch. Upon arrival, DNIS tells the switch which one of the 1-800 numbers was actually dialed. DNIS can be used by the Voice Server to automatically select between several voice applications. Often used as a synonym for "called number."

DTMF. Dual Tone Multiple Frequency. The tones generated by pressing keys on a telephone's keypad.

DTMF Simulator. A GUI tool that enables you to simulate DTMF input when testing your VoiceXML application on your desktop workstation. The VoiceXML browser communicates with the DTMF Simulator to accept DTMF input, and uses that input to fill in forms or select menu items within the VoiceXML application.

echo cancellation. Technology that removes echo sounds from the input data stream before passing what's left (that is, user speech) to the speech recognition engine. In a connection environment, this is configured on the telephony hardware; if echo cancellation is poor, you may need to turn off barge-in or switch to "half-duplex."

ECMAScript. An object-oriented programming language adopted by the European Computer Manufacturer's Association as a standard for performing computations in Web applications. ECMAScript is the official client-side scripting language of VoiceXML. Refer to the ECMAScript Language Specification, available at **http://www.ecma.ch/ecma1/stand/ECMA-262.htm**.

event. The VoiceXML browser throws an event when it encounters a **<throw>** element and certain specified conditions occur. Events are caught by **<catch>** elements that can be specified within other VoiceXML elements in which the event can occur, or inherited from higher-level elements. The VoiceXML browser supports a number of predefined events and default event handlers; you can also define your own events and event handlers.

form. One of two basic types of VoiceXML dialogs. Forms allow the user to provide voice or DTMF input by responding to one or more **<field>** elements. See also menu.

full-duplex. Applications in which the user and computer can speak concurrently. Full-duplex applications use echo cancellation to subtract computer output from the incoming data to determine what was user speech. See also "barge-in." Contrast with "half-duplex."

grammar. A collection of rules that define the set of all user utterances that can be recognized by the speech recognition engine at a given point in time. The VoiceXML browser makes different grammars active at different points in the dialog, thereby controlling the set of valid utterances that the speech recognition engine is listening for. Grammars support word substitution and word repetition.

GSM. Global System for Mobile Communication. The cellular telephone network.

GUI. Graphical User Interface. A type of computer interface consisting of visual images and printed text. Users can access and manipulate information using a pointing device and keyboard. Contrast with "speech user interface."

H.323. Audio communications protocol used by WebSphere Voice Server.

half-duplex. Applications in which the user should not speak while the computer is speaking because the speech recognition engine does not receive audio when sending audio to the user. Turn-taking problems can occur when the user speaks before the computer has finished speaking; using a unique tone to indicate the end of computer output can minimize these problems by informing users when they can speak. Contrast with "full-duplex."

help mode. A technique for providing general help information explicitly, in a separate dialog. Contrast with "self-revealing help." See "Choosing help mode or self-revealing help" on page 40.

II digits. Information Indicator Digits. A telephony service that provides information about the caller's line (for example, cellular service, pay telephone, etc.).

JMF. Java Media Framework.

JSP. Java Server Pages. One of many server-side mechanisms for generating dynamic Web content by transmitting data between an HTTP server and an external program. JSPs call Java programs, which are executed by the HTTP server.

JVM. Java Virtual Machine.

Lombard speech. The tendency of people to raise their voices in noisy environments, so that they can be heard over the noise.

machine directed. A dialog in which the computer controls interactions. Grammars are only active within their own dialogs. Contrast with "mixed initiative."

menu. One of two basic types of VoiceXML dialogs. Menus allow the user to provide voice or DTMF input by selecting one menu choice. See also form.

menu flattening. A feature of natural command grammars that enables the system to parse user input and extract multiple tokens. Mixed initiative dialogs can provide similar benefits.

mixed initiative. A dialog in which either the user or the computer can initiate interactions. You can use form-level grammars to allow the user to fill in multiple fields from a single utterance, or document-level grammars to allow the form's grammars to be active in any dialog in the same VoiceXML document; if the user utterance matches an active grammar outside of the current dialog, the application transitions to that other dialog. Contrast with "machine directed."

mixed-mode applications. Applications that mix speech and DTMF input.

µ-law. The compression and expansion algorithm used in primarily in North America and Japan when converting from analog to digital speech data.

multi-modal application. An application that has both a speech and a visual interface.

natural command grammar. A complex grammar that approaches natural language understanding in its lexical and syntactic flexibility, but unambiguously specifies all acceptable user utterances. Contrast with "natural language understanding (NLU)."

natural language understanding (NLU). A statistical technique for processing natural language, using text that is representative of expected utterances to create a dictation-like model. NLU does not use grammars; instead, it uses statistical information to tag and analyze

key words in an utterance. Contrast with "natural command grammar."

out-of-grammar (OOG) utterance. The user input was not in any of the active grammars.

persistence. A property of visual user interfaces is that information is persistent; that is, information remains visible until the user moves to a new visual page or the information changes. Contrast with "transience."

phone. The actual pronunciation of a sound. Phones have a variable duration of up to several seconds. Multiple phones can be categorized as the same phoneme. For example, the vowels in the words "bean" and "bead" are classified as the same phoneme; however, if you carefully monitor the shape of your lips and the position of your tongue and jaw when saying the two words, you can see that the actual sound of the vowel is different (that is, they are different phones). Contrast with "phoneme."

phoneme. A perceived pronunciation or category or pronunciation for a distinctive sound segment of a language. A change in the phoneme changes the meaning of a word. For example, "zip" and "sip" differ by only the initial sound, but are completely different words. Contrast with "phone."

prompt. Computer spoken output, often directing the user to speak.

pronunciation. A possible phonetic representation of a word that is stored in the speech recognition engine and referenced by one or more words in a grammar. A pronunciation is a string of sounds that represents how a given word is pronounced. A word may have several pronunciations; for example, the word "tomato" may have pronunciations "toe-MAH-toe" and "toe-MAY-toe."

PSTN. Public Switched Telephone Network.

prosody. The rhythm and pitch of speech, including phrasing, meter, stress, and speech rate.

recognition. When utterances are known and accepted by the speech recognition engine. Only words, phrases, and DTMF key sequences in active grammars can be recognized.

recognition window. The period of time during which the system is listening for user input. In a full-duplex implementation, the system is always listening for input; in a half-duplex implementation or when barge-in is temporarily disabled, a recognition window occurs only when the dialog is in a state where it is ready to accept user input.

self-revealing help. A technique for providing context-sensitive help implicitly, rather than providing general help using an explicit help mode. Contrast with "help mode."

servlet. One of many server-side mechanisms for generating dynamic Web content by transmitting data between an HTTP server and an external program. Servlets are dynamically loaded Java-based programs defined by the Java Servlet API (**http://java.sun.com/products/servlet/**). Servlets run inside a JVM on a Java-enabled server.

session. A session consists of all interactions between the VoiceXML browser, the user, and the document server. The session starts when the VoiceXML browser starts, continues through dialogs and the associated document transitions, and ends when the VoiceXML browser exits.

speech browser. See "VoiceXML browser."

speech recognition. The process by which the computer decodes human speech and converts it to text.

speech recognition engine. Decodes the audio stream based on the current active grammar(s) and returns the recognition results to the VoiceXML browser, which uses the results to fill in forms or select menu choices or options.

speech user interface. A type of computer interface consisting of spoken text and other audible sounds. Users can access and manipulate information using spoken commands and DTMF. Contrast with "GUI." See "DTMF."

spoke too soon (STS) incident. A recognition error that occurs when the user in a half-duplex application begins speaking before the turn-taking tone sounds and continues speaking over the tone and into the speech recognition window.

spoke way too soon (SWTS) error. A recognition error that occurs when the user in a half-duplex application finishes speaking before the turn-taking tone sounds.

stuttering effect. When a prompt in a full-duplex application keeps playing for more than 300 ms after the user begins speaking, users may interpret this to mean that the system didn't hear their input. As a result, the users stop what they were saying and start over again. This "stuttering" type of speech makes it difficult for the speech recognition engine to correctly decipher user input.

subdialog. Roughly the equivalent of function or method calls. Subdialogs can be used to provide a disambiguation or confirmation dialog, or to create reusable dialog components.

text-to-speech (TTS) engine. Generates computer synthesized speech output from text input.

token. The smallest unit of meaningful linguistic input. A simple grammar processes one token at a time; contrast with "menu flattening," "natural command grammar," and "natural language understanding (NLU)."

transience. A property of speech user interfaces is that information is transient; that is, information is presented sequentially and is quickly replaced by subsequent information. This places a greater mental burden on the user, who must remember more information than they need to when using a visual interface. Contrast with "persistence."

turn-taking. The process of alternating who is performing the next action: the user or the computer.

URI. Universal Resource Identifier. The address of a resource on the World Wide Web. For example: http://www.ibm.com.

URL. Universal Resource Locator. A subset of URI.

User to User Information. ISDN service that provides call set-up information about the calling party.

utterance. Any stream of speech, DTMF input, or extraneous noise between two periods of silence.

verbatim prompts. Menu choices that the user can select by repeating what the system said. Contrast with "data prompts."

voice application. An application that accepts spoken input and responds with spoken output.

VoiceXML. Voice eXtensible Markup Language. An XML-based markup language for creating distributed voice applications. Refer to the VoiceXML Forum Web site at http://www.voicexml.org.

VoiceXML browser. The "interpreter context" as defined in VoiceXML 2.0. The VoiceXML browser fetches and processes VoiceXML documents and manages the dialog between the application and the user.

"Wizard of Oz" testing. A testing technique that allows you to use a prototype paper script and two people (a user and a human "wizard" who plays the role of the computer system) to test the dialog and task flow before coding your application. See "Prototype phase ("Wizard of Oz" testing)" on page 16.

XML. eXtensible Markup Language. A standard metalanguage for defining markup languages. XML is being developed under the auspices of the World Wide Web Consortium (W3C).

Index

Special characters

<assign> element 82
<audio> element 82
<block> element 82
<break> element 82
<catch> element 82
<choice> element 82
<clear> element 83
<data> element 83
<disconnect> element 83
 element 83
<dtmf> element 83
<else> element 83
<elseif> element 83
<emp> element 83
<enumerate> element 84
<error> element 84
<exit> element 84
<field> element 84
<filled> element 84
<foreach> element 84
<form> element 84
<goto> element 84
<grammar> element 84
<help> element 84
<ibmlexicon> element 84
<ibmvoice> element 85
<if> element 85
<initial> element 85
<link> element 85
<log> element 85
<mark> element 85
<menu> element 85
<meta> element 85
<metadata> element 85
<noinput> element 85
<nomatch> element 85
<object> element 86
  sample 129
<option> element 86
<param> element 86
<prompt> element 86
<property> element 87
<pros> element 88
<record> element 88, 95
<reprompt> element 88
<return> element 88
<sayas> element 88
<script> element 88
<subdialog> element 89
<submit> element 89
<throw> element 89
<transfer> element 89
<value> element 89, 95
<var> element 89, 100
<vxml> element 89, 90

A

a-law 82, 95
accuracy
  alternative pronunciation, creating an 90
advantages of VoiceXML 4
ANI 101, 117
  sample 118
application development
  audio file 95
  design consistency 43
  dialogs 76
  duplex 21
  events 5
  fields in a form 76
  form 76
  grammar 59, 104
  information analysis 20
  menus 77
  prototyping 16
  publications xii
  queries 2
  transactions 3
  user interface 20
  voice application
    advantages of VoiceXML 4
    catch 82
    conversational access 2
    definition 1
    deploying 5
    design consistency 43
    duplex 21
    dynamic and static documents 4
    else statement 83
    elseif statement 83
    error 84
    events 5
    form 76
    forms 77
    goto statement 84
    grammar 59
    help 84
    if statement 85

application development *(continued)*
voice application *(continued)*
information analysis 20
interactions with users 6
link 85
noinput 85
nomatch 85
return statement 88
samples 129
throw 89
types 2
user interface 20
variable, assigning a 102
variable, assigning a value to 82
variable, clearing a 83
variable, declaring a 89, 100
variable, referencing a 102
audio 95
DTMF key
recording, terminating 103
during of recording 103
in queries 2
in transactions 3
playing back 95
prompt
creating 25
playing 86, 88
prosody 88
recommended format 82
size of recording 103
tones, using 27
user input, recording 88, 95
VoiceXML element 82
Automatic Number Identification 101, 117
sample 118

B

barge-in 22
user interface 21
block 77
boolean 92
British English 177

C

caching documents 96
call transfer 89, 119
caller
application, prototyping the 16
distorted speaking 23
identification 117
interactions with VoiceXML application 6
recording 88, 95
requirements 14
tasks to be performed 14
transferring 89
Canadian French 139
Chinese, Simplified 171
completetimeout 116
Computer Telephony Integration 89
confidence-level processing 127
content of a VoiceXML document
dynamic 81
static 81
conventions used in this book xv
currency 93

D

data flow 78
dates 93
deployment platforms 6
design, user interface 9
audio tones 27
breakdowns 18
commands 34
consistency factors 43
design phase 13
duplex, choosing 21
error recovery 64
grammars 59
grammars, designing 29
help, designing 40
human intervention 39
information analysis 20
interface, selecting an 20
Lombard speech 23
prompt style 31
prompts, designing 24
prototyping 16
publications xii
recognition problems, identifying 18
speaking too soon 65
speech vs DTMF 32
test phase 18
user requirements 14
user tasks 14
words, spelling 55
developing applications
audio file 95
design consistency 43
dialogs 76
duplex 21
events 5
fields in a form 76
form 76
grammar 59, 104
information analysis 20
menus 77
prototyping 16
publications xii
queries 2

developing applications *(continued)*
transactions 3
user interface 20
voice application
advantages of VoiceXML 4
catch 82
conversational access 2
definition 1
deploying 5
design consistency 43
duplex 21
dynamic and static documents 4
else statement 83
elseif statement 83
error 84
events 5
form 76
forms 77
goto statement 84
grammar 59
help 84
if statement 85
information analysis 20
interactions with users 6
link 85
noinput 85
nomatch 85
return statement 88
samples 129
throw 89
types 2
user interface 20
variable, assigning a 102
variable, assigning a value to 82
variable, clearing a 83
variable, declaring a 89, 100
variable, referencing a 102
Dialed Number Identification Service 101, 117, 118
sample 119
dialogs 76
block 77
duration of call 104
foreach element 84
form element 84
goto element 84
grammar accessible to 110
how to group 123
information prompt for form 85
menu, defining a 85
subdialog 77, 79
supported by VoiceXML 3
digits 93
DNIS 101, 117, 118
sample 119
document
dynamic content 81
fetching and caching 96, 124
grammar accessible to 110
linking 85
properties 85
State Table, Voice XML 125
static content 81
variables 89
DTMF input 2
as input mode 103
boolean input 92
currency input 93
date input 93
digit input 93
grammar 83, 107
in a menu 78
key to terminate recording 103
number input 94
telephone number input 94
time of day input 94
duration of call 104
dynamic grammar 109
dynamic VoiceXML documents 4, 81

E

ECMAScript 88
semantic error 97
URL 99
editor 5
elements of VoiceXML
assign 82
audio 82
block 82
break 82
catch 82
choice 82
clear 83
data 83
disconnect 83
div 83
dtmf 83
else 83
elseif 83
emp 83
enumerate 84
error 84
exit 84
field 84
filled 84
foreach 84
form 84
goto 84
grammar 84
help 84

elements of VoiceXML *(continued)*
ibmlexicon 84
ibmvoice 85
if 85
initial 85
link 85
log 85
mark 85
menu 85
meta 85
metadata 85
noinput 85
nomatch 85
object 86
sample 129
option 86
param 86
prompt 86
property 87
pros 88
record 88, 95
reprompt 88
return 88
sayas 88
script 88
subdialog 89
submit 89
throw 89
transfer 89
value 89, 95
var 89, 100
anonymous variable 100
application variable 100
dialog variable 100
document variable 100
session variable 101
vxml 89
word 90
emphasis for TTS output 83
end user
application, prototyping the 16
distorted speaking 23
identification 117
interactions with VoiceXML application 6
recording 88, 95
requirements 14
tasks to be performed 14
transferring 89
errors
error.badfetch event 97
error.semantic event 97
error.unsupported event 97
error.unsupported.format event 97
event 84
speaking too soon 65
errors *(continued)*
spelling 55
subdialogs 79
escape sequence 102
event 5, 96
cancel 97
catch 82
error 84
error.badfetch 97
error.semantic 97
error.unsupported 97
error.unsupported.format 97
exit 97
help 84, 97
link 85
noinput 85, 98
nomatch 85, 98
recurring 99
telephone.disconnect.hangup 98
telephone.disconnect.transfer 98
throw 89
examples
VoiceXML files 129

F

fetching and caching documents 96, 124
field 76
action to perform when filled 84
boolean 92
currency 93
date 93
defining in a form 84
digit 93
execution of 102
number 94
option 86
telephone number 94
time of day 94
file
audio, playing 82
containing grammar 106
fetching and caching 96
grammar 105
sample VoiceXML 129
form 76
block 77
boolean fields 92
currency fields 93
date fields 93
defining for a dialog 84
digit fields 93
field, defining a 84
grammar accessible to 110
information prompt 85
number fields 94

form *(continued)*
prompt, playing a 88
sample 80
subdialog 77, 79
telephone number fields 94
time of day fields 94
types 77
form item 76
<block> element 82
action to perform on a field, defining 84
boolean field 92
currency field 93
date field 93
digit field 93
execution of 102
input field, defining 84
number field 94
option 86
telephone number field 94
time of day field 94
French, Canadian 139

G

German 149
glossary of terms 189
grammar 84, 104
boolean field types 92
currency field types 93
date field types 93
definition 104
designing 59
digit field types 93
disabling 110
DTMF 83, 107
dynamic 109
element for defining 84
external and internal 106
header 105
hierarch 110
location 124
nomatch event 98
number field types 94
rules 105
scope 109
sounds-like spelling 114
static 107
syntax 105
telephone number field types 94
time of day field types 94
usability 29

H

hint and tips 123
HTTP xi
http-equiv attributes 85

I

IBM speech recognition engine
confidence level 103
shadow variables 102
timeout properties 115
utterance recognized 103
incompletetimeout 115
input field
action to perform, defining 84
defining in a form 84
interactive voice response 4
interface design, user 9
audio tones 27
breakdowns 18
commands 34
consistency factors 43
design phase 13
duplex, choosing 21
error recovery 64
grammars 59
grammars, designing 29
help, designing 40
human intervention 39
information analysis 20
interface, selecting an 20
Lombard speech 23
prompt style 31
prompts, designing 24
prototyping 16
publications xii
recognition problems, identifying 18
speaking too soon 65
speech vs DTMF 32
test phase 18
user requirements 14
user tasks 14
words, spelling 55
international phonetic alphabet xii
Invoking a State Table using Voice XML 125
IVR 4

J

Japanese 157
Java
JavaServer pages xii
servlet xii
Java applications 133

K

Korean 163

L

Lombard speech 23

M

markup language 3
  VoiceXML 75
menu 77
  choices, playing 84
  defining 85
  design considerations 48
  grammar accessible to 110
  input 78
  item, defining an 82
  sample 80
menu item 77
  <choice> element 82
  choices, playing 84
  design considerations 48
misrecognition
  spelling 55
mu-law 82, 95

N

notices 185
numbers 94

P

phonetic alphabet, international xii
program number ii
programming
  audio file 95
  design consistency 43
  dialogs 76
  duplex 21
  events 5
  fields in a form 76
  form 76
  grammar 59, 104
  information analysis 20
  menus 77
  prototyping 16
  publications xii
  queries 2
  transactions 3
  user interface 20
  voice application
    advantages of VoiceXML 4
    catch 82
    conversational access 2
    definition 1
    deploying 5
    design consistency 43
    duplex 21
    dynamic and static documents 4
    else statement 83
    elseif statement 83
    error 84
    events 5
programming *(continued)*
  voice application *(continued)*
    form 76
    forms 77
    goto statement 84
    grammar 59
    help 84
    if statement 85
    information analysis 20
    interactions with users 6
    link 85
    noinput 85
    nomatch 85
    return statement 88
    samples 129
    throw 89
    types 2
    user interface 20
    variable, assigning a 102
    variable, assigning a value to 82
    variable, clearing a 83
    variable, declaring a 89, 100
    variable, referencing a 102
prompt
  designing 24
  length 51
  playing 86, 88
  prosody 88
  style 31
  variable, embedding a 89
pronunciation
  overriding 90
  TTS output, controlling 88, 114
  words in boolean grammar 92
  words in currency grammar 93
  words in date grammar 93
  words in digits grammar 93
  words in number grammar 94
  words in telephone number grammar 94
  words in time of day grammar 94
proxy server 127
publications
  programming xii
  standards and specifications xi
  user interface design xii
publications, related xi

Q

queries 2

R

recorded audio 95
  during speech recognition 95, 100, 103
  in queries 2
recording user input 88, 95

related publications xi
rules of grammar 105

S

samples
  Automatic Number Identification 118
  Dialed Number Identification Service 119
  DTMF and voice input into a menu 78
  menu and form 80
  n-best 134
  object element 129
  VoiceXML files 129
servlet xii
shadow variables 102
Simplified Chinese 171
sounds-like spelling 114
specifications xi
  HTML 85
  HTTP xi, 85
  phonetic alphabet xii
  Unicode xii
  VoiceXML xi, 3
speech application
  advantages of VoiceXML 4
  audio file 95
  conversational access 2
  definition 1
  deploying 5
  design consistency 43
  dialogs 76
  duplex 21
  dynamic and static documents 4
  else statement 83
  elseif statement 83
  events 5
    cancel 97
    catch 82
    error 84
    error.badfetch 97
    error.semantic 97
    error.unsupported 97
    error.unsupported.format 97
    exit 97
    help 84, 97
    link 85
    noinput 85, 98
    nomatch 85, 98
    recurring 99
    telephone.disconnect.hangup 98
    telephone.disconnect.transfer 98
    throw 89
  fields in a form 76
  form 76
  goto statement 84
  grammar 59, 104
speech application *(continued)*
  if statement 85
  information analysis 20
  input mode 103
  interactions with users 6
  menus 77
  prototyping 16
  queries 2
  return statement 88
  samples 129
  transactions 3
  types 2
  user interface 20
  variable, assigning a 102
  variable, assigning a value to 82
  variable, clearing a 83
  variable, declaring a 89, 100
  variable, referencing a 102
speech recognition
  distorted utterances 23
  problems, identifying 18
  spelling 55
  string returned from engine 103
speech, synthesized
  emphasis for output 83
  in queries 2
  in transactions 3
  menu item 84
  pause in output, inserting 82
  prompt, playing a 86, 88
  prompts, designing 26
  pronunciation
    controlling 88
    overriding 90
    sounds-like 114
  prosody 88
  type of output text 83
spelling of a word
  common errors 55
standards xi
  HTML 85
  HTTP xi, 85
  phonetic alphabet xii
  Unicode xii
  VoiceXML xi, 3
State tables 133
static grammar 107
static VoiceXML documents 4, 81
stuttering effect 23
subdialogs 77, 79
supported environments 6
  Automatic Number Identification 117
  Dialed Number Identification Service 118
  transfer 89

synthesized speech
emphasis for output 83
in queries 2
in transactions 3
menu item 84
pause in output, inserting 82
prompt, playing a 86, 88
prompts, designing 26
pronunciation
controlling 88
overriding 90
sounds-like 114
prosody 88
type of output text 83

T

telephone
duration of call 104
numbers 94
telephone.disconnect.hangup event 98
telephone.disconnect.transfer event 98
telephone numbers 94
telephony 6
Automatic Number Identification 117
Dialed Number Identification Service 118
transfer 89
terms, glossary of 189
text-to-speech
emphasis for output 83
in queries 2
in transactions 3
menu item 84
pause in output, inserting 82
prompt, playing a 86, 88
prompts, designing 26
pronunciation
controlling 88
overriding 90
sounds-like 114
prosody 88
type of output text 83
time of day 94
timeout properties 115
trademarks used in this book 187
transactions 3
TTS
emphasis for output 83
in queries 2
in transactions 3
menu item 84
pause in output, inserting 82
prompt, playing a 86, 88
prompts, designing 26
pronunciation
controlling 88
TTS *(continued)*
pronunciation *(continued)*
overriding 90
sounds-like 114
prosody 88
type of output text 83

U

UK English 177
Unicode xii
URI 4
external references 104
URL
Active Server Pages Resource Index xiii
CGI scripts in Perl xii
ECMAScript xii, 99
HTML 85
HTTP 85
HTTP Specification xi
international phonetic alphabet xii
Java Servlet xii
JavaServer pages xii
VoiceXML specification xi
user
application, prototyping the 16
distorted speaking 23
identification 117
interactions with VoiceXML application 6
recording 88, 95
requirements 14
tasks to be performed 14
transferring 89
user interface design 9
audio tones 27
breakdowns 18
commands 34
consistency factors 43
design phase 13
duplex, choosing 21
error recovery 64
grammars 59
grammars, designing 29
help, designing 40
human intervention 39
information analysis 20
interface, selecting an 20
Lombard speech 23
prompt style 31
prompts, designing 24
prototyping 16
publications xii
recognition problems, identifying 18
speaking too soon 65
speech vs DTMF 32
test phase 18

user interface design *(continued)*
user requirements 14
user tasks 14
words, spelling 55

V

variable
anonymous 100
application 100
assigning 102
clearing 83
declaring 89, 100
dialog 100
document 100
document server, submitting to a 89
in a prompt 89
referencing 102
session 101
shadow 102
value, assigning a 82
voice application
advantages of VoiceXML 4
audio file 95
conversational access 2
definition 1
deploying 5
design consistency 43
dialogs 76
duplex 21
dynamic and static documents 4
else statement 83
elseif statement 83
events 5
cancel 97
catch 82
error 84
error.badfetch 97
error.semantic 97
error.unsupported 97
error.unsupported.format 97
exit 97
help 84, 97
link 85
noinput 85, 98
nomatch 85, 98
recurring 99
telephone.disconnect.hangup 98
telephone.disconnect.transfer 98
throw 89
fields in a form 76
form 76
goto statement 84
grammar 59, 104
if statement 85
information analysis 20
voice application *(continued)*
input mode 103
interactions with users 6
menus 77
prototyping 16
queries 2
return statement 88
samples 129
transactions 3
types 2
user interface 20
variable, assigning a 102
variable, assigning a value to 82
variable, clearing a 83
variable, declaring a 89, 100
variable, referencing a 102
Voice Extensible Markup Language 3, 75
Voice Server
program number ii
Voice Toolkit 5, 88
VoiceXML 75
advantages 4
comment 79
data flow, managing 78
definition 3
deploying an application 5
dialogs 3
dynamic content 81
dynamic documents 4
editor 5
elements
assign 82
audio 82
block 82
break 82
catch 82
choice 82
clear 83
data 83
disconnect 83
div 83
dtmf 83
else 83
elseif 83
emp 83
enumerate 84
error 84
exit 84
field 84
filled 84
foreach 84
form 84
goto 84
grammar 84
help 84

VoiceXML *(continued)*
elements *(continued)*
ibmlexicon 84
ibmvoice 85
if 85
initial 85
link 85
log 85
mark 85
menu 85
meta 85
metadata 85
noinput 85
nomatch 85
object 86, 129
option 86
param 86
prompt 86
property 87
pros 88
record 88, 95
reprompt 88
return 88
sayas 88
script 88
subdialog 89
submit 89
throw 89
transfer 89
value 89, 95
var 89, 100
vxml 89
word 90
property
completetimeout 116
incompletetimeout 115
sample files 129
sample using menu and form 80
specification xi, 75
static content 81
static documents 4
user interactions with an application 6
VoiceXML browser
audio file 95
cancel event 97
disconnecting from a telephone session 83
document fetching and caching 96
error.badfetch event 97
error.semantic event 97
error.unsupported event 97
error.unsupported.format event 97
exit event 97
exiting a browser session 84
help event 97
noinput event 98
VoiceXML browser *(continued)*
nomatch event 98
recurring events 99
telephone.disconnect.hangup event 98
telephone.disconnect.transfer event 98
unsupported objects 86
VRBE 133

W

Web site
Active Server Pages Resource Index xiii
CGI scripts in Perl xii
ECMAScript xii, 99
HTML 85
HTTP 85
HTTP Specification xi
international phonetic alphabet xii
Java Servlet xii
JavaServer pages xii
VoiceXML specification xi
WebSphere Voice Response 6
WebSphere Voice Server
program number ii
World Wide Web Consortium 3

Program Number: 5724-F72