



Programmer Guide
PowerScribe[®] SDK

Version 3.2

Trademarks

Nuance®, the Nuance logo, Dictaphone®, and PowerScribe® are trademarks or registered trademarks of Nuance Communications, Inc. or its affiliates in the United States and/or other countries. All other trademarks referenced herein are trademarks or registered trademarks of their respective owners.

Patents

The PowerMic II product is the subject of pending U.S and foreign patent applications.

Copyright Notice

This manual is copyrighted and all rights are reserved by Nuance Communications, Inc. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of Nuance Communications, Inc., 1 Wayside Road, Burlington, MA 01803.

Copyright © 2006-2009 Nuance Communications, Inc. All rights reserved.

Disclaimer

Nuance makes no warranty, express or implied, with respect to the quality, reliability, currentness, accuracy, or freedom from error of this document or the product or products referred to herein and specifically disclaims any implied warranties, including, without limitation, any implied warranty of merchantability, fitness for any particular purpose, or non-infringement. Nuance disclaims all liability for any direct, indirect, incidental, consequential, special, or exemplary damages resulting from the use of the information in this document. Mention of any product not manufactured by Nuance does not constitute an endorsement by Nuance of that product.

Published by Nuance Communications, Inc.
Burlington, Massachusetts, USA

Visit Nuance Communications, Inc. on the Web at www.nuance.com.

Contents

Part I: Introduction and Installation

Chapter 1: Introducing PowerScribe® Software Development Kit 1

PowerScribe SDK Object Model	2
Types of Applications You Can Develop	3
Types of Speech Recognition Technology	4
PowerscribeSDK Objects for Creating Reports	5
Report Object and Types of Reports	5
Other Objects for Use When Receiving Dictation	6
Creating Your Own Grammars for Reports	7
PowerscribeSDK Objects for Transient Data Retrieval	8
Note and CustomEditController Objects	8
Other PowerScribe SDK Objects/Tasks	9
Deploying an EventMapper	9
Creating Custom Training Module	9
Accessing Information About Logged In User	9
SDK Administrator Tasks	10
Creating Users and Assigning User Privileges	10
Working with Custom Shortcuts and Words	11
Taking System Actions on Users, Shortcuts, Categories, Groups, and Words	13

Chapter 2: Installing PowerScribe® SDK **15**

Understanding SDK Architecture	16
Installing the SDK Web Server and Recognition Server	17
Installing the SDK Client DLLs	18
Installing the SDK Developer DLLs from CD	19
Installing the SDK Developer DLLs from Server	19
Running the SDK Web Server	19
Creating Users in the SDK Administrator	20
Running the SDK Administrator	20
Adding a User	20
Assigning Privileges	21

Finding Help for SDK Administrator	21
Creating Your Custom SDK Installation Module	22
Locating SDK Code Samples	22
Setting Up Foot Pedals and Microphones	23
Chapter 3: Getting Started Developing SDK Application	25
Instantiating the PowerscribeSDK Object	26
Initializing Connection to the Server	26
Creating a Microphone Object	27
Configuring the Application to Listen to Events	27
Mapping Events in Javascript	27
Creating Event Delegates in C#	28
Pulling the Intialization Code Together	29
Turning on Logging	30
Creating the Login Functionality	30
Logging In the User	30
Anticipating Built-in SDK Responses to Login	31
Logging Out Dangling Logged In User	32
Handling Login Events	33
Handling the ProgressMessage Event	33
Handling the ProgressMessageEx Event	34
Handling the DownloadProgressMessage Event	34
Handling the LoginEnd Event	35
Taking Startup Actions after Successful Login	36
Loading Shortcuts and Words	36
Retrieving Version Information	37
Working with UserProfiles	38
Disabling Realtime Speech Recognition for Logged In User	38
Setting User Preferences	39
Turning On/Off Automatic Punctuation	39
Setting Correction and Selection Popup Box Options	39
Turning On/Off Display of Dictation Result Box	40
Turning On/Off Streaming of Dictated Text	41
Ensuring UserProfile Property Settings Take Effect	41
Retrieving UserProfile Property Settings	41
Determining Whether User Is Administrator	41
Determining Other Privileges of User	42
Releasing Event Maps before Exiting	43

Logging Out of the Application	44
Disconnecting from the Server	44
Code Summary for Initialize and Login	45
Code Summary for UserProfile Preference Settings	49

Part II: *PowerScribe SDK Application*

Chapter 4: Creating the Report	55
Understanding Types of Reports	56
Making SDK ActiveX Controls Available	56
Creating References to Libraries	56
Adding ToolBox Items in Visual Studio	57
Adding an Editor to Your Application	58
Embedding PlayerEditor Control in Your Application	59
Adding a LevelMeter to Your Application	60
Embedding LevelMeter Control in Your Application	60
Creating the Report Object	60
Associating PlayerEditor and LevelMeter with the Report Object	61
Associating PlayerEditor with the Report Object	61
Associating LevelMeter with the Report Object	62
Setting Up PlayerEditor Event Handlers	62
Mapping PlayerEditor Events	62
Handling PlayerEditor Events	62
Locking and Activating the Report	63
Locking the Report for Dictation or Editing	63
Activating the Report	64
Handling the ActivateEnd Event	64
Mapping the ActivateEnd Event	65
Handling the ActivateEnd Event	65
Adding Text/Changing the Report Content	65
Setting Dictation Mode	66
Handling the ReportChanged Event	66
Mapping the ReportChanged Event	67
Handling the ReportChanged Event	67
Working with Recognized Text—Handling the RecognizeEnd Event	67
Mapping the RecognizeEnd Event	67
Handling the RecognizeEnd Event	68
Saving the Report	68

Handling the SaveEndEx Event	69
Mapping the SaveEndEx Event	69
Handling the SaveEndEx Event	69
Working with the CanRecover Property	70
Preventing Recovery from Occuring	70
Taking Actions After Successful Save	70
Retrieving Any Remaining Audio/Clearing Audio Buffer	71
Marking the Report for Adaptation	71
Removing Text from Editor Display	71
Unlocking the Report	72
Disabling the PlayerEditor	72
Finalizing the SaveEndEx Event Handler	72
Handling the WaveFileProgress Event	73
Mapping the WaveFileProgress Event	73
Handling the WaveFileProgress Event	73
Reopening and Editing the Report	74
Preparing to Edit the Report	74
Opening the Report in View-Only Mode	75
Handling the LoadEnd Event	75
Mapping the LoadEnd Event	75
Handling the LoadEnd Event	76
Editing Report Content	76
Approving or Unapproving the Report	77
Deleting Reports	77
Exporting and Importing Reports to or from Disk	78
Exporting Standard Reports to Disk	78
Exporting Word Reports to Disk	78
Importing Standard Reports from Disk	78
Importing Word Reports from Disk	79
Code Summary	79
Chapter 5: Advanced Actions in Plain Reports	87
Exporting Reports	88
Retrieving Text of the Report from Server	88
Retrieving Text of the Report from Editor	89
Retrieving Audio of the Report	89
Retrieving Associated Concordance File	90
Importing Reports from Enterprise Express Speech or Another PowerScribe SDK Server	91

Locking the Report	91
Importing Existing Plain or XML Text File	91
Importing Existing Audio	92
Importing Associated Concordance File	92
Locking and Activating the Report	93
Pulling Together Code for Exporting and Importing Report	93
Importing Report from Third Party Device and Recognizing Audio Immediately	94
Generating Report Text from Audio	95
Saving Text of the Report	95
Handling the WaveFileProgress Event	95
Spell Checking Report Text	96
Sending Reports to Recognition Server for Batch Recognition	98
Disabling Real Time Speech Recognition	98
Preparing to Handle the WaveFileProgress and QueuedForRecognition Events	98
Saving Audio Files to the Server	99
Adding Audio File to the Recognition Server Queue	99
Handling the QueuedForRecognition Event	100
Using Recognition Accuracy Feedback (RAF) Tool	100
Code Summary	103
Chapter 6: Creating Structured Reports	109
Understanding Elements of Structured Report	110
Creating Structured Report Template	111
Creating Sections and Headings in XML Report Template (Manually)	111
Preparing to Handle Report and Sections Object Events	112
Mapping Report Object Events	112
Mapping Sections Object Events	113
Creating Structured Report Object	113
Creating Structured Report	113
Activating Structured Report	113
Adding and Removing Report Sections	114
Retrieving Sections Object	114
Adding Sections to Report	114
Removing Sections from Report	116
Handling SectionsChanged Report Event	117
Adding and Removing Subsections	117
Adding Subsections to Existing Sections Collection	117
Adding New Subsections Collection to Section	118

Removing Subsections from Report	118
Working with Report Section Information	119
Using Properties to Determine Report Content	119
Traversing Sections and Subsections with Section Properties	121
Displaying Report Structure	122
Handling SectionChanged Sections Event	123
Navigating and Moving Cursor in Report and Taking Action on Current Section	124
Determining Section of Cursor in Report	124
Moving Cursor to Another Section in Report	124
Adding Section before Currently Selected Section	124
Removing Currently Selected Section	125
Finding Particular Section	125
Exporting and Importing Structured Report	126
Showing Source of Text in the Editor	126
Displaying Text Source Markup	127
Customizing Display of Text Source Markup	128
Overview: Merging Reports	129
Merging Content of Two Structured Reports	129
Importing Selected Sections of Another Report	131
Appending Paragraph Text to Particular Section	133
Creating XML String of Text to Append	133
Appending Text to Section	134
Viewing Resulting Changes in Editor	134
Appending New Section to Report	134
Creating XML String of New Section	135
Appending New Section to Report	135
Viewing Resulting Changes in Editor	136
Merging Unformatted Text into Structured Report	136
Creating String of Text to Add	136
Positioning Cursor and Adding Text to Report	137
Viewing Resulting Changes in Editor	137
Merging Text into Plain Text Report	138
Creating String of Text to Add	138
Positioning Cursor and Adding New Section to Report	138
Viewing Resulting Changes in Editor	138
Merging Shortcut Text into Report	139
Code Summary	140

Chapter 7: Working with Report Editors	145
Choosing an Editor: PlayerEditor or Word PlayerEditor	146
Setting Up Word PlayerEditor	147
Embedding WordPlayerEditorCtl Control in Your Application	147
Associating WordPlayerEditorCtl with Report Object	148
Preparing to Work with Report Text	148
Changing Default Text Fonts and Colors in PlayerEditor	149
Finding and Replacing Text	149
Finding a String of Text	149
Replacing the String	150
Inserting Text or Tabs at Cursor	150
Inserting Text at the Cursor	150
Inserting Tabs at the Cursor	150
Handling TextSelChanged Event—Modifying Style of Text	151
Mapping PlayerEditorCtl and WordPlayerEditorCtl Events	151
Handling TextSelChanged Event to Indicate Format of Selected Text ..	151
Setting Style of Text	152
Sample Code That Parses Style Argument	153
Setting the Text Case	155
Copying or Cutting and Pasting Text	155
Undoing and Redoing Text Changes	156
Creating Numbered or Bulleted Lists	157
Modifying Default List Characteristics	157
Starting and Ending the List	160
Converting Selected Text to List Items	160
Converting Selected Text to List Items	160
Changing Start Number of Selected List Items	161
Choosing Clipboard Text Format	161
Handling CtrlKeyPress Event—Setting Up Control Keys	162
Handling CombineKeysPress Event—Setting Up Control Key Combinations ..	162
Handling FunctionKeyPress Event—Setting Up Function Keys	163
Handling NumLockKeyPress Event—Working with Locked Number Pad ..	164
Handling ButtonClick Event—Responding to Mouse Button Clicks	164
Retrieving All Text from PlayerEditor	165
Programmatically Positioning Cursor or Selecting Text	166
Refocusing Cursor in Report	166
Positioning Cursor in Report Text Programmatically	166
Determining Text Location Programmatically	169

Retrieving Audio Corresponding to Selected Text	169
Manipulating Audio Playback or Rewind Speed	170
Forcing Shortcuts to Expand in Editor	170
Expanding Shortcuts	170
Handling the ShortcutsExpanded Event	170
Zooming In or Out on Report Text	171
Dragging/Dropping Text into PlayerEditor	171
Using Microsoft Word Capabilities in Reports	172
Code Summary	172
Chapter 8: Dictating Outside a Report	183
Preparing to Accept Recognized Text Outside a Report	184
Creating the Note Object	185
Declaring the RecognizeEnd and ShortcutsExpanded Events	185
Creating PlayerEditor for the Note	186
Creating CustomEditController for the Note	186
Creating LevelMeter for the Note	187
Associating LevelMeter with Note	187
Setting PlayerEditor to the Current Editor for Note	187
Setting Windows Component to Current Editor for Note	187
Securing LevelMeter for Use by the Note	188
Setting Dictation Preferences for the Note	188
Setting Dictation Mode for the Note	188
Setting the Note to Receive Streamed Text	189
Securing Microphone for Use of the Note	189
Retrieving Recognized Text from Note—Handling the RecognizeEnd Event ..	189
Working with the Note Data	190
Retrieving Text from the Note	190
Retrieving Audio of the Note	190
Retrieving Concordance of the Note	190
Working with Shortcuts in a Note—Handling ShortcutsExpanded Event	191
Working with Note Audio File	191
Releasing the Microphone from Exclusive Use by the Note	192
Using the Note Object in JavaScript	192
Switching Between Note and Report	192
Switching Between Note and Report in JavaScript	192
Switching Between Note and Report in C#	193
C# Code Summary	195

JavaScript Code Summary for Dictating a Note	198
JavaScript Code Summary for Switching Between Note and Report	205
Chapter 9: Configuring and Customizing the Microphone	219
Locating Microphone Buttons	220
Creating a Microphone Object	220
Embedding Microphone Tuning Wizard	220
Requesting That User Set Up Foot Pedals and Microphones	222
Configuring Audio/Orientation Settings, Default Button Actions, Mic Light ..	223
Configuring Microphone Settings	224
Ensuring Microphone Settings Take Effect	227
Setting Transcribe and Navigate Preferences	227
Setting Transcribe Preferences	228
Setting Navigate Preferences	229
Beeping on Field or Section Navigation	231
Automatically Advancing to Next Shortcut Field	231
Mimicking Microphone with GUI Elements	232
Recording and Transcribing Audio	232
Advancing, Rewinding, Playing, and Stopping Audio	233
Handling Microphone Events	234
Mapping Microphone Events	234
Turning On Events from Microphone	235
Handling ButtonDown Event (Button Pressed)	235
Handling ButtonUp Event (Button Released)	237
Tracking Audio Position by Handling WavePosition Event	237
Handling MicConnectionChange Event	238
Handling ScanText Event	238
Customizing Programmable Microphone Buttons	239
Working with Headsets for Audio in Conjunction with Microphone or Foot Pedal Buttons	241
Code Summary	242
Chapter 10: Using Command and Control	247
What Is an SDK Grammar?	248
Steps to Defining and Using a Grammar	248
Creating XML Grammar Template	249
Grammar with Single Level	250
Grammar with Nested Item Levels	251
Grammar with Optional Phrases	252

Grammar with Multiple Verbs for Same Action	255
Creating a Grammar Object	256
Creating a Rules Object	256
Working with Rules	257
Preparing to Handle Grammar Events	258
Mapping Grammar Object Events	258
Switching to Command Mode	258
Handling Command Events	259
Handling Multi-Level Rules in a Grammar	260
Handling Multi-Level Rules with Optional Phrases	261
Handling Rules with Multiple Verbs for One Action	261
Deploying DefaultGrammar Commands	262
Implementing Predefined Actions	263
Implementing Custom Actions	264
Code Summary	265

Part III: SDK Administrator Functionality

Chapter 11: Getting Started Developing SDK Administrator Application	269
SDK Administrator Provided vs. Custom Administrator Application	270
Running the SDK Administrator	270
Understanding Features of SDK Administrator	270
Features You Can Include in Custom Administrator	271
Instantiating the PSAdminSDK Object	272
Initializing Administrator Application	272
Logging Out Dangling Logged In User	272
Configuring Administrator Application to Listen to Events	274
Instantiating EventMapper for PSAdminSDK Objects	274
Creating Event Delegates in C#	275
Releasing Event Maps Before Exiting	275
Logging Out of Application	276
Code Summary	276
Chapter 12: Working with Shortcuts, Categories, and Words .	279
Understanding Shortcuts and How They Are Used	280
Types of Shortcuts: Voice and Text	280
Shortcut Categories and Groups	280
How SDK Deploys Shortcuts and Words	280

Overview of Creating and Using Shortcuts	281
Creating and Editing Shortcuts	281
Steps to Creating Shortcuts	281
Creating the PSAdminSDK Object	282
Creating a Shortcuts Collection Object	282
Adding Shortcuts to the Collection	284
Modifying Short Phrase and Expanded Text of Existing Shortcuts	286
Assigning the Shortcut a Category	286
Retrieving Author, Global Status, and LastDateModified of Shortcut	287
Removing Shortcuts from the Collection	287
Creating and Merging Structured Shortcuts	288
Deploying Shortcuts	288
Steps to Deploying Shortcuts	288
Loading Shortcuts Collection into the In-Memory Language Model	289
Loading Single Shortcut into the In-Memory Language Model	289
Disabling and Re-Enabling Shortcuts	290
Verifying That Shortcuts Are Enabled	291
Creating Categories for Shortcuts	291
Creating a Collection of Categories	291
Adding a Category to the Collection	291
Removing a Category from the Collection	292
Assigning a Category to a Shortcut	292
Changing the Category Name or Description	292
Understanding Words	293
Overview of Creating and Using Words	293
Creating and Editing Words	294
Steps to Creating Words	294
Creating a Words Collection Object	294
Adding Words to the Collection	296
Retrieving and Modifying Particular Words	297
Removing Words from the Collection	298
Deploying Words	298
Steps to Deploying Words	298
Loading Words into the In-Memory Language Model	299
Loading Single Word into the In-Memory Language Model	299
Removing Single Word from the In-Memory Language Model	300
Retrieving a Read-Only Collection of Words	300
Retrieving Words Collection Containing Forms of a Punctuation Mark	300

Overview of Training Shortcuts, Words, or Punctuation Marks	301
Steps to Training Shortcuts, Words, or Punctuation Marks	301
Preparing to Train Voice Shortcuts, Words, or Punctuation Marks	302
Creating PhoneticTranscriber Object	302
Securing Microphone for PhoneticTranscriber	302
Securing LevelMeter for PhoneticTranscriber	302
Retrieving Pronunciation While Training Voice Shortcuts, Words, or Punctuation Marks	303
Training Audio of Shortcut or Word Using Custom GUI Buttons	303
Training Audio of Shortcut or Word Using Microphone Buttons	305
Training Audio of Punctuation Marks Using Custom GUI Buttons	306
Training Audio of Punctuation Marks Using Microphone Buttons	307
Adding Trained Shortcut to Collection, Loading into Memory	308
Using the Phonetic Transcription String to Add Shortcut to Collection	308
Revising Training for Existing Shortcut	309
Adding New Shortcut with Existing Phonetic Transcriber	309
Adding Trained Word or Punctuation Mark to Collection, Loading into Memory	310
Using the Phonetic Transcription String to Add Word to Words Collection .	310
Using the Phonetic Transcription String to Add Punctuation Mark to Words Collection	310
Revising Training for Existing Word	311
Adding New Word with Existing Phonetic Transcriber	311
Importing and Training Trigger Words	311
Selecting, Inserting and Auto Expanding Shortcuts in PlayerEditor— JavaScript	312
Auto Expanding Shortcuts After Drag and Drop into PlayerEditor—C#	314
Implementing Drag/Drop Feature in Drag Source	314
Using the Drag Source Application	317
C# Drag and Drop Code Summary	317
JavaScript Code Summary	324
Shortcuts ListView	324
Shortcuts Scripts	326
Additional Shortcuts Functions	334
Words ListView	335
Words Script	336

Chapter 13: Creating Users and Assigning Access Levels	343
Understanding Language Models and Users	344
Understanding Types of User Privileges	344
Steps to Creating Users	345
Creating the PSAdminSDK Object	346
Retrieving Language Models	346
Retrieving Language Models Collection	346
Selecting a Language Model from the Collection	347
Creating a User Object	347
Adding User to Collection and Assigning Login	347
Assigning User Privilege Levels	349
Granting Report Access Privileges	349
Granting Administrative Privileges	349
Setting Up and Assigning Format Profiles	351
Creating Custom Format Profiles	351
Retrieving Collection of Format Profiles	351
Assigning Format Profile to User	352
Assigning the User an Accent	353
Enabling and Disabling Logging for User	353
Enabling or Disabling Users	354
Working with Assigned Language Model	355
Modifying Language Model Assigned	355
Displaying Language Model Assigned	355
Working with User Acoustic Models	356
Retrieving and Modifying User Information	358
Retrieving the User Object	358
Changing User Password	358
Changing User Name and Badge Information	359
Changing User Report Capabilities	359
Changing User Administrative Capabilities	359
Switching to Batch Speech Recognition (on Server)	359
Removing a User	360
Code Summary	360
Chapter 14: Creating Groups in Your Application	371
Understanding Groups	372
Overview of Creating Groups	372
Creating and Populating Groups	372

Steps to Creating Groups	372
Creating the PSAdminSDK Object	373
Creating a Groups Collection Object	373
Adding Group Objects to the Groups Collection	374
Adding Shortcuts to a Group	375
Retrieving a Particular Group	375
Modifying Name/Description of the Group	376
Determining Who Owns the Group	376
Determining Whether Group Contains Shortcuts Available to All Users	376
Removing Shortcuts from a Group	377
Removing a Group from the Collection	377
Code Summary	378
Chapter 15: Creating Templates for Structured Reports and Shortcuts	385
How Application Uses Template Object	386
Steps to Using XmlTemplateEdit Object to Develop Report Templates	387
Creating Strings to Initialize XmlTemplateEdit Object	388
Creating Empty XML Template String	388
Creating XML Style Format String	388
Creating and Intializing the XmlTemplateEdit Object	390
Associating Editor with XmlTemplateEdit Template Builder Object	390
Creating Template with Content from Associated PlayerEditor	391
Steps to Using XmlTemplateEdit Object to Develop and Edit Structured Shortcuts	392
Using XmlTemplateEdit Object to Create or Modify Structured Shortcuts	392
Changing Text Style in XmlTemplateEdit Template	393
Preparing for XmlTemplateEdit Events	394
Developing Graphical Element That Reflects Template Structure	395
Handling Change to Section Name in Template	397
Correlating Template with Graphical Element	398
Positioning Cursor in Graphical Element Based on Click in XmlTemplateEdit Template	398
Positioning Cursor in XmlTemplateEdit Template Based on Click in Graphical Element	399
Indicating the Font of Text Selected in XmlTemplateEdit Template	400
Code Summary	401

Part IV: Advanced Tasks

Chapter 16: Executing Advanced SDK Administrator Tasks . . . 407

Copying Users from One SDK System to Another	408
Exporting Users	408
Importing Users	408
Exporting and Importing User Voice Models	409
Preparing to Handle Export and Import Events	409
Exporting User Voice Model	409
Handling EndExport Event	410
Importing User Voice Model	410
Handling the EndImport Event	411
Importing Users from Non-SDK System	411
Copying Words from One SDK System to Another	413
Exporting Words	413
Importing Words	413
Importing Words from Non-SDK System	413
Copying Shortcuts from One SDK System to Another	415
Exporting Shortcuts	415
Importing Shortcuts	415
Importing Shortcuts from Non-SDK System	416
Copying Categories from One SDK System to Another	417
Exporting Categories	417
Importing Categories	418
Importing Categories from Non-SDK System	418
Copying Groups from One SDK System to Another	419
Exporting Groups	419
Importing Groups	419
Importing Groups from Non-SDK System	420
Managing Reports	422
Code Summary	422

Chapter 17: Taking Over Control of the Microphone 427

Taking Over Control of the Microphone	428
Taking Custom Actions in ButtonDown Event Handler	428
Taking Custom Actions in Response to BOTTOM RIGHT Button	430
Taking Custom Actions in Response to DICTATE Button	430
Taking Custom Actions in Response to TRANSCRIBE Button	430

Taking Custom Actions in Response to PLAY/STOP Button	431
Taking Custom Actions in Response to REW Button	431
Taking Custom Actions in Response to BOTTOM LEFT Button	432
Restoring Default Microphone Functions	432
Code Summary	432
Chapter 18: Understanding Built-in Training and Adaptation	435
Operation of Built-in Training	436
Understanding Training Stages	437
Processing User for Training	437
Retrieving TrainingState of User	437
Working with Wav File Button	438
Popping Up Modal Training Dialog	439
Queueing User to Be Processed for Training	439
Queueing User to Be Processed for Adaptation	440
Option to Modify Built-in Training Pages	440
Chapter 19: Creating Custom Training Module	441
Steps to Creating Custom Training Module	442
Creating a Training Object	443
Creating an EventMapper Object	443
Securing Microphone and LevelMeter Exclusively for Training	444
Starting Training	444
Inserting Training Type into Your HTML	446
Inserting Training Text into Your HTML	447
Inserting Training Time into Your HTML	448
Handling Training Events	449
Responding to the NEXT Button	449
Responding to the PREVIOUS or BACK Button	450
Responding to the CANCEL Button	452
Responding to the START OVER Button	452
Releasing the Microphone and LevelMeter	452
Integrating Custom Training into Login Process	453
Adding Wave File Button to Custom Training	454
Code Summary	455

Part V: Appendices & Index

Appendix A: HeadingStyle and ParagraphStyle Options	463
Heading and Paragraph Styles	464
Appendix B: Empty Structured Report Template	465
Empty Structured Report Template	466
Appendix C: XML Template Schema	467
XML Template Schema	468
Appendix D: XML Style Format Template Schema	471
Style Formats Schema	472
Appendix E: PowerMic and PowerMic II Microphone Buttons .	473
PowerMic in PowerScribe® SDK	474
PowerMic II in PowerScribe® SDK	475
Appendix F: Audio Converter Tool for Upgrading Compressed Audio	477
Introducing Audio Converter	478
Installing Audio Converter Tool	478
Running Audio Converter Tool	479
Appendix G: Setting Up PowerMic II for Use on Virtual Machine	481
Requirements for Operating PowerMic II on VMWare Virtual Machine	482
Configuring USB Ports for PowerMic II	482
Notes on Configuring Dictaphone USB Device Ports on Virtual Machines ...	487

Introducing PowerScribe® Software Development Kit

Objectives

PowerScribe® Software Development Kit (SDK) is a set of COM objects and application programmer interfaces (APIs). The *SDK* helps you create a system that records speech to create documents (such as medical reports) and uses speech recognition to transcribe them. The system you create might also (or alternatively) record audio, then manage that audio as users manually transcribe it into text documents and then edit the text. In addition, the *SDK* helps you use speech recognition to retrieve transient realtime data from the speaker, such as part numbers.

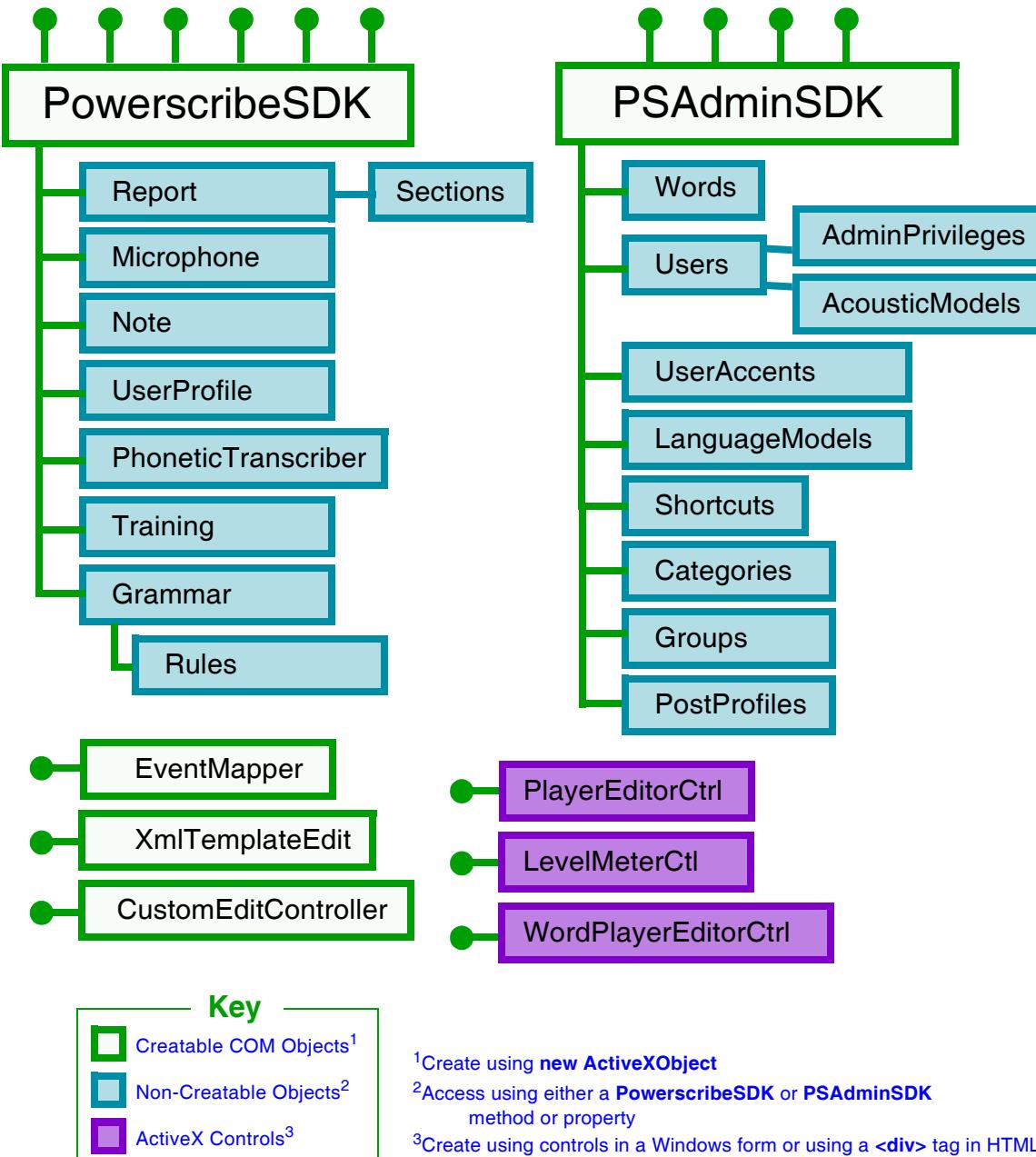
This chapter presents an overview of the object model and the types of capabilities that the *SDK* provides:

- [PowerScribe SDK Object Model](#)
- [Types of Applications You Can Develop](#)
- [Types of Speech Recognition Technology](#)
- [PowerscribeSDK Objects for Creating Reports](#)
- [PowerscribeSDK Objects for Transient Data Retrieval](#)
- [Other PowerScribe SDK Objects/Tasks](#)
- [SDK Administrator Tasks](#)

PowerScribe SDK Object Model

PowerScribe SDK is a toolkit for creating a custom report management system that captures dictated reports and uses speech recognition to transcribe them, then allows end users to correct the text and take other actions.

The object model for the *SDK* (shown below) contains several types of objects. Most of these objects belong to either the **PowerscribeSDK** or the **PSAdminSDK** object group.



Types of Applications You Can Develop

The *PowerScribe SDK* product contains tools to create two types of applications:

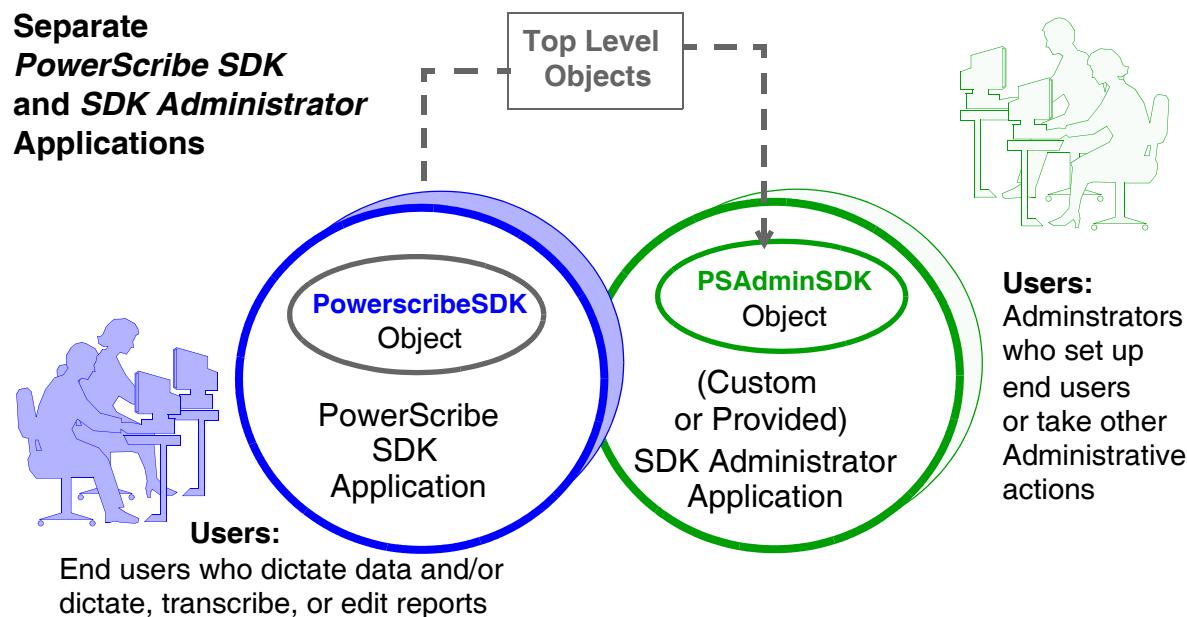
- *PowerScribe SDK* applications that deploy speech recognition (work with **PowerscribeSDK** object group of object model)
- *SDK Administrator* applications—Administer the system (work with **PSAdminSDK** object group in the object model)

You usually write your own *PowerScribe SDK* application(s) and utilize the *SDK Administrator* program provided; however, you can create custom *SDK Administrator* applications as well.

You are required to use the *SDK Administrator* application provided to manage logins, schedule purging of reports, and set system parameters, because you have limited access to these capabilities through the *SDK* APIs.

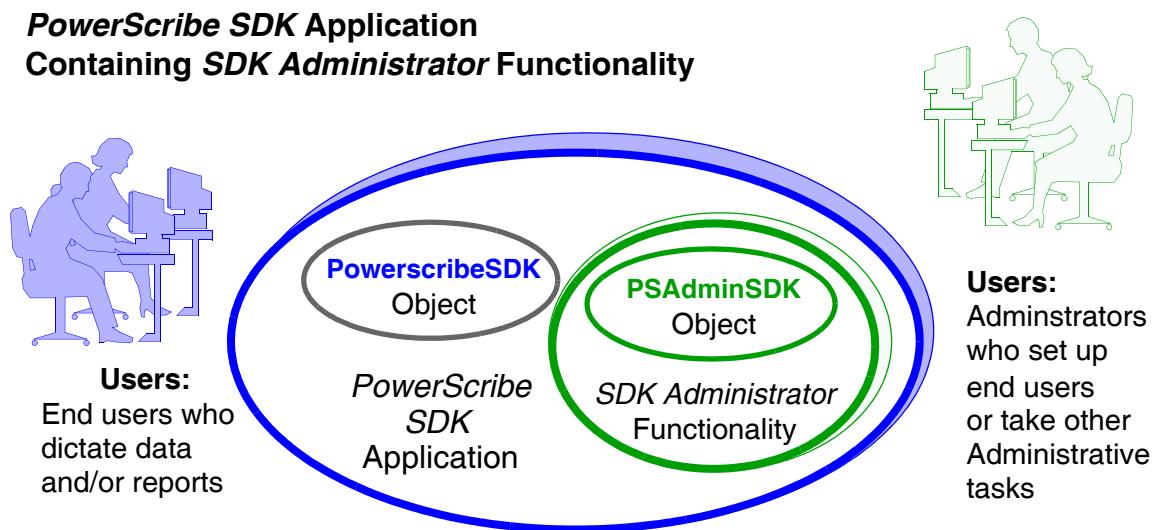
In your custom *SDK Administrator*, you can manage users, shortcuts, categories, groups, and custom words. In the course of managing users, you also handle assigning each user administrative privileges, a particular language model ID, and a postprocessor profile for formatting that user's reports. For more detail, refer to [SDK Administrator Provided vs. Custom Administrator Application on page 270](#).

As shown in the illustration below, *SDK Administrator* and *PowerScribe SDK* applications each have a different top level object.



Although the *SDK Administrator* and *PowerScribe SDK* applications can function separately, side-by-side, you might find it convenient to create a single application that lets a select group of users who are assigned appropriate administrative privileges carry out the administrative tasks, and all other users dictate data, dictate reports, or carry out related tasks.

Combining the two types of functionality in a single application is straightforward, because as shown in the illustration below, the **PSAdminSDK** object and **PowerscribeSDK** object can both be in the same application.



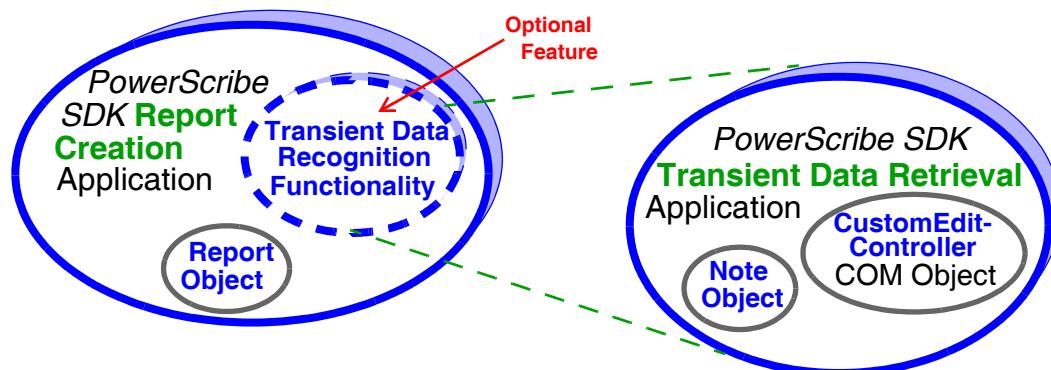
If you choose not to write a full *SDK Administrator* application, having an **PSAdminSDK** object inside your *PowerScribe SDK* application gives you access to user data and other administrative data that you set up with the *SDK Administrator* program provided.

Types of Speech Recognition Technology

PowerScribe SDK applications can utilize one or both of two major types of technology:

- Report creation—Recognize speech from dictation for creating and developing reports using the **Report** object. Stores text and audio of the report on the server.
- Transient data retrieval—Recognize speech from dictation to obtain transient data (such as a dictated part number to search for) using the **Note** object and **CustomEditController** creatable COM object. Does *not* store audio or recognized text of this data.

Note: These two types of technology can be integrated (see circle to left below), so that applications that deploy report creation can also deploy realtime transient data retrieval.



The **Note** object is available in JavaScript as well as in C#, Visual Basic, C++, and Java on Windows. The **CustomEditController** object is available in all of the same languages, except JavaScript, where you can generate the same functionality using the **Note** alone.

PowerscribeSDK Objects for Creating Reports

Let's begin by taking a look at the objects you would use in any *PowerScribe SDK* application designed for the purpose of dictating, transcribing, and editing reports. These objects are all under the **PowerscribeSDK** object in the object model.

Report Object and Types of Reports

You create reports with the *SDK* using the **Report** object. Each report can be one of two types:

- Structured
- Plain

Structured Reports

Structured reports contain predefined headings and sections created by applying an XML formatted template. The template determines the text that each heading should contain and the style of that heading text, as well as the style of the paragraphs entered under the heading. The structured report template can have up to nine heading levels. The end user can quickly enter text under each heading by typing or dictating, without having to enter the actual headings for each new report. The *SDK* provides this functionality as well as the ability to navigate the structure using the microphone.

A structured report also contains data about the source of the report text. The source of the text might be, for instance, typed, dictated, or reused. You can choose to display the source of the text by assigning different highlight colors and fonts to each source, then turning on the *display of source markup* to see each piece of text's source.

Plain Reports

Plain reports, by contrast, are text reports with no structure. If you do not provide an XML structure for the report, it defaults to a plain report. The plain report is not just ASCII text (although it can be), but normally contains fonts, and paragraph and heading styles you assign. What it does *not* contain is predefined headings and subheadings with predefined heading and paragraph fonts and styles that force it into a particular structure.

RTF Reports

When you save a plain report, the *SDK* saves it in *Rich Text Format (.rtf)*, so it retains fonts and styles you are using.

Word Format Reports

You can also create a plain report in a Microsoft Word editor. When you do, the *SDK* saves the report in Microsoft Word Document (**.doc**) format, so that you can open the file in Word or a compatible editor.

Other Objects for Use When Receiving Dictation

Other objects that you can use in the course of creating reports include:

- **Microphone**
- **PlayerEditorCtl** (or **WordPlayerEditorCtl**)
- **LevelMeterCtl**
- **XmlTemplateEdit**

The **PlayerEditorCtl**, **WordPlayerEditorCtl**, and **LevelMeterCtl** are all ActiveX controls, while the **XmlTemplateEdit** object is a creatable COM object.

Microphone Object

When you create a report in your *SDK* application, you can type text or dictate into a microphone. To work with a microphone, you are not required to have a **Microphone** object, but you can create this object to program customizable buttons on the microphone or even take over full control of all the buttons on the microphone in your application.

Another alternative you could take would be to have GUI buttons that initiate microphone actions, like **Record**, **Play**, **Stop**, **FastForward**, and **Rewind**. You can create such buttons and activate them using **Microphone** object methods.

PlayerEditorCtl ActiveX Control

After you dictate the report, the next step for a **Speech Recognition** user (a user with permission to dictate, transcribe, and edit) is to transcribe the audio—so the text appears in an editor. In *PowerScribe SDK* applications, you use a special editor called a **PlayerEditor**. The **PlayerEditor** is an editor that not only allows the user to see the text and modify it, but allows the user to play back the audio and watch the cursor track the text that corresponds to that audio. To integrate a **PlayerEditor** with all its capabilities into your application, you embed a **PlayerEditorCtl** ActiveX control in your Windows form or HTML.

WordPlayerEditorCtl ActiveX Control

If you would like to use a Microsoft Word-compatible editor to create plain reports that you can later open in Word, you can deploy a unique **PlayerEditor** designed to work with Word documents by embedding a **WordPlayerEditorCtl** ActiveX control in your Windows or HTML form. This **PlayerEditor** has not only all of the functionality of a **PlayerEditorCtl**, but the ability to utilize Microsoft Word's editing tools as well.

The *SDK* also provides a method you can call to retrieve a Microsoft Word document object, so that your application can work with Word commands that Microsoft provides.

LevelMeterCtl ActiveX Control

While dictating audio into a report in your application, a user needs to know that action is taking place. An object that shows changes in the volume of vocalizations spoken into the microphone is a **LevelMeter**, a bar that shows more or fewer tick marks as the volume of the speaker rises and falls. You can insert a **LevelMeter** in your application by embedding the **LevelMeterCtl** ActiveX control in your Windows form or HTML.

Creating Report Structures with XmlTemplateEdit Object

You can create an XML formatted template manually and pass it to a method when you create the report, or you can create the template programmatically using a special object called an **XmlTemplateEdit** object.

Creating Your Own Grammars for Reports

A grammar is a collection of specialized custom voice commands that you define in an XML file and can use only in **Command** mode. (In contrast with commands built in to the product that are operational in **Dictation** mode.)

A grammar might, for instance, include commands to create and edit reports or pop up a list of available shortcuts:

- Create Report
- Edit Report
- Display Shortcuts

Another grammar might be a set of voice commands for checking off options on a form, pressing buttons on a form, or using other specific phrases.

Grammar and Rules Objects

To create a grammar, you use the **Grammar** object.

When you define the grammar, you delineate the **Rules** of the grammar inside an XML file. Your application activates the collection of **Rules**, which contains the individual **Rule** objects, to activate the grammar.

You can create several sets of voice commands by creating a **Grammar** object for each one. Your application can then deploy one or more **Grammar** objects, depending on the needs of the person who is dictating.

PowerscribeSDK Objects for Transient Data Retrieval

When you create any *PowerScribe SDK* application not necessarily designed for the purpose of generating reports, you can embed in that application speech recognition for purposes other than reporting. Your application might, for instance, contain a form where a user can dictate the content of the fields, dictate a name or number to search for, or retrieve dictated numbers, letters, or words for other purposes.

Objects you would use in a realtime data retrieval application include two you also use for creating reports:

- **Microphone**
- **LevelMeterCtl**

In addition, a realtime data retrieval application also deploys two additional objects:

- **Note**
- **CustomEditController**

Note and CustomEditController Objects

To create fields in an application that receive dictation from a speaker for purposes other than creating a report, you use a **Note** object. The **Note** object, unlike a report, does not save the recognized text of the dictation; instead, your application uses the **Note** object to hold the recognized text in memory and use it to, for instance, look up a name or part number in a database. The **Note** is for text that is transient, like a lookup key, that can be disposed of when it is no longer needed.

The **Note** object is available in both JavaScript and *Visual Studio* applications.

In JavaScript, you can associate a **PlayerEditor** with the **Note** object and dictate into the editor. However, in *Visual Studio* applications using Windows components, you do not use the **PlayerEditor** to retrieve dictation into a **Note**.

Instead, for a GUI developed in a *Visual Studio* language (such as Visual Basic or C#) to receive dictation from a speaker, after you create a Windows form, you can set up any component on the form to receive dictation by deploying an *SDK* creatable COM object called a **CustomEditController** and using the **Note** object with that controller object.

The **CustomEditController** associates the component in the GUI with the **Note** object that receives the dictation.

Other PowerScribe SDK Objects/Tasks

Deploying an EventMapper

If you are developing your application in JavaScript, you can take advantage of a unique feature of the *PowerScribe SDK* product. This feature, called *Map Creation*, was designed to map events that the *SDK* COM objects fire, so that your JavaScript application can receive those events.

EventMapper Creatable COM Object

To use *Map Creation*, you instantiate an **EventMapper** creatable COM object. You then use that **EventMapper** to map each event you expect from a particular object to its event handler.

Later, when the event occurs, the associated handler responds to that event.

You use the **EventMapper** object to receive events from COM objects that are not ActiveX controls.

Creating Custom Training Module

Another object in the arsenal of the **PowerscribeSDK** object is the **Training** object. You use this object to apply a custom look and feel to the training pages in the product. The **Training** object provides the text for the pages, while you provide the custom colors and text styles, then determine the conditions under which the next page of text displays.

Accessing Information About Logged In User

Often when a user is logged in to a *PowerScribe SDK* application, the application needs to take action or know something about that user. To work with the logged in user, the application uses a **UserProfile** object. This object lets the application change the user's login ID and password, but also lets the application determine other aspects of that user—whether the user is an administrator, whether to use automatic punctuation in this user's recognized text, whether to stream text into the application for this user, and other aspects of the logged in user's profile.

SDK Administrator Tasks

Administrative tasks are those that an administrator level user normally takes. You can carry out many of these actions using the *SDK Administrator* application provided; however, if you want to customize the administrator functionality, you might want to create an *SDK Administrator* application of your own.

Let's take a look at the objects you use to carry out those tasks in your application. These objects are under the **PSAdminSDK** object in the object model.

You might write a separate *SDK Administrator* application where the user can carry out these tasks or you might integrate the tasks into a *PowerScribe SDK* application.

Creating Users and Assigning User Privileges

Before anyone can use either your *PowerScribe SDK* application or your *SDK Administrator* application, you must create users with the **PSAdminSDK** object.

Users and User Objects

You create users by retrieving the **Users** collection object (that already contains the original **admin** user) and adding new users to the collection. Each user has a **User** object that you retrieve to access that user's property settings and, in some cases, to modify those property settings.

AdminPrivileges Objects

One of the properties of a **User** object is the **AdminPrivilege** property. You use the **AdminPrivilege** property to return an **AdminPrivileges** object that has several properties of its own. You set the properties of the **AdminPrivileges** object to give the associated user particular types of administrative privileges.

PostProfiles Collection Object

Another object that you use when assigning **User** property values is the **PostProfile**, an object that you can define to set up how all the reports of that particular user are later formatted during postprocessing. You can utilize the default postprofile or you can configure a profile format using the *AutoformatProfile Customization* tool. Later, you can retrieve that profile from a **PostProfiles** collection and use it to set the **AutoFormatProfileID** property of a particular user.

LanguageModels and LanguageModel Objects

A language model is a vocabulary for a particular specialty that the *Speech Recognition* engine understands. For instance, the Radiology and Pathology models would be for specific fields in medicine.

Your application can retrieve the collection of all **LanguageModels** and then select a single **LanguageModel** object from the collection to assign to a particular speech recognition user (who can dictate reports and have the speech recognition engine transcribe them).

When the speech recognition user later logs in, the *PowerScribe SDK* application loads the associated language model from the site server into the memory of the machine where the user is running the *PowerScribe SDK* application, storing a copy of the model as the *in-memory language model*.

UserAccents and UserAccent Objects

PowerScribe SDK provides a collection of accents that you can choose from to assign an accent to a particular user. To retrieve that collection, your application can use the **UserAccents** collection object. The application can then retrieve a particular **UserAccent** from the collection and assign it to a particular user.

The collection includes accents such as **Indian English**.

AcousticModels and AcousticModel Objects

Each time a user logs in to your *PowerScribe SDK* application, that user speaks into an audio device such as a microphone, but the user can log in to another login session using a different type of device, such as a recorder or bluetooth device. Each time the user logs in with a different audio device, the *SDK* has the user teach the system his or her voice with that device, then generates a unique acoustic voice model for that user with that audio device. The user's voice model with that audio device is stored in an **AcousticModel** object in the **AcousticModels** collection. Because the voice model requirements may vary based on the audio device being used, if you want to know the training state of a user, you do not retrieve that data from the **User** or **UserProfile** object. Instead, the application retrieves the data about the training state of the logged in user through the **AcousticModel** of that user.

Working with Custom Shortcuts and Words

Shortcuts are single words or short phrases that trigger *SDK* functionality to insert entire predefined sections or multiple predefined paragraphs into a report, a process called *expanding the shortcut*. By saying a single word like **Start**, you could have your application insert several boilerplate paragraphs into a report. Similar short words or phrases might expand to include information that applies only in particular cases.

Your application can facilitate the process of creating shortcuts by helping the user define them, associating them with an audio file, and loading them for use. It can take similar actions for custom words that are not in your usual language model, but that you might define for a particular situation, such as generic drug names for a pharmaceutical application.

You use the **Shortcut** object to deploy this kind of functionality.

Shortcuts and Shortcut Objects

Although shortcuts can be dictated words or phrases (called *voice* shortcuts), they can also be typed words or groups of letters (called *text* shortcuts). For instance, instead of typing out a paragraph of standard input, you could type a series of letters that the application would automatically expand into a paragraph or complete section of the report.

To create shortcuts, you start by creating a **Shortcuts** collection object. You then add each **Shortcut** to the collection. Any shortcut in that collection can be for the currently logged in user only or globally available to all users.

Categories and Category Objects

Your *SDK Administrator* application can create user-defined purposes or types called **Categories** to assign to each shortcut. You create a category by first creating a collection of **Categories**, then adding particular **Category** objects to the collection, assigning each a name and description. Once the **Category** exists, you can assign that category to any particular shortcut using the **CategoryName** property of the **Shortcut** object. Each shortcut can have only one category.

Groups and Group Objects

Your *SDK Administrator* application can associate related or similar shortcuts with one another by putting them into groups. You create a group by first creating a collection of **Groups**, then adding particular **Group** objects to the collection, assigning each a name, description, and global status (global if available to all users, not global if specific to the creator of the group). Once the **Group** exists, you can add shortcuts to the group using the **Shortcut** object. Each shortcut can belong to more than one group.

Words and Word Objects

Sometimes the speech recognition engine has difficulty recognizing a word when a particular user dictates it. To help improve the quality of recognition, you can create custom words. You teach the speech recognition engine how those words sound when spoken by the particular user. You create such words using the **Word** object. You start by creating a **Words** collection object, and then add each individual **Word** to the collection.

PhoneticTranscriber Object

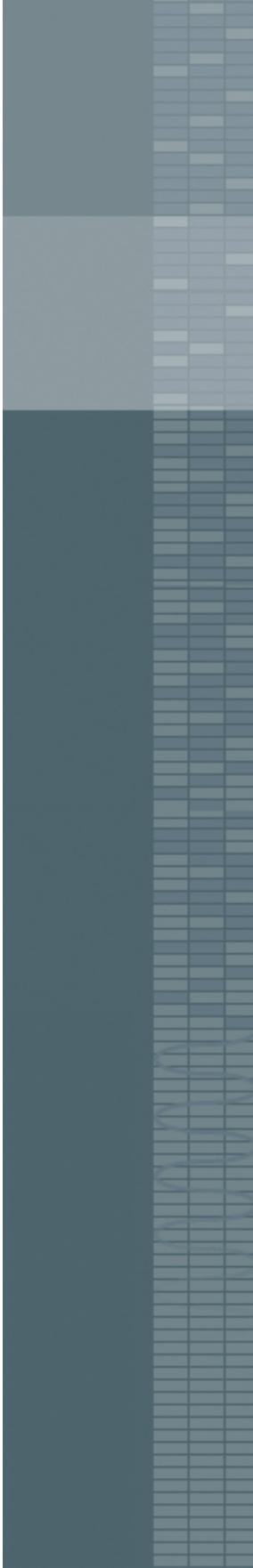
You can create an audio file to indicate how a custom shortcut or word would sound when it is dictated. You then pass the information from that file in a phonetic transcription string to the method that creates the shortcut or word. You retrieve that string from a **PhoneticTranscriber** object. This object helps you improve the quality of speech recognition for shortcuts and words.

Taking System Actions on Users, Shortcuts, Categories, Groups, and Words

Administrator level users might also take advantage of methods of the **PSAdminSDK** object that facilitate importing and exporting of users, voice models for those users, shortcuts, categories, groups, and words.

The administrator can use these capabilities to copy those objects from one *SDK* system to another.

Using XML formatted files, the administrator can also import user, shortcut, category, group, or word information from other (non-*SDK*) systems.



Chapter 2

Installing PowerScribe®

SDK

Objectives

This chapter tells you how to get started by installing the *PowerScribe SDK*:

- [Understanding *SDK* Architecture](#)
- [Installing the *SDK* Web Server and Recognition Server](#)
- [Installing the *SDK* Client DLLs](#)
- [Creating Users in the *SDK Administrator*](#)
- [Creating Your Custom *SDK* Installation Module](#)
- [Setting Up Foot Pedals and Microphones](#)



***Caution:** Be sure not to install PowerScribe and PowerScribe *SDK* on the same machine.*

Understanding SDK Architecture

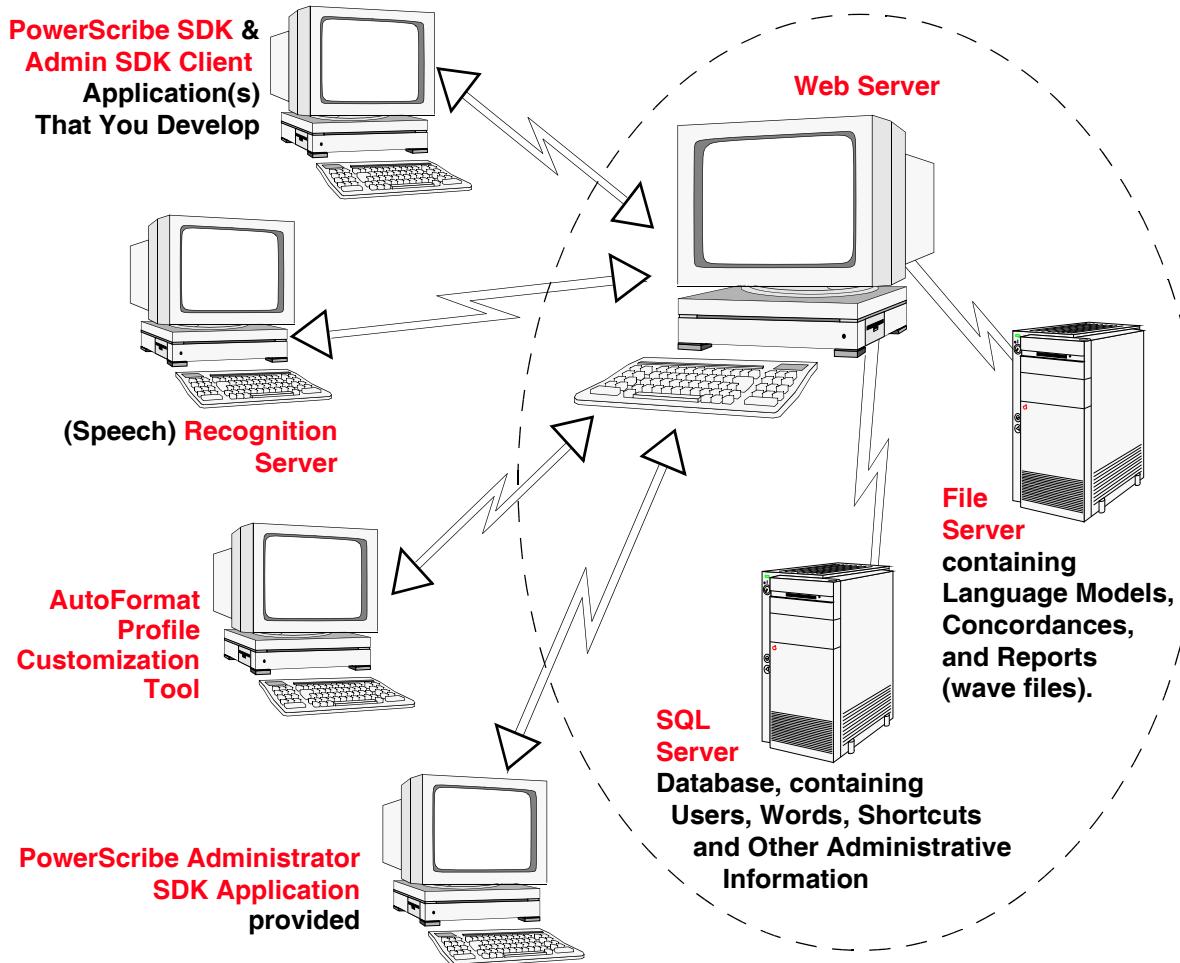
PowerScribe SDK is a software development kit that you use to develop a custom reporting system application whose purpose is to receive dictation and automatically transcribe it into reports. The *SDK* provides objects that you use to build two types of custom applications that work with the *SDK* web server and recognition server provided:

- An *SDK* application for dictation and transcription (using **PowerscribeSDK** objects)
- An *SDK Administrator* application for managing the system (using **PSAdminSDK** objects)

Your application interacts with the *PowerScribe SDK* server (*Web Server*). You can install all server and application components on the same physical machine or on remote machines.

Conceptual Illustration of Architecture*

with SDK Application, SDK Administrator Application, Web Server,
Recognition Server, Database, and File Server



*All components of the architecture can be installed on the same or separate machines. Frequently the Web Server, SQL Server, and File Server are on the same machine.

Installing the SDK Web Server and Recognition Server

Before you can develop an application using the *PowerScribe SDK*, you must have the *Web Server* and *Recognition Server* components of the product installed. Step-by-step instructions on how to install the these servers (or install an upgrade to them) are in the *PowerScribeSDK V2.1 Software Installation/Upgrade and Configuration* document (part number L-2139-001), on the product CD. When you install the *Web Server*, you install the *SDK Administrator* components with it. (When you write an *SDK Administrator* application, you create only the *interface* to the administrator functionality; the functionality still resides on the *Web Server*.)

If you are upgrading you should be sure to take the following steps first:

1. Back up the existing *PowerScribe SDK* database
2. Uninstall the old *PowerScribe SDK* application software

The steps you carry out to install the *PowerScribe SDK Web Server* are:

1. Install Microsoft SQL 2000 Server and SP4
2. Install Internet Information Services (IIS)
3. Install the Web Server Common Platform Installer (in the **CPI** directory)
4. Install the *PowerScribe SDK Database*
5. Install the Web Server
6. Install the Language Models

The steps you carry out to install the *Recognition Server* are:

1. Install Microsoft .NET Framework Version 1.1
2. Install the *Recognition Server* Software

After you upgrade you should be sure to take the following final steps:

Convert any compressed audio files stored on the server to the PCM format using the new Audio Converter tool. Instructions for running it are in [Appendix F: Audio Converter Tool for Upgrading Compressed Audio](#) (When you later compress the audio again, *PowerScribe SDK* uses the the algorithm you select for the **CompressionAlgorithm** parameter in the *SDK Administrator*.)

Installing the SDK Client DLLs

You can install the *PowerScribe SDK* application development toolkit (sometimes referred to as *client* development toolkit) functionality either from CD or from the server that you just installed. Once you have installed the application development tools, you must run the server to use those tools.

Installing the SDK Developer DLLs from CD

You must install the DLLs that enable the object model and APIs on the development machine. These COM objects allow you to work with JavaScript, C#, Visual Basic (6.0 or .NET), and C++. In addition, you must install a set of Java classes to develop your application in Java.

To install the SDK developer DLLs (client) from the product CD provided:

1. Insert the CD in the CD or DVD drive of the computer where you plan to develop code. Explore the CD and look for the **Redist** folder.
2. In the folder, find the **PowerscribeSDK.exe** file and run it. The InstallShield® wizard guides you through the installation process. This process automatically installs *PowerMic* and *PowerMic II* microphone drivers.
3. If you want to use Java for development, you need additional files. Look in the **Drivers** folder under **Redist** and find the **Java** folder. Copy the folder to your hard drive.

Installing the SDK Developer DLLs from Server

You must install the DLLs that enable the object model and APIs on the development machine.

To install the SDK developer DLLs (client) from the *PowerScribe SDK* server:

1. Open a browser on the machine where you plan to develop the SDK client application.
2. Enter the URL to the *PowerScribe SDK* server (substituting your server name for <servername>):
http://<servername>/pscribesdk/cab/
3. When the **Dictaphone PowerscribeSDK Components Setup** screen appears, click on the **Install** button and follow the instructions. The InstallShield® wizard guides you through the installation process.

To install Java classes for Java development, refer to [Installing the SDK Client DLLs on page 18](#).

Running the SDK Web Server

To develop code using the *SDK* developer DLLs, the *PowerScribe SDK Web Server* should be running. The *Web Server* begins running automatically as soon as you complete the installation or whenever you reboot your computer. It continues running as long as your server hardware is running. If for some reason, the *Web Server* goes down, *PowerScribe SDK Web Server* restarts when you boot the machine again. If the *PowerScribe SDK Web Server* stops running, you receive a 404 error in the browser when you try to connect to it.

Creating Users in the SDK Administrator

Before you can take any further action with the product, you need to establish a user account for yourself that gives you privileges required to create a report.

You create user accounts initially by using the *SDK Administrator* included with the product. Later, after you create your own *Administrator* using the **PSAdminSDK** object, you can create additional user accounts with that application instead.

Running the SDK Administrator

To run the *SDK Administrator*:

Open a web browser and enter the following for the URL, using the name of your server:

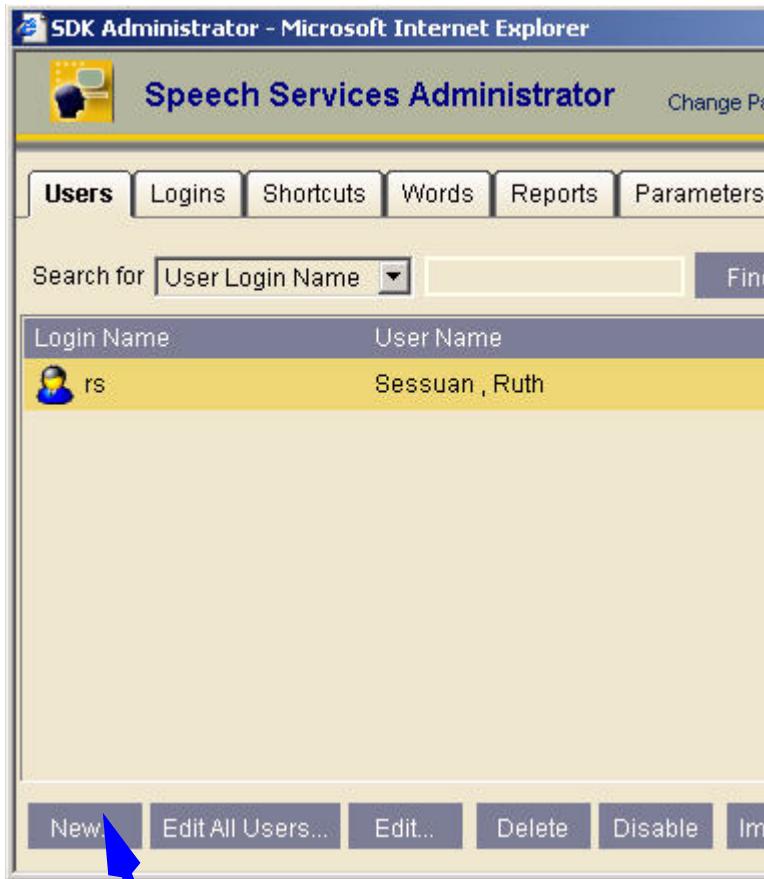
http://<servername>/pscribesdk/admin

The *SDK Administrator* GUI appears in the browser.

Adding a User

To create a user in the *SDK Administrator*:

1. Log in using the administrator login and password provided. You then see the **Users** tab displayed.
2. Create a new user by clicking the **New** button in the lower left corner of the window.



3. On the **General** tab, enter the identifying information, including the login and password of the user.

Assigning Privileges

In the **User Roles** group box, you indicate the type of privileges the user should have:

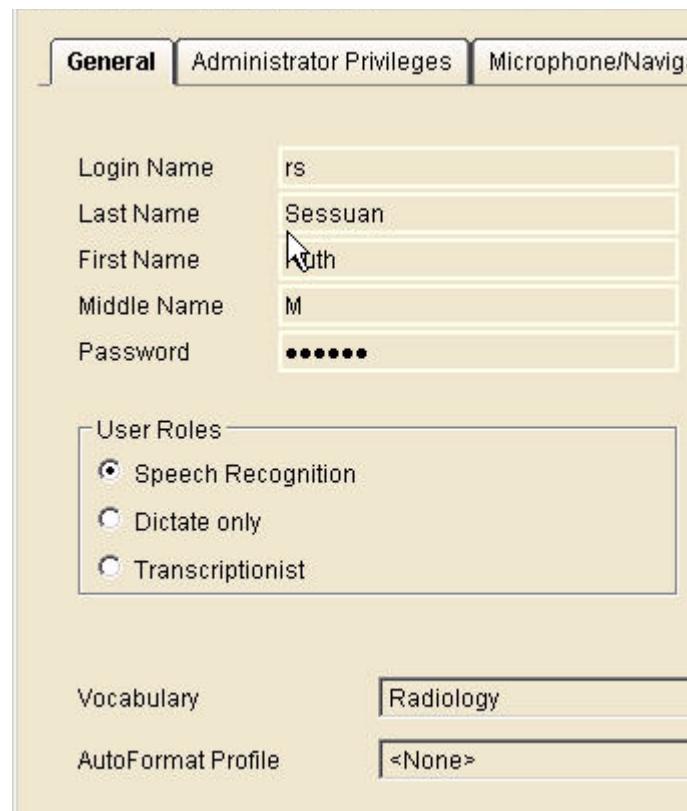
- Speech Recognition
- Dictate Only
- Transcriptionist

Speech Recognition—This type of user can dictate and indicate the dictation should be transcribed. The *PowerScribe SDK* then recognizes and transcribes the text. This type of user corrects the text on the screen using voice commands or on the screen.

Another way of referring to this type of user is as a *Self-Correcting* user, because this person corrects his or her own dictated text. However, the user still has the option of sending the report to a transcriptionist to be further corrected.

Dictate Only—This user can dictate reports to be manually transcribed. This user first dictates the report and then indicates the dictation should be transcribed. The *PowerScribe SDK* saves the audio but does not transcribe the report. Your application can implement logic that allows the user to edit the audio or sends the audio file to a transcriptionist. Another way of referring to this type of user is as an *Audio-Only* user.

Transcriptionist—This user can take only limited action, listening to dictated audio and transcribing it and/or editing the transcribed text of reports. This user cannot create reports, dictate into them, or edit them. In one typical workflow, a **Dictate Only** user dictates audio and the **Transcriptionist** user transcribes that audio.



Finding Help for SDK Administrator

To find out more about the *SDK Administrator* built-in to the product, refer to the Help file provided on the product CD or click the **Help** link in the *SDK Administrator* GUI.

Creating Your Custom SDK Installation Module

If you are a vendor/developer, after you develop your client application, you can create your own custom installation for your clients that packages the application with the SDK functionality.

To create the installation module:

1. On the *PowerScribe SDK* CD, find the **Redist** directory. In this directory look for the files indicated in subsequent steps and include those files in your installation module.
2. Find the **COM** folder and include the DLLs from it or include the entire directory in your installation package.
3. If your clients use the standard *PowerMic* and *PowerMic II* microphones, go to the **Drivers** directory and include the **PowerMicI** and **PowerMicII** subdirectories, which contain the microphone drivers, or include the entire **Drivers** directory in your installation package.
4. If your application is a Java application, you must include the Java classes from the **Java** folder. (Java applications still require the COM objects in the **COM** folder.)

Package the installation module in a format convenient for your clients.

Locating SDK Code Samples

After you have successfully installed the *PowerScribe SDK* on your server, you can access a directory of samples in six languages, including JavaScript, C++, C#, Visual Basic 6.0, Visual Basic .NET, and Java. Only the JavaScript samples are installed on your server when you install the product. The rest are available on the CD.

To access the samples already installed on your server, go to this URL:

<http://<servername>/pscribesdk/samples>

To find the source files for the installed samples, look under:

C:\Dictaphone\tomcat\webapps\pscribesdk\samples

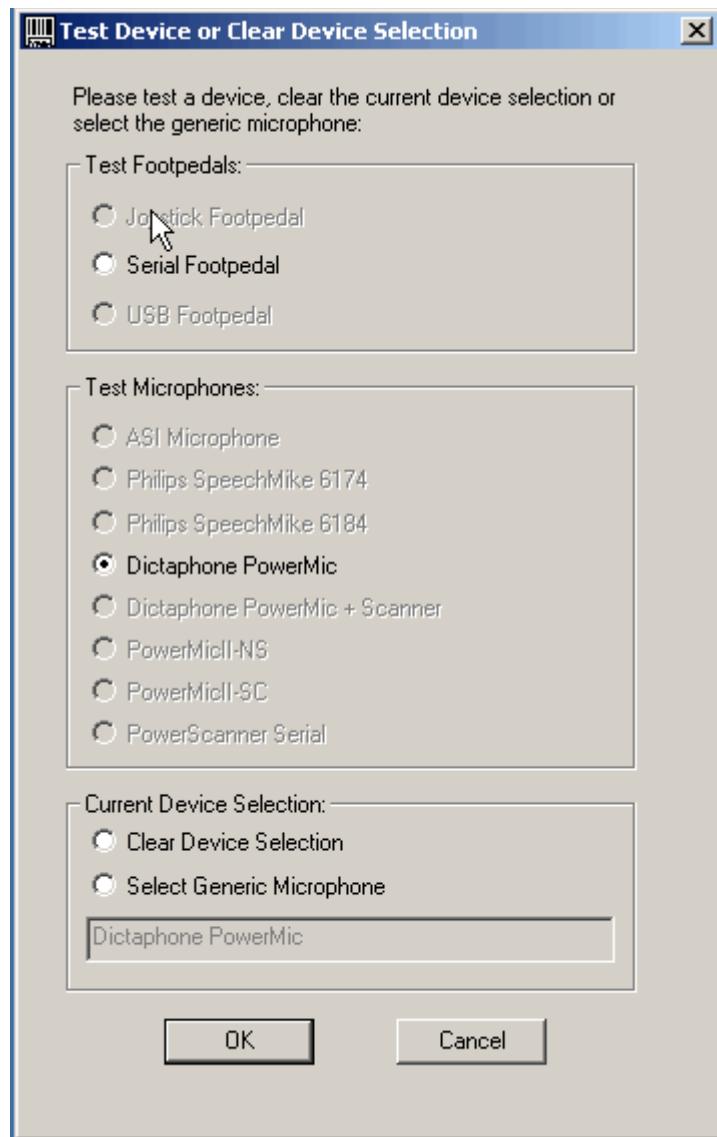
To use other samples, copy them from the **Samples** folder under the root on the CD and compile and run them in Visual Studio or in Java development tools. Each Visual Studio sample has an **.ini** file that contains the setting for the **wsdlLocation** or **PscribeSDKLocation** under **[PscribeSDK]**. Be sure that the location is either **localhost** (for locally installed server) or the name of the machine that is running your *PowerScribe SDK* server.

Setting Up Foot Pedals and Microphones

On each machine where a user might dictate through a microphone, you should install the microphone wizard.

To install the microphone wizard:

1. Copy the **Micwiz.exe** file from the product CD to the **C:\Windows\System32** directory on the machine.
2. Run the executable. You are then prompted to test a device.
3. If you are using a foot pedal, in the **Test Footpedals** group box, select **Serial Footpedal**.
4. If you are using a *PowerMic* microphone, in the **Test Microphones** group box, the **Dictaphone PowerMic** model should already be selected, so you do not need to take any action.
5. If you are using any other microphone model, in the **Current Device Selection** group box, choose **Select Generic Microphone**.
6. Click **OK**.



Getting Started Developing SDK Application

Objectives

The *PowerScribe® SDK* provides you with tools to create your own reporting system. This chapter uses JavaScript to present the fundamental steps required to start any *SDK* client application:

- [Instantiating the PowerscribeSDK Object](#)
- [Initializing Connection to the Server](#)
- [Configuring the Application to Listen to Events](#)
- [Creating the Login Functionality](#)
- [Handling Login Events](#)
- [Taking Startup Actions after Successful Login](#)
- [Working with UserProfiles](#)
- [Disabling Realtime Speech Recognition for Logged In User](#)
- [Setting User Preferences](#)
- [Releasing Event Maps before Exiting](#)
- [Logging Out of the Application](#)
- [Disconnecting from the Server](#)
- [Code Summary for Initialize and Login](#)



Note: The code in this manual was developed strictly for illustrative purposes. Sample code is included on the product CD in the **Samples** directory.

Instantiating the PowerscribeSDK Object

To begin interacting with the *PowerScribe SDK Web Server*, your application must instantiate a **PowerscribeSDK** object, the main object in the object model. You would instantiate the object in JavaScript as follows:

```
pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");
```

The first action you take on the **PowerscribeSDK** object is to initialize the object, so that you can then use the methods, properties, and events of the object.

Initializing Connection to the Server

Now that you have the **PowerscribeSDK** object, you initialize the **PowerscribeSDK** object using the **Initialize()** method. This method has three arguments, the first two required. The first argument is the URL to the *SDK* application on the server, the second the name of your application:

```
var url = "http://server2/pscribesdk/";  
  
pscribeSDK.Initialize(url, "JavaScript_Demo");
```

An optional third argument is a Boolean that indicates whether to use realtime speech recognition (occurs locally) or batch speech recognition (occurs on the *Recognition Server*).

The third argument is named *disableRealTimeRecognition* and if you do not pass a value for this argument, it is automatically set to **False**, and the *SDK* enables speech recognition on the machine running the application. The setting you pass for this parameter overrides the setting of the **UseRealTimeRecognition** property setting in the **UserProfile** of the logged in user.

If you want the application to use realtime speech recognition, you can pass **False** for the third argument:

```
pscribeSDK.Initialize(url, "JavaScript_Demo", False);
```

In a case where you want to prevent the application from taking up the bandwidth that would be required to download language models, you might want to turn off realtime speech recognition by passing **True** for the third argument:

```
pscribeSDK.Initialize(url, "JavaScript_Demo", True);
```

You would then carry out speech recognition on the server rather than on the local machine running your application.

Creating a Microphone Object

As soon as you plug it in, you can start working with the microphone in your application. By default, the microphone is there. You do not need to be logged in to the application to have it receive events from the microphone. Your application does not have to handle basic microphone events, such as button clicks that receive and transcribe dictation.

You might want to have buttons in your application that imitate the microphone. To have that capability, you need to have a **Microphone** object that you create through the **PowerscribeSDK** object using the **Microphone** property:

```
MyMicrophone = pscribeSDK.Microphone;
```

Once you have a **Microphone** object, you can use it to access associated methods, properties, and events and add custom functionality to your application. By default, you do not need the object. For more information on using the **Microphone** object and its events in your application, refer to [Chapter 9, Configuring and Customizing the Microphone](#).

Configuring the Application to Listen to Events

After you have initialized the connection to the *PowerScribe SDK* server, you need to set up the application to listen to events.

Mapping Events in Javascript

Because JavaScript does not provide a mechanism for listening to and handling COM object events, *PowerScribe SDK* provides a special mechanism for your application to retrieve events it triggers. The mechanism is called **Map Creation**, a flexible facility for mapping events to user defined functions. This mechanism was designed to be used inside JavaScript.

Instantiating the EventMapper Object

Before you can map events with the **Map Creation** facility of the product, you must instantiate an **EventMapper** object to catch events that the **PowerscribeSDK** object triggers. To indicate the **EventMapper** is catching **PowerscribeSDK** object events, let's name it **SDKsink**:

```
SDKsink = new ActiveXObject("PowerscribeSDK.EventMapper");
```

Mapping the Events to the Object

Once you have the **EventMapper** object, you then link events to user defined functions that your application must later call to handle events that the **PowerscribeSDK** object triggers. For example, for the **LoginEnd** event, you might use the **EventMapper** named **SDKsink** that you just created to map **LoginEnd** to a handler as follows:

```
SDKsink.LoginEnd = HandleLoginEnd;
```

The handler can have any name, but it makes sense to have its name clearly indicate the event that it is designed to handle. Also, when you create the handler, it must have the exact number of arguments the event sends when it is triggered. For instance, the **LoginEnd** event has a single argument, so the **HandleLoginEnd** event handler must be set up to receive that single argument, as shown below:

```
function HandleLoginEnd(rca)
{
    //Handles the LoginEnd event
}
```

You must define each handler to receive the exact number/type of arguments documented for the associated event, in this case one argument that receives an integer indicating the success or failure of the login process.

The mapping of the handler is not complete until you call the **Advise()** method of the **EventMapper** object and pass it the name of the **PowerscribeSDK** object that will be the source of the events:

```
SDKsink.Advise(pscribeSDK);
```

Later, after it finishes using events, before the application logs out the user, it must call the **Unadvise()** method to release the event mapper object.

You should take these same actions for the other login events that the **PowerscribeSDK** object triggers, **ProgressMessage**, **ProgressMessageEx**, and **DownloadProgressMessage**.

You need only call the **Advise()** method once for all events associated with the **PowerscribeSDK** object.

Creating Event Delegates in C#

In C#, you create the event delegates using the **PowerscribeSDK** object. For instance, if the object is named **m_psSDK**, you create the **ProgressMessage** and **LoginEnd** delegates as follows:

```
m_psSDK.ProgressMessage += new POWERSCRIBESDKLib._IPowerscribeSDKEvents
    _ProgressMessageEventHandler(m_psSDK_ProgressMessage);
m_psSDK.LoginEnd += new POWERSCRIBESDKLib._IPowerscribeSDKEvents
    _LoginEndEventHandler(m_psSDK_LoginEnd);
```

When you create the actual handler for the **LoginEnd** event, you pass it the **LoginResult** from the **POWERSCRIBESDKLib** in the **ErrorCode** argument:

```
private void m_psSDK_LoginEnd(POWERSCRIBESDKLib.LoginResult ErrorCode)
{
    // Take appropriate action
```

```

    }
}

```



Note: For samples showing how to set up events in Visual Basic 6.0, Visual Basic .NET, C++, and Java, refer to the sample code in the **Samples** directory under the root on the CD.

Pulling the Initialization Code Together

You might carry out the entire series of actions that initialize the application in a user defined **InitializePS()** function. It is advisable to call methods inside a **Try/Catch** block to help catch errors:

```

function InitializePS()
{
    try
    {
        // Instantiate PowerScribe SDK Object
        pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");

        // Initialize PowerScribe SDK Server

        var url = "http://server2/pscribesdk/";
        pscribeSDK.Initialize(url, "JavaScript_Demo");
        // The disableRealtimeRecognition argument of Initialize()
        // defaults to False, turning on realtime recognition

        // Create Event Mapper Objects for SDK Object, Microphone Object

        SDKsink = new ActiveXObject("PowerscribeSDK.EventMapper");
        MicSink = new ActiveXObject("PowerscribeSDK.EventMapper");

        // Map Event Handlers for Each Event to SDK Event Mapper Object

        SDKsink.LoginEnd = HandleLoginEnd;
        SDKsink.ProgressMessage = HandleProgressMessage;
        SDKsink.ProgressMessageEx = HandleProgressMessageEx;
        SDKsink.DownloadProgressMessage = HandleDownloadProgressMessage;
        SDKsink.Advise(pscribeSDK);

        // Create Microphone Object, Map Event Handler to Mic Evt Mapper Obj

        MyMicrophone = pscribeSDK.Microphone;
        MicSink.Advise(MyMicrophone);
    }
    catch(error)
    {
        alert("InitializePowerscribe: " + error.number + error.description);
    }
}

```

Turning on Logging

An action you might want to take after you initialize *PowerScribe SDK* is turning on logging to help you determine what has happened should any issues arise with your application. You take this action by calling the **EnableLogging()** method of the **PowerscribeSDK** object:

```
pscribeSDK.EnableLogging(true);
```

This method turns on (if you pass it **True**) or off (if you pass it **False**) client side logging for the duration of the current login session. This process logs all API calls your application makes. The *SDK* stores the logs in .txt files that you can find in the **C:\Documents and Settings\<username>\Local Settings\Application Data\Dictaphone\HSG\Trace\PowerscribeSDK** directory. If you call for technical support, you might be asked to provide these files.

If at any time you want your application to retrieve the log files, you can always retrieve the path to the log files using the **GetLogPath()** method of the **PowerscribeSDK** object:

```
var strLogFilePath = pscribeSDK.GetLogPath();
```

This method returns the path in a string and you can use that information to access any log file in the directory.

Creating the Login Functionality

After you have initialized the connection to the server, the user can log in to your application.

Logging In the User

You log the user in to your application by calling the **Login()** method of the **PowerscribeSDK** object. You are required to pass the method only the user's login name:

```
pscribeSDK.Login(userName);
```

All other parameters of this method are optional, but in most cases you will also want to retrieve a password, which you would pass as the second argument. The third argument, *Silent*, is obsolete, so it you can set it to either **True** or **False**, but you must pass it as a space holder. The fourth argument is a Boolean called *ShowEnrollment* that determines whether to display the **Training** pages (starting with the first one, called the **Enrollment** page) during the login process. The fifth argument associates an audio input device, such as a microphone, BlueTooth, Palm Pilot, or simiar device, with this login session of this particular user. To have **Login()** check the user's password, display the default **Training** pages, and retrieve audio with the *PowerMic II* microphone, call it this way:

```
psAudioDevicePowerMicII = 7;  
pscribeSDK.Login(username, userPswd, true, true, psAudioDevicePowerMicII);
```

Other devices you might pass to in the **Login()** method are listed in the table of known audio devices that follows.

AudioDeviceType	Value
psAudioDeviceUnknown	0
psAudioDevicePhillipsSpeechMic	1
psAudioDeviceASIMic	2
psAudioDevicePowerSerialMic	3
psAudioDevicePowerMicI	4
psAudioDeviceBoomerangMic	5
psAudioDeviceGenericMic	6
psAudioDevicePowerMicII	7
psAudioDeviceBluetooth	8
psAudioDeviceArrayMic	9
psAudioDevicePocketPC	10
psAudioDevicePalmPilot	11
psAudioDeviceDROlympus	12
psAudioDeviceOlympusDRec	13
psAudioDeviceOlympusDS330	14
psAudioDeviceOlympusDS2	15
psAudioDevicePC	16
psAudioDevicePCDynamic	17
psAudioDeviceTelephonyServer	18
psAudioDeviceTelephonyDictaphone	19
psAudioDeviceTelephonyDVIPS	20
psAudioDeviceDVI	21
psAudioDeviceLanier	22

Anticipating Built-in SDK Responses to Login

Once a user logs in, the *SDK* automatically presents a wizard it uses to adjust the volume of the microphone.



Caution: *The user must read all text in the dialog box presented for setting the microphone volume level. If the user does not read the text, the SDK logs that user out automatically and continues to present the dialog box each time that user logs in, until the person has read the text.*

After the user reads the text of the dialog box that sets the microphone volume, if this is the first time the user is logging in to the *SDK* application, the *SDK* automatically presents a wizard it uses to train the speech recognition engine to understand the user's speech patterns.



Caution: *The user must read all text in the first dialog presented for teaching (training) the speech recognition engine. If the user does not read the text, the SDK logs that user out automatically and continues to present that training dialog each time that user logs in, until the person has read at least the first training dialog (the **Enrollment** page).*

Once a user completes the training process, that person does not see the training dialogs again.

Your code can use custom dialog boxes and logic for training the speech recognition engine, but it must include the feature. For information on developing custom training, refer to [Chapter 19, Creating Custom Training Module on page 441](#).

Logging Out Dangling Logged In User



Note: *The login for a particular user never expires. As a result, if the user exits the application without logging off, the user remains logged in until your application logs him or her out.*

When a user first attempts to log in to your application, we strongly recommend that your code check to see if that user is logged in already, because when that user last finished using the application, if he or she did not log out, the session has not expired. In this situation, a *dangling logged in user* remains logged in forever unless your application logs that user out.

One way to manage this situation is to create a variable called **userLoggedInSDK** and initialize it to a setting of **False** to indicate the default situation that the user does not need to be logged out:

```
var userLoggedInSDK = false;
```

Now encase the call of the **Login()** method inside a **Try/Catch** block and set it up to detect error number -2146778148. When this error occurs, the user who is attempting to log in cannot because he or she is already logged in. Under these conditions, you set **userLoggedInSDK** to **True** so that the code can respond appropriately:

```
try
{
    pscribeSDK.Login(username, userPswd, true, true);
}
catch(error)
{
    if (error.number == -2146778148)
    {
        userLoggedInSDK = true;
    }
    else ...
}
```

When **userLoggedInSDK** is **True**, you need to log out the user and then log in the new user. You log the old user out by calling a method designed specifically for the situation of a dangling logged in user, the **LogoffUser()** method. You pass **LogoffUser()** the user name and password, then the name of the application to log out of, such as **JScriptDemo**:

```
if (userLoggedInSDK)
{
    pscribeSDK.LogoffUser(username, userPswd, "JScriptDemo");
    pscribeSDK.Login(username, userPswd, true, true);
}
```

To log the user back in, you then immediately call the **Login()** method.

Handling Login Events

Every time a login to an *SDK* application begins, the *SDK* notifies the application about the progress of the login through these events:

- **ProgressMessage**
- **ProgressMessageEx**
- **DownloadProgressMessage**
- **LoginEnd**

When the login completes, the *SDK* sends the application a notification that the login process has completed in the **LoginEnd** event.

Handling the ProgressMessage Event

Every time a login to an *SDK* application starts, the *SDK* notifies the application of progress during the login and training and enrollment processes by sending a message to the **ProgressMessage** handler. You name this handler using the name you associated with the event mapper earlier, **HandleProgressMessage**, and have it receive the progress message in an argument named **msg**:

```
function HandleProgressMessage(msg)
{
    // Show message
    // in a status bar
    window.status = msg;
}
```

 You must define each handler to receive the exact number/type of arguments documented for the associated event, in this case one argument that receives a string containing a message about the progress of the login.

The handler can take any action you deem appropriate. In the sample code shown above, the handler displays the message in a status bar.

Handling the ProgressMessageEx Event

Another event that the *SDK* triggers during the login and training process is the **ProgressMessageEx** event, which sends the percentage of files downloaded and a status message about the login operation in process. You name this handler using the name you associated with the event mapper earlier, **HandleProgressMessageEx**, and have it receive the percentage of files downloaded in the first argument and the status message in the second argument.

Let's call the two arguments of the event handler *percent* and *status*:

```
function HandleProgressMessageEx(percent, status)
{
    // Display progress bar
    // showing percentage
    // of files downloaded
    // status msg in status bar
}
```

You must define each handler to receive the exact number/type of arguments documented for the associated event, in the correct order. This handler receives a long containing the percentage of files downloaded so far, and a string containing the status message.

You could have this handler display, for instance, a progress bar showing the percentage of files downloaded along with the status message in a status bar.

Handling the DownloadProgressMessage Event

Another event that the *SDK* triggers during the login and training processes is the **DownloadProgressMessage** event, which sends more detailed login and training progress information to the application. You name this handler using the name you associated with the event mapper earlier, **HandleDownloadProgressMessage**, and have it receive five arguments, in this order:

- *totalTimeRemaining* — Time remaining to finish downloading all components.
- *totalPercentage* — Percentage of files downloaded.
- *stepTimeRemaining* — Time remaining to finish downloading the current component.
- *stepPercent* — Percentage of the current component that has been downloaded.
- *bsMsg* — Message giving the name of the component currently being downloaded.

Your handler can retrieve the values from these arguments and present the information on the screen to show the user what is happening:

```
function HandleDownloadProgressMessage(totalTime, totalPercent, stepTime,
stepPercent, statusMsg)
{
    // Track download process
}
```

Define the event handler to receive the exact number and type of arguments documented for the associated event, in order. This handler receives five arguments, four longs followed by a string.

Handling the LoginEnd Event

The **SDK** triggers the **LoginEnd** event when the login has completed. In the handler for this event, you can determine the result of the login based on the login result value passed to the handler. If the login result is **0** or **psLoginResultOk**, then the login succeeded.

Values for Login Result Argument of LoginEnd Event Handler

Constant	Value	Comments
psLoginResultOk	0	Constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.
psLoginResultFailed	1	

If the login is successful, you can take appropriate action in the **LoginEnd** event handler, such as displaying a window where the user can begin dictating reports:

```
function HandleLoginEnd(rc)
{
    if (rc == 0) //Using numeric value of the argument in JavaScript
    {
        try
        {
            window.open("ReportDictation.htm", "_self",
                        "height=600,width=400,status=yes", true);
        }
        catch(error)
        {
            if (error.number != -2146808285)
                alert(error.description);
        }
    }
}
```

If the login fails, you might suggest that the user contact a system administrator:

```
if (rc == 1)
{
    alert("Login failed. Contact your system administrator.");
}
```

Taking Startup Actions after Successful Login

The next few subsections show how you can take startup actions after the login **LoginEnd** event succeeds.

Loading Shortcuts and Words

Two actions you might want your application to take after the user has successfully logged in include loading shortcuts and loading custom words into the in-memory language model on the *Web Server*. You take these actions using the **LoadShortcuts()** and **LoadWords()** methods of the **PowerscribeSDK** object.

Loading Shortcuts

If you have dictated and stored custom shortcuts you want the speech recognition engine to use, you can load those shortcuts into your application's in-memory language model by calling **LoadShortcuts()** after the user logs in to the application. By default this method loads both shortcuts created by the logged in user and shortcuts created by the system administrator. To load all shortcuts, you call the method with no arguments (**True** by default):

```
pscribeSDK.LoadShortcuts();
```

To restrict shortcuts loaded to those that were created by a particular, you can pass **False** for the *IncludeGlobals* argument, followed by the login name of the user for the *author* argument, and finally the type of shortcuts to install, either voice (**psShortcutTypeVoice** or 1), text (**psShortcutTypeText** or 2), or both (**psShortcutTypeAll** or 0):

```
pscribeSDK.LoadShortcuts(false, "jsmith", psShortcutTypeAll);
```

If you have not created shortcuts, calling this method does not raise an exception; instead, the SDK ignores the method call.

After the shortcuts are installed they are enabled by default. For more detailed information about shortcuts, refer to [Chapter 12, Working with Shortcuts, Categories, and Words on page 279](#).

Loading Words

If you have (or a user has) dictated and stored custom words you want the speech recognition engine to use, you can load those words into your application's in-memory language model by calling **LoadWords()** after the user logs in to the application. By default this method loads both words created by the logged in user and words created by

the administrator of the system. To load all words, you can call the method without passing it any arguments or by passing it **True**:

```
pscribeSDK.LoadWords(true);
```

To restrict the words that load to those that were created by the logged in user, you can pass **False** for the *IncludeGlobals* argument:

```
pscribeSDK.LoadWords(false);
```

If you have no custom words, calling the method and passing it **False** does not raise an exception; instead, the *SDK* ignores the method call.

Creating and working with words is covered in more detail starting in [Overview of Creating and Using Words on page 293](#).

Retrieving Version Information

Other actions you might want to take in the **LoginEnd** event handler include checking the version of your application that is installed and checking the version of *PowerScribe SDK* and the **PowerscribeSDK.dll**.

Retrieving Major and Minor Version Numbers

To determine the major and minor versions of your application installed on the machine, you can use the **MajorVersion** and **MinorVersion** properties:

```
var majVersion;  
var minrVersion;  
  
majVersion = pscribeSDK.MajorVersion;  
minrVersion = pscribeSDK.MinorVersion;
```

You can use these properties to display the version information for the end user, who can indicate the version when requesting technical support.

Retrieving Version of PowerScribe SDK and DLL

To determine the version of *PowerScribe SDK* installed, you can use the **ProductVersion** property:

```
var versPSSDK;  
versPSSDK = pscribeSDK.ProductVersion;
```

To determine the version of **PowerscribeSDK.dll** installed, you can use the **FileVersion** property:

```
var versPSSDK_dll;  
versPSSDK_dll = pscribeSDK.FileVersion;
```

You can use these properties to be sure your application is compatible with the versions of *PowerScribe SDK* and the DLL installed.

Working with UserProfiles

You can use the **UserProfile** object to manipulate the logged in user from within your application. Most properties of the **UserProfile** object are read-only, but if you change one of the settable properties of the **UserProfile** object before you initialize the **PowerscribeSDK** object, those settings take effect once you do initialize the **PowerscribeSDK** object or when you call the **Update()** method of the **UserProfile** object.

Disabling Realtime Speech Recognition for Logged In User

To set up your application to carry out all speech recognition on the *Recognition Server* instead of having recognition carried out on the local workstation where the *SDK* application is installed, you disable realtime speech recognition for each user through the *SDK Administrator* application or you can disable it inside your application by modifying the user profile of the user after login.



Note: To disable realtime (local) speech recognition for the currently logged in speech recognition user of your application, you take this action before you call the **Initialize()** method of the **PowerscribeSDK** object. The change to the value does not take effect until you call that method.

To modify the logged in user's profile, you first create a **UserProfile** object using the **PowerscribeSDK** object:

```
objUserProfile = pscribeSDK.UserProfile;
```

Once you have the **UserProfile** object, you can turn off realtime speech recognition for that user by setting the **UseRealTimeRecognition** property of the object to **False**:

```
objUserProfile.UseRealTimeRecognition = false;
```

Now, the *SDK* automatically disables local speech recognition, and performs recognition on the *Recognition Server* in batch mode. (By default, this means that after the user presses the **TRANSCRIBE** button on the microphone, the *SDK* saves the report audio to the server, then queues it for recognition.) The **UseRealTimeRecognition** property setting is saved in the database, so after you shut down your application, the value is retained in the database.

By default, this property is set to **True**. When it is **True**, the *SDK* automatically loads the user speech information onto the machine where the user is running your application, and speech recognition occurs locally, in real time.

Setting User Preferences

In addition to setting the **UseRealTimeRecognition** property, you can set several other preferences for the currently logged in user by modifying the settings of these properties of the **UserProfile** object:

- **AutoPunctuation**
- **CorrectOptionPreference**
- **ShowDictationResultBox**
- **TextStreaming**

To change the settings for any of these property settings, you first create a **UserProfile** object using the **PowerscribeSDK** object:

```
objUserProfile = pscribeSDK.UserProfile;
```

Turning On/Off Automatic Punctuation

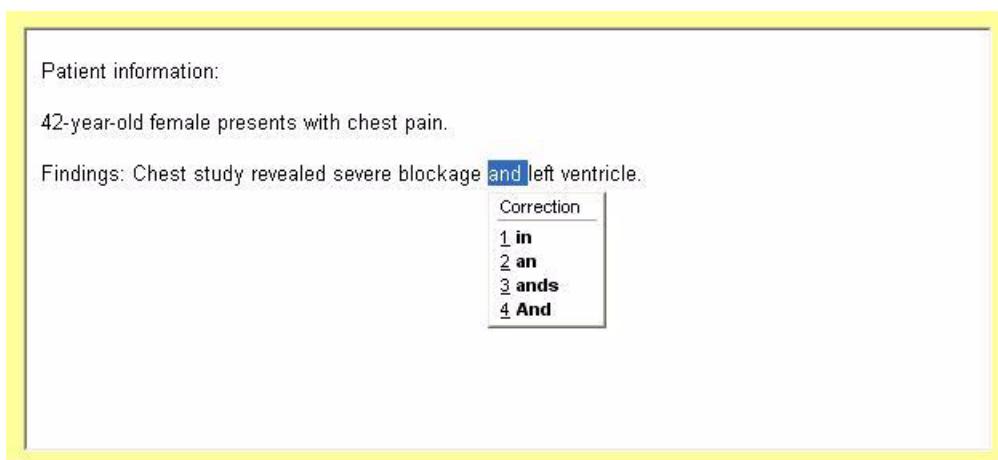
You can have the recognition engine automatically insert commas and periods when the speaker pauses, based on the length of the pause and the meaning of the sentence.

Once you have the **UserProfile** object, you can turn on automatic punctuation for the logged in user by setting the **AutoPunctuation** property of the object to **True**:

```
objUserProfile.AutoPunctuation = true;
```

Setting Correction and Selection Popup Box Options

You can have the recognition engine automatically pop up a **Correction** box that lists possible ways to correct recognition of a word you have selected in the body of the recognized text when you dictate the *Correct That* command. The popup box would appear as shown in the following illustration, where the user has selected the word *and* then dictated *Correct That*:



You can also have your application automatically select text when you dictate the *Select That* command. You can choose to have your application take action in response to either *Correct That* or *Select That*, or both.

Once you have the **UserProfile** object, you can set the **CorrectOptionPreference** property of the object to either **psCorrectOptNone** (0), **psCorrectOptSelectThat** (1), **psCorrectOptCorrectThat** (3), or **psCorrectOptBoth** (4). To have the **Correction** box pop up only when you dictate *Correct That*, you would set the property as follows:

```
var psCorrectOptCorrectThat = 3;  
objUserProfile.CorrectOptionPreference = psCorrectOptCorrectThat;
```

Your application could differentiate between the four possible settings by retrieving user preferences from a combo box in the GUI and executing logic to set the property:

```
<tr>  
    <td>CorrectOptions:</td>  
    <td><SELECT id="cbCorrectOptions" name="cbCorrectOptions">  
        <OPTION value="0" selected>None</OPTION>  
        <OPTION value="1">Select That</OPTION>  
        <OPTION value="2">Correct That</OPTION>  
        <OPTION value="3">Both</OPTION>  
    </SELECT>&nbsp;</td>  
</tr>  
  
try  
{  
    var val = cbCorrectOptions.options(cbCorrectOptions.selectedIndex) .value;  
    objUserProfile.CorrectOptionPreference = val;  
}  
catch(err)  
{  
    alert("UserProfile.CorrectOption Error: " + err.description);  
}
```

Turning On/Off Display of Dictation Result Box

You can have the recognition engine automatically display the recognized text in a **Result Box** like the one shown here.

Once you have the **UserProfile** object, you can turn on display of this box when the logged in user dictates by setting the **ShowDictationResultBox** property of the object to **True**:

```
objUserProfile.ShowDictationResultBox = true;
```

Patient information:
42-year-old female presents with chest pain.

Findings:
Chest study revealed severe blockage in left ventricle.

History: patient has no history of heart disease period a

Turning On/Off Streaming of Dictated Text

You can have the recognition engine automatically stream the text into the GUI rather than waiting for the user to select a transcribe button.

Once you have the **UserProfile** object, you can turn on streaming of text whenever this user dictates by setting the **TextStreaming** property of the object to **True**:

```
objUserProfile.TextStreaming = true;
```

Ensuring UserProfile Property Settings Take Effect

After you change the settings for one or more settable **UserProfile** properties, you should call the **Update()** method of the **UserProfile** object to ensure the changes take effect:

```
objUserProfile.Update();
```

Retrieving UserProfile Property Settings

When you set up a user through the *SDK Administrator* application provided with the product, you are actually setting several **UserProfile** properties whose values you can retrieve at runtime. For instance, you can determine whether the logged in user is set up as an administrator, or whether the user can author (dictate), transcribe, and/or correct reports.

Determining Whether User Is Administrator

To determine whether or not the logged in user is an administrator:

1. Retrieve the **UserProfile** object using the **PowerscribeSDK** object:

```
objUserProfile = pscribeSDK.UserProfile;
```

2. Retrieve the setting of the **Administrator** property of the **UserProfile** object:

```
boolAdminPriv = objUserProfile.Administrator;
if (boolAdminPriv == True)
{
    alert("The user is an Administrator");
}
else
{
    alert("The user is not an Administrator");
}
```

Determining Other Privileges of User

To determine other privileges assigned to the user:

1. To determine whether or not the user has privileges to create reports, you can retrieve the setting of the **Author** property of the **UserProfile** object:

```
boolAuthor = objUserProfile.Author;
```

2. Some users who author reports can transcribe their own audio and correct the resulting text, while others might send them to a transcriptionist/editor. Other users (usually transcriptionists) may not be able to author reports at all, but be set up to only transcribe and correct them. To determine whether or not the user has privileges to transcribe reports, you can retrieve the **Transcription** property of the **UserProfile** object:

```
boolTranscribe = objUserProfile.Transcription;
```

3. To determine whether or not the user has privileges to edit and correct reports, you can retrieve the setting of the **Correction** property of the **UserProfile** object:

```
boolCorrect = objUserProfile.Correction;
```

4. When you print out any information about the user, you can also retrieve the **UserName** property of the **UserProfile** object to include the user's full name (first and last names concatenated) in the statement:

```
if ((boolAuthor == true) && (boolTranscribe == false) && (boolCorrect ==  
false))  
{  
    alert(UserProfile.UserName " can author reports/send them to an editor");  
}
```

5. To print out information about the status of the user model for the currently logged in user, retrieve the **UserTrainingData** property of the **UserProfile** object. The data in this property is a string containing:

<VoiceModelStatus>: <UserID>

The **UserTrainingData** property's *VoiceModelStatus* might indicate one of the following the voice model statuses:

Voice Model Status String	Description
Initial Voice Model for user:	Voice model has been initialized.
Imported Voice Model for user:	Voice model has been imported.
Enroll Voice Model for user:	Voice model has been modified based on dictation of Enrollment page.
Basic Trained for user:	Voice model has been modified based on dictation of Basic training pages.

Voice Model Status String	Description
Extended Trained for user:	Voice model has been modified based on dictation of Extended training pages.
Adapted for user:	Voice model has been adapted based on dictation of a required number of reports.

You would retrieve the data by creating a string to hold the property setting returned:

```
var strTrainData = objUserProfile.UserTrainingData;
alert(objUserProfile.UserTrainingData);
```

The output for a user with an ID of 8 whose voice model has been adapted would be:

```
Adapted for user: 8
```

After you retrieve the training data, you could parse the string to retrieve the voice model status and apply that information in your application.

Releasing Event Maps before Exiting

Before you log out the user of the application, you release the event maps by calling **Unadvise()** on the **EventMapper** object and passing it the name of the object whose events you want to release:

```
SDKsink.Unadvise(pscribeSDK);
```

You usually call **Unadvise()** in your user defined shutdown procedure, inside a **Try/Catch** block:

```
function cleanup()
{
    try
    {
        if (SDKsink)
        {
            SDKsink.Unadvise();
            MicSink.Unadvise();
        }
    }
    catch(error)
    {
        alert("Cleanup " + error.description);
    }
}
```

Logging Out of the Application

You usually log out the user as part of a sequence of actions you take during a shutdown of the application. To log out the user, you use the **Logoff()** method instead of the **LogoffUser()** method (discussed earlier under [Creating the Login Functionality on page 30](#)).

You pass **Logoff()** the value **True** for its *bClearWaveCache* argument if you want it to delete all cached wave files after the logoff is successful. If you do not pass **True** for the argument, it defaults to **False**:

```
pscribeSDK.Logoff(True);
```

Disconnecting from the Server

To disconnect from the *PowerScribe SDK Web Server* after the user logs out of the application, you call the **Uninitialize()** method:

```
pscribeSDK.Uninitialize()
```

You can call the method inside a **Try/Catch** block to catch any exceptions that occur.

Code Summary for Initialize and Login

```

<HTML>
<OBJECT ID="pscribeSDK">
</OBJECT>

<HEAD>
<TITLE>Powerscribe SDK</TITLE>

<SCRIPT type="text/javascript">

var psAudioDevicePowerMicII = 7;

function InitializePS()
{
    try
    {
        // Instantiate PowerScribe SDK Object
        pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");

        // Initialize PowerScribe SDK Server

        var url = "http://server2/pscribesdk/";
        pscribeSDK.Initialize(url, "JScript_Demo");
        // The disableRealtimeRecognition argument of Initialize()
        //      defaults to False, turning on realtime recognition

        // Create EventMapper Objects for SDK Object, Microphone Object

        SDKsink = new ActiveXObject("PowerscribeSDK.EventMapper");
        MicSink = new ActiveXObject("PowerscribeSDK.EventMapper");

        // Map Handlers for Each Event to SDK EventMapper Object

        SDKsink.LoginEnd = HandleLoginEnd;
        SDKsink.ProgressMessage = HandleProgressMessage;
        SDKsink.ProgressMessageEx = HandleProgressMessageEx;
        SDKsink.DownloadProgressMessage = HandleDownloadProgressMessage;
        SDKsink.Advise(pscribeSDK);

        // Create Microphone Object, Map Handler to Mic EventMapper Obj

        MyMicrophone = pscribeSDK.Microphone;
        MicSink.Advise(MyMicrophone);
    }
    catch(error)
    {
        alert("InitializePowerscribe: " + error.number + error.description);
    }
    finally
    {
        DoLogin()
    }
}

```

```
        }

    }

function DoLogin()
{
    try
    {
        var userLoggedInSDK = false;
        try
        {
            pscribeSDK.Login(username, userPswd, true, true, psAudioDevicePowerMicII);
        }
        catch(error)
        {
            if (error.number == -2146778148)
            {
                // This error means user already logged in,
                // so set var to True
                userLoggedInSDK = true;
            }
            else
            {
                alert(error.description + "Error Number: " + error.number);
                return;
            }
        }
    }

    if (userLoggedInSDK) // If user is already logged in
    {
        //First log out the user
        PowerscribeSDK.LogoffUser(userName, userPswd, "JScript_Demo");

        //Then log in the user
        PowerscribeSDK.Login(userName, userPswd, true, true);
    }
    catch(error)
    {
        alert("DoLogin " + error.description + " Error:" + error.number);
    }
}

function HandleProgressMessage(msg)
{
    // Show message in status bar
    window.status = msg;
}

function HandleProgressMessageEx(percent, status)
{
    // Display progress bar showing percentage of files downloaded
    // Show status message in status bar
}
```

```
function HandleDownloadProgressMessage(totalTime, totalPercent, stepTime,
stepPercent, statusMsg)
{
    // Take appropriate action to track the download process

    if ((stepTime == 0) && (stepPercent == 100))
    {
        //Indicate last component downloaded.
    }
}

function HandleLoginEnd(rc)
{
    if (rc == 0)
    {
        try
        {
            window.open("ReportDictation.htm", "_self",
                        "height=600,width=400,status=yes", true);
        }
        catch(error)
        {
            if (error.number != -2146808285)
                alert(error.description);
        }
    }
    if (rc == 1)
    {
        alert("Login failed. Contact your system administrator.");
    }
}

function cleanup()
{
    try
    {
        if (SDKsink)
        {
            SDKsink.Unadvise();
            MicSink.Unadvise();
        }
    }
    catch(error)
    {
        alert("Cleanup " + error.description);
    }
}
```

```
function DoLogoff()
{
    try
    {
        pscribeSDK.Logoff();
    }
    catch(error)
    {
        alert(error.description);
    }
}

function UninitializePS()
{
    try
    {
        Cleanup();
        pscribeSDK.Uninitialize();
    }
    catch(error)
    {
        alert(error.description);
    }
}

function IsAdministrator()
{
    boolAdminPriv = objUserProfile.Administrator;
    if (boolAdminPriv == True)
    {
        alert("The user is an Administrator");
        return true;
    }
    else
    {
        alert("The user is not an Administrator");
        return false;
    }
}

</SCRIPT> </HEAD>

<BODY bgcolor="#fffff" language="javascript" onload="InitializePS()"
onunload="UninitializePS()">

<form>
...
<input language="javascript" type="button" name="btnLogin"
      value="Login" onClick="DoLogin()"><br>
<input language="javascript" type="button" name="btnLogout"
      value="Logout" onClick="DoLogoff()"><br>
...
</form> </BODY> </HTML>
```

Code Summary for UserProfile Preference Settings

```

<HTML>
<HEAD>
<META NAME="GENERATOR" Content="Microsoft Developer Studio">
<META HTTP-EQUIV="Content-Type" content="text/html; charset=iso-8859-1">
<TITLE>User Profile</TITLE>
</HEAD>
<BODY onload="setup()" bgColor="#CCFFCC">

<br>
<table align="center">
  <tr>
    <td>User Name:</td>
    <td><INPUT disabled name="user" style="FONT-STYLE: oblique"></td>
  </tr>
  <tr>
    <td>Training State:</td>
    <td>
      <INPUT disabled name="trstate" style="FONT-STYLE: oblique">
    </td>
  </tr>
  <tr>
    <td>Training Data:</td>
    <td>
      <INPUT disabled name="trdata" style="WIDTH: 170px;
      FONT-STYLE: oblique; HEIGHT: 22px" size="23">
    </td>
  </tr>
  <tr>
    <td>Correction:</td>
    <td>
      <INPUT type="checkbox" disabled name="chCorrection" value="">
    </td>
  </tr>
  <tr>
    <td>Transcription:</td>
    <td>
      <INPUT type="checkbox" disabled name="chTranscription" value="">
    </td>
  </tr>
  <tr>
    <td>Author:</td>
    <td>
      <INPUT type="checkbox" disabled name="chAuthor"
      value="">&ampnbsp&ampnbsp&ampnbsp&ampnbsp&ampnbsp&ampnbsp&ampnbsp&ampnbsp
    </td>
  </tr>
</table>

```

```
<tr>
    <td>Administrator:</td>
    <td>
        <INPUT type="checkbox" disabled name="chAdministrator" value="">
    </td>
</tr>
<tr>
    <td>Use Real Time Recognition:</td>
    <td><INPUT type="checkbox" name="chRealTime" value=""></td>
</tr>
<tr>
    <td>OldPassword:</td>
    <td>
        <INPUT type="password" name="oldpsd" style="FONT-STYLE: oblique">
    </td>
</tr>
<tr>
    <td>NewPassword:</td>
    <td>
        <INPUT type="password" name="newpsd" style="FONT-STYLE: oblique">
    </td>
</tr>
<tr>
    <td>ShowDictateResultBox:</td>
    <td>
        <INPUT id="cbShowDictResultBox" type="checkbox"
               name="cbShowDictResultBox">&nbsp;&nbsp;
    </td>
</tr>
<tr>
    <td>AutoPunctuation:</td>
    <td>
        <INPUT id="cbAutoPunctuation" type="checkbox"
               name="cbAutoPunctuation">&nbsp;&nbsp;
    </td>
</tr>
<tr>
    <td>TextStreaming:</td>
    <td><INPUT id="cbTextStream" type="checkbox" name="cbTextStream">
</td>
</tr>
<tr>
    <td>CorrectOptions:</td>
    <td><SELECT id="cbCorrectOptions" name="cbCorrectOptions">
        <OPTION value="0" selected>None</OPTION>
        <OPTION value="1">Select That</OPTION>
        <OPTION value="2">Correct That</OPTION>
        <OPTION value="3">Both</OPTION>
    </SELECT>&nbsp;</td>
</tr>
<tr>
    <td></td>
    <td></td>
</tr>
```

```

<tr>
    <td><INPUT style="FONT-STYLE: oblique" onclick="ChangePassword()" type="button" value="ChangePassword" name="changebtn">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</td>
    <td><INPUT style="FONT-STYLE: oblique" onclick="Apply()" type="button" value="Apply" name="Applybtn" id="Button1"></td>
</tr>
</table>

<SCRIPT>

// variables
var win = window.dialogArguments;
var gScript = win.top.script;
var pscribeSDK = gScript.pscribeSDK;
var objUserProfile;
function setup()
{
    var string = "";
    try
    {
        objUserProfile = pscribeSDK.UserProfile;
        BuildUI();
    }
    catch(err)
    {
        alert(err.description)
    }
}

function BuildUI()
{
    user.value = objUserProfile.UserName;
    trstate.value = objUserProfile.TrainingState;
    trdata.value = objUserProfile.UserTrainingData;
    chCorrection.checked = objUserProfile.Correction;
    chTranscription.checked = objUserProfile.Transcription;
    chAuthor.checked = objUserProfile.Author;
    chAdministrator.checked = objUserProfile.Administrator;
    chRealTime.checked = objUserProfile.UseRealTimeRecognition;
    cbShowDictResultBox.checked = objUserProfile.ShowDictationResultBox;
    cbAutoPunctuation.checked = objUserProfile.AutoPunctuation;
    cbTextStream.checked = objUserProfile.TextStreaming;
    cbCorrectOptions.selectedIndex = objUserProfile.CorrectOptionPreference;
}

function ChangePassword()
{
    try
    {
        objUserProfile.UpdateLogonPassword(oldpsd.value, newpsd.value);
    }
}

```

```
        catch(err)
        {
            alert(err.description)
        }
    }

function Apply()
{
    try
    {
        objUserProfile.UseRealTimeRecognition = chRealTime.checked;
    }
    catch(err)
    {
        chRealTime.checked = false;
        alert("UserProfile.UseRealTimeRecognition Error: " +err.description);
    }

    // Dictation Settings
    try
    {
        objUserProfile.ShowDictationResultBox = cbShowDictResultBox.checked;
    }
    catch(err)
    {
        cbShowDictResultBox.checked = false;
        alert("UserProfile.ShowDictationResultBox Error: " +err.description);
    }

    try
    {
        objUserProfile.AutoPunctuation = cbAutoPunctuation.checked;
    }
    catch(err)
    {
        cbAutoPunctuation.checked = false;
        alert("UserProfile.AutoPunctuation Error: " + err.description);
    }

    try
    {
        objUserProfile.TextStreaming = cbTextStream.checked;
    }
    catch(err)
    {
        cbTextStream.checked = false;
        alert("UserProfile.TextStreaming Error: " + err.description);
    }

    try
    {
        var val =cbCorrectOptions.options(cbCorrectOptions.selectedIndex) .value;
        objUserProfile.CorrectOptionPreference = val;
    }
}
```

```
        catch(err)
        {
            alert("UserProfile.CorrectOption Error: " + err.description);
        }

        try
        {
            objUserProfile.Update();
        }
        catch(err)
        {
            alert("UserProfile.Update Error:" + err.description);
        }
    }

    </SCRIPT>
</BODY>
</HTML>
```


Creating the Report

Objectives

Chapters in this book are designed to be read sequentially. If you have not read the preceding chapters, you should at least peruse them before proceeding with this one. This chapter covers how to create and manipulate a plain report in your *PowerScribe SDK* application:

- [Understanding Types of Reports](#)
- [Making *SDK* ActiveX Controls Available](#)
- [Adding an Editor to Your Application](#)
- [Adding a **LevelMeter** to Your Application](#)
- [Creating the **Report** Object](#)
- [Associating **PlayerEditor** and **LevelMeter** with the **Report** Object](#)
- [Setting Up **PlayerEditor** Event Handlers](#)
- [Locking and Activating the Report](#)
- [Saving the Report](#)
- [Handling the **SaveEndEx** Event](#)
- [Handling the **WaveFileProgress** Event](#)
- [Reopening and Editing the Report](#)
- [Handling the **LoadEnd** Event](#)
- [Approving or Unapproving the Report](#)
- [Deleting Reports](#)

Understanding Types of Reports

There are two types of reports, plain and structured.

A structured report contains predefined sections created with an XML schema. Each section contains a heading and one or more paragraphs, and can contain subsections. In a medical report, for instance, predefined headings might include:

- Patient Information
- Clinical Information
- Impressions
- Findings

By contrast, a plain report has no predetermined structure.

To quickly show how to create reports, this chapter covers the fundamentals of creating a plain report in an HTML file; the next chapter, [Chapter 5, Advanced Actions in Plain Reports](#), covers more advanced features of plain reports; and [Chapter 6, Creating Structured Reports](#), presents how to create structured reports.

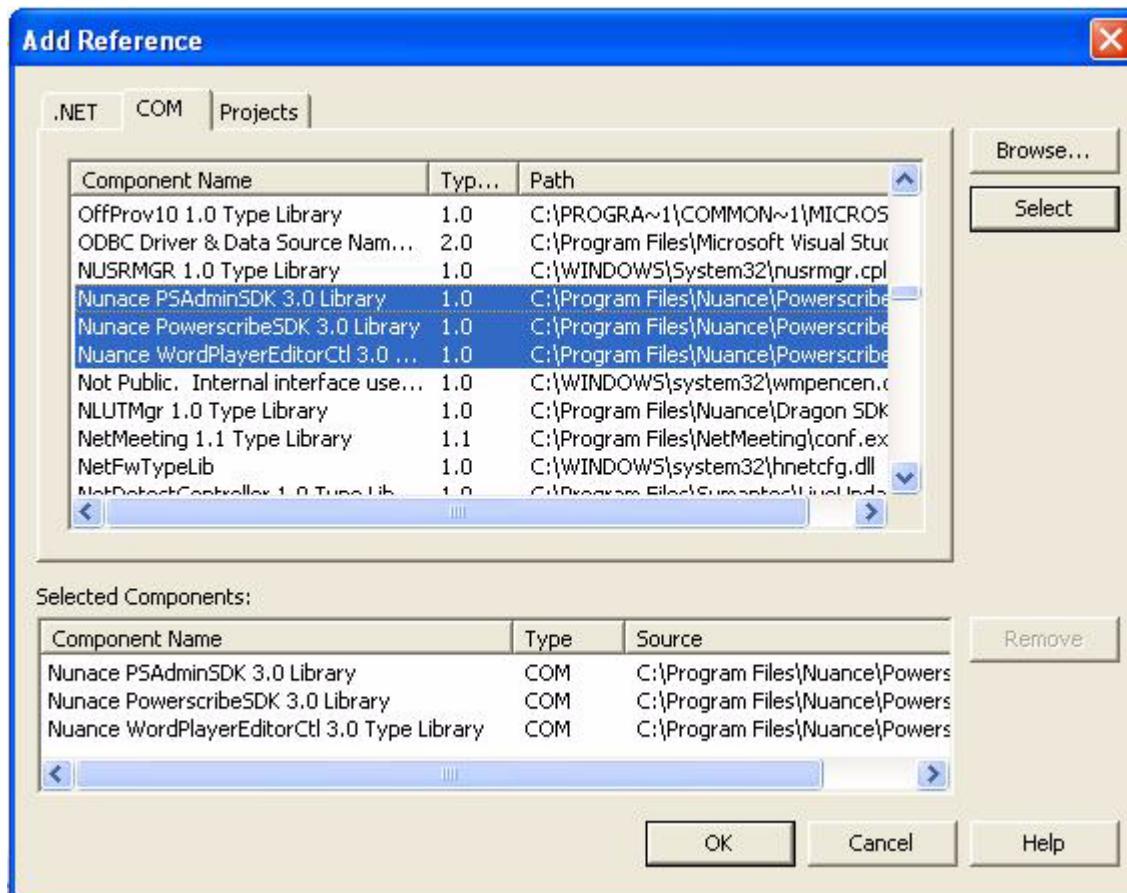
Making SDK ActiveX Controls Available

The *SDK* contains some ActiveX controls that you can use to add an editor where you view and revise the report text and a sound meter that indicates dictation is occurring. Before you can use these controls in a web application, you add them to the toolbox in Visual Studio. Later you can drag and drop the controls from the toolbox into the design view of the web page.

Creating References to Libraries

To create references to all the *PowerScribe SDK* libraries your application needs:

1. Go to the menu bar and selecting **Project => Add Reference**.
2. Click on the **COM** tab and select these libraries:
 - **Nuance PSAdminSDK 3.0 Library**
 - **Nuance PowerscribeSDK 3.0 Library**
 - **Nuance WordPlayerEditorCtl 3.0 Type Library**
3. Click the **Select** button and then when the libraries appear in the lower panel of the dialog, click **OK** to complete the process.

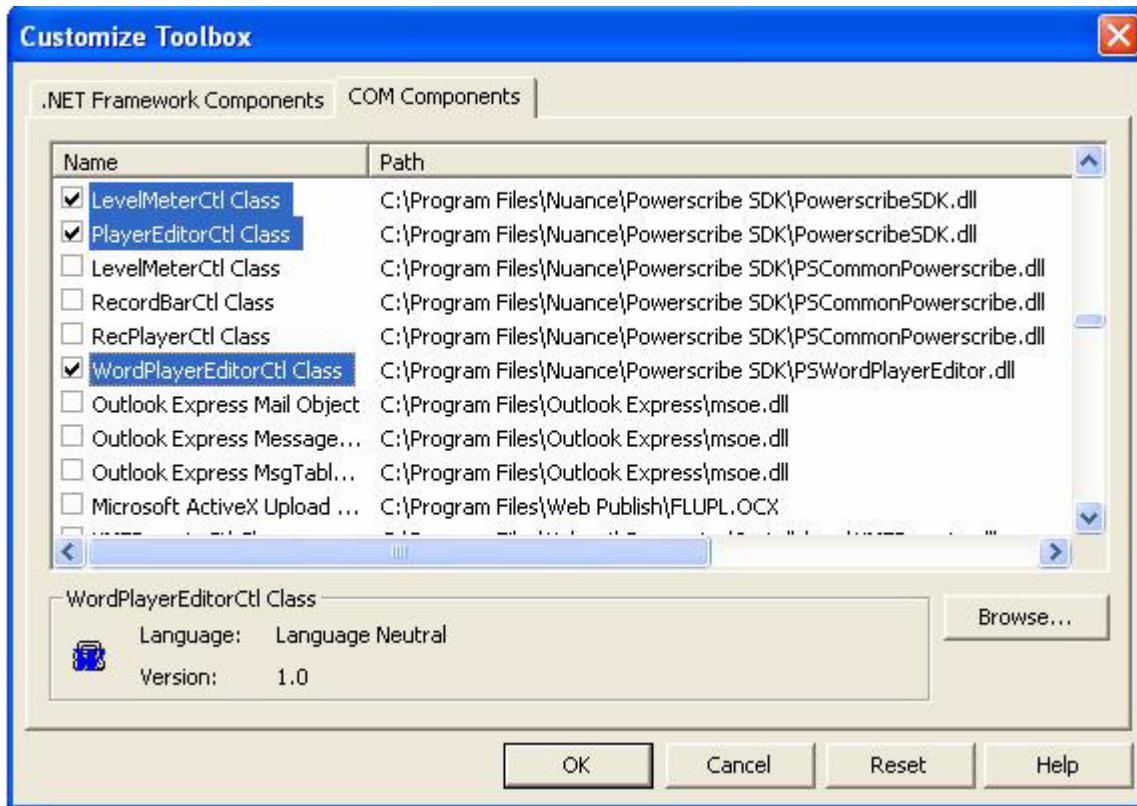


Adding ToolBox Items in Visual Studio

To add the controls to the HTML tab of the Visual Studio toolbox (for use in an HTML file of a JavaScript application):

1. Display the toolbox by going to the menu bar and selecting **View => Toolbox**.
2. Right-click in the open area of the **Toolbox** and select **Choose items...**
3. When the **Choose Toolbox Items** dialog box appears, click the **COM Components** tab.
4. In the list, click the check box for the **PlayerEditorCtl Class** or the **WordPlayerEditorCtl Class**. The **PlayerEditorCtl** ActiveX control creates an ordinary HTML editor in your web page; the **WordPlayerEditorCtl** ActiveX creates

a Word-based editor in your web page. The **WordPlayerEditorCtl** requires that both you (the developer) and the end user of your web page have Microsoft Word installed.



5. Check the check box for the **LevelMeterCtl** ActiveX control in the list and click **OK**.
6. To ensure the controls appear under the **HTML** tab, right-click in the toolbox and select **Show All**.

Once you have these controls available, you can add them to your application.

Adding an Editor to Your Application

Before you can create a new report, if you expect **Speech Recognition** (self-correcting) users to use your application to create and edit reports, the application needs an editor. In *PowerScribe SDK* you create an editor where you can see and edit transcribed text through the keyboard and/or play the audio back and edit text with voice commands using the microphone. Because it has the ability to manage audio as well as text, the *PowerScribe SDK* editor is called a *PlayerEditor*.

The **PlayerEditor** is not required if you expect to have strictly **Dictate Only** users using your application, but you might want to provide it to **Transcriptionist** users so that when they play back the audio, they can use the editor to generate and modify report text.

[Chapter 7, Working with Report Editors](#), tells you more about using the **PlayerEditor**.

An alternative to a **PlayerEditor** is a **CustomEditController** object. You can use a **CustomEditController** to turn any GUI component of your application into an *editor* that receives the recognized text resulting from dictation. The **CustomEditController** object lets you dictate text that may be for purposes other than producing a report. For more information, refer to [Chapter 8, Dictating Outside a Report](#).

Embedding PlayerEditor Control in Your Application

To add a **PlayerEditor** to your application, you use a **PlayerEditorCtl** ActiveX control.

To embed a PlayerEditorCtl ActiveX control in your program:

1. Select the **PlayerEditorCtl** control in the toolbox and drag and drop the control into your Visual Studio application form.
2. Assign the **PlayerEditor** a name. In the code shown here, the control has been named **objRptEditor**.

In the HTML source for a JavaScript application, the **PlayerEditorCtl** becomes embedded in a **<div>** block with the attribute settings shown below:

```
<div id="objRptEditor">
    <object id="PlayerEditor" ...
        classid="clsid:A470D150-C3FF-4584-9B0B-7D31AF4621C1">
    </object>
</div>
```

3. You then create a function that instantiates this type of ActiveX control:

```
function CreateObjectCtl(divID)
{
    var divRef = document.getElementById(divID);
    if (divRef)
    {
        divRef.innerHTML = divRef.innerHTML;
    }
}
```

4. Call the function to instantiate the **PlayerEditorCtl** control, passing it the value of the **id** attribute for the **<div>** block:

```
<script>
    CreateObjectCtl("objRptEditor");
</script>
```

Adding a LevelMeter to Your Application

A **LevelMeter** is a graphical bar that dynamically displays the perceived volume level of the dictation. The bar flickers to indicate that audio is being processed. Although your application does not have to contain a **LevelMeter**, the meter is useful to the person who is dictating.

Embedding LevelMeter Control in Your Application

To add a **LevelMeter** to your application, you use a **LevelMeterCtl** ActiveX control.

To embed a LevelMeterCtl ActiveX control in your program:

1. Select the **LevelMeter** control in the toolbox and drag and drop the control into your Visual Studio application form.
2. Assign the **LevelMeter** a name. In the code shown here, the control has been named **objLevelMeter**.

In the HTML source for a JavaScript application, the **LevelMeterCtl** becomes embedded in a <div> block with the attribute settings shown below:

```
<div id="objLevelMeter">
  <object id="LevelMeter" height="20" width="150" data="data:application/
    x-oleobject;base64,DfwvSUoKrU68UpZq4VaPzQADAACBDwAAEQIAAA==">
    <classid="clsid:492FFC0D-0A4A-4EAD-BC52-966AE1568FCD" viewastext>
  </object>
</div>
```

3. Call the same function to instantiate the **LevelMeterCtl** as you called to instantiate the **PlayerEditorCtl** control (under [Embedding PlayerEditor Control in Your Application on page 59](#)), passing it the value of the **id** attribute for the <div> block:

```
<script>
  CreateObjectCtl("objLevelMeter");
</script>
```

Creating the Report Object

To create a report in *PowerScribe SDK*, you use the **PowerscribeSDK** object and call its **NewReport()** method:

```
var objReport;
objReport = pscriveSDK.NewReport("ORD-120406");
```

If you want to create a plain report, you pass only one argument to the method, the *unique ID* to identify the report. To create a structured report, you would also pass a second argument, covered later in [Chapter 6, Creating Structured Reports](#).

The method returns a **Report** object that you can now use to take action on the report.

When you call this method, you should call it in a **Try/Catch** block to catch any of several exceptions that could occur:

```
try
{
    objReport = PowerscribeSDK.NewReport("ORD-120406");
}
catch(error)
{
    alert("New Report:" + error.description);
}
```

Among the exceptions you can handle in the **Try/Catch** block are errors caused by *PowerScribe SDK* not being initialized (-2146808287), no user being logged in (-2146808285), the unique ID already being in use (-2146808282), or an invalid unique ID being passed to the method (-2146808271).

```
catch(error)
{
    if (error.number == -2146808287) || (error.number == -2146808285)
    { // SDK not initialized or user not logged in
        alert("Cannot create new report until logged in. Please log in.")
    }

    if (error.number == -2146808282) || (error.number == -2146808271)
    { // The unique ID is already in use or not a valid report ID
        alert("Cannot use that ID. Please enter another report ID.")
    }
}
```

Once you have the **Report** object, you can call other methods to edit and manipulate the report. This manual presents more on those methods in [Chapter 5, Advanced Actions in Plain Reports](#).

Associating PlayerEditor and LevelMeter with the Report Object

Associating PlayerEditor with the Report Object

To associate the **PlayerEditor** in your application with the report object, you indicate the instance of the **PlayerEditor** object to use with the report by calling the **SetPlayerEditor()** method of the **Report** object:

```
objReport.SetPlayerEditor(objRptEditor);
```

This method returns the editor in the *PlayerEditor* argument you pass it. The **PlayerEditor** is automatically enabled by default. Later, you can choose to disable it after you close the report.

Associating LevelMeter with the Report Object

To associate the **LevelMeter** with the report, you indicate the instance of the **LevelMeter** object to use with the report by calling the **SetLevelMeter()** method of the **Report** object:

```
objReport.SetLevelMeter(objLevelMeter);
```

This method returns the **LevelMeter** in the *LevelMeter* argument you pass it. Although the **LevelMeterCtl** object has properties you could set, it is best to work with the default settings of this control.

Setting Up PlayerEditor Event Handlers

Some **PlayerEditorCtl** events that your application might want to handle are:

- **CtrlKeyPress(char)**—Occurs when the user presses **Ctrl** plus any key on the keyboard.
- **TextSelChanged(selStart, selEnd, style)**—Occurs when the user moves the cursor or selects text in the editor.
- **ButtonClick(psmkButton)**—Occurs when the user clicks one of the mouse buttons.

Your application can handle all these events, plus several more covered in [Chapter 7](#).

Mapping PlayerEditor Events

Because the **PlayerEditorCtl** is an ActiveX control, you do not need to map its events using the **EventMapper** object.

Handling PlayerEditor Events

You now create the handler functions for the **PlayerEditorCtl** events. Remember that you set this type of event handler up as you would for any ActiveX control in JavaScript, because the **PlayerEditor** is an ActiveX control. Pass each event handler name of the object that you set in the OBJECT tag as the value of the **for** attribute. Then, inside each handler, take appropriate action:

```
<script for="PlayerEditor" event="CtrlKeyPress(char)">
{
    // Handle Ctrl + Key
}
</script>
```

```

<script for="PlayerEditor" event="TextSelChanged(nStart,nEnd,bsStyle)">
{
    // Handle change in selected text
}
</script>

<script for="PlayerEditor" event="ButtonClick(psmkButton, xcoord, ycoord)">
{
    // Handle button click
}
</script>

```

To take action on text in the editor, you can use some methods of the **Report** object, explained in [Editing Report Content on page 76](#), but you mostly use methods and properties of the **PlayerEditor** object, covered in more detail in [Chapter 7, Working with Report Editors](#).

Locking and Activating the Report

Before you can accept dictation into the report, you must be sure it is locked, then activate it. Activating the report opens it for dictation or editing.

Locking the Report for Dictation or Editing

The reason you lock the report is to ensure no other user can dictate into the same report. You lock the report by calling the **ExclusiveLock()** method of the **Report** object and passing it **True** to lock it (**False** would unlock it):

```
objReport.ExclusiveLock(true);
```

If a user dictates into the report on one workstation, then goes to another workstation and tries to modify it there, the user is normally not allowed to work on the report on the second workstation. To let the user open and work on the same report on multiple workstations, your application can force an otherwise unavailable report to lock (be available to edit) on the second workstation by passing **True** for a second argument of the **ExclusiveLock()** method, called *ForceLock*:

```
objReport.ExclusiveLock(true, true);
```

 **Caution:** If your application does not save the audio and text of the report or modifications made to it on the first workstation, when the application locks the report on the second workstation using **ExclusiveLock()** with the ForceLock argument, additions and changes made on the first workstation will be lost.

Normally, speech recognition takes place on the workstation where your *PowerScribe SDK* application resides. This type of recognition is called *realtime* speech recognition. If instead of using realtime speech recognition, your application is having the *SDK* sends audio files to the *Recognition Server* to be transcribed, the file could be in the process of being transcribed on the *Recognition Server* when **ExclusiveLock()** tries to lock it. In this

situation, the method returns an error (-2146808270) that indicates the file is unavailable to be locked or activated.

Activating the Report

Once the report is locked, you can then activate it to receive dictation or modifications by using the **ActivateReport()** method of the **PowerscribeSDK** object and passing it the **Report** object:

```
try
{
    pscribeSDK.ActivateReport(objReport);
}
catch(error)
{
    alert(error.description);
    if (error.number == -2146808280)
        alert("You must have a PlayerEditor for this report.");
}
```

This method requires that the report have a **PlayerEditor** before it can be activated. If no **PlayerEditor** is available, the **ActivateReport()** method returns a -2146808280 error.

Another error **ActivateReport()** might return is 2146808279, indicating that the report has not been locked before attempting to activate it:

```
if (error.number == -2146808279)
    alert("You must lock the report before activating it.");
```

If your application has disabled speech recognition, forcing the *SDK* to send the file to the *Recognition Server* to be processed, **ActivateReport()** returns an error indicating the report is not available to activate, -2146808270:

```
if (error.number == -2146808279)
    alert("Report being processed on Recognition Server.");
```

Handling the ActivateEnd Event

Each time you call the **ActivateReport()** method of the **Report** object, to activate the report, the *PowerScribe SDK* application downloads report files from the server. When it finishes downloading those files, the **Report** object fires an **ActivateEnd** event.

After you receive the **ActivateEnd** event, the report is ready for the user to dictate into or edit.

Mapping the ActivateEnd Event

To use the **ActivateEnd** event, you should map it by creating an **EventMapper** object for it, then using that object to link the event handler to the corresponding event:

```
ReptSink = new ActiveXObject("PowerscribeSDK.EventMapper");
ReptSink.ActivateEnd = HandleActivateEnd;
```

And, finally, be sure to call the **Advise()** method of the **EventMapper** object and pass it the **Report** object (once you have created it):

```
ReptSink.Advise(objReport);
```

Handling the ActivateEnd Event

Inside the **ActivateEnd** event handler, you can take appropriate. Your handler needs to receive these three arguments in the order indicated—the HRESULT, the report ID, and any error message if an error has occurred:

```
function HandleActivateEnd(hresult, strID, strError)
{
    if (hresult < 0)
    {
        alert("Report " + strID + ": " + strError);
    }
    else
    {
        StartDictation();
    }
}
```

Once the **ActivateEnd** event occurs, the report has been activated and the user can begin dictation into or editing of the report. If an error has occurred, you can handle the error.

Adding Text/Changing the Report Content

Once the report has been activated, the user can dictate into the microphone or type into the editor to modify the report. The user might choose to:

- Dictate commands to edit the report
- Dictate new content for the report
- Dictate numbers into the report
- Spell words or names into the report

To set how the report should interpret the dictation, as commands, text, numbers, or spelled words, you set the **Mode** property of the **PowerscribeSDK** object.

Setting Dictation Mode

The default mode of the application is **Dictate** mode, a mode where the application receives what you speak into the microphone as dictation and receives a few commands that exist by default, such as *Bold That* or *Italicize That* to change the font of the selected text (see [Chapter 10](#) for information on **Command** mode).

Other modes you can put the application into include **DictateOnly**, where no commands are understood; **Number** mode, where digits are recognized; and **Letter** mode, where both letters and digits are recognized when the speaker spells the word. See the next table for a summary of the modes.

To put the application in **DictateOnly** mode, usually during initialization, you set the **Mode** property of the **PowerscribeSDK** object to **psSystemModeDictateOnly**:

```
var psSystemModeDictateOnly = 0;  
pscribeSDK.Mode = psSystemModeDictateOnly;
```

Settings for Mode Property of PowerscribeSDK Object

SystemMode Constant	Value	Explanation
psSystemModeDictateOnly	0	Dictate Only mode. No commands are recognized; all dictation is assumed to be speech and translated into text.
psSystemModeDictate	1	Dictate mode. Speech is translated to text and transcribed. No user defined commands are triggered in this mode. Only default grammar commands are triggered.
psSystemModeCommand	2	Command mode. User defined commands are triggered and all other speech, including default grammar commands, is discarded.
psSystemModeNumber	3	Number mode. Recognizes only numbers. Recognizes and transcribes numeric digits as the speaker says them.
psSystemModeLetter	4	Letter mode. Recognizes and transcribes letters and numbers as the speaker spells the word(s).

SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

Handling the ReportChanged Event

The first time you add audio or recognized text to a report or modify its text after either activating it or saving it, the **Report** object fires a **ReportChanged** event.

Your application can handle the **ReportChanged** event to take action in response to the report content changing.

Mapping the ReportChanged Event

To use the **ReportChanged** event, you should map it by creating an **EventMapper** object for it, then using that object to link the event handler to the corresponding event:

```
ReptSink = new ActiveXObject("PowerscribeSDK.EventMapper");
ReptSink.ReportChanged = HandleReportChanged;
```

And, finally, be sure to call the **Advise()** method of the **EventMapper** object and pass it the **Report** object (once you have created it):

```
ReptSink.Advise(objReport);
```

Handling the ReportChanged Event

Inside the **ReportChanged** event handler, you take appropriate action in response to the receipt of new or changed text in the report. The handler does not need to receive any arguments:

```
function HandleReportChanged()
{
    // Display the changed text
}
```

Some actions you might take in the **ReportChanged** event handler include:

- Highlighting the text that the user just changed.
- Saving the report.

Working with Recognized Text— Handling the RecognizeEnd Event

Each time the user clicks the microphone TRANSCRIBE button, the recognized text appears in the **PlayerEditor** and the **Report** object fires a **RecognizeEnd** event, which receives a string containing the path to the wave file that contains the audio corresponding to the recognized text.

Mapping the RecognizeEnd Event

To use the **RecognizeEnd** event, you should map it by creating an **EventMapper** object for it, then using that object to link the event handler to the corresponding event:

```
ReptSink = new ActiveXObject("PowerscribeSDK.EventMapper");
ReptSink.RecognizeEnd = HandleRecognizeEnd;
```

And, finally, be sure to call the **Advise()** method of the **EventMapper** object and pass it the **Report** object (once you have created it):

```
ReptSink.Advise(objReport);
```

Handling the RecognizeEnd Event

Your application can take action in response to the **RecognizeEnd** event, which you would set up as follows:

```
function HandleRecognizeEnd(string WaveFilePath)
{
    // Retrieve the audio for the text from wave file at WaveFilePath
    // Playback the audio in the wave file
}
```

Some actions you might take in the **RecognizeEnd** event handler include:

- Retrieving the path to the wave file.
- Retrieving the audio from the wave file.
- Playing the audio from the wave file.

Saving the Report

Each time the user makes a change, your application should save the report, regardless of whether the change is new dictation, transcription, or editing. The report remains open and the user can then dictate or edit the report and save it again. When the user has completed action on the report, you should save the report to the server, then handle the **SaveEndEx** event (which occurs only when you save to the server), where you can take actions to ensure the report is no longer open to edit.

When you save a report, you save all audio and text files associated with it. You can save those files as read-only files to the local machine's cache or save them to the server. When you call the **Save()** method of the **Report** object, you can indicate the files are to be saved to local cache by passing **True** to the method:

```
objReport.Save(true);
```

To indicate the files are to be saved to the server, either pass **False** to the method or omit the argument, as it defaults to **False**:

```
objReport.Save(false);
```

Your application can save only the current report and can save that report only if it is locked by the currently logged in user. Otherwise, this method returns an error code.

Handling the SaveEndEx Event

When your application calls the **Save()** method of the **Report** object or when a **Dictate Only** user presses the **Transcribe** button on the microphone, if the report is successfully saved to the server, the *SDK* fires the **SaveEndEx** event of the **Report** object.

Mapping the SaveEndEx Event

To use the **SaveEndEx** event, you should map it by creating an **EventMapper** object for it, then using that object to link the event handler to the corresponding event:

```
ReptSink = new ActiveXObject("PowerscribeSDK.EventMapper");
ReptSink.SaveEndEx = HandleSaveEndEx;
```

And, finally, be sure to call the **Advise()** method of the **EventMapper** object and pass it the **Report** object (once you have created it):

```
ReptSink.Advise(objReport);
```

Inside the handler, you take appropriate action, as outlined in the sections that follow.

Handling the SaveEndEx Event

The **SaveEndEx** event handler receives three arguments. The first is the result of the save operation in a result code argument. The result code can have several possible values:

Result Codes Passed to SaveEndEx Handler

Result Code	Description
0	S_OK
0x80040001	Read Failed
0x80040002	Convert Failed
0x80040003	Write Failed
0x80072EE1 to 0x80072F7C	Standard wininet errors indicated by last four digits. Refer to Wininet error codes and messages.

The two additional arguments that the **SaveEndEx** event handler receives are a string containing the unique ID of the report being saved and another string containing the textual description of the error that occurred if the save did not succeed. Your handler needs to receive these three arguments in the order indicated:

```
function HandleSaveEndEx(rc, rptUniqueID, errorStr)
{
}
```

If an error does occur, these extra arguments help you determine exactly what the error is and the particular report it affects. You can include the strings in an alert message:

```
alert(errorStr + "occurred for " + rptUniqueID);
```

In the **SaveEndEx** event handler, you should first check to see if the result code indicates the save was successful. If the save process in any way failed, you can determine whether or not the report audio and text can be recovered by checking to see if the **CanRecover** property of the **Report** object is **True**:

```
function HandleSaveEndEx(rc, rptUniqueID, errorStr)
{
    if (rc < 0)
    {
        if (objReport.CanRecover == true)
        {
            //Recover the report
        }
    }
}
```

Working with the **CanRecover** Property

If the *SDK* has set the **CanRecover** property to **True** for a report, your application should:

- Try saving it again in the current login session.
- If saving fails in the current session, after the user logs out and logs back in, the *SDK* automatically sets this property to **True** for the report. If the user logs in on the same machine where the report was last modified and where saving it failed, the *SDK* should recover the report automatically when **CanRecover** is **True**, but your application must save the report after the *SDK* recovers it to ensure no changes are lost.
- Display a message telling the user to recover the report on a particular machine.
(Remember that if the report was not saved on one machine, then edited on a second machine and saved there, the user can recover the report only on the second machine.)

Preventing Recovery from Occuring

You can prevent recovery from occurring by setting **CanRecover** to **False** before calling **Save()**. You might prevent recovery to avoid creating a situation where saving might put the copy saved on the server out of synch with the copy saved in local cache.

You can never set the property to **True**; only the *SDK* can set it to **True**.

Taking Actions After Successful Save

If the result code passed to the **SaveEndEx** event indicates the save was successful, you should take several other actions to finalize the report:

- Retrieve any audio still in the buffer
- Mark the report for adaptation

- Remove the displayed text from the editor
- Unlock the report.
- Disable the **PlayerEditor** until the user opens another report.

Retrieving Any Remaining Audio/Clearing Audio Buffer

After a call to the **Save()** method of the **Report** object, you can check to see if any audio remains in the buffer by calling the **HasAudioInMicBuffer()** method of the **Report** object:

```
if (objReport.HasAudioInMicBuffer)
{
    // Transcribe the audio
}
```

If you do not want to include the remaining audio in the report, you can call the **ClearAudioBuffer()** method of the **Report** object to erase it:

```
objReport.ClearAudioBuffer();
```

If you do not call this method to clear the audio buffer, the **Save()** method of the **Report** object will transcribe the remaining audio and add it to the report before saving it.

Marking the Report for Adaptation

Once a report is complete and has been saved, if you want the recognition server to use its audio to modify (*adapt*) the user speech acoustic models based on the report recognition results, you can mark the report for *adaptation*.

You take this action by calling the **MarkForAdaptation()** method of the **Report** object:

```
objReport.MarkForAdaptation();
```

Once your application has saved the report, the report is not yet closed; the report is still open as long as its text continues to display in the **PlayerEditor** and the report is still locked.

Removing Text from Editor Display

To remove the text from the **PlayerEditor** so that it does not appear that the report is still being edited, you can call the **ClearEditor()** method of the **Report** object:

```
objReport.ClearEditor();
```

The method removes the text from the editor text block, but keeps the report text intact in the database.

Unlocking the Report

To unlock the report, which officially closes it so that the application cannot take further action on it until it is opened (locked) again, call the **ExclusiveLock()** method of the **Report** object and pass it **False**:

```
objReport.ExclusiveLock(false);
```

Disabling the PlayerEditor

After the user has completed a report using the **PlayerEditor**, you might want to disable the **PlayerEditor** using the **EnablePlayer()** method and passing it **False** to disable it:

```
objRptEditor.EnablePlayer(false);
```

This action is entirely optional. If you do disable the editor, you can re-enable it by calling the method and passing it an argument of **True**.

Finalizing the SaveEndEx Event Handler

The complete **SaveEndEx** event handler code might look like the following:

```
function HandleSaveEndEx(rc, rptUniqueID, errorStr)
{
    if (rc != 0)
    {
        alert(error.rc)
        alert(errorStr + "occurred for " + rptUniqueID)

        if (objReport.CanRecover == true) {
            //Recover the report
        }
    }
    else
    {
        if (rc == S_OK)
        {
            objReport.MarkForAdaptation();
            objReport.ClearEditor();
            objReport.ExclusiveLock(false);
            objRptEditor.EnablePlayer(false); //optional
        }
    }
}
```

Handling the WaveFileProgress Event

Another event that the *SDK* fires when you save a file to the server with **Save()** or access the file on the server with **GetReport()** is the **WaveFileProgress** event. This event notifies the application about progress of the transfer of the file between the *SDK* machine running your application and the *PowerScribe SDK Web Server*.

Mapping the WaveFileProgress Event

Before you can use the **WaveFileProgress** event, you should map it by creating an **EventMapper** object, then using that object to link the event handler to the corresponding event:

```
ReptSink = new ActiveXObject("PowerscribeSDK.EventMapper");
ReptSink.WaveFileProgress = HandleWaveFileProgress;
```

And, finally, you call the **Advise()** method of the **PowerscribeSDK** object and pass it the **Report** object:

```
ReptSink.Advise(objReport);
```

Handling the WaveFileProgress Event

When the **WaveFileProgress** event fires, its handler receives four pieces of information:

- *Compressed*—Boolean set to **True** if the file is compressed, **False** if not.
- *Type*—Type of notification, indicated by a numeric value or its equivalent constant (see table that follows).

Notification Types WaveFileProgress Event Handler Receives

Constant	Value	Notes
psWaveFileProgressLoadInProgress	1	For this notification, <i>BytesRemaining</i> is the full size of the file being transferred.
psWaveFileProgressSaveInProgress	2	Wave file save is still in process.
psWaveFileProgressSaveEnd	3	Your application receives multiple SaveEnd notifications for a single file transfer; the first indicates the transfer is complete.
psWaveFileProgressLoadEnd	4	Wave file load has completed.
psWaveFileProgressFileSize	5	Wave file size has been determined.

- *BytesTransferred*—Number of bytes already transferred. Does not carry significant information for a **psWaveFileProgressSaveEnd** or **psWaveFileProgressLoadEnd** type notification. If the notification is a **psWaveFileProgressFileSize** notification, this argument contains the full size (number of bytes) of the file to be transferred.

- *BytesRemaining*—Number of bytes remaining to be transferred. Does not carry significant information if the notification is a **psWaveFileProgressSaveEnd**, **psWaveFileProgressLoadEnd**, or **psWaveFileProgressFileSize** notification. For a **psWaveFileProgressLoadInProgress** notification, this argument contains the total size of the file being transferred, instead of the number of bytes remaining to transfer.

You can take any action you consider appropriate in the **WaveFileProgress** event handler. Common actions taken here are to display and update a progress bar or status bar message.

```
function HandleWaveFileProgress(compress, type, bytesSent, bytesRemain)
{
    if (type == psWaveFileSaveInProgress)
        window.status = "Bytes Sent: "+bytesSent+"; Remaining: "+bytesRemain
    if (type == psWaveFileProgressFileSize)
        window.status = "Total of " + bytesSent + " Bytes Transferred"
}
```

Reopening and Editing the Report

To open the report that is already in the *PowerScribe SDK* database in the **PlayerEditor** for editing, you use the **GetReport()** method of the **PowerscribeSDK** object and pass it the *Unique ID* for the report:

```
objReport = pscribeSDK.GetReport("ORD-090206");
```

The method returns a **Report** object that you can then use to take other actions on the report.



Note: Be careful to check for the correct error code, as even when the error is essentially the same, the code returned may differ for another method. For instance, the **GetReport()** method returns an error code of **0x800A4E23** for **No user is logged in**, while the **LogoffUser()** method returns a different error code for the same error—**0x800A7535**.

Preparing to Edit the Report

Before you can work with the report, you must lock, then activate the report, just as you did after you created a new report in [Locking and Activating the Report on page 63](#).

If you proceed to lock and activate the report immediately, the **ActivateReport()** method of the **PowerscribeSDK** object goes to the server to retrieve the audio for the report, which slows down the process of opening the report. To save time expended retrieving data from the server, after your application retrieves the object using **GetReport()**, download all files associated with that report using the **LoadAll()** method of the **Report** object:

```
objReport.LoadAll();
```

Now, when you lock, then activate the report, the files are already on the local disk and your application can access them more rapidly.

Opening the Report in View-Only Mode

You can also make the report read-only by immediately calling the **SetReadOnly()** method of the **Report** object and passing it either **psReportReadonlyAll** (1) to prevent the user from recording new audio and changing the text or **psReportReadonlyAudio** (2) to prevent the user from recording new audio but allow changing the text:

```
objReport.SetReadOnly(psReportReadonlyAll);
```

The option to edit the text applies only to **Speech Recognition** users. **Dictate Only** users, who can never edit the text anyway, lose the ability to record new audio into the report when you pass either **psReportReadonlyAll** (1) or **psReportReadonlyAudio** to this method.

To later allow the user to edit the report, you can change its setting to allow both audio and text to be edited by passing the **SetReadOnly()** method the **psReportReadonlyNone** (0) constant:

```
objReport.SetReadOnly(psReportReadonlyNone);
```

Constants for Setting Report to Read-Only or Editable

Constant	Value	Notes
psReportReadonlyNone	0	Allow both report audio and text to be edited if user has appropriate privileges.
psReportReadonlyAll	1	Allow speech recognition users to dictate commands but not edit text or audio content. Do not allow dictate only users to record audio.
psReportReadonlyAudio	2	Allow speech recognition users to dictate commands but not new audio content. Allow the user to edit the text. Do not allow dictate only users to record audio.

Handling the LoadEnd Event

To ensure that **LoadAll()** downloads all report files without error, you can handle the **LoadEnd** event that the **Report** object triggers when **LoadAll()** completes.

Mapping the LoadEnd Event

To use the **LoadEnd** event, you should map it by creating an **EvenMapper** object for it, then using that object to link the event handler to the corresponding event:

```
ReptSink = new ActiveXObject("PowerscribeSDK.EventMapper");
ReptSink.LoadEnd = HandleLoadEnd;
```

And, finally, be sure to call the **Advise()** method of the **EventMapper** object and pass it the **Report** object (once you have created it):

```
ReptSink.Advise(objReport);
```

Handling the LoadEnd Event

Inside the **LoadEnd** event handler, you take appropriate action. Your handler needs to receive these three arguments in the order indicated—the HRESULT, the report ID, and any error message if an error has occurred:

```
function HandleLoadEnd(hresult, strID, strError)
{
    if (hresult > 0)
    {
        alert("Report " + strID + ": " + strError);
    }
    else
    {
        // Lock and activate report
    }
}
```

One action you might take in the **LoadEnd** event is to ensure that downloading of the report files succeeded by testing the HRESULT argument. As shown above, you can print the report ID and associated error message to help the user respond if an error occurs.

Editing Report Content

You handle most editing of report text using the **PlayerEditor** object, its methods, and its properties. However, *PowerScribe SDK* does provide a few methods of the **Report** object that you can also use to manipulate or work with the report content.

To insert a standard piece of text into the report currently displaying in the **PlayerEditor**, you might first select the location where you want to insert that text. You can select the location by indicating the character positions you want to overwrite. To overwrite the first 10 characters of the report, you would select characters **0** through **9**. To, for instance, insert text at the very top of the report, you could select characters **0** through **0** by calling the **SelectText()** method of the **Report** object and passing it **0** for the *start* character and **0** for the *end* character:

```
objReport.SelectText(0, 0);
```

You can then call the **InsertText()** method of the **Report** object and pass it the string of text to insert, such as a heading common to all reports:

```
objReport.InsertText("Merrimack County Hospital \r");
objReport.InsertText("PO Box 2080, Merrimack, NH 00001 \r \r");
```

The **InsertText()** method replaces the selected text with the string you pass it. In a case where character position 0 to 0 was selected, the method inserts the heading at the top of the report.

If you use **SelectText()** to select text you'd like to remove, you can remove it by calling the **DeleteSelText()** method:

```
objReport.DeleteSelText();
```

Approving or Unapproving the Report

When the user indicates the report content is correct and complete, called *approving* the report, your application can mark the report approved by calling the **MarkForAdaptation()** method of the **Report** object:

```
objReport.MarkForAdaptation();
```

You can still edit the report while it is in the adaptation queue, unless your application's logic denies the option.

Later, if for any reason the user decides to make changes to the report, he or she can make those changes without removing the report from the adaptation queue.

However, if you want to remove the report from the adaptation queue while it is being modified, you should have your application call the **MarkUnapproved()** method of the **Report** object:

```
objReport.MarkUnapproved();
```

This method removes the report from the adaptation queue as well as marking the report unapproved. If your application denies the option of editing the file while it is in the adaptation queue, it should allow editing again after the report becomes unapproved.

Deleting Reports



Caution: Your application should not allow users to delete a report unless there is a compelling reason to delete it. You should restrict who is allowed to delete reports.

If a compelling situation exists where a report must be deleted, if a user meets strict access requirements, you can have your application delete the report by calling the **DeleteReport()** method of the **PowerscribeSDK** object and passing it the unique ID for the report:

```
pscribeSDK.DeleteReport(reportID);
```

Exporting and Importing Reports to or from Disk

Exporting Standard Reports to Disk

When you are working with a standard (non-Word) **PlayerEditor**, the reports you create can be saved as **.rtf** files that you can send in email or open in Microsoft Word outside the application.

To export an **.rtf** report, it does not have to be open in a **PlayerEditor**, but you must have already saved it to the database using the **Save()** method. You then use the **GetDocFile()** method of the **Report** object.

Call **GetDocFile()** and pass it the path to save the report to, then the *psFileType*, always **psFileTypeRTF** for a standard report document:

```
var psFileTypeRTF = 1;  
objReport.GetDocFile("C:\Reports\" + uniqueID + ".rtf", psFileTypeRTF);
```

You should set the permissions on the file to protect it from being removed from the disk.

Exporting Word Reports to Disk

When you are working with a Word **PlayerEditor**, the reports you create can be saved as **.doc** files that you can work with in a Word **PlayerEditor**, but that you can also send in email or open in Microsoft Word outside the application.

To export a Word report, it does not have to be open in a Word **PlayerEditor**, but you must have already saved it to the database using the **Save()** method. You then use the **GetDocFile()** method of the **Report** object.

Call **GetDocFile()** and pass it the path to save the report to, then the *psFileType*, always **psFileTypeWord** for a Microsoft Word document:

```
var psFileTypeWord = 0;  
objReport.GetDocFile("C:\Reports\" + uniqueID + ".doc", psFileTypeWord);
```

You should set the permissions on the file to protect it from being removed from the disk.

Importing Standard Reports from Disk

Once you have exported an **.rtf** file, you can import it into your database and later open it in a **PlayerEditor**. To import the **.rtf** file, you use the **SetDocFile()** method. You pass the method the path to retrieve the report from, then the *PSFileType* for a standard report document, **psFileTypeRTF**:

```
objReport.SetDocFile("C:\Reports\" + uniqueID + ".rtf", psFileTypeRTF);
```

Importing Word Reports from Disk

Once you have exported a **.doc** file, you can import it into your database and later open it in the Word **PlayerEditor**. To import the **.doc** file, you use the **SetDocFile()** method. You pass the method the path to retrieve the report from, then the *PSFileType* for a Word document, **psFileTypeWord**:

```
objReport.SetDocFile("C:\Reports\" + uniqueID + ".doc", psFileTypeWord);
```

Code Summary

```
<HTML> <OBJECT ID="pscribeSDK"></OBJECT>
<HEAD> <TITLE>Powerscribe SDK</TITLE>
<!-- This code summary is not a complete program; it assumes
inclusion of required code from the previous chapter -->
<SCRIPT type="text/javascript">

//Create Event Mappers

PlayEditSink = new ActiveXObject("PowerscribeSDK.EventMapper");
ReptSink = new ActiveXObject("PowerscribeSDK.EventMapper");

//Use Event Mappers to Map Events to Their Handlers

ReptSink.ActivateEnd = HandleActivateEnd;
ReptSink.LoadEnd = HandleLoadEnd;
ReptSink.ReportChanged = HandleReportChanged;
ReptSink.SaveEndEx = HandleSaveEndEx;
ReptSink.WaveFileProgress = HandleWaveFileProgress;

//Create global report object variable
var objReport;

function CreateNewReport()
{
try
{
    objReport = PowerscribeSDK.NewReport("ORD-120406");
}
catch(error)
{
    if (error.number == -2146808287) || if (error.number == -2146808285)
    { // SDK not initialized or user not logged in
        alert("Cannot create a new report until you log in. Pls log in.")
    }

    if (error.number == -2146808282) || if (error.number == -2146808271)
    { // The unique ID is already in use or not a valid report ID
        alert("Cannot use that ID. Please enter another report ID.")
    }
}
```

```
        }

    }

    try
    {
        objReport.SetPlayerEditor(objRptEditor);

        objReport.SetLevelMeter(objLevelMeter);

        CreateObjectCtl(objRptEditor);
        CreateObjectCtl(objLevelMeter);
        objReport.ExclusiveLock(true);
        ReptSink.Advise(objReport); //Activate event handlers for rpt events
        pscribeSDK.ActivateReport(objReport);
    }
    catch(error)
    {
        alert(error.description);
        if (error.number == -2146808280)
            alert("You must have a PlayerEditor for this report.");

        if (error.number == -2146808279)
            alert("You must lock the report before activating it.");

        //Occurs with ActivateReport if rpt is being processed on Rec Server
        if (error.number == -2146808270)
            alert("Report being processed on Recognition Server.");
    }
}

function EditExistingReport()
{
    try
    {
        objReport = pscribeSDK.GetReport("ORD-090206");
        objReport.LoadAll();
    }
    catch(error)
    {
        alert(error.description);
    }
}

function HandleLoadEnd(hresult, strID, strError)
{
    if (hresult < 0)
    {
        alert("Report " + strID + ":" + strError);
    }
    else
    {
```

```
try
{
    objReport.SetPlayerEditor(objRptEditor);
    objReport.SetLevelMeter(objLevelMeter);
    objReport.ExclusiveLock(true);
    ReptSink.Advise(objReport); //Activate evt handlers of rpt evts
    pscribeSDK.ActivateReport(objReport);
}
catch(error)
{
    alert(error.description);
    if (error.number == -2146808280)
        alert("You must have a PlayerEditor for this report.");
    if (error.number == -2146808279)
        alert("You must lock the report before activating it.");
    if (error.number == -2146808270) //From ActivateReport only
        alert("Report being processed on Recognition Server.");
}

}

function HandleActivateEnd(hresult, strID, strError)
{
    if (hresult < 0)
    {
        alert("Report " + strID + ": " + strError);
    }
}

function HandleReportChanged()
{
    // Display the changed text
}

function HandleSaveEndEx(rc, rptUniqueID, errorStr)
{
    if (rc < 0)
    {
        alert(error.rc + ": " + errorStr + "occurred for " + rptUniqueID)
        if (objReport.CanRecover == true)
        {
            //Recover the report
        }
    }
    else
    {
        if (rc == 0)
```

```
{  
    if (objReport.HasAudioInMicBuffer)  
    {  
        // Transcribe the audio  
        objReport.ClearAudioBuffer();  
        objReport.Save(false);  
    }  
}  
else  
{  
    SaveAndApprove()  
}  
}  
  
}  
  
function HandleWaveFileProgress(compress, type, bytesSent, bytesRemain)  
{  
    if (type == psWaveFileSaveInProgress)  
        window.status = "Bytes Sent: "+bytesSent+"; Remaining: "+bytesRemain  
    if (type == psWaveFileProgressFileSize)  
        window.status = "Total of " + bytesSent + " Bytes Transferred"  
}  
  
}  
  
function SaveReport()  
{  
    objReport.Save(false);  
}  
function SaveAndApprove()  
{  
    objReport.MarkForAdaptation();  
    objReport.ClearEditor()  
    objReport.ExclusiveLock(false);  
}  
function UnapproveAndRevise()  
{  
    objReport.MarkUnapprove();  
    EditExitingReport();  
}  
  
function HandleCtrlKeyPress(char)  
{  
    // Handle the keystroke  
}
```

```
function HandleTextSelChanged(nStart, nEnd, bsStyle)
{
    // Handle the selected text
}

function HandleButtonClick(psmkButton, xcoord, ycoord)
{
    // Handle the mouse button click
}

function CreateObjectCtl(divID)
{
    var divRef = document.getElementById(divID);
    if (divRef)
    {
        divRef.innerHTML = divRef.innerHTML;
    }
}

function SetReadOnly(nReportReadonly)
{
    try
    {
        objReport.SetReadOnly(nReportReadonly);

        if (nReportReadonly == 1)
        {
            // Disable PlayerEditor
            objPlayerEditor.EnablePlayer(false);
        }
        if (nReportReadonly != 1)
        {
            // Enable PlayerEditor
            objPlayerEditor.EnablePlayer(true);

            if (nReportReadonly == 2)
            {
                alert("Audio Receiving Commands Only");
            }
        }
    }
    catch (e)
    {
        alert(e.description);
    }
}
```

```
//Expanded version of Cleanup() function from preceding chapter

function Cleanup()
{
    try
    {
        //Release EventMapper Objects
        SDKsink.Unadvise();
        MicSink.Unadvise();
        PlayEditSink.Unadvise();
        ReptSink.Unadvise();
    }
    catch(error)
    {
        alert("Cleanup " + error.description);
    }
}

</SCRIPT> </HEAD>

<BODY bgcolor="#fffff" language="javascript" onload="InitializePS()"
onunload="UninitializePS()">

<form>
...
<div id="objRptEditor">
    <object id="PlayerEditor" style="WIDTH: 605px; HEIGHT: 265px; BACK-
    GROUND-COLOR: #EFE7CE" data="data:application/x-oleobject;base64,UNFwpP/
    DhEWbC30xr0YhwRAHAAJAAAAAAAAAAAAAAA=">
        classid="clsid:A470D150-C3FF-4584-9B0B-7D31AF4621C1" viewastext>
    </object>
</div>

<div id="objLevelMeter">
    <object id="LevelMeter" height="20" width="150" data="data:application/
        x-oleobject;base64,DfwvSUoKrU68UpZq4VaPzQADAACBDwAAEQIAAA==">
        classid="clsid:492FFC0D-0A4A-4EAD-BC52-966AE1568FCD">
    </object>
</div>

<input value="Create Report" type="button" name="btnCreateRept" language=
    "javascript" onclick="CreateNewReport()" style="FONT-STYLE: bold">
<input value="Edit Report" type="button" name="btnEditRept" language=
    "javascript" onclick="EditExistingReport()" style="FONT-STYLE: bold">
<input value="Save" type="button" name="btnSave" language="javascript"
    onclick="SaveReport()" style="FONT-STYLE: bold">
...

```

```
<input value="Save & Approve" type="button" name="btnApprove"
       language="javascript" onclick="SaveAndApprove()" style="FONT-STYLE:
bold">

<input value="Unapprove & Revise" type="button" name="btnApprove"
       language="javascript" onclick="UnapproveAndRevise()" style="FONT-
STYLE: bold">

<table border="0" cellpadding="0" cellspacing="3" id="Table4">

  <tr>

    <td>

      <input value="Report Read-Only" type="button" name="btnReadOnly"
             language="javascript" disabled onclick="SetReadOnly(1)"
             style="FONT-STYLE: oblique" ID="btnReadOnly">
    </td>

    <td>

      <input value="Command-Only Audio" type="button"
             name="btnReadOnlyAudio" language="javascript" disabled
             onclick="SetReadOnly(2)" style="FONT-STYLE: oblique"
             ID="btnReadOnlyAudio">
    </td>

    <td>

      <input value="Report Read-Write" type="button" name="btnReadOnlyOff"
             language="javascript" disabled onclick="SetReadOnly(0)"
             style="FONT-STYLE: oblique" ID="btnReadOnlyOff">
    </td>

  </tr>
</table>
</form>
</body>
```


Advanced Actions in Plain Reports

Objectives

This chapter covers how to take advanced actions on plain reports:

- [Exporting Reports](#)
- [Importing Reports from *Enterprise Express Speech* or Another *PowerScribe SDK* Server](#)
- [Pulling Together Code for Exporting and Importing Report](#)
- [Importing Report from Third Party Device and Recognizing Audio Immediately](#)
- [Spell Checking Report Text](#)
- [Disabling Real Time Speech Recognition](#)
- [Sending Reports to Recognition Server for Batch Recognition](#)
- [Handling the **QueuedForRecognition** Event](#)
- [Using *Recognition Accuracy Feedback \(RAF\)* Tool](#)

Exporting Reports

To export a report to be moved to another *PowerScribe SDK* server, you use methods to:

- Retrieve the **Report** object using the **GetReport()** method of the **PowerscribeSDK** object
- Optionally associate it with a **PlayerEditor** (to export from editor rather than database)
- Export existing report text marked with its status
- Export existing report audio
- Export a file that correlates the text to the audio (a concordance file)

Retrieving Text of the Report from Server

After you create the **Report** object (see [Creating the Report Object](#) in [Chapter 4](#)), to retrieve all text of a report from the database and place it in a string, you create a variable to receive returned text and then call the **GetText()** method of the **Report** object. You pass the method two arguments—the first the *TextType*, either preliminary or corrected, the second argument the *TextFormatType* of the report, either plain, XML, RTF, or unknown:

```
var rptText;  
rptText = objReport.GetText(psTextTypeCorrected, psTextFormatTypePlain);
```

TextType Enum Values

TextType	Value	Description
psTextTypePreliminary	1	Text has been recognized, but is still raw; does not contain expanded shortcuts and is not used for adaptation.
psTextTypeCorrected	2	Text has been corrected in the PlayerEditor . Default.

SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

If you do not know the format of the text, you can pass **psTextFormatTypeUnknown** for the *TextFormatType* argument, and *PowerScribe SDK* then determines the format for you.

TextFormatType Enum Values

TextFormatType	Value	Description
psTextFormatTypeUnknown	0	Text that might or might not have formatting.
psTextFormatTypePlain	1	Text with no formatting. Default.
psTextFormatTypeXML	2	Report content containing XML formatting.
psTextFormatTypeRTF	3	Report content containing text in rich text format (RTF).

By default, the *TextType* is **psTextTypeCorrected** and the *TextFormatType* is plain, so you can make the same call shown earlier with no arguments:

```
rptText = objReport.GetText();
```

Retrieving Text of the Report from Editor



Note: *GetText()* goes to the server to get the text. If the report is activated and displaying in the editor, you should call **GetEditorText()** instead, because **GetEditorText()** retrieves the text from the editor, producing better performance than going to the database would.

Before you can retrieve report text from an editor, you should create a **PlayerEditor** object and associate the editor with the report, as you did in [Adding an Editor to Your Application on page 58](#) and [Associating PlayerEditor and LevelMeter with the Report Object on page 61](#).

To retrieve all text of an active report from the **PlayerEditor** and place it in a string, you create a variable to receive the returned text, then call the **GetEditorText()** method of the **Report** object. The method takes an argument called *TextFormatType*, set to **psTextFormatTypePlain** by default:

```
var rptText = objReport.GetEditorText(psTextFormatTypePlain);
```

You can also retrieve a plain report as an RTF file by passing **psTextFormatTypeRTF** for the format type:

```
var rptText = objReport.GetEditorText(psTextFormatTypeRTF);
```

Retrieving Audio of the Report

To retrieve (export) the audio file for a report from the server and download it to any destination you want, you call the **GetWaveFile()** method of the **Report** object and pass it two arguments, the full path to the file to store the audio in as the first argument and whether to compress the audio or not in the second argument (see table):

```
objReport.GetWaveFile(C:\report.wav, psWaveFileTypeCompressed);
```

WaveFileType Enum Values

WaveFileType	Value	Description
psWaveFileTypeNonCompressed	1	Indicates to return the file uncompressed. Default.
psWaveFileTypeCompressed	2	Indicates to return the file compressed.
SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.		

You can pass a third (optional) argument to **GetWaveFile()** to indicate you want to export the wave file from the local machine's cache instead of from the server. Passing **True** for this argument indicates you want to export the file from the local machine's cache, while

passing **False** or not passing the argument indicates the file should be exported from the server:

```
objReport.GetWaveFile(C:\report.wav, psWaveFileTypeCompressed, true);
```

Once you have the audio file, you can use **Microphone** object commands to work with the **Report** audio wave file, including **Play()**, **Rewind()**, **Forward()**, **Stop()**, **GoToStart()**, and **GoToEnd()**. For details, refer to [Advancing, Rewinding, Playing, and Stopping Audio on page 233](#).

Retrieving Associated Concordance File

To export the entire report from the database, you need to also export a proprietary *PowerScribe* format file, called a *concordance* file, that aligns the text with the location of each word in the audio when the audio plays back in the **PlayerEditor**. To export this file, you call the **GetConcordanceFile()** method of the **Report** object. You pass the method two arguments, the path to export the concordance file to, and the format of the concordance file, **psConcordanceFilePowerScribe** (this format is the default, as this argument is optional):

```
objReport.GetConcordanceFile("C:\report.ast", psConcordanceFilePowerScribe);
```

The root name of the concordance file always matches the root name of its audio source file. The extension on the *PowerScribe SDK* concordance file name is **.ast**.

ConcordanceFileType Enum Values

ConcordanceFileType	Value	Description
psConcordanceFilePowerScribe	0	File from another <i>PowerScribe SDK</i> server. Default.
psConcordanceFileExSpeech	1	File from an <i>Enterprise Express Speech</i> server.

SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

You can pass a third (optional) argument to **GetConcordanceFile()** to indicate you want to export the concordance file from the local machine's cache instead of from the server. Passing **True** for this argument indicates you want to export the file from the local machine's cache. Passing **False** or not passing the argument indicates the file should be exported from the server:

```
objReport.GetConcordanceFile("C:\report.ast", psConcordanceFilePowerScribe, true);
```

Importing Reports from Enterprise Express Speech or Another PowerScribe SDK Server

To import a report that you have exported from another *PowerScribe SDK* server or from an *Enterprise Express Speech* server, you use a series of methods to:

- Create the **Report** object using the **NewReport()** method of the **PowerscribeSDK** object
- Lock the report
- Optionally associate the report with a **PlayerEditor** (to later display it in editor)
- Import existing report text and indicate its status
- Import existing report audio
- Import a file that correlates the text to the audio (a concordance file)
- Activate the report

Locking the Report

Before you begin importing the report to the server, you should lock it immediately, to ensure no one else locks the file.

Importing Existing Plain or XML Text File

After you create the report object (see [Creating the Report Object](#) in [Chapter 4](#)), to import existing text into that report in the *PowerScribe SDK* database, you use the **SetTextEx()** method and pass it the *TextType* (see table), the *TextFormatType* (see table), and the actual text (in either a string variable or a literal string).

TextType Enum Values

TextType	Value	Description
psTextTypePreliminary	1	Text has been recognized, but is still raw; does not contain expanded shortcuts and is not used for adaptation.
psTextTypeCorrected	2	Text has been corrected in the PlayerEditor . Default.

For instance, to import report text from a string called **fullReportString** and indicate the report text has been corrected, you would call **SetTextEx()** and pass it first the **psTextTypeCorrected** constant, second the *TextFormatType*, and finally the string

variable containing the report text (you might set this variable by retrieving text from a file):

```
psTextTypeCorrected = 2;  
psTextFormatTypePlain = 1;  
  
objReport.SetText(psTextTypeCorrected, psTextTypePlain, fullReportString);
```

TextFormatType Enum Values

TextFormatType	Value	Description
psTextFormatTypeUnknown	0	Text that might or might not have formatting.
psTextFormatTypePlain	1	Text with no formatting. Default.
psTextFormatTypeXML	2	Report content containing XML formatting.

SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

After the import is successful, the report text is flagged as corrected in the database.

Importing Existing Audio

If you have an audio (wave) file in addition to (or instead of) report text, you can import that file into the *PowerScribe SDK* database by calling the **SetWaveFile()** method and passing it the external path to the report's wave file:

```
objReport.SetWaveFile("C:\report.wav")
```



Note: The wave file must be in PCM 11 kHz, 16 bit Mono format.

The method also has an optional second argument to indicate whether the file should be imported as compressed or uncompressed. You can pass it one of the two constants indicated in the next table; the argument defaults to **psWaveFileTypeNonCompressed**.

WaveFileType Enum Values

WaveFileType	Value	Description
psWaveFileTypeNonCompressed	1	Indicates to return the file uncompressed. Default.
psWaveFileTypeCompressed	2	Indicates to return the file compressed.

SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

Importing Associated Concordance File

If you have imported both text and audio for the same report, you should also import a file that aligns the text with the location of each word in the audio when the audio plays back in the **PlayerEditor**, called a *concordance* file. To import this file, you call the

SetConcordanceFile() method of the **Report** object. This file can be in one of two proprietary formats, a *PowerScribe .ast* file or an *Enterprise Express Speech* file. You indicate to the method the path where the file is located and its type by passing those two pieces of information as arguments:

```
objReport.SetConcordanceFile("C:\report.ast", psConcordanceFilePowerScribe);
```

ConcordanceFileType Enum Values

TextType	Value	Description
psConcordanceFilePowerScribe	0	File from another <i>PowerScribe SDK</i> server. Default.
psConcordanceFileExSpeech	1	File from an <i>Enterprise Express Speech</i> server.
<i>SDK</i> constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.		

The root name of the concordance file always matches the root name of its audio and text source files. The extension on the *PowerScribe* format concordance file name is **.ast**. The extension on the *Enterprise Express Speech* format concordance file name can be any extension you assigned it.

Locking and Activating the Report

Finally, you can now lock and activate the report, just as you did when you created the report through dictation in [Chapter 4, Creating the Report](#).

Pulling Together Code for Exporting and Importing Report

The code for exporting a report from the database can be consolidated into a single function:

```
function ExportReportText()
{
    var rptText;
    var rptName.Text; // Retrieved from input field
    objReport = pscribeSDK.GetReport(rptName.Text);

    rptText = objReport.GetText(psTextTypeCorrected, psTextFormatTypePlain);
    StoreReportStringInFile(rptText, "C:\ReportText\report.txt");

    objReport.GetWaveFile("C:\" + rptName.Text + ".wav",
                           psWaveFileCompressed);
```

```
    objReport.GetConcordanceFile("C:\" + rptName.Text + ".ast",
                                  psConcordanceFilePowerScribe);
}
```

The code for importing a *PowerScribe* report can be consolidated into a single function:

```
function ImportReport_and_DisplayInEditor()
{
    var objReport;
    var rptText;
    var rootname;

    objReport = pscribeSDK.NewReport("rptName.Text"); // Name from field
    objReport.ExclusiveLock(true);
    rptText = ReadFileToString(rptText.txt);

    objReport.SetTextEx(psTextTypeCorrected, psTextFormatTypePlain, rptText);
    objReport.SetWaveFile("C:\" + rptName.Text + ".wav",
                          psWaveFileTypeCompressed);
    // Retrieve the root name of the wave file & store in rootname var
    objReport.SetConcordanceFile("C:\" + rootname + ".ast",
                                 psConcordanceFilePowerScribe);

    pscribeSDK.ActivateReport(objReport);
}
```

Importing Report from Third Party Device and Recognizing Audio Immediately

If a person has dictated a report into an alternative device, such as a personal digital assistant (PDA), you can import the audio from that device into the **PlayerEditor**, where speech recognition occurs when you open the report for recognition.



Note: The reports must be:

- Wave files in PCM 11 kHz, 16 bit Mono format
- Not compressed

Before you proceed, refer to [Chapter 4, Creating the Report](#) to take these actions:

- Create the **Report** object
- Create the **PlayerEditor** and associate it with the report
- Lock and activate the report

Then you use a series of methods to:

- Generate the report text from the audio
- Run speech recognition on the wave file to generate report text
- Save the report
- Handle the associated events

Generating Report Text from Audio

After you create the **Report** object (see [Chapter 4, Creating the Report](#)), associate it with a **PlayerEditor** object, and lock and activate the report file, you are ready to import an an audio file into the report.

To import a wave file from a PDA or similar device into that report and have *PowerScribe SDK* recognize the audio, transcribe it into text, and display it in the **PlayerEditor**, you call the **RecognizeWaveFile()** method of the **Report** object and pass it the full path to the audio file on your local workstation:

```
objReport.RecognizeWaveFile("C:\report.wav");
```

Executing this method puts the text of the report into the text file associated with the **Report** object and immediately displays that text in the **PlayerEditor**.

Saving Text of the Report

Once the text that **RecognizeWaveFile()** generates appears in the editor, you save the report to store it in the database:

```
objReport.Save();
```

Handling the WaveFileProgress Event

After the wave file has been transferred into the database on the server, the **WaveFileProgress** event handler receives a **psWaveFileProgressSaveEnd** notification to indicate the file transferred has completed. You can take appropriate action in response to that notification as well as other notifications the event receives:

```
function HandleWaveFileProgress(compress, type, bytesSent, bytesRemain)
{
    ...
    if (type == psWaveFileProgressSaveEnd)
    {
        window.status = "File Transfer Completed";
    }
}
```

Spell Checking Report Text

To run a spell checker on a report after it has been through speech recognition, you call the **CheckSpelling()** method of the **Report** object and pass it one option:

- Check all text in the file.
- Allow the user to choose whether to check all text in the file or check only selected text (this dialog box pops up only when the user has selected text in the report) . Refer to the table shown here for the options.

Spell Checking Options and Values

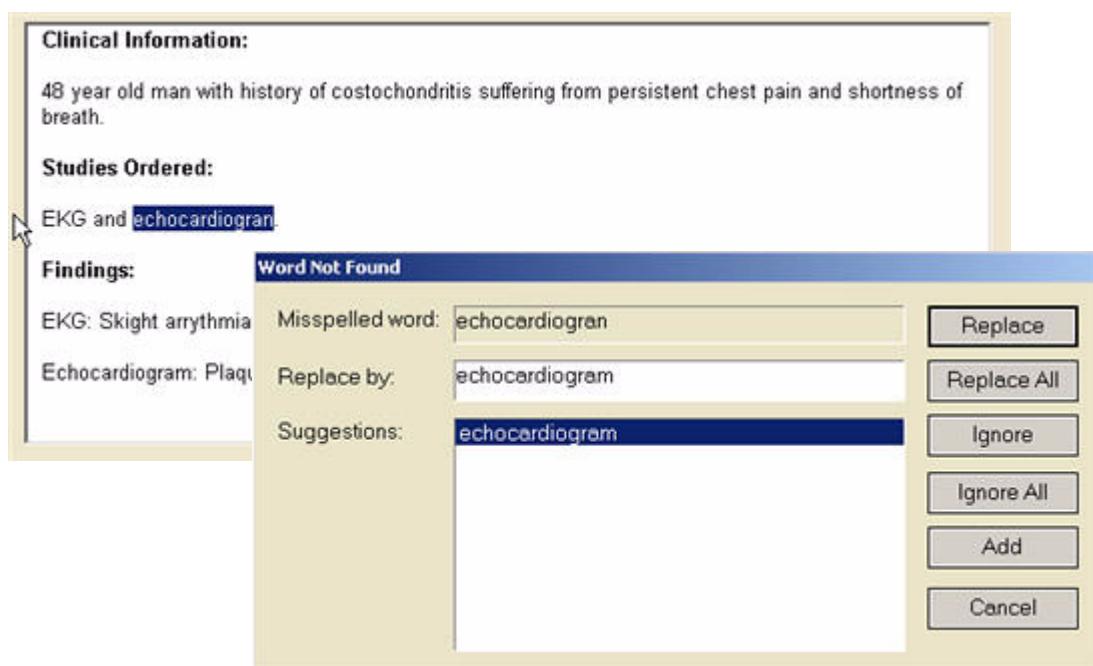
Spell Checking Option Constant	Value
psSpellCheckerCheckAll	0
psSpellCheckerShowSelectionDialog	1

You must define them to use *SDK* constants in JavaScript.

For example, to have the method always spell check the entire file and pop up a dialog box when it finds a potential misspelling, you would pass the **0** option, **psSpellCheckerCheckAll**:

```
var SpellingResult;
SpellingResult = objReport.CheckSpelling(psSpellCheckerCheckAll);
```

The illustration below shows a report being spell checked and the **Word Not Found** dialog box that pops up when it finds the misspelled word.

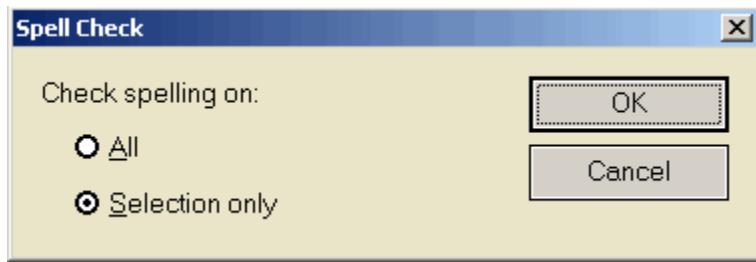


If the user clicks the **Add** button in this dialog box, *SDK* adds the word to the **Words** table in the **General** category, then updates the spellchecker dictionary the next time the user logs in.

The user might want to spell check only a particular block of selected text. To offer this option whenever the user selects text inside the report, you would pass the **1** option, **psSpellCheckerShowSelectionDialog**:

```
SpellResult = objReport.CheckSpelling(psSpellCheckerShowSelectionDialog);
```

When you pass this option, if the user does not select specific text, the spell checker behaves as it does for option **0**. But once the user selects text, a dialog box pops up where the user can choose to spell check only the selected text.



If the spell checker runs without any problems, the method returns **psSpellResultNoError**.

If an error occurs during spell checking, the method returns a **SpellResultCode** indicating the error. Possible errors are listed in the table that follows.

SpellResultCode Values

SpellResultCode Constant	Value	SpellResultCode Constant	Value
psSpellResultNoMisspell	2	psSpellResultBadCustom	14
psSpellResultAlreadyOpen	4	psSpellResultNotLoaded	16
psSpellResultOpenFailed	5	psSpellResultReplaceError	17
psSpellResultCreateFailed	6	psSpellResultCacheError	18
psSpellResultInvalidDictionary	7	psSpellResultNothingToCheck	20
psSpellResultWriteFailure	9	psSpellResultDialogError	21
psSpellResultReadFailure	10	psSpellResultOutOfMemory	22
psSpellResultBadFileName	11	psSpellResultOutOfStringSpace	24
psSpellResultNotUpdateable	12	psSpellResultFileExists	26
psSpellResultDuplicateWord	13	psSpellResultCancelled (sic)	27

SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

Sending Reports to Recognition Server for Batch Recognition

In some cases, you might want to design an application that works at sites where real time speech recognition is disabled. This setup forces the *Recognition Server* to transcribe the audio, instead of having recognition carried out on the local workstation where the *SDK Client* is installed.

In other cases, a person might have dictated multiple reports into an alternative device, such as a personal digital assistant (PDA). When you import the audio for these reports, you can then queue the audio files for speech recognition on the *Recognition Server*.

In either of these situations, you need to take several steps to add the report to the recognition queue and handle the resulting event:

- Disable real time speech recognition on initialization of *PowerScribe SDK*.
- Prepare to handle the **WaveFileProgress** and **QueuedForRecognition** events
- Put the audio files into the *Recognition Server* queue
- Handle the **QueuedForRecognition** event

Disabling Real Time Speech Recognition

To set up your application to carry out all speech recognition on the *Recognition Server* instead of having recognition carried out on the local workstation where the *SDK Client* is installed, you disable real time speech recognition when you call the **Initialize()** method of the **PowerscribeSDK** object to start up *PowerScribe SDK*. To do so, you pass **True** for the *disableRealTimeRecognition* argument of the method:

```
pscribeSDK.Initialize(url, "JavaScript_Demo", true);
```

Your application then carries out speech recognition on the server rather than locally.

Preparing to Handle the WaveFileProgress and QueuedForRecognition Events

If you have not already created an event mapper for the **WaveFileProgress** event, create one and link it to the event as shown in [Handling the WaveFileProgress Event on page 73](#).

Create an event mapper for the **QueuedForRecognition** event in the same way.

Be sure to call the **Advise()** method of the **PowerscribeSDK** object so that you can receive the events:

```
ReptSink.Advise(objReport);
```

Saving Audio Files to the Server

You can save audio files to the server as shown earlier under [Importing Existing Audio on page 92](#), using the **SetWaveFile()** method of the **Report** object:

```
objReport.SetWaveFile("C:\report.wav")
```

After you call this method, it triggers the **WaveFileProgress** event and sends a **psWaveFileProgressSaveEnd** notification.

Adding Audio File to the Recognition Server Queue

When the **WaveFileProgress** event handler receives a **psWaveFileProgressSaveEnd** notification, that notification indicates the wave file transfer has completed successfully. In response to the notification, you can send the audio file into the *Recognition Server* queue using the **QueueForRecognition()** method of the **Report** object:

```
function HandleWaveFileProgress(compress, type, bytesSent, bytesRemain)
{
    ...
    if (type == psWaveFileProgressSaveEnd)
    {
        window.status = "File Transfer Completed";
        objReport.QueueForRecognition();
    }
}
```

Alternatively, you can call the **QueueForRecognition()** method in the **SaveEndEx** event handler instead of the **WaveFileProgress** event handler.

Another way you can put the audio file into the queue for recognition is to handle a **Microphone** object event that occurs when the user presses the transcribe (left) button on the microphone (**Microphone.Left**). In this situation, the *SDK* saves the wave file, automatically sends the audio into the queue for recognition, then unlocks the report. For more information on using **Microphone** object events, refer to [Chapter 9, Configuring and Customizing the Microphone](#).

When the file has been successfully added to the *Recognition Server* queue, that action triggers a **QueuedForRecognition** event.

The server carries out recognition on each wave file in a background process.

Handling the QueuedForRecognition Event

Each time a wave file is successfully added to the queue for recognition on the *Recognition Server*, the *SDK* triggers the **QueuedForRecognition** event. In the handler for this event, you can indicate on the status line that the report has been added to the queue:

```
function HandleQueuedForRecognition()
{
    window.status = "Report has been queued for Speech Recognition";
}
```

Later, when you decide to edit the report, before you activate it, you should check the value of the **InRecognition** property of the **Report** object to determine if the report is still in recognition or if recognition is complete:

```
if (objReport.InRecognition == false)
{
    objReport.ExclusiveLock(true);
    pscribeSDK.ActivateReport(objReport);
}
```

If recognition is complete, you can open the report for editing by locking and activating the report, then take other action on it. You lock and activate the report just as you did with a new report in [Locking and Activating the Report on page 63](#).

Using Recognition Accuracy Feedback (RAF) Tool

One of the built-in tools in *PowerScribe SDK* is the *Recognition Accuracy Feedback* (RAF) tool. This tool helps troubleshoot speech recognition issues. You teach this tool the phrases that give a particular user the most problems with recognition and, later, every time that user dictates those phrases, the tool stores the recognition information about those phrases in a RAF data file to be used for troubleshooting.

To use the RAF tool, your application can:

1. Create a report that receives dictation of the problematic phrase:

```
objRAFreport = pscribeSDK.NewReport("RAF_Info");
```
2. Enable the RAF tool before activating the report, by setting the **EnableRAF** property of the **Report** object to **True** (it is **False** by default):

```
objRAFreport.EnableRAF = true;
```

3. As you would with any other report, you generate a **PlayerEditor** and **LevelMeter**, and associate them with the report before you lock and activate the report. You might take all these actions in a custom **SetupRAF()** function that you call during initialization of the *PowerScribe SDK*:

```
function SetupRAF()
{
    try
    {
        objRAFreport = pscribeSDK.NewReport("RAF_Info");
        objRAFreport.EnableRAF = true;
        objRAFreport.SetPlayerEditor(PlayerEditorRAF);
        objRAFReport.SetLevelMeter(LevelMeterRAF);
    }
    catch(error)
    {
        alert(error.description);
    }
}
```

4. Provide an **Expected Correlating Text** text entry box in the GUI where the user can enter the correct text that he or she expects to be transcribed whenever the problematic phrase is dictated.
5. Provide a **Comment** text entry box in the GUI where the user can enter an explanation about the problematic phrase.
6. When the user starts the *Recognition Accuracy Feedback* feature (by perhaps pressing a button), your application should lock and activate the RAF report:

```
try
{
    objRAFReport.ExclusiveLock(true);
    pscribeSDK.ActivateReport(RAFreport);
}
catch(error)
{
    alert(error.description);
    // Handle error
}
```

7. After the report is active, the user can then dictate the problematic phrase that requires greater recognition accuracy, enter the correct correlating text in the **Expected Correlating Text** box, and enter any explanation in the **Comments** box.
8. Immediately after the text has been dictated and the expected corresponding text and comments entered, your application saves this information for recognition purposes by calling the **SaveRAFInfo()** method of the **Report** object and passing it three arguments:
 - The string of text you expect to receive for the particular phrase just dictated (from **Correlating Text**)
 - A string containing user comments about the phrase (from **Comments**)

- The full path location on the local machine to save the RAF data to

The **SaveRAFInfo()** method call might look like this:

```
objReport.SaveRAFInfo(strPhrase, strComment, "C:\RAFdata");
```

The location to save the RAF data is optional. If you do not pass it to the method, the location defaults to the path you set the PS_PARAMETER to in the *SDK Administrator* application, a location on the server.

You might call the **SaveRAFInfo()** method in response to a button in the GUI:



Note: You do not have to save the report before calling **SaveRAFInfo()**, as the method saves all report information:

```
function btnSendRecogAccuracyDataToServer_onclick()
{
    try
    {
        // Save Expected Correlating Text for the audio/any comments
        RAFreport.SaveRAFInfo(strPhrase , strComment, "C:\RAFdata");
    }
    catch(error)
    {
        alert(error.description);
    }
}
```

The RAF tool stores the comments from the second argument and later prints them into logs it updates each time the user dictates the problematic phrase. The RAF tool logs data into ASR log files, and postprocessing log and dump files.

To improve recognition, the RAF tool retains several pieces of information:

- Wave file containing dictated audio of the problematic phrase.
- Associated user's acoustic model (**.sig** and **.usr** files) and user ID information.
- Concordance (**.ast**) file that coordinates the dictated audio of the problematic phrase to the expected correlating text.
- Recognized output in a plain report (**.rtf**) file.
- Expected correlating text for the problematic phrase in a text (**.txt**) file.

Code Summary

```

<HTML>
<OBJECT ID="pscribeSDK">
</OBJECT>
<HEAD>
<TITLE>Powerscribe SDK</TITLE>
<!-- This code summary is not a complete program; it assumes
inclusion of required code from the previous chapter --&gt;
&lt;SCRIPT type="text/javascript"&gt;
function InitializePS()
{
    // Bold statements in this function differ from earlier chapters
    try
    {
        // Instantiate PowerScribe SDK Object
        pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");

        // Initialize PowerScribe SDK Server

        var url = "http://server2/pscribesdk/";
        // Set the third arg of Initialize to True to turn on
        // server-based recognition and disable local realtime recognition
<b>pscribeSDK.Initialize(url, "JavaScript_Demo", true);

        // Create Event Mapper Objects for SDK Object, Microphone Object

        SDKsink = new ActiveXObject("PowerscribeSDK.EventMapper");
        MicSink = new ActiveXObject("PowerscribeSDK.EventMapper");
        ReptSink = new ActiveXObject("PowerscribeSDK.EventMapper");

        // Map Event Handlers for Each Event to SDK Event Mapper Object
        // Bold handler is new for this chapter; others from ch 3 & 4

        SDKsink.LoginEnd = HandleLoginEnd;
        SDKsink.ProgressMessage = HandleProgressMessage;
        SDKsink.ProgressMessageEx = HandleProgressMessageEx;
        SDKsink.DownloadProgressMessage = HandleDownloadProgressMessage;
        SDKsink.Advise(pscribeSDK);
        ReptSink.SaveEndEx = HandleSaveEndEx;
        ReptSink.WaveFileProgress = HandleWaveFileProgress;
ReptSink.QueuedForRecognition = HandleQueuedForRecognition;

        // Create Microphone Object, Map Event Handler to Mic Evt Mapper Obj

        MyMicrophone = pscribeSDK.Microphone;
        MicSink.Advise(MyMicrophone);
    }
    catch(error)
    {
        alert("InitializePowerscribe: " + error.number + error.description);
    }
}

```

```
        }

    }

// Create global report object variable
var objReport;

function ExportReportText()
{
    var rptText;
    var rptName.Text; // Retrieved from input field
    objReport = pscribeSDK.GetReport(rptName.Text);

    rptText = objReport.GetText(psTextTypeCorrected, psTextFormatTypePlain);
    StoreReportStringInFile(rptText, "C:\ReportText\report.txt");

    objReport.GetWaveFile("C:\" + rptName.Text + ".wav",
                           psWaveFileTypeCompressed);

    objReport.GetConcordanceFile("C:\" + rptName.Text + ".ast",
                                 psConcordanceFileTypePowerScribe);
}

function ImportReport_and_DisplayInEditor()
{
    var objReport;
    var rptText;
    var rootname;

    objReport = pscribeSDK.NewReport("rptName.Text"); // Name from field
    objReport.ExclusiveLock(true);
    ReptSink.Advise(objReport); // Activate event handlers for rpt events
    rptText = ReadFileToString(rptText.txt);

    objReport.SetTextEx(psTextTypeCorrected, psTextFormatTypePlain, rptText);
    objReport.SetWaveFile("C:\" + rptName.Text + ".wav",
                           psWaveFileTypeCompressed);
    // Retrieve the root name of the wave file & store in rootname var
    objReport.SetConcordanceFile("C:\" + rootname + ".ast",
                                 psConcordanceFileTypePowerScribe);

    pscribeSDK.ActivateReport(objReport);
}
```

```
function OpenReportForEdit()
{
    // Before opening the report to edit, be sure it is not still
    // being recognized on the Speech Recognition Server

    if (objReport.InRecognition == false)
    {
        objReport.ExclusiveLock(true);
        pscribeSDK.ActivateReport(objReport);
    }
}

function GetReportFromPDA()
{
    objReport.RecognizeWaveFile("C:\report.wav");
    // After text appears in the PlayerEditor, save the report
    objReport.Save();
}

function SpellCheckReport()
{
    var SpellResult ;
    var psSpellCheckerCheckAll = 0;
    var psSpellCheckerShowSelectionDialog = 1;

    SpellResults = objReport.CheckSpelling(psSpellCheckerShowSelectionDialog);
}

function SetUpRAF()
{
    try
    {
        RAFreport = pscribeSDK.NewReport("RAF_Info");
        RAFreport.EnableRAF = true;
        RAFreport.SetPlayerEditor(PlayerEditorRAF);
        RAFreport.SetLevelMeter(LevelMeterRAF);
    }
    catch(error)
    {
        alert(error.description);
    }
}
```

```
function ActivateRecogFdback()
{
    try
    {
        RAFReport.ExclusiveLock(true);
        pscribeSDK.ActivateReport(RAFreport);
    }
    catch(error)
    {
        alert(error.description);
        // Handle error
    }

    // Retrieve expected text and associated comments user entered
    objReport.SaveRAFInfo(strPhrase, strComment, "C:\RAFdata");
}

function btnSendRAF_DataToServer_onclick()
{
    try
    {
        // Save the Expected Correlating Text and any comments
        RAFreport.SaveRAFInfo(strPhrase , strComment, "C:\RAFdata");
    }
    catch(error)
    {
        alert(error.description);
    }
}

function HandleWaveFileProgress(compress, type, bytesSent, bytesRemain)
{
    ...
    if (type == psWaveFileProgressSaveEnd)
    {
        window.status = "File Transfer Completed";
        objReport.QueueForRecognition();
    }
}

function HandleQueuedForRecognition()
{
    window.status = "Report has been queued for Speech Recognition";
    // Take other action
}

</SCRIPT>
</HEAD>
```

```
<BODY bgcolor="#fffff" language="javascript" onload="InitializePS() "
onunload="UninitializePS()">

<form>
...
<div id="objRptEditor">
    <object id="PlayerEditor" style="WIDTH: 605px; HEIGHT: 265px; BACK-
    GROUND-COLOR: #EFE7CE" data="data:application/x-oleobject;base64,UNFwpP/
    DhEWbC30xr0YhwRAHAAJAAAAAAAAAAAAAAA="

    classid="clsid:A470D150-C3FF-4584-9B0B-7D31AF4621C1" viewastext>
    </object>
</div>

<div id="objLevelMeter">
    <object id="LevelMeter" height="20" width="150" data="data:application/
        x-oleobject;base64,DfwvSUoKrU68UpZq4VaPzQADAACBDwAAEQIAAA==">
    classid="clsid:492FFC0D-0A4A-4EAD-BC52-966AE1568FCD">
    </object>
</div>
...

// Expected Correlating Text Entry Box for RAF

// Comment Text Entry Box for RAF

<input value="Open Report for Edit" type="button" name="btnOpenRept"
language="javascript" onclick="OpenReportForEdit()" style="FONT-
STYLE: bold">
<input value="Export Report" type="button" name="btnExportRept" language=
"javascript" onclick="ExportReport()" style="FONT-STYLE: bold">
...
<input value="Import Report" type="button" name="btnImportRept" language=
"javascript" onclick="ImportReport_and_DisplayInEditor()"
style="FONT-STYLE: bold"> ...
<input value="Retrieve PDA Report" type="button" name="btnGetReportFromPDA"
language="javascript" onclick="GetReportFromPDA()"
style="FONT-STYLE: bold">...
<input value="Spell Check Report" type="button" name="btnSpellCheck"
language="javascript" onclick="SpellCheckReport()" style="FONT-STYLE:
bold">...
<input value="Set Up Accuracy Feedback" type="button" name="btnSetUpRAF"
language="javascript" onclick="SetUpRAF()" style=
"FONT-STYLE: bold">
<input value="Activate Recognition Feedback" type="button"
name="btnActRecognition" language="javascript"
onclick="ActivateRecognFdback()" style="FONT-STYLE: bold">
<input value="Send RAF Data to Server" type="button"
name="btnSendRAF_DataToServer" language="javascript"
onclick="SendRAF_DataToServer()" style="FONT-STYLE: bold">

</form>
</body>
```


Chapter 6

Creating Structured Reports

Objectives

In addition to plain reports, you can create structured reports. This chapter covers structured reports. Plain reports are covered in [Chapter 4](#) and [Chapter 5](#).

- [Understanding Elements of Structured Report](#)
- [Creating Structured Report Template](#)
- [Preparing to Handle **Report** and **Sections** Object Events](#)
- [Creating Structured **Report** Object](#)
- [Adding and Removing Report Sections](#)
- [Handling **SectionsChanged Report** Event](#)
- [Adding and Removing Subsections](#)
- [Working with Report Section Information](#)
- [Traversing Sections and Subsections with Section Properties](#)
- [Handling **SectionChanged Sections** Event](#)
- [Navigating and Moving Cursor in Report and Taking Action on Current Section](#)
- [Showing Source of Text in the Editor](#)
- [Merging Reports](#)

Understanding Elements of Structured Report

A structured report contains predefined headings, such as *Clinical Information*, *Allergies*, *Examination*, *Findings*, and *Diagnosis* in a medical report (see illustration below). The structure is defined in an XML template that complies with the *Clinical Language Understanding (CLU)*.
.xsd format provided on the product CD. When you create the structured report, you indicate the particular XML template that contains the standard headings the report should include.



*This type of report requires a **PlayerEditor** with that has been created using the **PlayerEditorCtl** ActiveX control; you cannot work with a structured report in a **WordPlayerEditor** created with the **WordPlayerEditorCtl** ActiveX control.*

Each heading and associated paragraph belongs to a **section** of the report. Sections can also contain up to nine **subsections**, like **Medications** and **Foods** shown under **Allergies** in the adjacent report illustration. Some predefined paragraph text might also be included as part of the template.

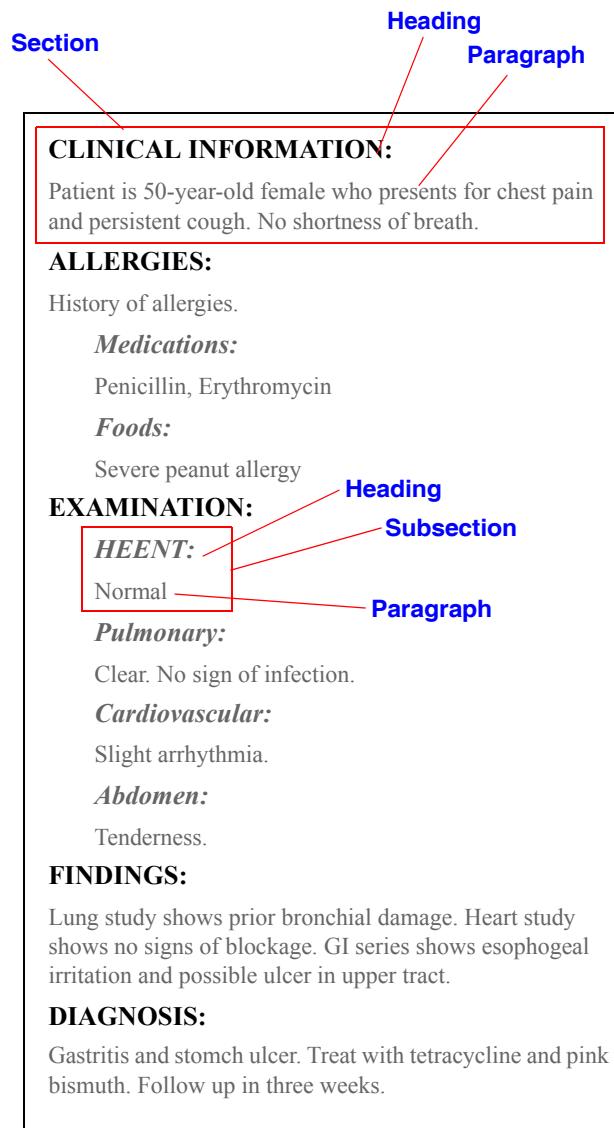
After the report is activated, it presents itself to the person displaying the predefined headings only. The headings appear automatically and the user cannot change them; they are read-only. The cursor initially appears under the first heading.

To start the structured report, the user proceeds to add a paragraph of text under the first heading either by dictating or typing it.

After the user completes the paragraph by clicking the microphone button, by default an additional click of the microphone button or a spoken voice command can move the cursor to the start of the next section. The structured report provides this default navigation capability without requiring you to write any special code.

The flexibility of the *SDK* allows you to write code to implement custom navigation as well.

Structured Report Elements



Creating Structured Report Template

Before you can create a structured report, you must create an XML report template that defines the sections and headings that the report should contain and an XML file that defines the font styles the report should apply to its headings and paragraphs.

(You can also create an XML shortcut template. This chapter focuses on working with the XML report template only. For information on working with shortcuts, refer to [Chapter 12, Working with Shortcuts, Categories, and Words on page 279](#).)

Creating Sections and Headings in XML Report Template (Manually)

Before you work with the template in your application, you need to create an XML file that defines the headings and paragraphs you want in the template. Here is a barebones sample showing the outline of what the XML should contain:

```
<?xml version="1.0" encoding="UTF-8" ?>
<clu xmlns="http://www.dictaphone.com/HSG/CLU/Extraction/2002-12-02"
      xmlns:t="http://www.dictaphone.com/HSG/CLU/Extraction/2002-12-02"
      xmlns:html="http://www.w3.org/1999/xhtml" >

  <doc>
    <body>
      <section>
        <heading>CLINICAL INFORMATION:</heading>
      </section>
      <section>
        <heading>ALLERGIES:</heading>
      </section>
      ...
    </body>
  </doc>
</clu>
```

You can have multiple sections within sections (up to nine levels), each with a heading:

```
<section>
  <heading>ALLERGIES</heading>
  <section>
    <heading>Medications:</heading>
  </section>
  <section>
    <heading>Food:</heading>
  </section>
</section>
```

Expanding XML Template Section

This partial XML template focuses on a single section and heading that have been expanded to include style and font information:

```
<?xml version="1.0" encoding="UTF-8" ?>
<clu xmlns="http://www.dictaphone.com/HSG/CLU/Extraction/2002-12-02"
      xmlns:t="http://www.dictaphone.com/HSG/CLU/Template/2002-12-02"
      xmlns:html="http://www.w3.org/1999/xhtml" >
<doc>
  <body>
    <section html:style="font-size:large;font-family:times;
                      font-weight:normal;font-style:normal">
      <heading html:style="font-family:times;font-size:14pt;
                      font-weight:bold;">CLINICAL INFORMATION:
      </heading>
    </section>
    ...
  </body>
</doc>
</clu>
```

The diagram shows two red boxes highlighting specific attributes in the XML code. One box surrounds the 'html:style' attribute of the 'section' element, with a blue callout pointing to it labeled 'Paragraph style for section'. The other box surrounds the 'html:style' attribute of the 'heading' element, with a blue callout pointing to it labeled 'Heading style for section'.

Preparing to Handle Report and Sections Object Events

Two types of events are triggered when report sections change:

- **SectionsChanged** (*Sections* is plural) event—**Report** object triggers when structure (section added or removed) of the report changes
- **SectionChanged** (*Section* is singular) event—**Sections** object triggers when content of a single section changes

Mapping Report Object Events

To prepare to handle the events that the **Report** object fires, you work with the **ReptSink EventMapper** object you created in [Mapping the SaveEndEx Event on page 69](#). You use the **EventMapper** to map the **SectionsChanged** (note that *Sections* is plural) event, triggered by the **Report** object when a change occurs to the structure of the report:

```
ReptSink.SectionsChanged = HandleStructureOfSectionsChanged;
```

Later, after you retrieve the **Report** object, you call **Advise()** method of the **PowerscribeSDK** object and pass it the structured **Report** object.

Mapping Sections Object Events

To prepare to handle the events that the **Sections** object fires, create an **EventMapper** object for those events. For instance, you could create an **EventMapper** named **SectionsSink**:

```
SectionsSink = new ActiveXObject("PowerscribeSDK.EventMapper");
```

You should also link an event handler to the **SectionChanged** (note that *Section* is singular) event, triggered by the **Sections** object whenever the content of a section changes:

```
SectionsSink.SectionChanged = HandleContentOfSectionChanged;
```

Later, after you retrieve a **Sections** object, you call the **Advise()** method of the **PowerscribeSDK** object and pass it the **Sections** object.

Creating Structured Report Object

When you create a structured report, you use the same method that you used to create a plain report, but in this case, you pass it data about the structure as well as the report identifier.

Creating Structured Report

To create a structured report in *PowerScribe SDK*, when you call the **NewReport()** method of the **PowerscribeSDK** object, in addition to passing it the report ID (first argument), you pass it a string containing the XML report template (second argument):

```
var objStructRept;  
objStructRept = pscribeSDK.NewReport("ORD-120406", xmlString);
```

If you want to create the template dynamically, you can pass an empty report template (one with nothing in its **<body>** tag) to the **NewReport()** method and then have your application create the template programmatically using the **Report** and **Sections** objects. Refer to [Appendix B, Empty Structured Report Template](#), for an empty template to use. Once you have the structured **Report** object, activate its **EventMapper**:

```
ReptSink.Advise(objStructRept);
```

Activating Structured Report

You take the same actions to activate the structured report that you took to activate a plain text report, including these actions, covered in [Chapter 4](#):

- [Associating PlayerEditor and LevelMeter with the Report Object on page 61](#)
- [Setting Up PlayerEditor Event Handlers on page 62](#)
- [Locking and Activating the Report on page 63](#)

Once the report is active, you see the report displayed in the **PlayerEditor**.

Adding and Removing Report Sections

Before you can add or remove sections from the report, you must activate the report.



Caution: If the report is not active when you add sections to or remove sections from it, when you subsequently activate the report, the previously defined sections are retrieved from the XML template and overwrite all your changes to the report sections. To prevent this problem, be sure to activate the report before you add or remove sections.

Retrieving Sections Object

Before you can add or remove sections from the report, you need to retrieve a **Sections** object, which contains a collection of all sections in the report. Once you have the structured report's **Report** object, you can retrieve the sections of the report and take action on them using the **GetSections()** method of the **Report** object:

```
objSections = objStructRept.GetSections();
```

The **Sections** object you retrieve is the collection of all sections at the top level in the report.

To ensure that the **EventMapper** for the **Sections** object is activated, when the application initializes, be sure to call the **Advise()** method of the **PowerscribeSDK** object and pass it the **Sections** object:

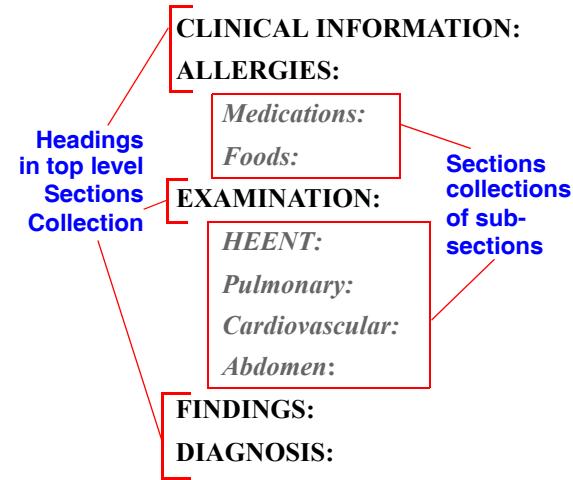
```
SectionsSink.Advise(objSections);
```

Adding Sections to Report

To add sections to a report, you add them to the collection of sections on the same level. You can also add subsections to a collection within the section. The adjacent illustration shows **Sections** collections at different levels:

- CLINICAL INFORMATION, ALLERGIES, EXAMINATION, FINDINGS, DIAGNOSIS (Top level)
- Medications, Foods (2nd level)
- HEENT, Pulmonary, Cardiovascular, Abdomen (2nd level)

To add sections to the top level of the report, you work with the **Sections** object you just obtained using the **GetSections()** method.



Once you have a **Sections** object (a collection of all top level sections in the report), you can add a section to that collection by calling the **Add()** method of that object and passing the method five arguments, in this order:

- *HeadingName*—String of text to appear in the report as the name of the new section.
- *HeadingStyle*—String holding the HTML style of the section's heading, such as:

```
"font-family:Times;font-size:14pt;font-weight:bold;
font-style:normal"
```

- *ParagraphStyle*—String holding the HTML style of the section's paragraphs, such as:

```
"font-family:Times;font-size:10pt;font-weight:normal;
font-style:normal"
```

For options allowed in *HeadingStyle* and *ParagraphStyle* arguments, refer to [Appendix A, HeadingStyle and ParagraphStyle Options](#).

- *SectionAddType*—Where to add the section, either at the beginning of the report, the end of the report, or before an existing section, using either a **SectionAddType** constant or its numeric equivalent (refer to the table below).
- *SiblingHeadingName*—If you are inserting the section before an existing section, string containing the name of the existing section. The method ignores this argument for any *SectionAddType* other than **psSectionAddBefore**.

SectionAddType	Value	Explanation
psSectionAddtoBegin	0	Insert section at the beginning of the Sections collection, before all other sections on the same level.
psSectionAddtoEnd	1	Insert section at the end of the Sections collection, after all other sections on the same level.
psSectionAddBefore	2	Insert section before another section in the Sections collection, passed in <i>SiblingHeadingName</i> argument.

Constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

A call to the method that would add a PATIENT INFORMATION heading in front of the CLINICAL INFORMATION heading might look like this:

```
var psSectionAddBefore = 2;

objSections.Add("Patient Information", htmlHead, htmlPara,
    psSectionAddBefore, "Clinical Information");
```

The structure of the report changes and the new resulting structure displays in the **PlayerEditor**, as shown in the adjacent illustration.

When this method completes, the **Report** object fires a **SectionsChanged** event.

Removing Sections from Report



Caution: Removing a section removes all subsections under the section as well.

To remove a section from the report, you again use the **Sections** collection object that you retrieved and use the **Item()** method of that object to retrieve a **Section** (singular) object from the collection:

```
objRmvSect = objSections.Item(2)
```

You call the **Remove()** method of the **Sections** object and pass the **Section** object of the section to remove:

```
objSections.Remove(objRmvSect);
```

PATIENT INFORMATION:

CLINICAL INFORMATION:

ALLERGIES:

Medications:

Foods:

EXAMINATION:

HEENT:

Pulmonary:

Cardiovascular:

Abdomen:

FINDINGS:

New heading
inserted in top
level Sections
Collection

DIAGNOSIS:

PATIENT INFORMATION:

ALLERGIES:

Medications:

Foods:

EXAMINATION:

HEENT:

Pulmonary:

Cardiovascular:

Abdomen:

FINDINGS:

DIAGNOSIS:

Heading
removed from the
top level Sections
Collection

The update of the report then displays in the **PlayerEditor** with the section removed (see the adjacent illustration).

When this method completes, the **Report** object fires a **SectionsChanged** event.

Handling SectionsChanged Report Event

The **SectionsChanged** event occurs for the top level section of the report when that section or any section or a subsection within that section, at any level, changes, usually in response to the **Add()** or **Remove()** method of the **Sections** object taking action. When the event occurs, your application needs an event handler to handle it:

```
function HandleStructureOfSectionsChanged()
{
    //Update GUI display of report structure with new and removed sections
}
```

The handler receives no arguments. A typical action to take in this handler (when the structure changes) is to update any GUI display that shows the report's sections to the end user.

Adding and Removing Subsections

Adding Subsections to Existing Sections Collection

To add new subsections to a particular section in the **Sections** collection:

1. Retrieve a **Section** (singular) object for an individual section within the **Sections** collection using the **Item()** method of the **Sections** object; pass the method a numeric index to locate the particular section in the collection (section numbering starts at 1). In the adjacent report, to retrieve the section called EXAMINATION, you would pass the **Item()** method an argument of 3:

```
objExamSect = objSections.Item(3);
```

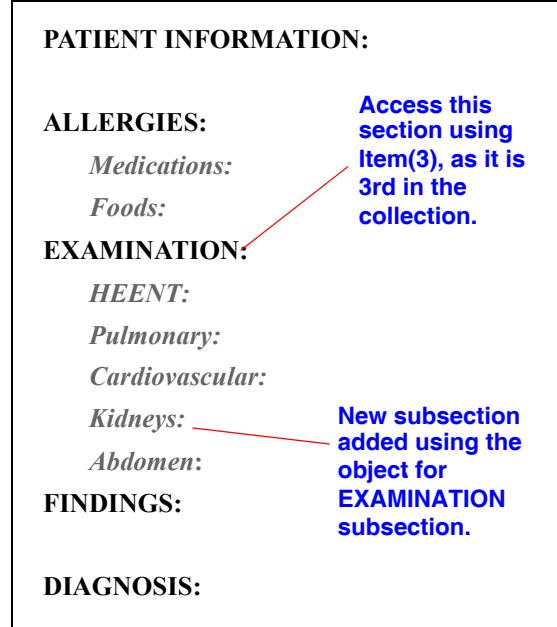
2. To use the **psSectionAddBefore** constant in JavaScript, define it:

```
var psSectionAddBefore = 2;
```

3. Add a subsection under that section using the **Subsections** property of the **Section** object. The **Subsections** property actually returns another **Sections** object, so you are using the **Add()** method on that **Sections** object:

```
objExamSect.Subsections().Add("Kidneys:", htmlHead, htmlPara,
    psAddSectionBefore, "Abdomen");
```

After you have added the subsection to the report, the **Report** object fires a **SectionsChanged** event.



Adding New Subsections Collection to Section

To add a new collection of subsections to a section using the **objExamSect Sections** object you retrieved earlier:

1. Generate a **Section** object for the subsection under EXAMINATION:

```
objKdnySect = objExamSect.Item(5)
```

2. Using the **Section** object's **Subsections** property, create a child **Sections** collection and call the **Add()** method on that collection. Pass the **Add()** method the same arguments you use to add a higher level section (for the fourth argument, 1 means to add the section to the end of the collection):

```
objKdnySect.Subsections().Add("Left:",  
    htmlHead, htmlPara, 1);  
objKdnySect.Subsections().Add("Right:",  
    htmlHead, htmlPara, 1);
```

You could also add a subsection at any level using **Subsections** and **Item()** together with the top level **Sections** object you retrieved from **GetSections()**. For instance, to add **Left** under **Kidneys**:

```
objSections.Item(3).Subsections.Item(5).  
Add("Left:", htmlHead, htmlPara, 1);
```

After you have added a subsection, the **Report** object fires a **SectionsChanged** event.

Removing Subsections from Report

To remove a subsection:

1. To retrieve the particular subsection to remove, use the **Subsections** property and the **Item()** method together on the **Sections** collection you retrieved from the **GetSections()** method:

```
objRmvSect = objSections.Item(3).Subsections.Item(1);
```

2. Now that you have a **Section** object for the section to remove, you must call the **Remove()** method in the **Sections** collection that the section belongs to. You find that collection using the **Subsections** property, then call the method on the collection and pass the method the object for the section to remove:

```
objSections.Item(3).Subsections.Remove(objRmvSect);
```

You can see the resulting change in the next illustration.

PATIENT INFORMATION:

ALLERGIES:

Medications:

Foods:

EXAMINATION:

HEENT:

Pulmonary:

Cardiovascular:

Kidneys:

Left:
Right:

Abdomen:

FINDINGS:

DIAGNOSIS:

New Sections
collection
added using
Subsections
property of
the Kidneys
Section
object

You could also take the same action starting with the object for an existing subsection, such as the **EXAMINATION** section:

```
objRmvSect = objExamSect.Subsections.Item(1);
objExamSect.Subsections.Remove(objRmvSect);
```

The adjacent illustration shows the first subsection under EXAMINATION (HEENT) has been removed.

You can use the same technique to remove any subsection in the report hierarchy from any starting position; for instance, to start at the top level and remove both **Left** and **Right** from under **Kidneys**, you would use **Item()** and **Subsections** repeatedly to traverse to that level of the report:

```
for (var i = 1; i <= objSections.Count; i++)
{
    objRmvSect = objSections.Item(3).Subsections.Item(5).Subsections.Item(i);
    objSections.Item(3).Subsections.Item(5).Remove(objRmvSect);
}
```

This code removes both the individual section or subsection and all of its subsections.

After you have removed the subsection, the **Report** object fires a **SectionsChanged** event.

PATIENT INFORMATION:

ALLERGIES:	First Section (HEENT) removed from the collection of subsections under EXAMINATION .
<i>Medications:</i>	
<i>Foods:</i>	
EXAMINATION:	
<i>Pulmonary:</i>	
<i>Cardiovascular:</i>	
<i>Abdomen:</i>	
<i>Kidneys:</i>	
Left:	
Right:	
<i>Extremities:</i>	

Working with Report Section Information

As you did earlier, to work with a section of the report, you retrieve its **Section** (singular) object using the **Item()** method:

```
objSection = objSections.Item(1)
```

Using Properties to Determine Report Content

You can then use the **Section** object to determine facts about that existing report section. The **Section** object has the following properties that give you information about the report content:

- **HeadingName**, read-only.
- **Text**, read-only.
- **HasData**, read-only.
- **HeadingStyle**, read-write.
- **ParagraphStyle**, read-write.
- **Attribute**, read-write.

The **HeadingName**, **Text**, and **HasData** properties of the **Section** object are read-only, while you can either get or set the values of **HeadingStyle**, **ParagraphStyle**, and **Attribute**. You can use these properties even when a report is not active.

Finding the Section Heading

To find the text in a heading, you retrieve the **HeadingName** property of the **Section**:

```
var stringHead = objSection.HeadingName;
```

The **HeadingName** property is read-only, so you cannot set the property. However, you can change the font and style of text in the heading and paragraph of a section by setting the **HeadingStyle** and **ParagraphStyle** properties of its **Section** object using the HTML style sheet for headings and paragraphs (see [Appendix A](#)). For instance, to set the heading font to 20-point Arial bold and the paragraph style to 16-point Courier, not bold:

```
objSection.HeadingStyle = "font-family:arial;font-size:20pt;
                           color:#00FF00;style:bold"
objSection.ParagraphStyle = "font-family:courier;font-size:16pt;
                            color:#00FF00;style:normal"
```

Retrieving the Content of Section

You can retrieve the content from report sections even when the report is not active. After you have the **Section** object, you can work with the information in that section, for instance, retrieving the heading from the **HeadingName** property or body text from the **Text** property of the **Section** object:

```
alert("Heading of the section:" + objSection.HeadingName);
alert("Text in the section:" + objSection.Text);
```

Determining Whether the Section Contains Text

You can determine whether a particular report section contains any text (data) by checking the value of its **HasData** property; you can then retrieve the text from the section using the **Text** property:

```
if (objSection.HasData == true)
{
    var sectionText = objSection.Text;
    ...
}
```

Getting and Setting Attributes of the Section

Each section in the template can have attributes. You create and initialize any attributes you would like in the <**section**> tag of the XML. For instance, the tag shown below creates two attributes, **Level** and **Type**:

```
<section Level="1";Type="Emergency Room" ...>
```

Using the **Section** object, you can retrieve or set these attributes with the **Attribute** property. For instance, you can retrieve the setting of the **Level** attribute by passing its name to the **Attribute** property:

```
var secLevel = objSection.Attribute("Level");
```

```

if (secLevel == "1")
{
    alert("The section is in the top level collection.");
}

```

You can change the setting of the **Type** attribute of the section by passing its name to the **Attribute** property and setting it as follows:

```
objSection.Attribute("Type") = "Office Visit";
```

You can use these attributes as you would like in your application.

Traversing Sections and Subsections with Section Properties

Your application can traverse the sections and subsections in the report using the **Item()** method of the **Section** object with the **Subsections** and **Parent** properties of the **Section** object.

The **Subsections** property returns the subsections collection (**Sections** and **Subsections** are both **Sections** objects, each at different levels). For instance, to retrieve the subsections collection under the EXAMINATION section, you use the **objExamSect** object, call **Item()**, and call **Subsections**:



*Note: Because subsections are **Sections** objects, you can take all the same actions on a **Subsections** collection that you can take on a **Sections** collection, including executing the **Add()**, **Remove()**, and **Item()** methods.*

```
objSBsections = objExamSect.Item(5).Subsections;
```

By contrast, the **Item()** method returns the **Section** (singular) object for the individual section within the collection:

```
objSBsection = objExamSect.Item(5).Subsections.Item(2);
```

If the subsection does not exist, the **Subsections** property returns an empty **Sections** collection that you can populate using **Add()**; however, if **Item()** does not find a section, it returns a null value.

To traverse back up the report section hierarchy, you can retrieve the parent section of any subsection using the **Parent** property of its **Section** object:

```
prevObjSection = objSBsections.Item(2).Parent;
```

Displaying Report Structure

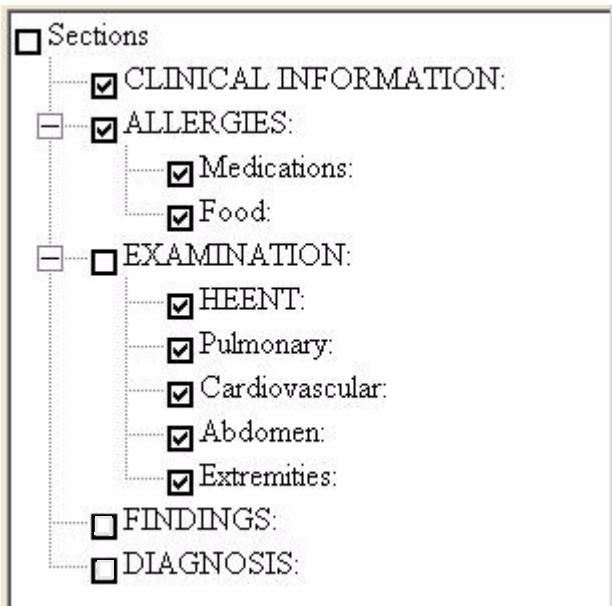
One way you can put the properties of the **Section** object to practical use is by building a graphical display that shows the end user the structure of the report and marks sections as dictated or not dictated, to help the user track report progress.

The **Item()** method of the **Sections** object used with the **Subsections** property of the **Section** object is especially useful when you want to loop through all of the sections in a report and take action using each section.

For instance, to build a tree-like structure of all the sections in a report, similar to the one shown in the adjacent illustration, that has “checked off” the sections containing text, you could create a recursive function to retrieve the **Section** object for each section using the **Item()** method. Alternatively, your custom function could recursively print out all section and subsection heading names in a format that shows the report structure and indicates the sections that contain text (without a graphical control):

```
var objSections = objStructRept.GetSections();
PrintSubsections(objSections, 1) //Calls custom function shown below

function PrintSubsections(objSections, secLvl)
{
    if (objSections.Count > 0)
    {
        for (i = 1; i <= objSections.Count; i++)
        {
            if (objSections.Item(i).HasData == true)
            {
                // Use secLvl to format indentation
                document.write(objSections.Item(i).HeadingName + " DONE");
            }
            else
            {
                // Use secLvl to format indentation
                document.write(objSections.Item(i).HeadingName + " EMPTY");
            }
            objSBsections = objSections.Item(i).Subsections;
            PrintSubsections(objSBsections, secLvl+1); //Call recursively
        }
    }
}
```



The resulting printout might look as follows:

CLINICAL INFORMATION: DONE

ALLERGIES: DONE

Medications: DONE

Foods: EMPTY

EXAMINATION: DONE

HEENT: DONE

Pulmonary: EMPTY

Cardiovascular: DONE

Abdomen: EMPTY

Extremities: EMPTY

FINDINGS: DONE

DIAGNOSIS: EMPTY

When the application receives a **SectionsChanged** event from the **Report** object, in the handler for the event, the application can update the structure of the tree or reprint the formatted list to add a new section that might have been added or remove a section that might have been removed.

When the application receives a **SectionChanged** event (content changed in a section or one of its subsections) from the **Sections** object, in the associated event handler, the application can add or remove a check mark for that section in the tree or reprint the formatted list with the word DONE or EMPTY after the section whose content changed. See also [Handling SectionChanged Sections Event on page 123](#).

Handling SectionChanged Sections Event

In the **SectionChanged** event handler of the **Sections** object, you can take action in response to a section's content changing, whether due to dictation or typing. The event is fired each time the text (data) of a section has changed. The event passes the handler two arguments—a path to the section changed and a Boolean, **True** if there is data in the section and **False** if not:

```
function HandleContentOfSectionChanged(HeaderPathName, HasData)
{
    if (HasData)
        // Update the tree to show section contains/does not contain text/data
}
```

The **HeaderPathName** always starts at the top level heading and concatenates the section headings with a semicolon separating them, so that the path to a section might contain:

"Examination;Kidneys;Left"

The event fires from the top level, so that the full path is always known without traversing through the levels of the report to find the changed text.

You can use the **HasData** Boolean to update a display of the progress of the report, as discussed earlier under [Displaying Report Structure on page 122](#).

Navigating and Moving Cursor in Report and Taking Action on Current Section

To navigate (move the cursor) through a report displaying in the **PlayerEditor**, you can use two methods of the **Report** object, **GetEditorCurrentSection()** and **SetEditorCurrentSection()**.

Determining Section of Cursor in Report

To work with a section in the report, you first select the section of the report where the cursor is currently positioned using the **GetEditorCurrentSection()** method of the **Report** object:

```
objSection = objStructRept.GetEditorCurrentSection();
```

You can then take action on that section using **PlayerEditor** object methods and properties.

Moving Cursor to Another Section in Report

You can decide to move the cursor to a particular section of the report. First you would select the section to move the cursor to, using the **Item()** method of the **Sections** object:

```
objSection = objSections.Item(2);
```

You then pass the **Section** object retrieved to the the **SetEditorCurrentSection()** method to move the cursor to that section in the report:

```
objStructRept.SetEditorCurrentSection(objSection);
```

Recognized text from dictation or typed text now starts from the cursor position.

You can manipulate text in the section or revise text in the section using the **PlayerEditor** instance of the **PlayerEditorCtl**, discussed at length in [Chapter 7, Working with Report Editors on page 145](#).

Adding Section before Currently Selected Section

To insert a new section called **TREATMENT** in front of the section that is selected in the **PlayerEditor** (where the cursor is):

1. Return the currently selected section using the **GetEditorCurrentSection()** method of the **Report** object:

- ```
objCurrSection = objStructRept.GetEditorCurrentSection();
```
2. Retrieve that currently selected section's name using the **HeadingName** property of the **Section** object and pass that section name to the **Add()** method as one to insert the new section in front of (before):

```
objSections.Add("TREATMENT:", htmlHead, htmlPara, psSectionAddBefore,
objCurrSection.HeadingName);
```

## Removing Currently Selected Section

**To remove the section that the cursor is currently located in:**

1. Retrieve the top level sections collection using **GetSections()**:

```
objSections = objStructRept.GetSections();
```
2. Use the **GetEditorCurrentSection()** method to retrieve the section where the cursor is, then retrieve the **Parent** property of the selected section:

```
objCurrSection = objStructRept.GetEditorCurrentSection();
objParentSection = objCurrSection.parent;
```
3. Under the parent section, find the **Sections** collection that the current section belongs to; call the **Remove()** method on that collection and pass it the object for the section to remove:

```
ObjParentSection.Subsections.Remove(objCurrSection);
```

## Finding Particular Section

**To find a particular section:**

1. Retrieve the top level sections collection using **GetSections()**:

```
var objSections = objStructRept.GetSections();
```
2. Search the **Sections** collections recursively for a section whose name matches the selected section passed to the function:

```
function FindSection(objSelSection, secLvl, objSections)
{
 if (objSections.Count > 0)
 {
 for (i = 1; i <= objSections.Count; i++)
 {
 var HeadingCompare = objSections.Item(i).HeadingName
 if (HeadingCompare == objSelSection.HeadingName)
 {
 return objSections.Item(i);
 }
 else
 {
 objSBsections = objSections.Item(i).Subsections;
 }
 }
 }
}
```

```
 FindSection(objSelSection, secLvl+1, objSBsections);
 }
}
}
```

Once you have found the section, you can work with that section of the report.

## Exporting and Importing Structured Report

To export a structured report from a *PowerScribe SDK Server* and then import it into another *PowerScribe SDK Server*, you proceed as covered in [Exporting Reports on page 88](#) and [Importing Reports from Enterprise Express Speech or Another PowerScribe SDK Server on page 91](#) in [Chapter 5](#), only when you export or import the existing file, you indicate the *TextFormatType* is XML rather than plain.

## Showing Source of Text in the Editor

Most of the paragraph text in a report is inserted by dictation. Some text, however, can be from other sources, including:

- Typed
- From an expanded shortcut
- Merged from another report (reused)
- Provided by the template

The source of the text is stored in the XML file along with the structured report itself. Knowing the source of the text helps *PowerScribe SDK* work more efficiently by using only dictated text for adaptation of text recognition algorithms.

Your application can use the source of the data to carry out statistical analysis of the effort used to create the report. Such analysis might help answer questions like:

- What percentage of the report was dictated?
- How much work do shortcuts save the user?
- How much of the report was creating by reusing text from another report?
- How much work does pre-defined paragraph text in the template save the user?

Although the source of the text is stored with the structured report, when the report displays in the **PlayerEditor**, by default you have no indication of where the text originated. For instance, if some of the text was dictated, but other portions of it are the result of expanded shortcuts or typing, that fact remains hidden.

If you want to know the source of each piece of text in the XML report, you can display it by turning on what is called *text source markup*, a style setting that the *SDK* applies to each piece of text so that you can see its source.

## Displaying Text Source Markup

You display a highlight (or mark up) that indicates the source of each piece of text in the report by calling **ShowDataSourceMark()**, a method of the **PowerscribeSDK** object. You pass the method an argument of **True** to turn on highlighting that marks the source of the text or **False** to turn it off:

```
pscribeSDK.ShowDataSourceMark(true);
```

By default, the following highlight colors indicate five possible sources of text origin:

### Default Colors for Indicating Source of Text

| Source    | Default Color | Text Source Code Name     | Value |
|-----------|---------------|---------------------------|-------|
| Typing    | White         | psSourceMarkTypeTyping    | 0     |
| Dictation | Blue          | psSourceMarkTypeDictation | 1     |
| Reused    | Yellow        | psSourceMarkTypeReuse     | 2     |
| Shortcut  | Red           | psSourceMarkTypeShortcut  | 3     |
| Template  | Gray          | psSourceMarkTypeTemplate  | 4     |

Text source code names, like other constants in the *PowerScribe SDK*, are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

The sample report displayed in a **PlayerEditor** below shows text source markup turned on; in this case, the text is mostly dictated (blue), but under **Allergies/Food**, it has been typed (the default color of white has been changed to orange in this case).

The screenshot shows a clinical report with various sections and highlighted text. Red arrows point from the following text elements to specific annotations:

- ALLERGIES:** "History of allergies" is highlighted in blue and points to the annotation "Dictated text highlighted in blue".
- Medications:** "Penicillin, Erythromycin" is highlighted in blue and points to the annotation "Dictated text highlighted in blue".
- Food:** "Severe peanut allergy" is highlighted in orange and points to the annotation "Typed text highlighted in a custom color (normally ‘highlighted’ in white)".
- EXAMINATION:** "Dr. Theodore Jones examined patient in ER." is highlighted in gray and points to the annotation "Text that is part of the template highlighted in gray (Dr. Jones might use this template for all Emergency Room patients.)".
- HEENT:** "Normal" is highlighted in blue and points to the annotation "Text that is part of the template highlighted in gray (Dr. Jones might use this template for all Emergency Room patients.)".

Template paragraph text appears with a gray highlight; template headings are not highlighted.

If a template had been merged with a report, expanded shortcuts, and merged or reused text would appear in other colors. For example, below is an example of *reused* text, text that has been imported from another report:



## Customizing Display of Text Source Markup

You can change the markup colors for each text source by calling the **SetDataSourceMarkStyles()** method and passing it an XML string that indicates the new colors to use.

The XML string would contain the markup style information in the following format, specifying the background color for each type of text:

```
<?xml version="1.0" encoding="utf-8" ?>
<pssdk xmlns="http://www.dictaphone.com/HSG/PSSDK/2005-07-21"
 xmlns:html="http://www.w3.org/1999/xhtml">
 <dataSourceMarkStyle>
 <source id="Tp" name="Typing" style="background-color:#FFFFFF"/>
 <source id="D" name="Dictation" style="background-color:#00FFFF"/>
 <source id="R" name="Reuse" style="background-color:#FFFF00"/>
 <source id="S" name="Shortcut" style="background-color:#FF0000"/>
 <source id="T" name="Template" style="background-color:#C0C0C0"/>
 </dataSourceMarkStyle>
</pssdk>
```

In addition to changing the background color that highlights text from each source, you can also modify other attributes for text from that source, including **font-size**, **font-family**, **font-weight**, **font-style**, **color** (text color, rather than background color), **text-decoration** (such as underscore). Refer to [Appendix A](#) for possible attribute settings.

The call of **SetDataSourceMarkStyles()** might look like this, using the literal string:

```
pscribeSDK.SetDataSourceMarkStyles("<?xml version='1.0'...</pssdk>")
```

Or the call might use a variable that contains the entire XML string:

```
var xmlSourceMarkStyle = "<?xml version='1.0'...</pssdk>";
pscribeSDK.SetDataSourceMarkStyles(xmlSourceMarkStyle);
```

To then redisplay the report with the new source markings, be sure to call the **Refresh()** method on the report:

```
objStructRept.Refresh();
```

# Overview: Merging Reports

Your application can merge report content, structured or not, into another report in several ways:

- Merge two reports with the same template
- Merge selected sections of a report with another report
- Append paragraphs to a particular section
- Append entirely new sections to the report
- Merge new unformatted text into a structured or plain text report
- Merge a shortcut into a report

**To merge new content, your application should carry out the following major steps:**

1. Be sure the report is activated and currently displaying in the **PlayerEditor**.
2. Create an XML string that contains the new content to merge into the existing report.
3. Call the **MergeTemplate()** method of the **Report** object and pass it the string of XML you want to merge and a **psMergeSrcType** to indicate the text source style you want the new text to have.

## Merging Content of Two Structured Reports

You can merge an entire structured report into the one currently displaying in the **PlayerEditor** and any sections with the same name will be merged. Suppose you are working with two reports created using the same template.

**To merge data from each section of the second report into a section of the same name in the original report:**

1. Retrieve the report object of the report you want to merge into the original report by calling the **GetReport()** method of the **PowerscribeSDK** object and passing it the report ID:

```
objRept = pscribeSDK.GetReport("ORD-071006");
```

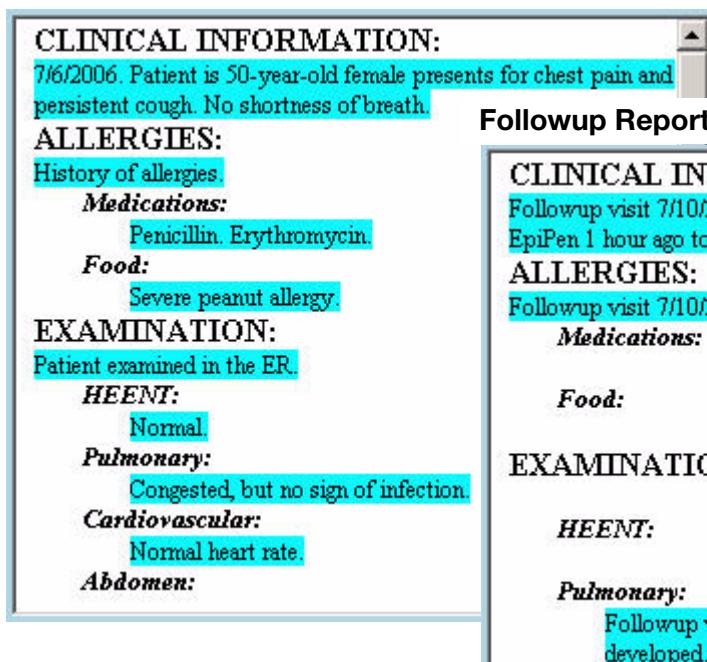
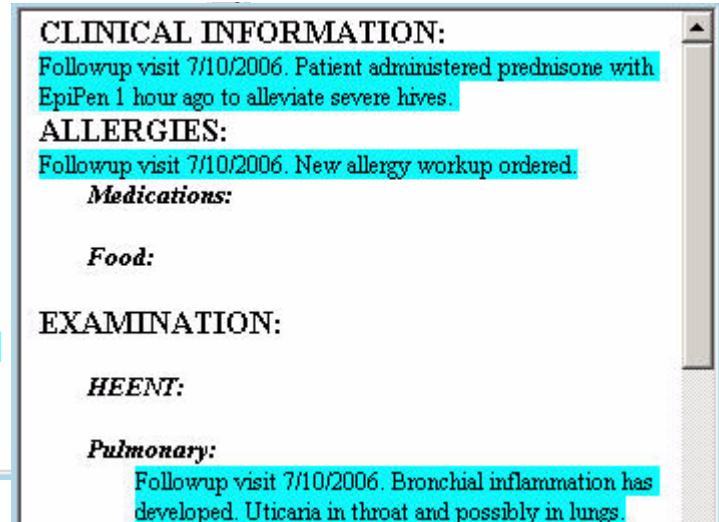
2. To retrieve the contents of the report you want to merge into the original report, call **GetText()** on it and be sure to pass XML for the *TextFormatType* (second) argument:

```
xmlRptText = objRept.GetText(psTextTypeCorrected, psTextFormatTypeXML);
```

The entire contents of the structured report is returned into the **xmlRptText** string.

3. Open the original report for editing by calling the **GetReport()** method of the **PowerscribeSDK** object and passing it the report ID:

```
objStructRept = pscribeSDK.GetReport("ORD-063006");
```

**Original Report:****Followup Report to Merge into Original:**

- After you have instantiated a **PlayerEditorCtl** object, associate the original report with the **PlayerEditor**, then lock and activate the report:

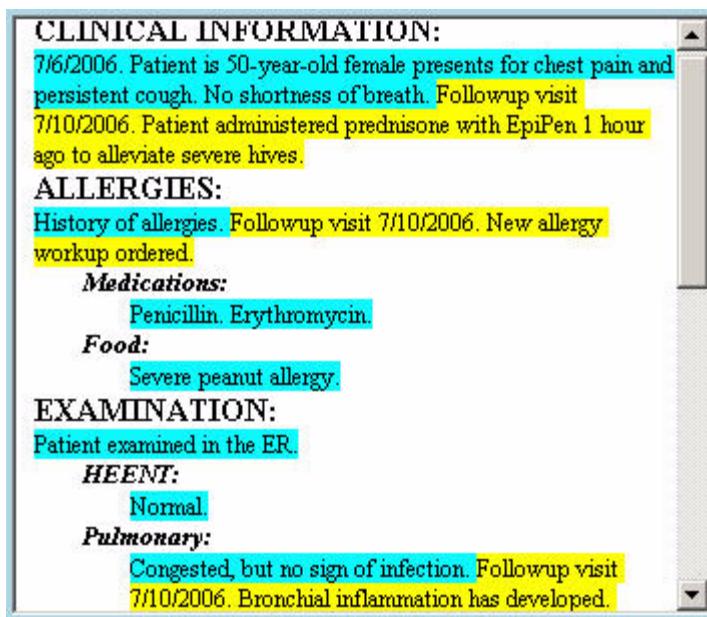
```
objStructRept.SetPlayerEditor(objRptEditor);
objStructRept.ExclusiveLock();
pscribesSDK.ActivateReport(objStructRept);
```

Once the original report is displaying in the **PlayerEditor**, you use the **MergeTemplate()** method of the **Report** object to *merge* the second report into the original:

- Create a variable to receive the status code that the method returns:

```
var psMergeCode;
```

- Call the method and pass it two arguments—a string containing the XML report to merge and, optionally, a *psMergeSrcType* value indicating how to mark the source of the text (see next table). For instance, to mark the merged text as **Reused**, you would pass the **psMergeSrcTypeReuse** constant:

**Result of Merged Reports:**

```
var psMergeSrcTypeReuse = 2;
psMergeCode = objStructRept.MergeTemplate(xmlRptText, psMergeSrcTypeReuse);
```

#### psMergeSrcType Enumerations

| Text Source                                                                                                                                        | Constant               | Value |
|----------------------------------------------------------------------------------------------------------------------------------------------------|------------------------|-------|
| Dictation                                                                                                                                          | psMergeSrcTypeDict     | 1     |
| Reused (default)                                                                                                                                   | psMergeSrcTypeReuse    | 2     |
| Shortcut                                                                                                                                           | psMergeSrcTypeShortcut | 3     |
| Constants for merge sources are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants. |                        |       |

New text appears at the end of each corresponding section, highlighted in the **Reused** color.

## Importing Selected Sections of Another Report

Suppose you have a pharmacology report (like the one shown on the next page) and you want to integrate this information with the studies performed on the same patient in the emergency room. You can append selected sections of this report to the end of the original report, effectively *reusing* them. To select the sections of a report to reuse, you can make a copy of the report and delete the sections you do **not** want to reuse from the copy, retaining only the sections you want. You can then merge the remaining copied report sections into the original report.

#### To import selected sections from another report:

1. Copy the report whose sections you want to use into a temporary report using the **GetReport()** method of the **PowerscribeSDK** object:

```
var objTempRpt = pscribeSDK.GetReport("PHR-102006");
```

2. After you have instantiated a **PlayerEditorCtl** object, associate the temporary report with the **PlayerEditor**, then lock and activate the report:

```
objTempRpt.SetPlayerEditor(objRptEditor);
objTempRpt.ExclusiveLock();
pscribeSDK.ActivateReport(objTempRpt);
```

3. After you activate the report, it appears in the **PlayerEditor**. You can then remove extraneous sections by first getting the top level **Sections** collection that contains them, with **GetSections()**, then finding the sections with a **HeadingName** property that matches the name for each section to remove, then removing it:

```
objSections = objTempRpt.GetSections();
for (var i = 1; i <= objSections.Count; i++)
{
 if (objSection.Item(i).HeadingName == "PATIENT NAME AND PC PHYSICIAN")
 objRmvSect = objSections.Item(i);
```

```
objSections.Item(i).Remove(objRmvSect);
}
```

4. Once you have removed extraneous sections, retrieve the remaining content of the temporary report using the **GetText()** method. Because the temporary report is structured, retrieving its text also retrieves the XML markup. The method stores the content in a string you can pass to **MergeTemplate()** as the XML template:

```
var xmlText;
xmlText = xmlTemp.GetText(psTextTypeCorrected, psTextFormatTypeXML);
```

5. After you save and close the temporary report, open the original report in the **PlayerEditor** and call the **MergeTemplate()** method, passing the temporary report text as the template and indicating you want to mark the text source as **Reused**:

```
var psMergeSrcTypeReuse = 2;
objStrRept.MergeReport
(xmlText,
psMergeSrcTypeReuse);
```

The resulting modified report contains the pharmacology report sections appended to the end, as shown in the adjacent illustration (middle sections removed for brevity).

#### Report with Different Template to Merge into Original

##### PATIENT NAME AND PC PHYSICAN:

Julia Jameison, patient of Dr. Simoneau.

##### PHARMACOLOGY:

Patient prescribed several medications.

###### *Antibiotics:*

No regular use of antibiotics.

###### *Immunosuppressants:*

Prednisone, 100 mg EpiPen. Historically used once every 2 months.  
Efudex, 20 mg, used daily. Salegen, 10 mg, 3 times a day.

###### *Psychoactive:*

Amitriptyline, 50 mg daily.

#### Modified Report After Merge:

##### CLINICAL INFORMATION:

7/6/2006. Patient is 50-year-old female presents for chest pain and persistent cough. No shortness of breath.

##### ALLERGIES:

History of allergies.

###### *Medications:*

Penicillin. Erythromycin.

###### *Food:*

Severe peanut allergy.

##### EXAMINATION:

Patient examined in the ER.

###### *HEENT:*

Normal.

...

##### PHARMACOLOGY:

Patient prescribed several medications.

###### *Immunosuppressants:*

Prednisone, 100 mg EpiPen. Historically used once every 2 mo:  
Efudex, 20 mg, used daily. Salegen, 10 mg, 3 times a day.

###### *Psychoactive:*

Amitriptyline, 50 mg daily.

# Appending Paragraph Text to Particular Section



**Caution:** Be sure the report is active and displaying in the **PlayerEditor** before attempting to merge content into the report.

You construct the XML string of text to *merge* into the report based on the kind of change you want to make to the report. If you want to add text to an existing section, you create an XML string that contains that section's heading and the new paragraph text. The **MergeTemplate()** method then adds the new text to the end of the section with the matching heading.



**Note:** The XML string must use the same format as an XML template for a structured report.

## Creating XML String of Text to Append

To add a new line of text to the end of a particular section in the report, you would package that text into an XML string using the **.xsd** format:

```
<clu xmlns="http://www.dictaphone.com/HSG/CLU/Extraction/2002-12-02"
 xmlns:t="http://www.dictaphone.com/HSG/CLU/Extraction/2002-12-02"
 xmlns:html="http://www.w3.org/1999/xhtml" >

<doc>
 <body>
 <section>
 <heading>FINDINGS:</heading>
 <p>07/10/06 followup visit:
 Lung inflammation due to allergic response
 to Tetracycline.</p>
 </section>
 </body>
</doc>
</clu>;
```

Because the XML string contains the name of an existing section, **MergeTemplate()** can determine where to insert the text in the report.

## Appending Text to Section

To actually append the string to the end of the **FINDINGS** section, you use the **MergeTemplate()** method of the **Report** object. To use this method, you need to create a variable to retrieve the merge result status code it returns:

```
var psMergeCode;
```

You then call the method and pass it two arguments—a string containing the XML and, optionally, a *psMergeSrcType* value indicating the source of the text to show later when the source markup is displayed. For instance, to mark the merged text as **Reused**, you would pass the **psMergeSrcTypeReuse** constant:

```
var psMergeSrcTypeReuse = 2;
psMergeCode = objStructRept.MergeTemplate(xmlStr,psMergeSrcTypeReuse);
```

By default the *psMergeSrcType* of text added using **MergeTemplate()** is **Reused**.

## Viewing Resulting Changes in Editor

Before you merge the new text into the section, the report displays the section like this in the **PlayerEditor** (in this case, the text markup shows this text was dictated):

**Before:**

---

**FINDINGS:**

Lungs study shows prior bronchial damage. Heart study shows no sign of blockage. GI series shows esophageal irritation and possible ulcer in upper tract.

After the merge, the new text from the XML string appears appended to the end of the section. The new text is considered reused, so it is marked in a different color.

**After:**

---

**FINDINGS:**

Lungs study shows prior bronchial damage. Heart study shows no sign of blockage. GI series shows esophageal irritation and possible ulcer in upper tract. 07/10/06 followup visit: Lung inflammation due to allergic response to tetracycline.

## Appending New Section to Report



**Caution:** Be sure the report is active and displaying in the **PlayerEditor** before attempting to merge content into the report.

You construct the XML string of text to *merge* into the report based on the kind of change you want to make to the report. When you create an XML string that contains a new section with or without paragraphs, the **MergeTemplate()** method adds that new section with its paragraphs to the end of the report.

## Creating XML String of New Section

To add a new section called **PHARMACOLOGY** to the end of the report and have it contain new text describing a particular treatment, you would package that text in an XML string using the **.xsd** format:

```
<clu xmlns="http://www.dictaphone.com/HSG/CLU/Extraction/2002-12-02"
 xmlns:t="http://www.dictaphone.com/HSG/CLU/Extraction/2002-12-02"
 xmlns:html="http://www.w3.org/1999/xhtml" >
 <doc>
 <body>
 <section>
 <heading>PHARMACOLOGY:</heading>
 <p>Patient prescribed several medications.</p>
 </section>
 </body>
 </doc>
</clu>
```

Because the string contains the name of a section that does not yet exist, calling **MergeTemplate()** adds the section to the end of the report as a new section. **PHARMACOLOGY** becomes the heading of the section and the material marked as a paragraph becomes text within the section.

## Appending New Section to Report

To actually add the new section to the end of the report, you use the **MergeTemplate()** method of the **Report** object and pass it the string, plus the constant that indicates the text source should be **Reused**:

```
var psMergeCode;
var psMergeSrcTypeReuse = 2;
psMergeCode = objStructRept.MergeTemplate(xmlStr,psMergeSrcTypeReuse);
```

## Viewing Resulting Changes in Editor

If DIAGNOSIS is the last section of the report, before the merge, the bottom of the report displays as follows, with the text source markup showing the text was dictated:

**Before:**

---

**DIAGNOSIS:**

Gastritis and stomach ulcer. Treat with tetracycline and pink bismuth. Followup in 3 weeks.

After the merge, the new section from the XML string appears appended to the end of the report. The new section text is considered reused, so it is marked in a different color:

**After**

---

**DIAGNOSIS:**

Gastritis and stomach ulcer. Treat with tetracycline and pink bismuth. Followup in 3 weeks.

**PHARMACOLOGY:**

Patient prescribed several medications.

## Merging Unformatted Text into Structured Report



***Caution:*** Be sure the report is active and displaying in the **PlayerEditor** before attempting to merge content into the report.

Because unformatted paragraph text requires no sections or structure, when you add unformatted (plain) text to a structured report using **MergeTemplate()**, the method automatically inserts the text where the cursor is located in the report or at the end of the report.

### Creating String of Text to Add

To add new text to an existing structured report, you create a string to hold the text but do not include any XML or HTML coding in the string:

```
txtStr = "Tetracycline and pink bismuth for three weeks."
```

## Positioning Cursor and Adding Text to Report

To add the new text to the Psychoactive section under PHARMACOLOGY, you position the cursor in that section:

**PHARMACOLOGY:**  
Patient prescribed several medications.

**Immunosuppressants:**  
Prednisone, 100 mg EpiPen. Historically used once every 2 months.  
Efudex, 20 mg, used daily. Salegen, 10 mg, 3 times a day.

**Psychoactive:** |

Insert the cursor where new text should appear.

1. Select the section to move the cursor to with the **Item()** method of the **Sections** object:

```
objSection = objSections.Item(2);
```

2. Pass the **Section** object retrieved to the the **SetEditorCurrentSection()** method to move the cursor to that section in the report:

```
objStructRept.SetEditorCurrentSection(objSection);
```

3. Call the **MergeTemplate()** method and pass it the string of text to insert at the cursor position:

```
var statusMergeCode;
statusMergeCode = objStructRept.MergeTemplate(txtStr);
```

## Viewing Resulting Changes in Editor

After you have executed the merge, the new text appears at the cursor location:

**PHARMACOLOGY:**  
Patient prescribed several medications.

**Immunosuppressants:**  
Prednisone, 100 mg EpiPen. Historically used once every 2 months.  
Efudex, 20 mg, used daily. Salegen, 10 mg, 3 times a day.

**Psychoactive:**  
Amitriptyline, 50 mg daily.

# Merging Text into Plain Text Report



*Caution: Be sure the report is active and displaying in the PlayerEditor before attempting to merge content into the report.*

Because a plain text report has no sections or structure, any text you add to a plain text report using **MergeTemplate()** is inserted where the cursor is located in the report.

## Creating String of Text to Add

To add new text to an existing plain text report you create a string to hold the text but do not include any XML or HTML coding in the string:

```
txtStr = "TREATMENT: Tetracycline and pink bismuth for 3 weeks."
```

## Positioning Cursor and Adding New Section to Report

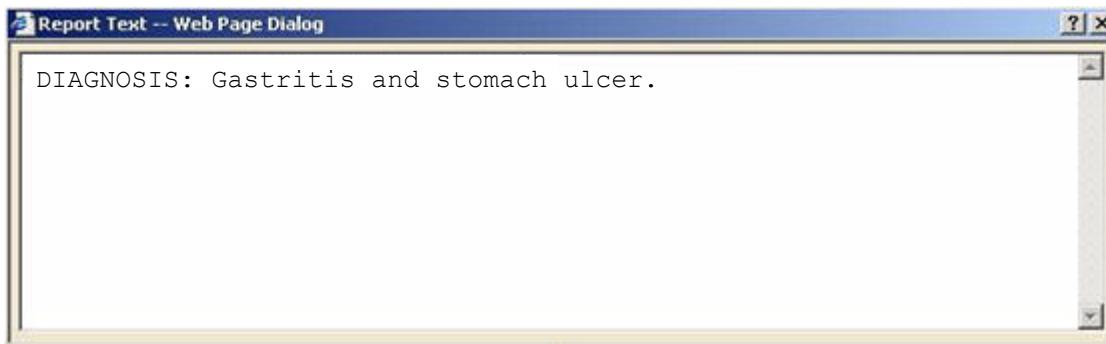
To add the new text after the **DIAGNOSIS** section at the end of the plain text report, you first use the mouse to position the cursor where you want to add the text.

You then call the **MergeTemplate()** method and pass it the string of text. You do not have to pass it the second argument to indicate how to mark the text source, because only structured reports store the source of the text:

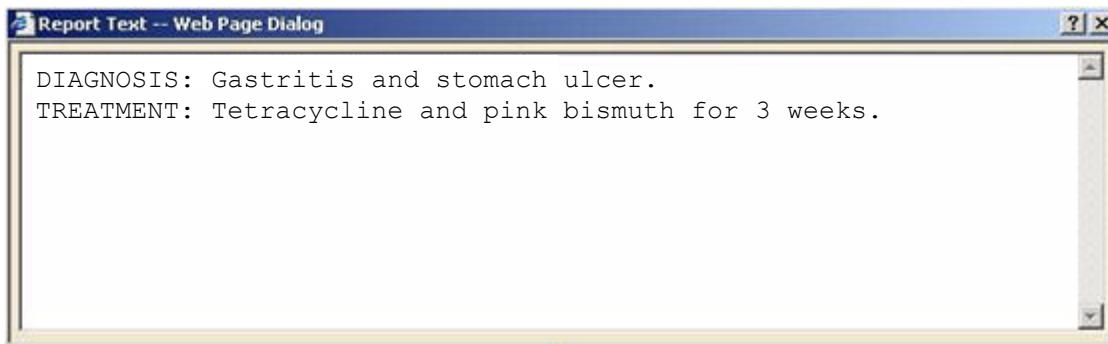
```
var statusMergeCode;
statusMergeCode = objReport.MergeTemplate(txtStr);
```

## Viewing Resulting Changes in Editor

Before the new text is merged into the report, this is how it appears:



After the merging of the new text into the plain report, the report lines appear as follows:



## Merging Shortcut Text into Report



***Caution:*** Be sure the report is active and displaying in the **PlayerEditor** before attempting to merge any shortcuts into the report.

You can also use the **MergeTemplate()** method to merge a shortcut into the report. When you do, you can make the method call using the **psMergeSrcTypeShortcut** constant:

```
var psMergeSrcTypeShortcut = 3;
psMergeStatusCode = objStructRept.MergeTemplate(xmlShCutTxt,
 psMergeSrcTypeShortcut);
```

For more information shortcuts and merging shortcuts into a report, refer to [Chapter 12, Working with Shortcuts, Categories, and Words on page 279.](#))

# Code Summary

```
<HTML>
<OBJECT ID="pscribeSDK">
</OBJECT>
<HEAD>
<TITLE>Powerscribe SDK</TITLE>
<!-- This code summary is not a complete program; it assumes
inclusion of required code from the previous chapters -->
<SCRIPT type="text/javascript">

function InitializePS()
{
 //Bold statements in this function differ from earlier chapters
 try
 {
 // Instantiate PowerScribe SDK Object
 pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");
 // Initialize PowerScribe SDK Server
 var url = "http://server2/pscribesdk/";
 pscribeSDK.Initialize(url, "JavaScript_Demo");
 // The disableRealtimeRecognition argument of Initialize()
 // defaults to False, turning on realtime recognition

 // New Event Mapper Objects for This Chapter in Bold
 SDKsink = new ActiveXObject("PowerscribeSDK.EventMapper");
 PlayEditSink = new ActiveXObject("PowerscribeSDK.EventMapper");
 ReptSink = new ActiveXObject("PowerscribeSDK.EventMapper");
 SectionsSink = new ActiveXObject("PowerscribeSDK.EventMapper");

 // New Event Handlers for This Chapter in Bold
 ...
 ReptSink.SectionsChanged = HandleStructureOfSectionsChanged;
 SectionsSink.SectionChanged = HandleContentOfSectionChanged;
 }
 catch(error)
 {
 alert("InitializePowerscribe: " + error.number + error.description);
 }
}

function CreateStructReport()
{
 // Extracting XML report template into string with custom function:
 var xmlString = ReadXmlToString(structXml);
 objStructRept = pscribeSDK.NewReport("ORD-120406", xmlString);
 ReptSink.Advise(objStructRept);
}
```

```
function ReadXmlToString()
{
 // Custom function that reads XML from a file and puts it into a string
}

function AddSection()
{
 objSections = objStructRept.GetSections();
 SectionsSink.Advise(objSections);

 var psSectionAddtoBegin = 0;
 var psSectionAddtoEnd = 1;
 var psSectionAddBefore = 2;

 // Get sectionToAdd.text & InsertBefore choice from GUI

 var objCurrSection = objStructRept.GetEditorCurrentSection();

 objSections.Add(sectionToAdd.text, htmlHead, htmlPara,
 psSectionAddBefore, objCurrSection.HeadingName);
}

function DeleteSection
{
 objCurrSection = objStructRept.GetEditorCurrentSection();
 objParentSection = objCurrSection.parent;

 objSections = objStructRept.GetSections();
 objParentSection.Subsections.Remove(objCurrSection);
}

function PrintSubsections(objSections, secLvl)
{
 if (objSections.Count > 0)
 {
 for (i = 1; i <= objSections.Count; i++)
 {
 if (objSections.Item(i).HasData == true)
 {
 // Use secLvl to format indentation
 document.write(objSections.Item(i).HeadingName + " DONE");
 }
 else
 {
 // Use secLvl to format indentation
 document.write(objSections.Item(i).HeadingName + " EMPTY");
 }
 objSBsections = objSections.Item(i).Subsections; //Get Collectn
 PrintSubsections(objSBsections, secLvl+1); //Call recursively
 }
 }
}
```

```
function SetMarkupColors()
{
 var xmlSourceMarkStyle = ReadXmlToString(xmlfile.xml);
 pscribeSDK.SetDataSourceMarkStyles(xmlSourceMarkStyle);

 objStructRept.Refresh();
}

function SelectSection()
{
 objStructReport.SetEditorCurrentSection(objSection);
 return true;
}

function FindSection(objSelSection, secLvl, objSections)
{
 if (objSections.Count > 0)
 {
 for (i = 1; i <= objSections.Count; i++)
 {
 var HeadingCompare = objSections.Item(i).HeadingName
 if (HeadingCompare == objSelSection.HeadingName)
 {
 return objSections.Item(i);
 }
 else
 {
 objSBsections = objSections.Item(i).Subsections;
 FindSection(objSelSection, secLvl+1, objSBsections);
 }
 }
 }
}

function MergeReports(objTempRept, objStructRept)
{
 var xmlRptText;
 var psMergeCode;
 var psMergeSrcTypeReuse = 2;

 xmlRptText = objTempRpt.GetText(psTextTypeCorrected, psTextFormatTypeXML)
 psMergeCode = objStructRept.MergeTemplate(xmlRptText,
 psMergeSrcTypeReuse)
}

function HandleContentOfSectionChanged(HeaderPathName, HasData)
{
 if (HasData)
 // Update tree to show section contains data
```

```

 else
 // Update tree to show section does not contain data
 }

 function HandleStructOfSectionsChanged
 {
 // Since a report section has been added or removed, get
 // top collection of sections/pass it to custom function
 // called PrintSections()

 var objSections = objStructRept.GetSections();
 PrintSubsections(objSections, 1);
 }


```

</SCRIPT>

</HEAD>

<BODY bgcolor="#fffff" language="javascript" onload="InitializePS()" onunload="UninitializePS()">

<form> ...

<div id="objRptEditor">

<object id="PlayerEditor" style="WIDTH: 605px; HEIGHT: 265px; BACK-GROUND-COLOR: #EFE7CE" data="data:application/x-oleobject;base64,UNFwpP/DhEWbC30xr0YhwRAHAAJAAAAAAAAAAAAAAA=" classid="clsid:A470D150-C3FF-4584-9B0B-7D31AF4621C1" viewastext>
 </object>
</div>

<div id="objLevelMeter">

<object id="LevelMeter" height="20" width="150" data="data:application/x-oleobject;base64,DfwvSUoKrU68UpZq4VaPzQADAACBDwAAEQIAAA==" classid="clsid:492FFC0D-0A4A-4EAD-BC52-966AE1568FCD">
 </object>
</div>

...

<input value="Create Structured Report" type="button" name="btnStructRept" language="javascript" onclick="CreateStructReport()" style="FONT-STYLE: bold">
<input value="Add Section" type="button" name="btnAddSection" language="javascript" onclick="AddSection()" style="FONT-STYLE: bold">
<input value="Delete Section" type="button" name="btnDeleteSection" language="javascript" onclick="DeleteSection()" style="FONT-STYLE:bold">
<input value="Merge into Current Report" type="button" name="btnMergeRept" language="javascript" onclick="MergeReports()" style="FONT-STYLE: bold">
<input value="Select Section" type="button" name="btnSelectSection" language="javascript" onclick="SelectSection()" style="FONT-STYLE:bold">
<input value="Find Section" type="button" name="btnFindSection" language="javascript" onclick="FindSection()" style="FONT-STYLE:bold">

</form>

</body>



# *Working with Report Editors*

## Objectives

If you have not already read the preceding chapters about creating reports, you should at least peruse them before reading this one.

This chapter covers how to use the *SDK* to program the section of your application that interacts with report text in a **PlayerEditor**, whether it is a **PlayerEditorCtl** or a **WordPlayerEditorCtl**:

- [Choosing an Editor: PlayerEditor or Word PlayerEditor](#)
- [Changing DefaultText Font/Colors in PlayerEditor](#)
- [Finding and Replacing Text](#)
- [Handling \*\*TextSelChanged\*\* Event—Modifying Style of Text](#)
- [Setting Style of Text](#)
- [Copying or Cutting and Pasting Text](#)
- [Creating Numbered or Bulleted Lists](#)
- [Converting Selected Text to List Items](#)
- [Retrieving All Text in \*\*PlayerEditor\*\*](#)
- [Retrieving Audio Corresponding to Selected Text](#)
- [Manipulating Audio Playback or Rewind Speed](#)
- [Forcing Shortcuts to Expand in Editor](#)
- [Zooming In or Out on Report Text](#)
- [Dragging/Dropping Text into PlayerEditor](#)
- [Using Microsoft Word Capabilities in Reports](#)

# Choosing an Editor: PlayerEditor or Word PlayerEditor

Your application can include a **PlayerEditor** area, where recognized text appears after a **Speech Recognition** user dictates a report. The **PlayerEditor** displays the report content so that you can view or edit the content.

You can create two possible types of **PlayerEditors**:

- Standard
- Microsoft Word-based (all workstations creating or accessing reports must have Microsoft Word Version 2000 or higher installed)

The Microsoft Word-based **PlayerEditor** displays the recognized report text in a Word document format and provides Word format editing tools for the user. To use formatting capabilities in a plain **PlayerEditor**, you must add those capabilities manually. If you embed a Word **PlayerEditor**, you can rely on Word to provide the user options such as bold, italic, and underscored text.



*Note: Your application should choose to use one type of editor, either Standard or Word, but should not use both. A report created in a Word **PlayerEditor** should never be opened in the Standard **PlayerEditor**, as the report would lose its formatting.*

The basics of how to add a **PlayerEditor** to your application were covered earlier, in [Chapter 4](#), under [Embedding PlayerEditor Control in Your Application on page 59](#) and [Associating PlayerEditor with the Report Object on page 61](#).

This chapter first presents setting up your application to work with a Word **PlayerEditor**. It then presents what you can do with the **PlayerEditorCtl** or **WordPlayerEditorCtl** control's methods, properties, and events.

All methods available for the standard **PlayerEditorCtl** control are available for a **WordPlayerEditorCtl** control; however, several of those methods are not really needed for Word reports, because Word handles the functionality for you.



*Note: Unless otherwise stated, all methods, properties and events in this chapter are available for both **PlayerEditorCtl** and **WordPlayerEditorCtl** objects.*

# Setting Up Word PlayerEditor

You can choose to embed a Microsoft Word-based **PlayerEditor** in your application if you make the **WordPlayerEditorCtl** ActiveX control available first, as explained in [Making SDK ActiveX Controls Available on page 56](#).

## Embedding WordPlayerEditorCtl Control in Your Application

To add a **WordPlayerEditor** to your application, you use a **WordPlayerEditorCtl** ActiveX control.

### To embed a WordPlayerEditorCtl ActiveX control in your program:

1. Select the **WordPlayerEditorCtl** control in the toolbox and drag and drop the control into your Visual Studio application form.
2. Assign the **WordPlayerEditor** a name. In the code shown here, the control has been named **objPlayerEditor**.

In the HTML source for a JavaScript application, the **WordPlayerEditorCtl** becomes embedded in a **<div>** block with the attribute settings shown below (the **classid** is the piece of information that must be correct):

```
<div id="objWordRptEditor">
 <object id="WordPlayerEditor" ...
 classid="clsid:9D6DE7B5-9D53-4E79-8564-2DD8783B1ADA">
 </object>
</div>
```

3. You then create a function that instantiates this type of ActiveX control:

```
function CreateObjectCtl(divID)
{
 var divRef = document.getElementById(divID);
 if (divRef)
 {
 divRef.innerHTML = divRef.innerHTML;
 }
}
```

4. Call the function to instantiate the **PlayerEditorCtl** control, passing it the value of the **id** attribute for the **<div>** block:

```
<script>
 CreateObjectCtl("objPlayerEditor");
</script>
```

## Associating WordPlayerEditorCtl with Report Object

To associate the **WordPlayerEditorCtl** in your application with the report object, first create a report (for details refer to [Creating the Report Object on page 60](#)), then you indicate the instance of the **WordPlayerEditorCtl** control to use with the report by calling the **SetPlayerEditor()** method of that **Report** object:

```
var objReport;
objReport = pscribeSDK.NewReport("ORD-120406");
objReport.SetPlayerEditor(objWordRptEditor);
```

This method returns the editor in the **WordPlayerEditorCtl** control that you pass it. The **Word PlayerEditor** is automatically enabled by default. Later, you can choose to disable it after you close the report and re-enable it when you start another report.

## Preparing to Work with Report Text

To edit the text of a report, you start by calling the **GetReport()** method of the **PowerscribeSDK** object and passing it the report ID of the report:

```
objReport = pscribeSDK.GetReport("ORD-090206");
```

The method returns a **Report** object that you can then use to lock the report. After you lock and activate the report using the **PowerscribeSDK** object (see below), if you have associated a **PlayerEditor** with the report object, the text of the report displays in that **PlayerEditor**:

```
objReport.ExclusiveLock(true);
objReport.SetPlayerEditor(objPlayerEditor);
pscribeSDK.ActivateReport(objReport);
```

Finally, map **PlayerEditor** events as covered in [Mapping PlayerEditor Events on page 62](#).

You are now ready to work with the text inside the report, using the **PlayerEditorCtl** or **WordPlayerEditorCtl** control.

# Changing Default Text Fonts and Colors in PlayerEditor

When a report displays in the **PlayerEditor**, if the user is not displaying the colors that indicate the source of each piece of text (called *source markup*), the application uses the default color and font for all text as well as the background color of the text.

You can change the default font and colors using the **SetStyle()** method of the **PlayerEditorCtl** control. You pass the method a string containing all style information that should apply to new text as the default:

```
objPlayerEditor.setStyle("font-family:times;font-size:10pt;font-style: normal;font-weight:normal;font-decoration:underline;color:#000000;list-type: none;tab-count:0")
```

These settings apply to all subsequently added text in the **PlayerEditor**. Previously existing text is not affected by changing the default settings.



**Note:** In structured reports, the default font and colors apply to paragraph text only, not headings, as headings are part of the structure of the report, rather than its content.

## Finding and Replacing Text

When you begin a search, the application starts searching at the current location of the cursor.

The application can find the current cursor location in the **PlayerEditor** without you having to write any special code.

Alternatively, your application can position the cursor at a location to start searching from. You can position the cursor programmatically in several ways presented in [Programmatically Positioning Cursor or Selecting Text on page 166](#).

### Finding a String of Text

To find a particular string of text in the report, you can call the **Find()** method of the **PlayerEditorCtl** control and pass it four arguments:

- *strFind*—The string to find
- *bMatchCase*—**True** to match the case of the text in the string
- *bUp*—**False** to search from the top down or **True** to search from the bottom up
- *nStart*—Cursor position to start the search from, **0** for the first character in the report, **-1** for the current cursor location
- *nEnd*—Cursor position where search should stop, **-1** to search to the end of the file; applies to searching down only

For instance, to find the string retrieved in **searchString**, make the search case sensitive, search the entire file from the top down and start at the current cursor position and stop at the end of the file, you would call the **Find()** method as follows:

```
var strStartPos;
strStartPos = objPlayerEditor.Find(searchString, True, False, -1, -1)
```

This method returns the location of the first character of the string found and also selects and highlights the string in the **PlayerEditor**. You can next choose to replace the string, or you could choose to copy or cut the string or take any other action on it, such as change the font or style.

## Replacing the String

To replace the string that **Find()** has just found and highlighted in the **PlayerEditor**, you call the **ReplaceSel()** method of the **PlayerEditorCtl** control and pass it the new string to replace the selected text in the editor:

```
objPlayerEditor.ReplaceSel(replaceString)
```

Whenever you use **ReplaceSel()**, it replaces any text that is selected at the time. If you have not selected any text, the method inserts the text at the current location of the cursor.

# Inserting Text or Tabs at Cursor

## Inserting Text at the Cursor

### To insert text into the report currently displaying in the PlayerEditor:

With no text selected, begin dictating or typing text. Typed text begins at the cursor location. After it is recognized by the *Speech Recognition* engine, dictated text appears beginning at the cursor location.

## Inserting Tabs at the Cursor

### To insert a tab or tabs into the report currently displaying in the PlayerEditor:

With no text selected, call the **InsertTabs()** method of the **PlayerEditorCtl** control and pass it the number of tabs to insert:

```
objPlayerEditor.InsertTabs(1);
```

In any **PlayerEditor**, each tab is five characters.

# Handling TextSelChanged Event— Modifying Style of Text

## Mapping PlayerEditorCtl and WordPlayerEditorCtl Events

Be sure to map the **PlayerEditor** events; see [Mapping PlayerEditor Events on page 62](#).

## Handling TextSelChanged Event to Indicate Format of Selected Text

When the user has selected text in the **PlayerEditor** or moved the cursor, the *SDK* fires a **TextSelChanged** event. When you create a handler to handle this event, the handler should receive three arguments:

- *selStart*—Starting position of the selected text (position of cursor if no text is selected)
- *selEnd*—Ending position of the selected text (position of cursor if no text is selected)
- *style*—Style of the selected text, in a .css style sheet formatted string:

```
"font-family:Arial;font-size:9pt;font-style:italic;font-weight:bold;
text-decoration:underline;color:#000000;source-type:0;list-type:none"
```

The handler would be a handler for an ActiveX control's event, so it takes the format shown below in HTML and can simply call a user-defined function that sets the styles in the GUI:

```
<script for="PlayerEditor" event="TextSelChanged(nStart,nEnd,bsStyle)">
{
 SetStylesInGUI(nStart, nEnd, bsStyles);
}
</script>
```

You can make use of the arguments passed to the handler to, for instance, highlight the formatting button that reflects the style of the currently selected text. First you can check the style of the selected text by parsing the *style* argument string passed to the handler and then, for example, turn on or off the highlight of the button to reflect the style.

You might also set a status of the button to indicate whether or not the style could be applied, making it **False** when the text already has that style (for example, is already bold) or **True** if it does not (that is, could still be made bold). Later your application could use that status in the button click handler to respond to the user clicking the button.

You should minimize the amount of action you take in the **TextSelChanged** event handler—for instance, set the status of the button, then exit and work with the selected text

in your user defined formatting button's button click handler. You can use the **PlayerEditorCtl** control's methods to act on the selected text in several ways:

- Change the font style
- Change the font case (uppercase, lowercase, initial capital letters)
- Underline the text
- Insert tabs in front of lines of text
- Copy or cut and paste (insert) text
- Modify the style of numbers or bullets used in lists
- Make new text entered into numbered or bulleted lists deploy a particular style
- Change a series of lines of the report text into a numbered or bulleted list
- Change the color of the text

You can also undo and redo changes to the text.



*Note: In structured reports, you can select and take action on paragraph text only, not headings, as headings are part of the structure of the report, rather than its content.*

## Setting Style of Text

You can set the style and case of any text selected in the **PlayerEditor**.

One capability the *SDK* provides for managing the style and case of the text in the **PlayerEditor** is the ability to retrieve the style of the text selected whenever a **TextSelChanged** event occurs. If your application has buttons for changing text to bold, italic, and so on, you can highlight the button that matches the selected text style.

You can also set a status attribute of the button to indicate whether the style should be toggled on or off when the button is clicked.

### To utilize the style setting capability:

1. In the **TextSelChanged** event handler, check to see if text has been selected:

```
if (intStart != intEnd)
```
2. Then work with the *style* argument to determine the style of the text; the argument contains a string of style sheet formatted attribute information that looks like this:  
`"font-family:arial;font-size:9pt;font-style:italic;font-weight:bold;text-decoration:underline;color:#000000;source-type:0;list-type:none"`  
You can parse this string for the font style characteristics and set a status variable for the style attribute to **True** if the text is not in the specified style or **False** if it already is. For instance, you could set a **btnBoldStatus** variable to **True** if the text is *not* bold.
3. Exit the function called by the handler. You should keep action taken in the handler to a minimum.

- To toggle the selected text between bold and not bold, in the button click handler you call the **SetSelBold()** method of the **PlayerEditorCtl** control and pass it the setting of the style button, **True** to bold it or **False** to unbold it; in this example, you pass the **btnBoldStatus** that indicates the current style of the text:

```
objPlayerEditor.SetSelBold(btnBoldStatus);
```

The font for the entire word changes even if you select only part of the word.



**Note:** Once the user clicks the cursor outside the **PlayerEditor**, the focus of the cursor is no longer in the report. For instance, the user might have selected text in the GUI, then clicked the **Bold** button. After you call the **SetSelBold()** method to bold the selected text, you need to refocus the cursor on the report, so that the user can continue to work inside the **PlayerEditor** seamlessly—for instance, to continue dictating.

- To refocus the application in the **PlayerEditor**, call the **SetEditorFocus()** method:

```
objPlayerEditor.SetEditorFocus();
```

The cursor returns to its last location in the **PlayerEditor**.

The procedure above applies to any style you might want to assign, using the methods shown in the next table. See also [Setting the Text Case on page 155](#) and [Converting Selected Text to List Items on page 160](#).

#### Text Formatting Methods of PlayerEditorCtl Control

Format Method	Arguments	Action Taken
SetSelBold()	True or False	Bold or unbold selected text.
SetSelItalic()	True or False	Italicize or unitalicize selected text.
SetSelUnderline()	True or False	Underline or remove underline from selected text.

## Sample Code That Parses Style Argument

The code below shows one way of working with the formatted string the handler receives in the *style* argument; in this case the handler breaks the string into substrings for each attribute (separated by the semicolon):

```
function SetStylesInGUI(nStart, nEnd, bsStyles)
{
 var sStyles = new String(bsStyles);
 var arrStyles = new Array();
 arrStyles = sStyles.split(";");
}
```

The handler then initializes some local variables to contain the style of the selected text, making them all **False** (meaning the text is not using that style) until examination of the elements in the array proves otherwise:

```
var bBold = false;
var bItalic = false;
var bUnderline = false;
```

```
var bOrderedList = false;
var bUnorderedList = false;
```

The handler then loops through the array and for each attribute sets the corresponding variable to either **True** or **False** based on whether or not **font-weight** is equal to **bold**, **font-style** is equal to **italic**, or **text-decoration** is equal to **underline**:

```
for (var i = 0; i < arrStyles.length; i++)
{
 var arrStyle = arrStyles[i].split(":");
 var sAttributeName = arrStyle[0];
 var sAttributeValue = new String(arrStyle[1]);
 if (sAttributeName == "font-weight")
 bBold = (sAttributeValue == "bold")?true:false;
 else if (sAttributeName == "font-style")
 bItalic = (sAttributeValue == "italic")?true:false;
 else if (sAttributeName == "text-decoration")
 bUnderline = (sValue == "underline")?true:false;
```

The handler also checks to see whether the **list-type** is set to **ordered** or **unordered**:

```
else if (sAttributeName == "list-type")
{
 if (sAttributeValue == "ordered")
 bOrderedList = true;
 else if (sAttributeValue == "unordered")
 bUnorderedList = true;
}
```

Finally, the handler sets a series of global variables to be used later by the button click handlers for the associated formatting buttons:

```
btnBoldStatus = bBold;
btnItalicStatus = bItalic;
btnUnderlineStatus = bUnderline;
btnOrderedListStatus = bOrderedList;
btnUnorderedListStatus = bUnorderedList;
}
```

You could also work with other attributes. For instance, this code ignores **tab-count**, **font-size**, and **color** attributes, to name a few of them. The application can later use the button status information by passing it to a formatting method of the **PlayerEditorCtl** control.

## Setting the Text Case

To change the capitalization (case) of the selected text, call the **SetSelTextCase()** method of the **PlayerEditorCtl** control and pass it the **WordCaseType** constant that corresponds to the capitalization you want; for instance, to capitalize the first letter of each selected word, you'd call the method as follows:

```
objPlayerEditor.SetSelTextCase(WCT_CapFirst);
```

The **WordCaseType** constants you can pass to **SetSelTextCase()** are shown in the next table.

**WordCaseType Constants of PlayerEditorCtl Control SetSelTextCase() Method**

Constant	Value	Explanation
WCT_Normal	0	Do not change the case.
WCT_AllUpper	1	Change selected words to uppercase.
WCT_AllLower	2	Change selected words to lowercase.
WCT_CapFirst	3	Change selected words to initial capped.
<i>SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.</i>		

## Copying or Cutting and Pasting Text

You can copy and paste or cut and paste selected text.

**To copy (or cut) and paste text from a particular location in the report displaying in the PlayerEditor to another location in the report:**

1. Copy the selected text using the **SendCommand()** method of the **PlayerEditorCtl** control and passing it the **psEditorCommandCopy** constant:

```
objPlayerEditor.SendCommand(psEditorCommandCopy);
```

Or cut the selected text using the **SendCommand()** method of the **PlayerEditorCtl** control and passing it the **psEditorCommandCut** constant:

```
objPlayerEditor.SendCommand(psEditorCommandCut);
```

The method places the copied or cut text on the Windows clipboard.

2. Move the cursor to the location where you want to paste the text. You can place the cursor at any location in the text. For all types of reports, you can move the cursor around by calling the **SendCommand()** method; for instance, to send the cursor to the end of the current line, you would pass this method **psEditorCommandEnd**:

```
objPlayerEditor.SendCommand(psEditorCommandEnd);
```

In a structured report, the pasting location might be a particular **Section** you selected with the **Item()** method of the **Sections** object; you can pass the object for that **Section**

- to the **SetEditorCurrentSection()** method of the **Report** object to position the cursor at the beginning of that section of the report. Refer to [Programmatically Positioning Cursor or Selecting Text on page 166](#) for more information.
3. Paste the text you selected with the **Copy** command or cut with the **Cut** command by calling the **SendCommand()** method of the **PlayerEditorCtl** control and passing it the **psEditorCommandPaste** constant:
- ```
objPlayerEditor.SendCommand(psEditorCommandPaste);
```

EditorCommand Constants of the PlayerEditor Object SendCommand() Method

| Constant | Value | Explanation |
|----------------------|--------------|---|
| psEditorCommandCopy | 1 | Copy selected text in the editor and store the copy on the clipboard. |
| psEditorCommandPaste | 2 | Paste text from the clipboard to the editor. |
| psEditorCommandCut | 3 | Cut selected text from the editor; store it on the clipboard. |
| psEditorCommandHome | 4 | Move the cursor to the beginning of the current line. |
| psEditorCommandEnd | 5 | Move the cursor to the end of the current line. |
| psEditorCommandUndo | 6 | Undo the last change to the text. |
| psEditorCommandRedo | 7 | Redo the last undone change to the text. |

SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

Undoing and Redoing Text Changes

To undo the last action you took with the **SendCommand()** method of the **PlayerEditorCtl** control, call the method again and pass it the **psEditorCommandUndo** constant:

```
var psEditorCommandUndo = 6;  
objPlayerEditor.SendCommand(psEditorCommandUndo);
```

To redo the last action you undid, call **SendCommand()** and pass it **psEditorCommandRedo**:

```
var psEditorCommandRedo = 7;  
objPlayerEditor.SendCommand(psEditorCommandRedo);
```

Creating Numbered or Bulleted Lists

When your application begins to turn lines of text into a numbered list, it uses the default settings for the characteristics of the list:

- Numbered rather than bulleted
- Arabic numerals for numbers
- Period after each number
- Starting at 1

You can modify these settings using the **SetListDefaultOption()** method. This

Defaults for Lists

- | Arabic |
|--------|
| 1. |
| 2. |
| 3. |

method changes the default settings for all new lists subsequently created; the defaults remain in force while the application is running and are lost when it powers down, unless your application saves them and an initialization routine restores them on startup.

After you set the format for the numbered list, you then start the list using the **StartList()** method. All subsequent lines of text entered into the report become numbered list items until you call the **EndList()** method to end the list.

If you are using a **WordPlayerEditorCtl** ActiveX control, as an alternative to the methods outlined here, you might choose to use programming features of Microsoft Word by deploying the **GetInternalObject()** method of the **WordPlayerEditorCtl** (see [Using Microsoft Word Capabilities in Reports on page 172](#) for more information).

Modifying Default List Characteristics

Before you start creating a list, you establish the characteristics of the list. For a numbered list, you choose from numbering options; for a bulleted list, choose from bullet options.

A numbered list can be numbered in one of several styles:

Options for Numbering Lists

| Arabic | Uc Roman | Lc Roman | Uc Letters | Lc Letters |
|--------|----------|----------|------------|------------|
| 1 | I | i | A | a |
| 2 | II | ii | B | b |
| 3 | III | iii | C | c |

The numbered list can also be punctuated in several ways:

Options for Punctuating Numbered Lists

| Plain | Period | Parenthesis | Parentheses |
|-------|--------|-------------|-------------|
| 1 | 1. | 1) | (1) |
| 2 | 2. | 2) | (2) |
| 3 | 3. | 3) | (3) |

Another way that list numbering can be punctuated that applies only to Word reports is with any prefix and/or suffix you choose:

Additional Options for Punctuation of Numbered Lists (Word Reports Only):

| | | | | |
|------------------------------|-----------------------------------|------------------------------|----------------------------------|------------------------------|
| hyphens
-1-
-2-
-3- | # and hyphen
#1-
#2-
#3- | # and %
#1%
#2%
#3% | \$ and =
\$1=
\$2=
\$3= | * and ,
*1,
*2,
*3, |
|------------------------------|-----------------------------------|------------------------------|----------------------------------|------------------------------|

Modifying Defaults for Numbered Lists

To set the default characteristics of numbered lists in the PlayerEditor:

1. Call the **SetListDefaultOption()** method and pass it three arguments:
 - *psNumListTypeOrdered*—The type of list from **psNumListType** table, in this case an ordered list, whether numerically ordered or alphabetically ordered.

psNumListType Options for SetListDefaultOption() Method

| psNumListType | Value | Explanation |
|--|-------|----------------------------|
| psNumListTypeUnknown | 0 | Use the default list type. |
| psNumListTypeOrdered | 1 | Ordered list. |
| psNumListTypeUnordered | 2 | Unordered list. |
| SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants. | | |

- *psNumListStyle*—Style of the numbers or letters for the list, from the next table.

psNumListStyle Options for SetListDefaultOption() Method

| Numbered Lists | | | Bulleted Lists | | |
|--|-----|-------------------|----------------------|-----|---------|
| psNumListStyle | Val | Example | psNumListStyle | Val | Example |
| psNumListStyleArabic | 4 | 1, 2, 3, 4, 5 | psNumListStyleBullet | 1 | 1 |
| psNumListStyleUpperRoman | 5 | I, II, III, IV, V | psNumListStyleSquare | 2 | n |
| psNumListStyleLowerRoman | 6 | i, ii, iii, iv, v | psNumListStyleCircle | 3 | I |
| psNumListStyleUpperLetter | 7 | A, B, C, D, E | | | |
| psNumListStyleLowerLetter | 8 | a, b, c, d, e | | | |
| SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants. | | | | | |

- *bsFormat*—String that indicates how the numbers or letters should join the list item; for example, they could be followed by a period (see following table).

bsFormat String Settings for SetListDefaultOption() Method

| Fomat String | Explanation |
|-------------------|--|
| 1 | Displays the number only. |
| 1. | Displays number or letter followed by period: 1., 2., 3., and so on. Default. |
| 1) | Displays number followed by a parenthesis: 1), 2), 3), and so on. |
| (1) | Encloses the number in parentheses: (1), (2), (3), and so on. |
| <prefix>1<suffix> | Inserts <i>prefix</i> before the number, <i>suffix</i> after it (Word reports only). |

For example, to create a list numbered using lowercase Roman numerals followed by a single parenthesis, you would call the method as follows:

```
objPlayerEditor.SetListDefaultOption(psNumListTypeOrdered,
                                     psNumListStyleLowerRoman, "1") ;
```

In a Word report, to create a list numbered using lowercase letters where each letter is preceded by an asterisk and followed by a comma, you would use this method call:

```
objWordRptEditor.SetListDefaultOption(psNumListTypeOrdered,
                                      psNumListStyleLowerLetter, "*1, ") ;
```

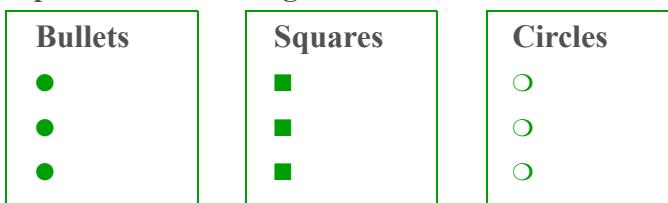
By default, every numbered list starts with the number **1** or the letter **A**; you cannot change the default starting point with this method.

2. Start the list following steps under [Starting and Ending the List on page 160](#).

Modifying Defaults for Bulleted Lists

A bulleted list can have one of three types of bullets—plain bullets, squares, or circles.

Options for Bulleting Lists



To set the default characteristics of bulleted lists in the PlayerEditor:

1. Call the **SetListDefaultOption()** method and pass it two arguments:
 - *psNumListTypeUnordered*—The type of list, in this case an unordered list.
 - *psNumListStyle*—Style of the bullets in the list, from the table: regular bullets, square bullets, or circle bullets.

For a bulleted list, you do not pass the third argument of **SetListDefaultOption()**; if you pass a third argument, the method ignores it.

For example, to create a bulleted list that uses square bullets, use this method call:

```
objPlayerEditor.SetListDefaultOption(psNumListTypeUnordered,  
                                     psNumListStyleSquare);
```

2. Start the list following steps under [Starting and Ending the List on page 160](#).

Starting and Ending the List

To have new lines of text become list items after you have set the list characteristics using SetListDefaultOption():

1. After you have set the list characteristics, to start the list, call the **StartList()** method of the **PlayerEditorCtl** control:

```
objPlayerEditor.StartList();
```

All new dictation or typed text is now formatted with each line as a list item.

2. To end the list, call the **EndList()** method of the **PlayerEditorCtl** control:

```
objPlayerEditor.EndList();
```

Converting Selected Text to List Items

You can convert selected text to a numbered or bulleted list using the **SetSelList()** method of the **PlayerEditorCtl**. You can also use this method to change the number of the first numbered step.

Converting Selected Text to List Items

To convert selected lines of text in the PlayerEditor to a numbered or bulleted list:

1. Select the lines of text to be converted into a numbered or bulleted list.
2. Call the **SetSelList()** method and pass it four arguments:
 - *bList*—**True** to make text lines into a list; **False** to make list items into ordinary paragraphs. If you set this argument to **False**, the method ignores all other arguments.
 - *eType*—Type of list, **psNumListTypeOrdered** or **psNumListTypeUnordered**. Or you can use **psNumListTypeUnknown** to use the list type from the default settings.
 - *eStyle*—Style of the numbers or letters in an ordered list or style of the bullets to use in an unordered list. See tables under [Modifying Default List Characteristics on page 157](#).
 - *nStart*—Integer indicating the number to start the numbered list.
 - *bsFormat*—String that indicates how the numbers or letters should join the list item. See table under [Modifying Default List Characteristics on page 157](#).

For example to create a numbered list that uses lowercase Roman numerals with a single parenthesis after each, starting with three, you would make the following method call:

```
objPlayerEditor.SetSelList(True, psNumListTypeOrdered,  
psNumlistStyleLowerRoman,  
3, "1));
```

The resulting list would be formatted as follows:

```
iii) Amoxicillin  
iv) Penicillin  
v) Cipro
```

Changing Start Number of Selected List Items

To change the number of the first item in a numbered list:

1. If the user has not already selected the numbered list item to modify, select the first numbered item in the numbered list (see [Programmatically Positioning Cursor or Selecting Text on page 166](#)).
2. Determine the number you want to have start the list; you might store the number in a variable:

```
newFirstItemNumber = 4;
```

3. Call **SetSelList()** as outlined under [Converting Selected Text to List Items on page 160](#) and pass it the new number for the first item as the *nStart* argument:

```
objPlayerEditor.SetSelList(True, psNumListTypeOrdered,  
psNumlistStyleLower-  
Roman, newFirstItemNumber, "(1));
```

The resulting list would be formatted as follows:

```
(iv) Amoxicillin  
(v) Penicillin  
(vi) Cipro
```

Choosing Clipboard Text Format

If you are working with a **PlayerEditorCtl**, you can paste text into the report from the Windows clipboard. You can set the format that pasted text should take by setting the **SetCanPasteType** property of the **PlayerEditorCtl** to either **psEditorPasteTypeAll (0)** for all the formats that the Rich Edit Control of the rich text editor supports or **psEditorPasteTypeTextOnly (1)** for plain text only:

```
objPlayerEditor.SetCanPasteType(psEditorPasteTypeAll);
```

Handling CtrlKeyPress Event—Setting Up Control Keys

Another event that the **PlayerEditorCtl** and **WordPlayerEditorCtl** controls fire is the **CtrlKeyPress** event. The ActiveX control fires this event whenever the user holds down the CTRL key and presses an alphabetic key on the keyboard at the same time.

The handler for the event should receive a *char* argument, which contains the character the user has pressed. The handler (sample shown below) can take a different action for each keyboard key:

```
<script for="PlayerEditor" event="CtrlKeyPress(char)">
{
    switch (char)
    {
        case "A":
            // Take action
        case "B":
            // Take action
        ...
    }
</script>
```

This functionality is more useful in a standard **PlayerEditor** than in a Word **PlayerEditor**.

Handling CombineKeysPress Event—Setting Up Control Key Combinations

Another event that the **PlayerEditorCtl** and **WordPlayerEditorCtl** controls fire is the **CombineKeysPress** event. The ActiveX control fires this event whenever the user holds down the CTRL key and presses an alphabetic key on the keyboard at the same time, holds down the ALT key and presses an alphabetic key, or holds down both CTRL and ALT while pressing an alphabetic key.

The handler for the event should receive three arguments—the first two Booleans for the **Ctrl** and **Alt** keys, the third the alphabetic key pressed, from **A** to **Z**. The handler (sample shown below) can take a different action for each combination of keyboard keys:

```
<script for="PlayerEditor" event="CombineKeysPress(bCtrl, bAlt, cKey)">
{
    if ((Ctrl == true) && (Alt == false))
    {
        switch (Key)
        {
```

```
        case "S":
            // Save the report to the server
        case "P":
            // Print the entire report from the server
        ...
    }
    else if ((Ctrl == false) && (Alt == true))
    {
        switch (Key)
        {
            case "S":
                // Save the report to local cache
            case "P":
                // Print the entire report from local cache
            ...
        }
    else if ((Ctrl == true) && (Alt == true))
    {
        alert("Please press either Ctrl or Alt, but not both");
    }
}
</script>
```

This functionality lets you customize your application's response to all CTRL and ALT key combinations.

Handling FunctionKeyPress Event—Setting Up Function Keys

Another event that the **PlayerEditorCtl** and **WordPlayerEditorCtl** controls fire is the **FunctionKeyPress** event. The ActiveX control fires this event whenever the user presses a function key (**F1** through **F12**) on the keyboard.

The handler for the event should receive an *fKeyNum* argument, which contains the number of the particular function key the user has pressed. The handler (sample shown below) can take a different action for each function key:

```
<script for="PlayerEditor" event="FunctionKeyPress( iNum )">
{
    switch (fKeyNum)
    {
        case 1:
            // Take action
        case 2:
            // Take action
        ...
    }
}
```

```
}
```

```
</script>
```

This functionality lets you customize your application's response to each function key.

Handling NumLockKeyPress Event—Working with Locked Number Pad

Another event that the **PlayerEditorCtl** and **WordPlayerEditorCtl** controls fire is the **NumLockKeyPress** event. The ActiveX control fires this event whenever the user presses a special key while **NumLock** is on. The special keys are dash (-), forward slash (/), plus (+), and asterisk (*).

The handler for the event should receive an *sKey* argument, which contains the special character the user has pressed. The handler (sample shown below) can take a different action for each special key:

```
<script For="PlayerEditor" EVENT="NumLockKeyPress (sKey)">
{
    switch (sKey)
    {
        case "-":
            // Take action
        case "/":
            // Take action
        case "+":
            // Take action
        case "*":
            // Take action
        ...
    }
}</script>
```

This functionality is more useful in a standard **PlayerEditor** than in a Word **PlayerEditor**.

Handling ButtonClick Event—Responding to Mouse Button Clicks

Another event that the **PlayerEditorCtl** control fires is the **ButtonClick** event.(The **WordPlayerEditorCtl** ActiveX control does not fire this event.) The **PlayerEditorCtl** ActiveX control fires this event whenever the user clicks the left, middle, or right mouse button. You do not have to handle this event, but if you do, your handler should receive three arguments, the *mkButton* pressed, and the *x* and *y* coordinates of the location clicked. The

event sets the *mkButton* argument to one of the following values based on the particular button that has been clicked:

- **psMKButtonLeft** or 1
- **psMKButtonMid** or 2
- **psMKButtonRight** or 3

The handler for the event can take appropriate action in response to the mouse button click:

```
<script for="PlayerEditor" event="ButtonClick(psmkButton, xcoord, ycoord)">
{
    if (mkButton == 1)
    {
        // Take left mouse button action on x,y location
    }
    else if (mkButton == 2)
    {
        // Take middle mouse button action on x,y location
    }
    else if (mkButton == 3)
    {
        // Take right mouse button action on x,y location
    }
}
</script>
```

Retrieving All Text from PlayerEditor

Earlier you saw that you can retrieve the text in the **PlayerEditor** using the **GetEditorText()** method of the **Report** object. An alternative way to retrieve all report text displaying in the **PlayerEditor** is to call the **GetText()** method of the **PlayerEditorCtl** control instead. This method is similar to the **GetEditorText()** method of the **Report** object, only it always returns the text in plain text format, never in XML format.

To call the **GetText()** method of the **PlayerEditorCtl** control, you pass it the **psEditorTextTypePlain** constant:

```
var psEditorTextTypePlain = 1;
objPlayerEditor.GetText(psEditorTextTypePlain);
```



Note: The **GetText()** of the **Report** object, unlike the **GetText()** method of the **PlayerEditor** ActiveX control, retrieves the saved text of a report from the database, rather than from the **PlayerEditor**.

The **GetText()** method of the **PlayerEditorCtl** returns all report text in a single string, unformatted. You can use this method to copy all text in one report and reuse it in another report or take other appropriate action with it.

Programmatically Positioning Cursor or Selecting Text

You can use the **PlayerEditorCtl** control's methods to position the cursor from your application. You can also select particular text in or retrieve text from the **PlayerEditor** so that you can later take action on that text.

In structured reports, you can select and take action on paragraph text only, not on headings.

These are actions your application can take independent of the user:

- Refocus the cursor in the report after the user has pressed a button or used another GUI indicator
- Position the cursor in the report text
- Select a particular string of text
- Select and retrieve all report text

Refocusing Cursor in Report

If the user clicks the cursor outside the **PlayerEditor** on any GUI element, the focus of the cursor is no longer in the report. For instance, suppose you have a **BOLD** button. The user has selected text in the GUI, then clicked the **BOLD** button. After you call the **SetSelBold()** method to bold the selected text, the cursor needs to be refocused on the report, so that the user can continue to work inside the **PlayerEditor** seamlessly. To refocus the application in the **PlayerEditor**, call the **SetEditorFocus()** method:

```
objPlayerEditor.SetEditorFocus();
```

The cursor returns to its last location in the **PlayerEditor**.

Positioning Cursor in Report Text Programmatically

Before you take action on report text, you might want to move the cursor around in the report. You can use a combination of **Sections** object methods and **PlayerEditorCtl** control methods to control where the cursor is located in the active report currently displaying in the **PlayerEditor**. You can move the cursor:

- To a particular section of the report.
- To the beginning or end of a particular line of the report
- To a particular character location in the report
- To the next field in a plain report or section in a structured report

Positioning Cursor in Particular Report Section

To position the cursor in a particular report section (in a structured report), you first retrieve the **Section** object associated with the section of the report, then call the **SetEditorCurrentSection()** method of the **Report** object and pass it that **Section** object:

```
objSection = Sections.Item(12);  
objReport.SetEditorCurrentSection(objSection);
```

This method places the cursor at the beginning of that section of the report.

Moving Cursor to Start or End of Report Line

To move the cursor to the beginning of the line of text it is currently located on, you can use the **SendCommand()** method of the **PlayerEditorCtl** control and pass it the **psEditorCommandHome** constant:

```
objPlayerEditor.SendCommand(psEditorCommandHome);
```

Similarly, to move the cursor to the end of the line of text it is currently located on, you can use the **SendCommand()** method of the **PlayerEditorCtl** control and pass it the **psEditorCommandEnd** constant:

```
objPlayerEditor.SendCommand(psEditorCommandEnd);
```

Moving Cursor to Particular Character Location

If you know the location of the character whose position you want to start searching from (**1** is the first character and they are numbered sequentially), you can select that character in the report using the **SetSel()** method of the **PlayerEditorCtl** control, which places the cursor at that character. To position the cursor at the first character in the report, you would call the **SetSel()** method and pass it **1** for the *nStart* character argument and **1** for the *nEnd* character argument:

```
objPlayerEditor.SetSel(1, 1);
```

You might position the cursor at the beginning of the report in order to, for instance, insert a masthead that contains the name and address of the organization.

Moving Cursor to the Next Field or Section

To move the cursor to the next field in a plain report or section in a structured report using keystroke combinations, you can call the **NavigateTo()** method of the **Report** object and pass it one of the **NavigatePreference** constants, followed by a **NavigateDirection** constant. The first of these constants indicates whether to navigate through fields or sections. The second indicates whether to navigate to the first, last, next, or previous field or section. See the tables that follow.

NavigateTo() Method NavigatePreference Constants for type Argument

| NavigatePreference | Value | Explanation |
|---------------------------|--------------|--|
| psNavigateTypeAll | 0 | Navigates through both sections and square bracketed field names in structured reports or through only field names in plain reports. When it moves to a field name, selects both the field name and the square brackets enclosing it. Default. |
| psNavigateTypeNextField | 1 | Navigates through square bracketed fields and, in structured reports, skips report sections. Selects the next field in square brackets. |
| psNavigateTypeNextSection | 2 | In structured reports, navigates the cursor to the next section and ignores all square bracketed fields. In plain reports, where there are no sections, this preference behaves the same as psNavigateTypeNextField . |

NavigateTo() Method NavigateDirection Constants for direction Argument

| NavigateDirection | Value | Explanation |
|-----------------------------|--------------|--|
| psNavigateDirectionFirst | 0 | Navigates to first set of square brackets or first section, depending on the first argument you passed. When it moves to a field name, selects both the field name and the square brackets enclosing it. Default. |
| psNavigateDirectionLast | 1 | Navigates to last set of square brackets or last section of the report, depending on the first argument you passed. When it moves to a field name, selects both the field name and the square brackets enclosing it. |
| psNavigateDirectionNext | 2 | Navigates to next set of square brackets or next section, depending on the first argument you passed. When it moves to a field name, selects both the field name and the square brackets enclosing it. |
| psNavigateDirectionPrevious | 3 | Navigates to previous set of square brackets or previous section, depending on the first argument you passed. When it moves to a field name, selects both the field name and the square brackets enclosing it. |

SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

You might call the **NavigateTo()** method inside the **CombineKeysPress** event handler:

```
<script for="PlayerEditor" event="CombineKeysPress (bCtrl, bAlt, cKey)">
{
    if ((bCtrl == true) || (bAlt == true))
    {
        switch (cKey)
        {
```

```
        case "N":  
            // Move cursor to next field  
            objReport.NavigateTo(psNavigateTypeNextField, psNavigateDirectionNext  
            ...  
        }  
    }  
</script>
```

Determining Text Location Programmatically

Your application can find the location of text selected in the **PlayerEditor**.

To find the location of text selected in the PlayerEditor:

1. To determine the location of the first character in a string of selected text, call the **GetSel()** method of the **PlayerEditorCtl** control and pass it the **psEditorSelTypeBegin** constant. The method returns the integer location of the first character selected:

```
var firstSelChar = objPlayerEditor.GetSel(psEditorSelTypeBegin);
```

2. To determine the location of the last character in a string of selected text, call the **GetSel()** method and pass it the **psEditorSelTypeEnd** constant. The method then returns the integer location of the last character selected:

```
var lastSelChar = objPlayerEditor.GetSel(psEditorSelTypeEnd);
```

3. You can then select the text by calling the **SetSel()** method and passing it the first character of the text to select and the last character of the text to select:

```
var strSelected = objPlayerEditor.SetSel(firstSelChar, lastSelChar);
```

Retrieving Audio Corresponding to Selected Text

After your application or the user has selected a particular piece of text in the report, you can retrieve the portion of the associated audio that corresponds to the selected text by calling the **GetDictatedWave()** method of the **Report** object. You pass the method the start position of the selected text, the end position, and the full path to the wave file to store the audio in (for the code below, the values of **firstSelChar** and **lastSelChar** were retrieved using the **GetSel()** method in [Determining Text Location Programmatically on page 169](#)):

```
objReport.GetDictatedWave(firstSelChar, lastSelChar, "C:\WaveFiles\phrase.wav");
```

Manipulating Audio Playback or Rewind Speed

If you have transcription users who strictly listen to audio and transcribe or edit text of the report, you may want to provide GUI components that allow those users to speed up or slow down the audio playback and rewind speeds. When the user clicks a button, for instance, your application could call the **PlaybackSpeed()** method of the **Report** object to increment or decrement the speed of the audio. You pass the method a constant that indicates whether the speed should be faster or slower, **psMicSpeedTypeFaster** (1) or **psMicSpeedTypeSlower** (2); each time you call the **PlaybackSpeed()** method, the speed changes by a single increment in the direction you indicate, so you may want to call the method multiple times:

```
objReport.PlaybackSpeed(psMicSpeedTypeFaster);
```

To modify the speed at which the audio rewinds, your application can call the **WindingSpeed()** method of the Report object and pass it one of the same two constants that it can pass to **PlaybackSpeed()**:

```
objReport.WindingSpeed(psMicSpeedTypeFaster);
```

Again, each call to the **WindingSpeed()** method changes the speed by a single increment, so you may want to call the method multiple times.

Both the playback and rewinding speeds set by calling these methods apply only to the report currently being edited.

Forcing Shortcuts to Expand in Editor

Expanding Shortcuts

You can have the voice shortcuts in the report currently displaying in the editor expand to display their long text in the editor by calling the **ExpandShortcuts()** method of the **Report** object:

```
objReport.ExpandShortcuts();
```

After you take this action, then save the report, the report is stored with all shortcuts permanently expanded.

Handling the ShortcutsExpanded Event

Whenever voice shortcuts are expanded, whether by calling the **ExpandShortcuts()** method, using buttons on the microphone, or calling **MergeTemplate()** with a source type of **psMergeSrcTypeShortcut**, the **Report** object fires the **ShortcutsExpanded** event.

The event has one argument that indicates the number of shortcuts that were just expanded by the last method call or action to expand them:

```
function HandleShortcutsExpanded(nNumber)
{
    window.status (nNumber + "shortcuts expanded")
}
```



Note: If you expand a shortcut and later delete that expanded text from the report, the count of the number of shortcuts expanded (stored in the **NumberOfShortcuts** property) still includes the deleted one, because it is still in the audio.

Zooming In or Out on Report Text

If you are working with a **PlayerEditorCtl**, you can zoom in or out to change the size of the text displaying in the **PlayerEditor**. You take this action by calling the **SetZoom()** method of the **PlayerEditorCtl** and passing it the percentage of the actual font size you want to display (minimum is **10** percent):

```
objPlayerEditor.SetZoom(txtPercentView);
```

Dragging/Dropping Text into PlayerEditor

You can drag text from *Microsoft WordPad*, *Visual Studio*, or any plain text editor on your computer that lets you drag text and then drop that text into a **PlayerEditor** displaying a plain text report. When you drag and drop the text, the **PlayerEditorCtl** makes that text part of the report. You take no special action in your application to make this technology function.

Similarly, you can drag text from a *Microsoft Word* file and drop it into a **WordPlayerEditor** displaying a *Microsoft Word* formatted report. The **WordPlayerEditorCtl** then integrates that text into the report.

In either the **PlayerEditor** or **WordPlayerEditor** you can enhance this dragging and dropping process to also expand shortcuts or apply templates found in the dragged text.

To take this action you need to write extra code outlined in [Auto Expanding Shortcuts After Drag and Drop into PlayerEditor—C# on page 314](#).

Using Microsoft Word Capabilities in Reports



Nuance Technical Support does not support programming in Word. For technical support on using the Word Document object, please contact Microsoft technical support.

If you are working with a Word **PlayerEditor**, you can create a **WordDocument** object and then use the Microsoft Word functionality available to a **WordDocument** object.

To retrieve the **WordDocument** object, you use the **GetInternalObject()** method of the **WordPlayerEditorCtl**:

```
var WordDoc = objWordRptEditor.GetInternalObject();
```

You can now use this object to access **WordDocument** methods, properties, and events through your *PowerScribe SDK* application.

Code Summary

```
<HTML>
<OBJECT ID="pscribeSDK">
</OBJECT>
<HEAD>
<TITLE>Powerscribe SDK</TITLE>
<!-- This code summary is not a complete program; it assumes
inclusion of required code from the previous chapters -->
<SCRIPT type="text/javascript">

function InitializePS()
{
    try
    {   // Instantiate PowerScribe SDK Object
        pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");
        // Initialize PowerScribe SDK Server
        var url = "http://server2/pscribesdk/";
        pscribeSDK.Initialize(url, "JavaScript_Demo");
    }
    catch ...
}

function CreateStructReport()
{
    var xmlString = ReadXmlToString(structXml);
    var objReport;
    objStructRept = pscribeSDK.NewReport(strReportID, xmlString);
```

```
//Use EventMapper to Map COM Object Events to Their Handlers
//But DO NOT use EventMapper for events of ActiveX controls, like
//the PlayerEditorCtl

ReptSink.ActivateEnd = HandleActivateEnd;
ReptSink.LoadEnd = HandleLoadEnd;
ReptSink.ReportChanged = HandleReportChanged;
ReptSink.SaveEndEx = HandleSaveEndEx;
ReptSink.WaveFileProgress = HandleWaveFileProgress;

ReptSink.Advise(objStructRept);
...
objStructRept.ExclusiveLock(true);
objStructRept.SetPlayerEditor(objPlayerEditor);
pscribeSDK.ActivateReport(objStructRept);

// Set Global Paragraph Style
objPlayerEditor.setStyle("font-family:times;font-size:10pt;font-style:
    normal;font-weight:normal;font-decoration:underline;color:#000000;
    list-type:none;tab-count:0")
}

function ReadXmlToString()
{
    // Custom function, reads XML from a file and puts it into a string
}

function EditReport()
{
    objStructRept = pscribeSDK.GetReport(strReportID);
}

function SearchAndReplace()
{
    var strStartPos;

    strStartPos = objPlayerEditor.Find(searchString.text, True, False, -1)
    // Find() method automatically highlights the string when found
    objPlayerEditor.ReplaceSel(replaceString)
    // ReplaceSel() replaces the highlighted text with replacement string
}

function SetStylesInGUI(nStart, nEnd, bsStyles)
{
    var sStyles = new String(bsStyles);
    var arrStyles = new Array();
    arrStyles = sStyles.split(";");
    var bBold = false;
    var bItalic = false;
    var bUnderline = false;
```

```
var bOrderedList = false;
var bUnorderedList = false;

for (var i = 0; i < arrStyles.length; i++)
{
    var arrStyle = arrStyles[i].split(":");
    var sAttributeName = arrStyle[0];
    var sAttributeValue = new String(arrStyle[1]);
    if (sAttributeName == "font-weight")
        bBold = (sAttributeValue == "bold")?true:false;
    else if (sAttributeName == "font-style")
        bItalic = (sAttributeValue == "italic")?true:false;
    else if (sAttributeName == "text-decoration")
        bUnderline = (sValue == "underline")?true:false;
    else if (sAttributeName == "list-type")
    {
        if (sAttributeValue == "ordered")
            bOrderedList = true;
        else if (sAttributeValue == "unordered")
            bUnorderedList = true;
    }
}

btnBoldStatus = bBold;
btnItalicStatus = bItalic;
btnUnderlineStatus = bUnderline;
btnOrderedListStatus = bOrderedList;
btnUnorderedListStatus = bUnorderedList;
}

function InsertTab()
{
    // Inserts tab at cursor location in the report displaying in Editor
    objPlayerEditor.InsertTabs(1);
}

function BoldText()
{
    objPlayerEditor.SetSelBold(btnBoldStatus);
    objPlayerEditor.SetEditorFocus();
}

function ItalicizeText()
{
    objPlayerEditor.SetSelItalic(btnItalicStatus);
    objPlayerEditor.SetEditorFocus();
}

function UnderlineText()
{
    objPlayerEditor.SetSelUnderline(btnUnderScStatus);
```

```
    objPlayerEditor.SetEditorFocus();
}

function NumberedList()
{
    // Converts selected text to numbered list using Arabic numbers/periods after
    objPlayerEditor.SetSelList(True, psNumListTypeOrdered, psNumListStyleArabic,
        1, "1.");
    objPlayerEditor.SetEditorFocus();
}

function BulletedList()
{
    // Converts selected text to bulleted list with square bullets
    objPlayerEditor.SetSelList(True, psNumListTypeUnordered,
psNumListStyleSquare);
    objPlayerEditor.SetEditorFocus();
}

function SetTextCaseInitialCap()
{
    // if strCase indicates to initial cap selected text:
    objPlayerEditor.SetSelTextCase(WCT_CapFirst);
    objPlayerEditor.SetEditorFocus();
}

function SetTextCaseAllCap()
{
    // if strCase indicates to capitalize all selected text:
    objPlayerEditor.SetSelTextCase(WCT_AllUpper);
    objPlayerEditor.SetEditorFocus();
}

function SetTextCaseAllLower()
{
    // if strCase indicates to lowercase selected text:
    objPlayerEditor.SetSelTextCase(WCT_AllLower);
    objPlayerEditor.SetEditorFocus();
}

<script for="PlayerEditor" event="CtrlKeyPress(char)">
{
    switch (char)
    {
        case "A":
            // Take action
        case "B":
            // Take action
        ...
    }
}
</script>
```

```
<script for="PlayerEditor" event="CombineKeysPress(bCtrl, bAlt, cKey)">
{
var psNavigateTypeNextField = 1;
var psNavigateDirectionNext = 2;

if ((bCtrl == true) && (bAlt == false))
{
    switch (cKey)
    {
        case "S":
            // Save the report to the server
        case "P":
            // Print the entire report from the server
        case "N":
            // Move cursor to next field
            objReport.NavigateTo(psNavigateTypeNextField, psNavigateDirectionNext
            ...
    }
else if ((bCtrl == false) && (bAlt == true))
{
    switch (cKey)
    {
        case "S":
            // Save the report to local cache
        case "P":
            // Print the entire report from local cache
            ...
    }
else if ((bCtrl == true) && (bAlt == true))
{
    alert("Please press either Ctrl or Alt, but not both");
}
}
</script>

<script for="PlayerEditor" event="FunctionKeyPress(fKeyNum)">
{
switch (fKeyNum)
{
    case 1:
        // Take action
    case 2:
        // Take action
        ...
}
</script>
```

```
<script for="PlayerEditor" event="NumLockKeyPress(sKey)">
{
    switch (sKey)
    {
        case "-":
            // Take action
        case "/":
            // Take action
        case "+":
            // Take action
        case "*":
            // Take action
        ...
    }
}
</script>

<script for="PlayerEditor" event="ButtonClick(psmkButton, xcoord, ycoord)">
{
    if (psmkButton == 1)
    {
        // Take left mouse button action on x,y location
    }
    else if (psmkButton == 2)
    {
        // Take middle mouse button action on x,y location
    }
    else if (psmkButton == 3)
    {
        // Take right mouse button action on x,y location
    }
}
</script>

function ZoomInOut(txtPercentView)
{
    objPlayerEditor.SetZoom(txtPercentView);
}

function Expand()
{
    objReport.ExpandShortcuts();
}

// Set Paste from Clipboard to RTF or Plain Text Format
function PastePref(nSetting)
{
    var psEditorPasteTypeAll = 0;
```

```
var psEditorPasetTypeTextOnly = 1;

if (nSetting == 0)
{
    objPlayerEditor.SetCanPasteType(psEditorPasteTypeAll);
}
else if (nSetting == 1)
{
    objPlayerEditor.setCanPasteType(psEditorPasteTypeTextOnly);
}
}

function HandleShortcutsExpanded(nNumber)
{
    window.status (nNumber + "shortcuts expanded")
}

function GetAudioSelectedText(wavefilename)
{
    var fstSelChar = objPlayerEditor.GetSel(psEditorTypeBegin);
    var lstSelChar = objRptEdtior.GetSel(psEditorTypeEnd);
    objReport.GetDictatedWave(fstSelChar, lstSelChar, "C:\Wavefies\" +
        wavefilename + ".wav");
}

function ChangePlayBackSpeed(plusOrMinus)
{
    var psMicSpeedTypeFaster = 1;
    var psMicSpeedTypeSlower = 2;

    if (plusOrMinus == 1)
        objReport.PlaybackSpeed(psMicSpeedTypeFaster);

    if (plusOrMinus == 2)
        objReport.PlaybackSpeed(psMicSpeedTypeSlower);
}

function ChangeRewindSpeed(plusOrMinus)
{
    var psMicSpeedTypeFaster = 1;
    var psMicSpeedTypeSlower = 2;

    if (plusOrMinus == 1)
        objReport.WindingSpeed(psMicSpeedTypeFaster);

    if (plusOrMinus == 2)
        objReport.WindingSpeed(psMicSpeedTypeSlower);
}
```

```
// Retrieve all text in PlayerEditor

function GetAllTextNoFormat()
{
    var psEditorTextTypePlain = 1;
    var strHoldAllText;

    strHoldAllText = objPlayerEditor.GetText(psEditorTextTypePlain);

    // Save plain text in a file
}

</SCRIPT>
</HEAD>

<BODY bgcolor="#fffff" language="javascript" onload="InitializePS()" onunload="UninitializePS()">

<form>
...
<div id="objPlayerEditor">
    <object id="PlayerEditor" style="WIDTH: 605px; HEIGHT: 265px; BACKGROUND-COLOR: #EFE7CE" ...
        classid="clsid:A470D150-C3FF-4584-9B0B-7D31AF4621C1" viewastext>
    </object>
</div>

<div id="objLevelMeter">
    <object id="LevelMeter" height="20" width="150" ...
        classid="clsid:492FFC0D-0A4A-4EAD-BC52-966AE1568FCD" viewastext>
    </object>
</div>

...
<!-- Action Buttons for Creating Report, Search &amp; Replace, Edit Report, and Insert Tab --&gt;

&lt;input value="Create Structured Report" type="button" name="btnStructRept"
    language="javascript" onclick="CreateStructReport()" style="FONT-STYLE: bold"&gt;
&lt;input value="Find &amp; Replace" type="button" name="btnFindandReplace"
    language="javascript" onclick="SearchAndReplace()" style="FONT-STYLE:bold"&gt;
&lt;input value="Edit Report" type="button" name="btnEditReport"
    language="javascript" onclick="EditReport()" style="FONT-STYLE:bold"&gt;
&lt;input value="Insert Tab" type="button" name="btnInsertTab"
    language="javascript" onclick="InsertTab()" style="FONT-STYLE:bold"&gt;
...
</pre>
```

```
<!-- Toggle Buttons for Formatting Text as Bold, Italic, Underlined,
Initial Cap, All Caps, All Lowercase, Numbered or Bulleted Lists-->

<input value="Bold" type="button" name="btnBold" language="javascript"
      onclick="BoldText()" style="FONT-STYLE: bold">
<input value="Italics" type="button" name="btnItalics" language=
      "javascript" onclick="ItalicizeText()" style="FONT-STYLE:bold">
<input value="Underscore" type="button" name="btnUnderline" language=
      "javascript" onclick="UnderlineText()" style="FONT-STYLE:bold">
<input value="Initial Cap Text" type="button" name="btnInitCap"
      language="javascript" onclick="SetTextCaseInitialCap()" style=
      "FONT-STYLE:bold">
<input value="Capitalize All" type="button" name="btnAllCaps"
      language="javascript" onclick="SetTextCaseAllUpper()" style="FONT-STYLE:
      bold">
<input value="Lowercase All" type="button" name="btnAllLowercase"
      language="javascript" onclick="SetTextCaseAllLower()" style="FONT-STYLE:
      bold">
<input value="Numbered List" type="button" name="btnNumbList"
      language="javascript" onclick="NumberedList()" style="FONT-STYLE:
      bold">
<input value="Bulleted List" type="button" name="btnBulletedList"
      language="javascript" onclick="BulletedList()" style="FONT-STYLE:
      bold">
...
<!-- Buttons for Moving Cursor to Start/End of Line -->
<input value="Start of Line" type="button" name="btnMoveCursorBegin"
      language="javascript" onclick="objPlayerEditor.SendCommand
      (psEditorCommandHome);" style="FONT-STYLE:bold">
<input value="End of Line" type="button" name="btnMoveCursorEnd"
      language="javascript" onclick="objPlayerEditor.SendCommand
      (psEditorCommandEnd);" style="FONT-STYLE:bold">
...
<!-- Buttons for Moving Cursor to Start of Section Cursor Located In -->
<input value="Start of Section" type="button" name="btnMoveToSectionStart"
      language="javascript" onclick="objReport.SetEditorCurrentSection
      (objSection);;" style="FONT-STYLE:bold">
...
<!-- Buttons for Copying, Cutting, Pasting Selected Text -->
<input value="Copy" type="button" name="btnCopy" language="javascript"
      onclick="objPlayerEditor.SendCommand(psEditorCommandCopy);"
      style="FONT-STYLE:bold">
<input value="Cut" type="button" name="btnCut" language="javascript"
      onclick="objPlayerEditor.SendCommand(psEditorCommandCut);"
      style="FONT-STYLE:bold">
<input value="Paste" type="button" name="btnPaste" language="javascript"
      onclick="objPlayerEditor.SendCommand(psEditorCommandPaste);"
      style="FONT-STYLE:bold">
...

```

```

<!-- Buttons for Undoing and Redoing Last Action on Selected Text -->
<input value="Undo" type="button" name="btnUndo" language="javascript"
   onclick="objPlayerEditor.SendCommand(psEditorCommandUndo);"
   style="FONT-STYLE:bold">
<input value="Redo" type="button" name="btnRedo" language="javascript"
   onclick="objPlayerEditor.SendCommand(psEditorCommandRedo);"
   style="FONT-STYLE:bold">
...
<!-- Button for Retrieving all Text without Format to Store in File -->
<input value="Copy" type="button" name="btnCopy" language="javascript"
   onclick="objPlayerEditor.GetAllTextNoFormat();" style="FONT-STYLE:bold">
...
<!-- Button for Expanding Shortcuts -->
<input value="Expand Shortcuts" type="button" name="btnCopy"
   language="javascript" onclick="Expand();" style="FONT-STYLE:bold">

<!-- Radio Buttons for Zooming In/Out by Percentage -->
<p><style="FONT-STYLE:bold">Zoom In/Out by Selected Percentage</p>
<input value="Zoom to 80%" type="button" name="btnRTF"
   language="javascript" onclick="ZoomInOut(80)" style="FONT-STYLE:bold">
<input value="Zoom to 100%" type="button" name="btnPlain"
   language="javascript" onclick="ZoomInOut(80)" style="FONT-STYLE:bold">

<!-- Radio Buttons for Choosing Type of Text to Paste from Clipboard -->
<input value="Past as RTF Format" type="button" name="btnRTF"
   language="javascript" onclick="PastePref(1)" style="FONT-STYLE:bold">
<input value="Paste As Plain Text" type="button" name="btnPlain"
   language="javascript" onclick="PastePref(0)" style="FONT-STYLE:bold">

<!-- Buttons for Increasing or Decreasing Audio Playback Speed -->
<input value="Speed Up Audio" type="button" name="btnSpeedup"
   onclick="ChangePlaybackSpeed(1)" style="FONT-STYLE:bold">
<input value="Slow Down Audio" type="button" name="btnSlow"
   onclick="ChangePlaybackSpeed(2)" style="FONT-STYLE:bold">
...
<!-- Buttons for Increasing or Decreasing Audio Rewind Speed -->
<input value="Speed Up Rewind" type="button" name="btnFastRwd"
   onclick="ChangeRewindSpeed(1)" style="FONT-STYLE:bold">
<input value="Slow Down Rewind" type="button" name="btnSlowRwd"
   onclick="ChangeRewindSpeed(2)" style="FONT-STYLE:bold">

<!-- Field for wave file name and button for saving report to that file -->
<input type="text" value="Wave File Name" name="txtFileName" size="32">
<input value="Save WaveFile to File" type="button" name="btnSaveWave"
   language="javascript" onclick="GetAudioSelectedText(txtFileName)">
...
</form>
</body>

```


Dictating Outside a Report

Objectives

This chapter covers how to use the *SDK* to recognize dictated audio for purposes other than creating a report. Some of this information is more straightforward if you've already read earlier chapters about creating reports and training words. Although the code snippets in this chapter are in C#, code summaries in both C# and JavaScript are provided at the end of the chapter.

The chapter covers several topics:

- [Preparing to Accept Recognized Text Outside a Report](#)
- [Creating the Note Object](#)
- [Declaring the RecognizeEnd and ShortcutsExpanded Events](#)
- [Creating PlayerEditor for the Note](#)
- [Setting Dictation Preferences for the Note](#)
- [Securing Microphone for Use of the Note](#)
- [Retrieving Recognized Text from Note—Handling the RecognizeEnd Event](#)
- [Working with the Note Data](#)
- [Working with Shortcuts in a Note—Handling ShortcutsExpanded Event](#)
- [Working with Note Audio File](#)
- [Releasing the Microphone from Exclusive Use by the Note](#)
- [Using the Note Object in JavaScript](#)

Preparing to Accept Recognized Text Outside a Report

Sometimes you want to dictate audio for purposes other than creating a report. For instance, you might want to be able to dictate a part number or customer ID to look up in a database—dictating transient information that your application does not need to save but needs to retrieve and use immediately, then discard. The *SDK* provides functionality for retrieving transient and disposable data that does not require persistence.

To receive recognized dictation of transient data in your *PowerScribe SDK* application, you use a **Note** object rather than a **Report** object. The **Note** object receives dictation without requiring that the audio become a report. You can think of a **Note** object as a disposable container that holds the dictation data until it is otherwise managed.

Like a **Report** object, the **Note** object works with an editor. The Note can use two possible types of editors:

- **PlayerEditor** like a **Report** uses (in JavaScript)
- **CustomEditController** object that lets you use any Windows GUI component, such as a text box, to receive dictation (in C++, C#, or Visual Basic .NET)

The **Note** can also deploy a **LevelMeter** (optional) and requires a **Microphone** object to receive the dictation. You secure the editor, **LevelMeter**, and **Microphone** objects for the exclusive use of the **Note** object, just as you do for exclusive use of the **PhoneticTranscriber** object when training words (see [Preparing to Train Voice Shortcuts, Words, or Punctuation Marks](#)).

Like a **Report** object, once it receives dictation, the **Note** has audio, text, and concordance files. However, unlike a **Report** object, the **Note**'s files are not stored in the database; instead, this object holds the related files in memory and those files exist only until your application retrieves the recognized text and discards the **Note**.

To set up your application for dictation into a **Note** object:

- Create a **Note** object
- Declare events of the **Note**
- Create a **PlayerEditor** or **CustomEditController** and associate it with the **Note**
- (Optional) Create a **LevelMeter** and associate it with the **Note**
- Set **Dictation** preferences such as the **Dictate** mode or **Command** mode and whether or not to receive the text in a stream
- Secure the **Microphone** object for the exclusive use of the **Note** until the note is complete
- Dictate into the microphone and retrieve the data from the **Note**
- Handle the **RecognizeEnd** event and, optionally, handle the **ShortcutsExpanded** event
- Work with data from the **Note**
- Release the **Microphone** object from exclusive use of the **Note** object

Creating the Note Object



Note: PowerScribe SDK provides the **CustomEditController** object only in C#, Visual Basic .NET, and other Visual Studio languages that use graphical components.

Before you create a **Note** object, you need to declare the **Note** in the class of your application alongside the **PowerscribeSDK** object and initialize each of them to null. The **Note** object, like the **PowerscribeSDK** object, is in the **POWERSCRIBESDKLib** library:

```
private POWERSCRIBESDKLib.PowerscribeSDK pscribeSDK;
private POWERSCRIBESDKLib.Note clsNote;

pscribeSDK = null;
clsNote = null;
```

In addition, you might need a string you will use to retrieve and hold the text of the **Note**:

```
public string strLookupKey;
```

You then create the **Note** object by calling the **NewNote()** method of the **PowerscribeSDK** object and passing it an empty string if you do not have an XML template you want to use to format the note:

```
clsNote = pscribeSDK.NewNote("");
```

If you want to have the text in the field be in bold or italicized or have other text properties, you can pass **NewNote()** an XML template, like a template you would use to format a structured report.

```
clsNote = pscribeSDK.NewNote("<clu xmlns=\"http://www.dictaphone.com
...</clu>");
```



Note: PowerScribe SDK has no method to save the content of a **Note** object the way it does for a **Report** object. Instead, **Notes** are designed to exist only in memory; your application can choose to save **Note** data by using non-SDK functionality.

Declaring the RecognizeEnd and ShortcutsExpanded Events

Each time the **Note** receives dictation and recognizes the audio, it fires the **RecognizeEnd** event. To be able to later handle this event, after you create the **Note** object, you declare the event:

```
clsNote.RecognizeEnd +=new _INoteEvents_RecognizeEndEventHandler
    (clsNote_RecognizeEnd);
```

After the **Note** receives dictation and recognizes the audio, it also automatically expands any shortcuts from that audio, if the audio contains shortcuts. If shortcuts have been expanded, the

Note object also fires the **ShortcutsExpanded** event. To later be able to handle this event, you need to declare it:

```
clsNote.ShortcutsExpanded += new _INoteEvents_ShortcutsExpandedEventHandler  
    (clsNote_RecognizeEnd);
```

Creating PlayerEditor for the Note

To create a **PlayerEditor** for the **Note**, you first declare the ActiveX control in the class. If you drag the **PlayerEditorCtl** from the **ToolBox** into the application, the declaration appears in the code as follows:

```
private AxPOWERSCRIBESDKLib.AxPlayerEditorCtl axPlayerEditorCtl1;
```

Creating CustomEditController for the Note

A **CustomEditController** is a creatable COM object that you use to turn any Windows graphical component into a **PlayerEditor**-like area that accepts the recognized text resulting from dictation. To create a **CustomEditController**, you first declare it in the class of your application:

```
private POWERSCRIBESDKLib.CustomEditController clsCustEdit;
```

In addition, you should create the Windows graphical component that you want the **CustomEditController** to use; for instance, you could create a text box where the user can enter a key to be looked up in the database and its declaration would look like this:

```
private System.Windows.Forms.TextBox txtLookupKey;
```

You then create the **CustomEditController** instance:

```
this.clsCustEdit = new CustomEditController();
```

You then associate the Windows component with the **CustomEditController** by calling the **SetHWnd()** method of the **CustomEditController** object and passing it the name of the graphical component as a long integer (**Int32**):

```
clsCustEdit.SetHWnd(txtLookupKey.Handle.ToInt32());
```

Finally, you associate the **CustomEditController** with the **Note** by calling the **SetCustomEditControl()** method of the **Note** object and passing it the **CustomEditController** object:

```
clsNote.SetCustomEditControl(clsCustEdit);
```

Creating LevelMeter for the Note

To create a **LevelMeter** for the **Note**, you drag the **LevelMeter** onto the form and Visual Studio adds the declaration of the ActiveX control to the class:

```
private AxPOWERSCRIBESDKLib.AxLevelMeterCtl axLevelMeterCtl1;
```

Associating LevelMeter with Note

Before the **Note** object can use the editor or **LevelMeter**, you:

- Set one of two possible objects to be the current **PlayerEditor** for the **Note**:
 - The **PlayerEditor**
 - or
 - The Windows component associated with the **CustomEditController**
- Optionally, secure the **LevelMeter** to work exclusively with the **Note**

Setting PlayerEditor to the Current Editor for Note

To indicate that the instance of the **PlayerEditor** ActiveX control in the application is the current editor for the **Note**, you call the **SetPlayerEditor()** method of the **Note** object and pass it the instance of the control:

```
clsNote.SetPlayerEditor(axPlayerEditorCtl1);
```

Setting Windows Component to Current Editor for Note

The Windows component where you expect to see the recognized text from dictation appear in your application's GUI may be only a text box, but in this situation it also needs to be an editor for the dictation results.

To indicate that the Windows GUI component associated with the **CustomEditController** object is the current *editor* for the **Note**, you call the **SetPlayerEditor()** method of the **Note** object and pass it the **CustomEditController** COM object:

```
clsNote.SetPlayerEditor(clsCustEdit);
```

Securing LevelMeter for Use by the Note

You are not required to have a **LevelMeter** that shows the volume level during dictation, but you can have one for a **Note**.

To both instantiate the **LevelMeter** object and set the instantiation of it to be used by the **Note** object, call the **SetLevelMeter()** method of the **Note** object and pass it the particular **LevelMeter** object retrieved by **GetOcx()**:

```
clsNote.SetLevelMeter((axLevelMeterCtl) this.axLevelMeterCtl1.GetOcx());
```

Setting Dictation Preferences for the Note

Now, you can set up dictation preferences for the **Note** object just as you would for a **Report** object. You take these actions to accomplish that goal:

- Set the mode of dictation for the **Note** object
- Optionally, set the **Note** to receive streamed text

Setting Dictation Mode for the Note

You can set up the note to receive dictated words, default grammar commands, numbers, or letters. If, for instance, you want to have the report ID be a series of characters followed by numbers, such as HSP-100126, you would want to use a mode called **Letter** mode, which recognizes both letters and numbers when the user dictates individual characters and digits. To set the note to **Letter** mode, you set the **Mode** property of the **Note** object to the **psSystemModeLetter** constant (see all modes in next table):

```
clsNote.Mode = POWERSCRIBESDKLib.SystemMode.psSystemModeLetter;
```

Settings for Mode Property of Note Object

| SystemMode Constant | Value | Explanation |
|-------------------------|-------|--|
| psSystemModeDictateOnly | 0 | Dictate Only mode. No commands are recognized; all dictation is assumed to be speech and translated into text. |
| psSystemModeDictate | 1 | Dictate mode. Speech is translated to text and transcribed. No user defined commands are triggered in this mode. Only default grammar commands are triggered. |
| psSystemModeCommand | 2 | Command mode. User defined commands are triggered and all other speech, including default grammar commands, is discarded. |
| psSystemModeNumber | 3 | Number mode. Recognizes only numbers. Recognizes and transcribes numeric digits as the speaker says them. |
| psSystemModeLetter | 4 | Letter mode. Recognizes and transcribes letters and numbers as the speaker spells the word(s). |

Setting the Note to Receive Streamed Text

To have the recognized text resulting from the dictation stream into the field rather than waiting for the user to click a **Transcribe** button on the microphone, you can set the **TextStreaming** property of the **Note** object to **True**:

```
clsNote.TextStreaming = true;
```

If you prefer to force the user to click a **Transcribe** button before inserting the text in the field, you can set the property to **False**:

```
clsNote.TextStreaming = false;
```

Securing Microphone for Use of the Note

You then secure the microphone for exclusive use by the **Note** object by calling the **SetMicrophone()** method of the **Note** object and passing it the **Microphone** object:

```
clsNote.SetMicrophone(m_Microphone);
```

Retrieving Recognized Text from Note—Handling the RecognizeEnd Event

To begin receiving dictation, you use the microphone to dictate the **Note**, just as you would to dictate a report. When you transcribe text or when streaming text ends and the recognition engine has recognized the audio, you receive the **RecognizeEnd** event that the **Note** object fires.

To exit this event handler quickly, do not take extensive action in the event handler; instead, acknowledge that the **Note** audio has been recognized and take action on the text outside of the handler:

```
private void clsNote_RecognizeEndEventHandler()
{
    // Acknowledge note audio has been recognized and exit.
}
```

Working with the Note Data

Once you have received the recognized text in the **Note**, you can now retrieve that text, or retrieve the audio or concordance file associated with the text. You might take these actions in button event handlers.

Retrieving Text from the Note

You then retrieve the text from the **Note** and put it in a string by calling the **GetText()** method of the **Note** object. You pass the method the **TextFormatType** you want for the text, in this case **psTextFormatTypePlain**, from the **POWERSCRIBESDKLib** library:

```
private void SearchDatabase(strLookupKey)
{
    strLookupKey = clsNote.GetText(POWERSCRIBESDKLib.TextFormatType.
        psTextFormatTypePlain);
    MessageBox.Show("Searching database for " + strLookupKey);
    // Search Database
}
```

Retrieving Audio of the Note

To retrieve the audio file for the **Note**, you can retrieve the file using the **GetWaveFile()** method of the **Note** object:

```
clsNote.GetWaveFile("C:\LookupKey.wav");
```

Retrieving Concordance of the Note

To work with the associated concordance file for the **Note**, you can retrieve the file using the **GetConcordanceFile()** method of the **Note** object:

```
clsNote.GetConcordanceFile("C:\LookupKey.ast");
```

Working with Shortcuts in a Note—Handling ShortcutsExpanded Event

When you dictate into a **Note**, you can use shortcuts, just as you would in a **Report**. Once you transcribe the recognized text, the shortcuts immediately expand and in addition to firing the **RecognizeEnd** event, the **Note** object then fires the **ShortcutsExpanded** event.

You should declare this event if the user might dictate a shortcut into the text box:

```
clsNote.ShortcutsExpanded +=new _INoteEvents_ShortcutsExpandedEventHandler
    (clsNote_ShortcutsExpanded);
```

The handler receives a single argument, the number of shortcuts expanded:

```
private void clsNote_ShortcutsExpanded(int nNum)
{
    MessageBox.Show("Shortcuts Expanded: " + nNum);
}
```

You might display this information for the user or deploy it in an analysis of the efficiency of dictation.

Working with Note Audio File

After you retrieve the audio file for the **Note** using the **GetWaveFile()** method of the **Note** object (see [Retrieving Audio of the Note](#)), if you want to determine the position of the cursor in the wave file, you can retrieve the **CurrentWavePosition** property of the **Note** object:

```
int cursorLocation;
cursorLocation = clsNote.CurrentWavePosition;
```

When you want to determine the length of the wave file in milliseconds, you can retrieve the **SoundLength** property of the **Note** object:

```
int audioLength;
audioLength = clsNote.SoundLength;

MessageBox.Show("Length of Audio = " + audioLength);
```

You can also use **Microphone** object commands to work with the **Note** audio. For instance, you can use the same **Microphone** object commands with the **Note** object that you would use with a **Report** object's wave file, including **Play()**, **Rewind()**, **Forward()**, **Stop()**, **GoToStart()**, and **GoToEnd()**. For details, refer to [Advancing, Rewinding, Playing, and Stopping Audio on page 233](#).

Releasing the Microphone from Exclusive Use by the Note

When you have finished receiving text in the field, you can release the microphone from exclusive use by the **Note** by calling the **ReleaseMicrophone()** method of the **Note** object:

```
clsNote.ReleaseMicrophone();
```

Using the Note Object in JavaScript

In JavaScript, you can use the **Note** object to retrieve audio containing transient data, but you cannot use a **CustomEditController** to retrieve the results of the recognition. Instead, you must use a **PlayerEditorCtl** to retrieve the resulting text.

Switching Between Note and Report

You can have one or more fields or **PlayerEditors** receiving dictation of transient data in the same application with one or more **PlayerEditors** receiving dictation for reports. However, the microphone works with only one field or **PlayerEditor** at a time, so in this scenario, you need to control which field or editor is receiving dictation at a given time.

As you might remember from earlier chapters, any **Microphone** object that is not secured for the use of another object automatically defaults to working with the currently active **Report** object.

If you start out having the user dictate an identifier into a field where the application uses a **Note**, first you secure the microphone for exclusive use of the **Note**. Then, once you retrieve the text from the **Note** with **Note.GetText()** (and save it in a string), you can then switch the focus to the currently active report by releasing the microphone with **ReleaseMicrophone()**. When you release the microphone, if a report is active, the *SDK* gives control of the microphone back to the report. Otherwise, the microphone remains available for any report you might start or for any other field to secure for its exclusive use.

Switching Between Note and Report in JavaScript

You might want to use a **Note** to receive dictation of an identifier when that identifier needs to be spelled out letter by letter and perhaps includes numbers. To then switch the focus back to the **Report** and allow the user to go back to the **Note** and dictate the next identifier, your application needs to be able to go back and forth between the **Note** and **Report** seamlessly.

For instance, if sections of a report are each about a different department identified by an alphanumeric code, after you create the Note object and its events, then create a PlayerEditor for the Note, you can take several actions in JavaScript:

1. Have a button the user clicks to indicate he or she wants to dictate a department identifier. When the user clicks the button, secure the microphone for exclusive use of the **Note** and insert the cursor there by setting the focus to the **Note's PlayerEditor**:

```
objNote.SetMicrophone(objMicrophone);  
objNotePlayerEditor.SetEditorFocus();
```

2. Dictate the department code into the **Note** in **Letter** mode (like a **Spell** mode).
3. In the **RecognizeEnd** event handler, retrieve the department code from the **Note** using **Note.GetText()** and save it as a string in memory:

```
strDeptCode = objNote.GetText(psTextFormatTyePlain);
```

4. Release the **Microphone** from exclusive use by the **Note**:
5. If you have more than one **PlayerEditor** where you can receive dictation, set the cursor to focus in the **PlayerEditor** for the **Report**:

```
objRptPlayerEditor.SetEditorFocus();
```

6. Insert the identifier string into the **Report** and then dictate the data about that department:
7. Repeat the above steps for the next section in the report.

This way, your report remains in **Dictate** mode, receiving ordinary spoken sentences, and you can switch back and forth between dictating the department code into the field and dictating sentences into the report.

By going back and forth between the **Note** and the **PlayerEditor**, you can efficiently dictate this type of report.

Switching Between Note and Report in C#

You might want to use a **Note** to receive dictation of an identifier when that identifier needs to be spelled out letter by letter and perhaps includes numbers. To then switch the focus back to the **Report** and allow the user to go back to the **Note** and dictate the next identifier, your application needs to be able to go back and forth between the **Note** and **Report** seamlessly.

For instance, if sections of a report are each about a different department identified by an alphanumeric code, after you create the Windows component for the GUI (such as a TextBox), you can take several actions in C#:

1. Create a **Note** object and declare events of the **Note**.
2. Create a **CustomEditController**.

3. Associate the **CustomEditController** with the component and with the **Note**. This makes the GUI component a place the user can later dictate the department identifier.
4. Create a button that the user clicks to indicate he or she wants to dictate a department identifier. When the user clicks the button, set the **Dictation** preferences to **Letter** mode and secure the microphone for exclusive use of the **Note**:

```
private void btnDeptIdentifier_onClick()
{
    // Create Custom Edit Control
    this.clsCustEdit = new CustomEditController();

    // Associate CustomEditControl with TextBox
    clsCustEdit.SetHWnd(txtDeptCode.Handle.ToInt32());

    this.clsNote == null;

    // Create Note and Associate with CustomEditController
    this.clsNote = this.clsSDK.NewNote("");
    clsNote.SetCustomEditControl(clsCustEdit);

    // Secure LevelMeter for Use by the Note
    clsNote.SetLevelMeter((LevelMeterCtl)this.axLevelMeterCtl1.GetOcx());

    // Set Dictation Mode of Note to Letter Mode
    clsNote.Mode = SystemMode.psSystemModeLetter;

    // Secure Microphone for Use by the Note
    clsNote.SetMicrophone(null);
}
```

5. Dictate the department code into the **Note** in **Letter** mode (like a **Spell** mode).
6. In the **RecognizeEnd** event handler, retrieve the department code from the **Note** using **Note.GetText()** and save it as a global string in memory:

```
private void clsNote_RecognizeEnd()
{
    MessageBox.Show("Department Code Recognized");
    strDeptCode = clsNote.GetText(POWERSCRIBESDKLib.
        TextFormatType.psTextFormatTypePlain);
}
```

7. Release the **Microphone** from exclusive use by the **Note**:
8. If you have more than one **PlayerEditor** where you can receive dictation, set the cursor to focus in the **PlayerEditor** for the **Report**:
9. Insert the identifier string into the **Report** and then dictate the data about that department:
10. Repeat the above steps for the next section in the report.

C# Code Summary

```
// Sample of Retrieving Lookup Key from GUI Using Note Object
// This code is not complete. A similar working sample is available
// in the \Samples\CS directory on the product CD.

...
using POWERSCRIBESDKLib;
using PSADMINSDKLib;

namespace SdkNoteSample
{
    public class Form1 : System.Windows.Forms.Form
    {
        private POWERSCRIBESDKLib.PowerscribeSDK clsSDK;
        private POWERSCRIBESDKLib.CustomEditController clsCustEdit;
        private POWERSCRIBESDKLib.Note clsNote;
        ...

        // More buttons and other components
        ...

        private AxPOWERSCRIBESDKLib.AxPlayerEditorCtl axPlayerEditorCtl1;
        private AxPOWERSCRIBESDKLib.AxLevelMeterCtl axLevelMeterCtl1;
        private System.Windows.Forms.TextBox txtLookupKey;
        private System.Windows.Forms.Button btnSearchData;
        private System.Windows.Forms.Button btnAnalyzeAudio;
        private System.Windows.Forms.Button btnSaveNoteAudio;
        private System.Windows.Forms.Button btnSaveConcordance;

        public string strLookupKey;

        private System.ComponentModel.Container components = null;

        public Form1()
        {
            // Required for Windows Form Designer support
            //
            InitializeComponent();

            // TODO: Add any constructor code after InitializeComponent call
            //
            clsSDK = null;
            clsCustEdit = null;
            clsNote = null;
            ...
        }

        ...

        private void InitsSDK()
        {
            try
            {
```

```
//Initialize
...
// Create Custom Edit Control
this.clsCustEdit = new CustomEditController();
// Associate CustomEditControl with TextBox
clsCustEdit.SetHWnd(txtLookupKey.Handle.ToInt32());
PrepareToReceiveLookupKey();
TraceOK(sFuncName);
}
catch(Exception ex)
{
    TraceError(sFuncName,ex);
}
}

...
private void PrepareToReceiveLookupKey()
{
    if(this.clsNote == null)
    {
        // Create Note and Associate with CustomEditController
        this.clsNote = this.clsSDK.NewNote("");
        clsNote.SetCustomEditControl(clsCustEdit);

        // Secure LevelMeter for Use by the Note
        clsNote.SetLevelMeter((LevelMeterCtl)this.axLevelMeterCtl1.GetOcx());
    }

    // Set Dictation Mode of Note to Letter Mode
    clsNote.Mode = SystemMode.psSystemModeLetter;

    // Set Note to Use or Not Use Streaming Text
    clsNote.TextStreaming = false;

    // Secure Microphone for Use by the Note
    clsNote.SetMicrophone(null);
}

// Handle RecognizeEnd Event of Note object
private void clsNote_RecognizeEnd()
{
    MessageBox.Show("Lookup Key Recognized");
}

private void btnSearchData_Click(object sender, System.EventArgs e)
{
    SearchDatabase(strLookupKey);
}
```

```
private void SearchDatabase(string strLookupKey)
{
    strLookupKey = clsNote.GetText(POWERSCRIBESDKLib.TextFormatType
        .psTextFormatTypePlain);

    MessageBox.Show("Searching for " + strLookupKey);

    AudioAnalysis();
    // Search Database for Text Matching Key
    ...
}

private void btnAnalyzeAudio_Click(object sender, System.EventArgs e)
{
    AudioAnalysis();
}

private void btnSaveNoteAudio_Click(object sender, System.EventArgs e)
{
    clsNote.GetWaveFile("C:\LookupKeyAudio.wav");
}

private void btnSaveConcordance_Click(object sender, System.EventArgs e)
{
    clsNote.GetConcordance("C:\LookupKeyConcordance.ast");
}

private void AudioAnalysis()
{
    // Retrieve the length of the audio file and the cursor pos in that file
    MessageBox.Show("Audio for lookup key is " + clsNote.SoundLength);
    MessageBox.Show("Cursor position is " + clsNote.CurrentWavePosition);

    ...
}

// Handle ShortcutsExpanded Event of Note Object

private void clsNote_ShortcutsExpanded(int nNum)
{
    MessageBox.Show("Total Shortcuts Expanded: " + nNum.ToString());
}

private void Uninitialize()
{
    clsNote.CleanupNote;
    this.clsSDK.Uninitialize();
}
```

```
// Release the Microphone from Exclusive Use by Note Object  
  
private void CleanupNote()  
{  
    clsNote.ReleaseMicrophone();  
}  
  
}
```

JavaScript Code Summary for Dictating a Note



Note: When you use the **Note** object in JavaScript, you do not need the **CustomEditController** object. Instead, you use the **PlayerEditor** as the text box that receives dictation for the **Note**.

```
<html>  
    <object ID="pscribeSDK">  
    </object>  
<head>  
  
<title>SDK Note Sample</title>  
  
<style> <!-- body,td,a,p,.h{font-family:arial,sans-serif;}  
        .h{font-size: 20px;}  
        .h{color:;}  
        .q{text-decoration:none; color:#0000cc;}  
        .clsBoldBtn { width:25px; height:25px; background-color: #efe7ce;  
        border: thin solid #FFFE8;  
        background-image: url(..../images/btnBold.bmp); }  
        .clsItalicBtn { width:25px; height:25px; background: #efe7ce;  
        border: thin solid #FFFE8;  
        background-image: url(..../images/btnItalic.bmp); }  
        .clsUnderlineBtn { width:25px; height:25px; background: #efe7ce;  
        border: thin solid #FFFE8;  
        background-image: url(..../images/btnUnderline.bmp); }  
        .clsUListBtn { width:25px; height:25px; background: #efe7ce;  
        border: thin solid #FFFE8;  
        background-image: url(..../images/btnUList.bmp); }  
        .clsOListBtn { width:25px; height:25px; background: #efe7ce;  
        border: thin solid #FFFE8;  
        background-image: url(..../images/btnOList.bmp); }  
        //--></style>  
  
<script language="javascript" id="utils" src="..../scripts/utils.js">  
</script>  
</head>
```

```
<body bgcolor="#FFFF99" text="#000000" link="#0000cc" vlink="#551a8b"
alink="#ff0000" language="javascript" onload="Initialize()"
onunload="Cleanup()">

<center style="FONT-SIZE: medium; FONT-STYLE: oblique; BACKGROUND-COLOR:
#FFFF99">
    PowerScribe SDK Programmer Guide <br>
    Note Sample Application
</center>
<br>
<center style="BACKGROUND-COLOR: #FFFF99">
    <input value="LogOut" type="button" name="btnLogout" language=
    "javascript" onclick="DoLogOff()" style="FONT-STYLE: oblique">
</center>

<br>
<center style="FONT-SIZE: smaller; BACKGROUND-COLOR: #FFFF99"><font
style="FONT-SIZE: x-small; COLOR: black; FONT-STYLE: oblique" size="3">
    Using Buttons, Set the Dictation Mode to Letter or Number,<br>
    then Dictate a Lookup Key or Identifier and Press the Appropriate Button
</font>
</center>

<br>
<center style="FONT-SIZE: x-small; FONT-STYLE: oblique; BACKGROUND-COLOR:
#FFFF99">

    <table border="0" cellpadding="0" cellspacing="3" id="table2">
        <tr>
            <td>
                <input value="Set Letter Mode" type="button" name="btnSetLetter"
                    language="javascript" onclick="SetLetterMode() "
                    style="FONT-STYLE: oblique">
            </td>
            <td>
                <input value="Set Number Mode" type="button" name="btnSetNumber"
                    language="javascript" onclick="SetNumberMode() "
                    style="FONT-STYLE: oblique">
            </td>
        </tr>
    </table>

    <br>
    <table border="0" cellpadding="0" cellspacing="3" id="table3">
        <tr>
            <td style="FONT-STYLE: oblique; FONT-SIZE: x-small;">Lookup Key:</td>
    </tr>
</table>
```

```
</td>
</tr>
</table>

<center>

<div id="PlayerEditorObj">

    <object id="objPlayerEditor" style="WIDTH: 425px; HEIGHT: 20px;
        BACKGROUND-COLOR: FFFF99" height="32" width="432"
        data="data:application/x-oleobject;base64,
        UNFwpP/DhEWbC30xr0YhwRAHAAJAAAAAAAAAAAAAAA=">
        classid="clsid:A470D150-C3FF-4584-9B0B-7D31AF4621C1" viewastext>
    </object>
</div>

<script>
    CreateObjectCtl("PlayerEditorObj");
</script>

<table cellspacing="3" cellpadding="0">
    <tr>
        <td colspan="3">
            <div id="LevelMeterObj">
                <object id="objLevelMeter" height="20" width="150"
                    data="data:application/
                    x-oleobject;base64,DfwvSUoKrU68UpZq4VaPzRAHAACBDwAAEQIAAA==">
                    classid="clsid:492FFC0D-0A4A-4EAD-BC52-966AE1568FCD">
                </object>
            </div>
            <script>
                CreateObjectCtl("LevelMeterObj");
            </script>
        </td>
        <td colspan="3">
            <input language="javascript" style="FONT-STYLE: oblique" disabled
                onclick="GetPosition()" type="button" value="Wave Position"
                name="btnWavepos">
        </td>
        <td colspan="2">
            <input maxlength="256" size="10" name="Pos" disabled>
        </td>
    </tr>
</table>
<br>
</center>

<script event="TextSelChanged(nStart,nEnd,bsStyle)" for="objPlayerEditor">
{
    EditorTextSelChanged(nStart, nEnd, bsStyle);
}
```

```
}

</script>

<center>
  <table>
    <tr>
      <td>
        <input value=" Search Database " type="button" name="btnSearch"
               language="javascript" disabled onclick="SearchDatabase()"
               style="FONT-STYLE: oblique">
      </td>
    </tr>
  </table>

  <p style="FONT-STYLE: oblique; FONT-SIZE: x-small;">Save Audio, Text, and/
  or<br> Concordance of Key</p>
  <table border="0" cellpadding="0" cellspacing="3" id="Table4">
    <tr>
      <td>
        <input value="Save Audio of Key in File" type="button"
               name="btnSaveWave" language="javascript" disabled
               onclick="SaveWave()" style="FONT-STYLE: oblique">
      </td>
      <td>
        <input value="Save Text of Key in File" type="button"
               name="btnSaveText" language="javascript" disabled
               onclick="SaveText()" style="FONT-STYLE: oblique">
      </td>
      <td>
        <input value="Save Concordance of Key in File" type="button"
               name="btnSaveConcordance" language="javascript" disabled
               onclick="SaveConcordance()" style="FONT-STYLE: oblique">
      </td>
    </tr>
  </table>
  <br>
</center>

<script>
// Text Format Type
psTextFormatTypeUnknown = 0;
psTextFormatTypePlain = 1;
psTextFormatTypeXML = 2;
psTextFormatTypeRTF = 3;

// Definitions
var gSel = "Selection";
```

```
// global objects
var gScript = window.top.script;
var PowerscribeSDK = gScript.pscribeSDK;

var psSystemModeDictateOnly = 0;
var psSystemModeDictate = 1;
var psSystemModeCommand = 2;
var psSystemModeNumber = 3;
var psSystemModeLetter = 4;

var strLookupKey;
var NoteSink;

function InitEvents()
{
    NoteSink = new ActiveXObject("PowerscribeSDK.EventMapper");
    NoteSink.RecognizeEnd = HandleRecognizeEnd;
    NoteSink.ShortcutsExpanded = HandleShortcutsExpanded;
}

function Cleanup()
{
    try
    {
        if (NoteSink)
        {
            NoteSink.Unadvise();
        }

        objNote = null;
    }
    catch(error)
    {
        alert("Cleanup" + error.description);
    }
}

function Initialize()
{
    try
    {
        btnSetLetter.disabled = false;
        btnSetNumber.disabled = false;
        btnSearch.disabled = false;
        btnWavepos.disabled = false;
        btnSaveWave.disabled = true;
        btnSaveText.disabled = true;
        btnSaveConcordance.disabled = true;
    }
}
```

```
    objNote = null;
    PowerscribeSDK.LoadWords();
    PowerscribeSDK.LoadShortcuts();
    InitEvents();
}
catch(error)
{
    alert(error.description);
}
}

function DoLogOff()
{
try
{
    btnSetLetter.disabled = true;
    btnSetNumber.disabled = true;
    btnWavepos.disabled = true;
    btnSearch.disabled = true;
    btnSaveWave.disabled = true;
    btnSaveText.disabled = true;
    btnSaveConcordance.disabled = true;
    objPlayerEditor.EnablePlayer(false);

    objNote.ReleaseMicrophone();

    PowerscribeSDK.Logoff();
    Cleanup();
}
catch(error)
{
    alert(error.description);
}
window.open("loginscreen.htm", "_self", "height=800,width=1000,
    status=yes", true);
}

function SetLetterMode()
{
if (objNote == null)
{
    objNote = top.script.pscribeSDK.NewNote("");
}
objNote.SetLevelMeter(objLevelMeter);
objNote.SetPlayerEditor(objPlayerEditor);
objPlayerEditor.SetEditorFocus();
objNote.SetMicrophone(top.script.objMicrophone);
objNote.Mode = psSystemModeLetter;
NoteSink.Advise(objNote);
}
```

```
function SetNumberMode()
{
    if (objNote == null)
    {
        objNote = top.script.pscribeSDK.NewNote("");
    }
    objNote.SetLevelMeter(objLevelMeter);
    objNote.SetPlayerEditor(objPlayerEditor);
    objPlayerEditor.SetEditorFocus();
    objNote.SetMicrophone(top.script.objMicrophone);
    objNote.Mode = psSystemModeNumber;
    NoteSink.Advise(objNote);
}

function SearchDatabase()
{
    btnSaveWave.disabled = false;
    btnSaveText.disabled = false;
    btnSaveConcordance.disabled = false;
    strLookupKey = objNote.GetText(psTextFormatTypePlain);
    alert("Searching database for " + strLookupKey);
    // Search the database for a match
}

function SaveWave()
{
    objNote.GetWaveFile("C:\LookupKey.wav");
    alert("Lookup Key audio has been saved.");
}

function SaveText()
{
    // save strLookupKey to File
    alert("Lookup Key saved in file.");
}

function SaveConcordance()
{
    objNote.GetConcordanceFile("C:\LookupKey.ast");
    alert("Lookup Key concordance file has been saved.");
}

function GetPosition()
{
    if (objNote != null)
    {
        Pos.value = objNote.CurrentWavePosition;
    }
    objPlayerEditor.SetEditorFocus();
}
```

```

//Events
function HandleRecognizeEnd()
{
    alert("The Lookup Key has been recognized");
}

function HandleShortcutsExpanded(nNum)
{
    alert("Total of " +nNum+ " shortcuts have been expanded");
}

</script>
</body>
</html>

```

JavaScript Code Summary for Switching Between Note and Report

```

<html>
    <object ID="pscribeSDK">
    </object>
<head>
<title>Switching between Note and Report</title>
    ...
<script language="javascript" id="utils" src="../scripts/utils.js">
</script>
</head>

<body bgcolor="#FFFF99" text="#000000" link="#0000cc" vlink="#551a8b"
      alink="#ff0000" language="javascript" onload="Initialize()"
      onunload="Cleanup()">

    <center style="FONT-SIZE: medium; FONT-STYLE: oblique; BACKGROUND-COLOR:
#FFFF99">
        PowerScribe SDK Programmer Guide <br>      Sample Application
    </center>
    <br>
    <center style="BACKGROUND-COLOR: #FFFF99">
        <input value="LogOut" type="button" name="btnLogout"
               language="javascript" onclick="DoLogOff()"
               style="FONT-STYLE: oblique">
        <input value="Back" type="button" name="btnBack" language="javascript"
               onclick="DoBackToMenu()" style="FONT-STYLE: oblique">
    </center>

```

```
<br>
<center style="FONT-SIZE: smaller; BACKGROUND-COLOR: #FFFF99">
    <font style="FONT-SIZE: x-small; COLOR: black; FONT-STYLE:
        oblique" size="3">
        First, Indicate Report ID for a New or Existing Report.<br>
    </font>
<br>
<table border="0" cellpadding="0" cellspacing="3" id="table3">
    <tr>
        <td style="FONT-STYLE: oblique; FONT-SIZE: x-small;">
            <strong>Unique Report ID:</strong></td>
        <td><input maxlength="256" size="10" name="RepID" enabled></td>
    </tr>
</table>

<br>
<table border="0" cellpadding="0" cellspacing="3" id="table2">
    <tr>
        <td>
            <input value="New Report" type="button" name="btnReport"
                language="javascript" enabled onclick="NewReport()"
                style="FONT-STYLE: oblique">
        </td>
        <td>
            <input value="Edit Report" type="button"
                name="btnEditReport"
                language="javascript" enabled onclick="EditReport()"
                style="FONT-STYLE: oblique">
        </td>
    </tr>
</table>
</center>

<br>
<center style="FONT-SIZE: smaller; BACKGROUND-COLOR: #FFFF99">
    <font style="FONT-SIZE: x-small; COLOR: black; FONT-STYLE:
        oblique" size="3">
        Second, using Buttons Below, Dictate a Department Code
        to Add to the Report: <br>
    1. Set the Dictation Mode for the Department Code to Letter or Number.<br>
    2. Dictate a Department Code, Spelling its Letters and/or Numbers.<br>
    </font>
</center>
```

```

<br>
<center style="FONT-SIZE: x-small; FONT-STYLE: oblique;
    BACKGROUND-COLOR: #FFFF99">
<table border="0" cellpadding="0" cellspacing="3" id="table2">
<tr>
<td>
    <input value="Set Letter Mode" type="button"
        name="btnSetLetter" language="javascript"
        onclick="SetLetterMode()" style="FONT-STYLE: oblique"></td>
<td>
    <input value="Set Number Mode" type="button"
        name="btnSetNumber" language="javascript"
        onclick="SetNumberMode()" style="FONT-STYLE: oblique">
</td>
</tr>
</table>

<br>
<table border="0" cellpadding="0" cellspacing="3" id="table3">
<tr>
<td style="FONT-STYLE: oblique; FONT-SIZE: x-small;">
    <strong>Department Code:</strong></td>
</tr>
</table>
<center>
<div id="PlayerEditorObj">
    <object id="objNotePlayerEditor" style="WIDTH: 425px; HEIGHT: 20px;
        BACKGROUND-COLOR: #FFFF99" height="32" width="432"
        data="data:application/x-oleobject;base64,
        UNFwpP/DhEWbC30xr0YhwRAHAAJAAAAAAAAAAAAAAA=">
        classid="clsid:A470D150-C3FF-4584-9B0B-7D31AF4621C1" viewastext>
    </object>
</div>

<script>
    CreateObjectCtl("PlayerEditorObj");
</script>

<table cellspacing="3" cellpadding="0">
<tr>
<td colspan="3">
    <div id="LevelMeterObj">
        <object id="objLevelMeter" height="20" width="150"
            data="data:application/x-oleobject;base64,
            DfwvSUoKrU68UpZq4VaPzRAHAACBDwAAEQAIAA==">
            classid="clsid:492FC0D-0A4A-4EAD-BC52-966AE1568FCD">

```

```
        </object>
    </div>
    <script>
        CreateObjectCtl("LevelMeterObj");
    </script>
</td>
<td colspan="3">
    <input language="javascript" style="FONT-STYLE: oblique"
        onclick="GetPosition()" type="button" value="Wave Position"
        name="btnWavepos">
</td>
<td colspan="2">
    <input maxlength="256" size="10" name="Pos" >
</td>
</tr>
</table>
<br>
</center>

<script event="TextSelChanged(nStart,nEnd,bsStyle"
                    for="objNotePlayerEditor">
{
    EditorTextSelChanged(nStart, nEnd, bsStyle);
}
</script>

<center>
Third, to Insert the Department Code into the Report:<br>
1. Click the Insert Department Code button below.<br>
2. Start Dictating the Content of the Report for that department.<br>

<td>
    <input value="Insert Department Code" type="button" name="btnDeptCode"
        language="javascript" enabled onclick="InsertDepartmentCode() "
        style="FONT-STYLE: oblique">
</td>

<table border="0" cellpadding="0" cellspacing="3" id="table3">
<tr>
    <br>
    <td style="FONT-STYLE: oblique; FONT-SIZE: x-small;">
        <strong>Report In Process:</strong></td>
    </tr>
</table>

<div id="PlayerEditorObj">
    <object id="objRptPlayerEditor" style="WIDTH: 425px; HEIGHT: 200px;
```

```
BACKGROUND-COLOR: #FFFF99" height="232" width="432"
data="data:application/x-oleobject;base64,UNFwpP/
DhEWbC30xr0YhwRAHAAJAAAAAAAAAAAAAAAAB="
classid="clsid:A470D150-C3FF-4584-9B0B-7D31AF4621C1" viewastext>
</object>
</div>

<table border="0" cellpadding="0" cellspacing="3" id="table4">
<tr>
<td></td>
<td>
<input value="Save Report" type="button" name="btnComplete"
language="javascript" enabled onclick="CompleteReport()"
style="FONT-STYLE: oblique">
</td>
<td></td>
</tr>
</table>
</center>

<script>

// Text Format Type
psTextFormatTypeUnknown = 0;
psTextFormatTypePlain = 1;
psTextFormatTypeXML = 2;
psTextFormatTypeRTF = 3;

// Definitions
var gSel = "Selection";

// global objects
var gScript = window.top.script;
var PowerscribeSDK = gScript.pscribeSDK;
var objReport = gScript.Report;
var gReportArg = null; // Object passed into Report dialog

var psSystemModeDictateOnly = 0;
var psSystemModeDictate = 1;
var psSystemModeCommand = 2;
var psSystemModeNumber = 3;
var psSystemModeLetter = 4;

// String to retrieve dictated department code (note) in
var strDeptCode;
```

```
// EventMapper Object Names
var NoteSink;
var MicrophoneSink;
var ReportSink;

var CurID;
var CurRepID;
var psTextTypeCorrected = 2;

function InitEvents()
{
    NoteSink = new ActiveXObject("PowerscribeSDK.EventMapper");
    NoteSink.RecognizeEnd = HandleRecognizeEnd;
    NoteSink.ShortcutsExpanded = HandleShortcutsExpanded;

    MicrophoneSink = new ActiveXObject("PowerscribeSDK.EventMapper");
    MicrophoneSink.ButtonClick = HandleButtonClick;
    MicrophoneSink.Advise(top.script.objMicrophone);

    RptSink = new ActiveXObject("PowerscribeSDK.EventMapper");
    RptSink.RecognizeEnd = HandleRptRecognizeEnd;
    RptSink.SaveEndEx = HandleRptSaveEndEx;
}

function Cleanup()
{
    try
    {
        if (NoteSink)
        {
            NoteSink.Unadvise();
        }
        if (RptSink)
        {
            RptSink.Unadvise();
        }
        if (MicrophoneSink)
        {
            MicrophoneSink.Unadvise();
        }
    }
    catch(error)
    {
        alert("Cleanup" + error.description);
    }
}
```

```
function Initialize()
{
    try
    {
        InitEvents();
        objNote = null;
        PowerscribeSDK.LoadWords();
        PowerscribeSDK.LoadShortcuts();
        InitEvents();
    }
    catch(error)
    {
        alert(error.description);
    }
}

function DoLogOff()
{
    try
    {
        objNotePlayerEditor.EnablePlayer(false);
        objNote.ReleaseMicrophone();
        objRptPlayerEditor.EnablePlayer(false);
        PowerscribeSDK.Logoff();
        Cleanup();
    }
    catch(error)
    {
        alert(error.description);
    }

    //top.script.bLoginFinished = false;
    window.open("loginscreen.htm", "_self", "height=800,width=1000,
               status=yes", true);
}

function DoBackToMenu()
{
    try
    {
        objNotePlayerEditor.EnablePlayer(false);
        objNote.ReleaseMicrophone();
        objRptPlayerEditor.EnablePlayer(false);
        Cleanup();
    }
    catch(error)
    {
```

```
        alert(error.description);
    }

    //top.script.bLoginFinished = false;
    window.open("selectreporttype.htm", "_self",
    "height=800,width=1000,status=yes", true);
}

function SetLetterMode()
{
    objNote = top.script.pscribeSDK.NewNote("");
    objNote.SetLevelMeter(objLevelMeter);
    objNote.SetPlayerEditor(objNotePlayerEditor);
    objNotePlayerEditor.EnablePlayer(true);
    objNotePlayerEditor.SetEditorFocus();

    objNote.SetMicrophone(top.script.objMicrophone);
    objNote.Mode = psSystemModeLetter;
    NoteSink.Advise(objNote);
}

function SetNumberMode()
{
    objNote = top.script.pscribeSDK.NewNote("");
    objNote.SetLevelMeter(objLevelMeter);
    objNote.SetPlayerEditor(objNotePlayerEditor);
    objNotePlayerEditor.EnablePlayer(true);
    objNotePlayerEditor.SetEditorFocus();

    objNote.SetMicrophone(top.script.objMicrophone);
    objNote.Mode = psSystemModeNumber;
    NoteSink.Advise(objNote);
}

function NewReport()
{
    try
    {
        var RepTemp;
        var Pos;
        var psTextTypeCorrected = 2;
        try
        {
            // report ID cannot consist of only spaces
            var tmp = RepID.value;
            while (tmp.search(" ") != -1)
                tmp = tmp.replace(" ", "");
        }
    }
}
```

```
if (tmp.length > 0)
    RepTemp = PowerscribeSDK.NewReport(RepID.value);
else
{
    alert("Report ID cannot consist of only spaces.");
    return;
}
}
catch(error)
{
    alert("New Report: " + error.description);
    if (CurRepID)
        RepID.value = CurRepID;
    return;
}

if (SaveUnlockExistingReport() == false)
    return;
varPos = 1;
objReport = null;
objReport = RepTemp;
objReport.SetPlayerEditor(objRptPlayerEditor);
// objReport.SetLevelMeter(objLevelMeter);
objReport.ExclusiveLock(true);
// level = objLevelMeter.RedLevel;
varPos = 2;

try
{
    PowerscribeSDK.ActivateReport(objReport);
    RptSink.Advise(objReport);
    top.script.objMicrophone.ReceiveEvents(false);
    objRptPlayerEditor.EnablePlayer(true);
    objRptPlayerEditor.SetEditorFocus();
    if (objNote != null)
        objNote.ReleaseMicrophone();
}
catch (error)
{
    alert("Error switching to report: " + error.description);
}

try
{
    PowerscribeSDK.ActivateReport(objReport);
}
catch(error)
{
    alert(error.description);
```

```
        }
        varPos = 3;
        CurID = 1;
        CurRepID = RepID.value;
    }
catch(error)
{
    alert("Error:" + error.description);
}
}

function EditReport()
{
try
{
    var RepTemp;
    var varPos;
    try
    {
        // report ID cannot consist of only spaces
        var tmp = RepID.value;

        while (tmp.search(" ") != -1)
            tmp = tmp.replace(" ", "");

        if (tmp.length > 0)
            RepTemp = PowerscribeSDK.GetReport(RepID.value);
        else
        {
            alert("Report ID cannot consist of only spaces.");
            return;
        }
    }
    catch(error)
    {
        alert(error.description);
        if (CurRepID)
            RepID.value = CurRepID;
        return;
    }

    if (SaveUnlockExistingReport() == false)
        return;

    varPos = 1;
    objReport = null;
    objReport = RepTemp;
    objReport.SetPlayerEditor(objRptPlayerEditor);
```

```
// objReport.SetLevelMeter(objLevelMeter);
objReport.ExclusiveLock(true);
varPos = 2;

try
{
if (objNote != null)
    objNote.ReleaseMicrophone();
}
catch (error)
{
    alert("Error switching to report: " + error.description);
}

try
{
    PowerscribeSDK.ActivateReport(objReport);
}
catch(error)
{
    alert(error.description);
}
top.script.objMicrophone.ReceiveEvents(false);
varPos = 3;
CurID = 1;
CurRepID = RepID.value;
}
catch(error)
{
    alert("ErrorNum:" + error.number + " Error:" + error.description);
}
}

function CompleteReport()
{
try
{
    top.script.objMicrophone.ReceiveEvents(true);
    if (SaveUnlockExistingReport() == false)
        return;
    objReport.MarkForAdaptation();
    objReport.ClearEditor();
    CurID = null;
    objReport.ExclusiveLock(false);
}
catch(error)
{
    alert("CompleteReport:" + error.description);
```

```
        }

    }

function SaveUnlockExistingReport()
{
    try
    {
        if (CurID)
        {
            if (CurID == 1)
            {
                objReport.Save(false);
            }
        }
        return true;
    }
    catch(error)
    {
        alert("SaveUnlockExistingReport" + error.description);
        return false;
    }
}

function GetPosition()
{
    if (objNote != null)
    {
        Pos.value = objNote.CurrentWavePosition;
        objNotePlayerEditor.SetEditorFocus();
    }
    else
    {
        Pos.value = objReport.CurrentWavePosition;
        objRptPlayerEditor.SetEditorFocus();
    }
}

function InsertDepartmentCode()
{
    var psEditorCommandEndDoc = 11;
    strDeptCode = objNote.GetText(psTextFormatTypePlain);
    alert("Retrieved Department Code for Report " + strDeptCode);

    try
    {
        if (objNote != null)
            objNote.ReleaseMicrophone();
    }
    catch (error)
```

```
{  
    alert("Error switching to report: " + error.description);  
}  
objRptPlayerEditor.SetEditorFocus();  
objRptPlayerEditor.SendCommand(psEditorCommandEndDoc);  
objReport.InsertText(strDeptCode);  
}  
  
//Events  
function HandleRecognizeEnd()  
{  
    alert("The Section Heading has been recognized");  
}  
  
function HandleShortcutsExpanded(nNum)  
{  
    alert("Total of " +nNum+ " shortcuts have been expanded");  
}  
  
function HandleRptRecognizeEnd()  
{  
    alert("The Report dictation has been recognized");  
}  
  
function HandleRptSaveEndEx(TextType, TextFormatType, strText)  
{  
    alert("Report successfully saved");  
}  
  
function HandleButtonClick(ButtonID, x, y)  
{  
    if (CurID)  
    {  
        if (ButtonID == gScript.psMicButtonBottomLeft)  
        {  
            //Complete the report  
            if (CurID == 1)  
            {  
                if (SaveUnlockExistingReport() == false)  
                    return;  
                objReport.MarkForAdaptation();  
                objReport.ClearEditor();  
                CurID = null;  
                objRptPlayerEditor.EnablePlayer(false);  
            }  
        }  
    }  
}
```

```
</script>
</body>
</html>
```

Chapter 9

Configuring and Customizing the Microphone

Objectives

Before you proceed with this chapter, if you plan to use the *PowerMic* or *PowerMic II* microphone in your application, be sure it is set up on each workstation where dictation will take place. Refer to [Setting Up Foot Pedals and Microphones on page 23](#).

This chapter focuses on how to have your application mimic the microphone and work with its customizable buttons. (For information on complete takeover of the microphone, see [Chapter 17, Taking Over Control of the Microphone on page 427](#).)

- [Locating Microphone Buttons](#)
- [Creating a Microphone Object](#)
- [Embedding Microphone Tuning Wizard](#)
- [Requesting That User Set Up Foot Pedals and Microphones](#)
- [Configuring Audio/Orientation Settings, Default Button Actions, Mic Light](#)
- [Setting Transcribe and Navigate Preferences](#)
- [Mimicking Microphone with GUI Elements](#)
- [Customizing Programmable Microphone Buttons](#)

Locating Microphone Buttons

Both the *PowerMic* and the *PowerMic II* microphones have buttons with specific report dictating and template navigating functions (refer to [Appendix E](#) for more detail). As soon as you plug it in, you can start working with the microphone in your application. By default, the microphone is present and most of its buttons have preprogrammed capabilities.

In addition, each microphone has some programmable (customizable) buttons.

Although it is always optional, there are several ways that you can work with a **Microphone** object in your application:

- Embed a microphone tuning wizard in your application.
- Create fields and buttons in your application to configure microphone preference settings.
- Program the customizable buttons on either the *PowerMic* or *PowerMic II* microphone.
- Create buttons in your application that imitate and replace the microphone, allowing the user to work with a headset or other alternative equipment.
- Have your application disable the standard capabilities of all *PowerMic* or *PowerMic II* microphone buttons and take over the entire functioning of the microphone.

Let's start by creating the microphone object, then embedding the microphone tuning wizard.

Creating a Microphone Object

You create a **Microphone** object using the **PowerscribeSDK** object's **Microphone** property:

```
objMicrophone = pscribeSDK.Microphone;
```

Once you have a **Microphone** object, you can use its methods, properties, and events.

Embedding Microphone Tuning Wizard

The microphone tuning wizard provided with the *SDK* presents screens to tune the volume of the microphone for the individual end user.

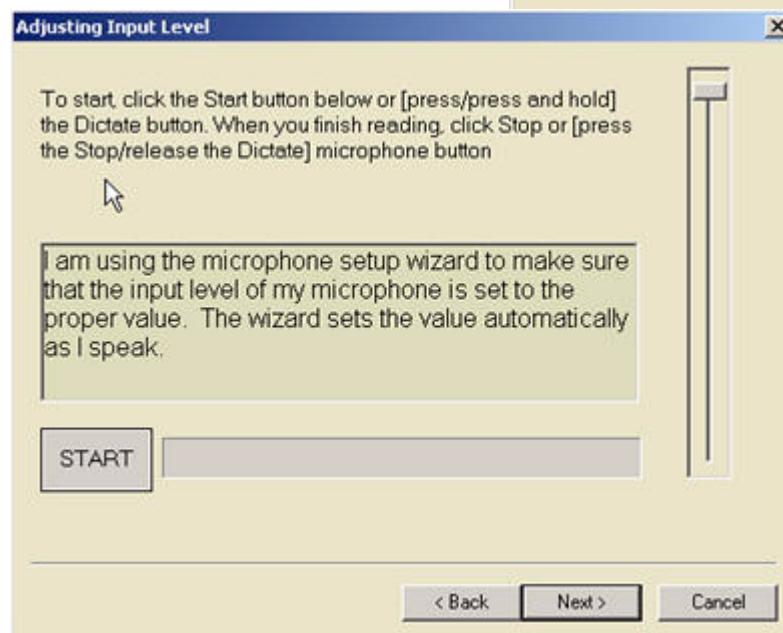
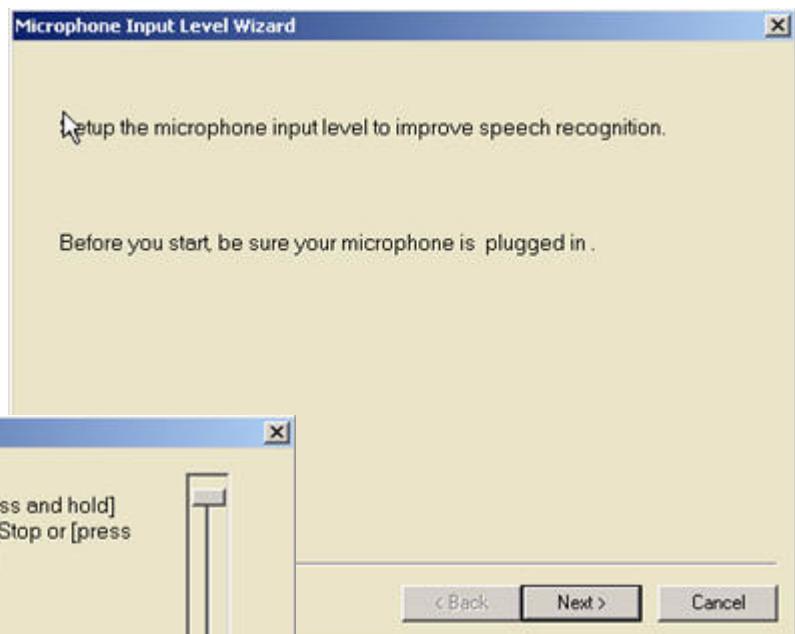
The tuning wizard is automatically embedded in your application by default as long as the **EnableGainWizard** property of the **PowerscribeSDK** object is set to **True**; however, if you'd like to have a button that invokes the wizard on demand, you can have that button call the **Tuning()** method of the **Microphone** object:

```
objMicrophone.Tuning();
```

The illustrations that follow show the successive screens that the tuning wizard presents.

When the user starts the microphone tuning wizard, the first wizard dialog box directs the user to ensure the microphone is plugged in.

Caution: When you invoke the *Tuning()* method, be sure the microphone is plugged in beforehand; if you plug it in afterwards, buttons on the microphone will not function.

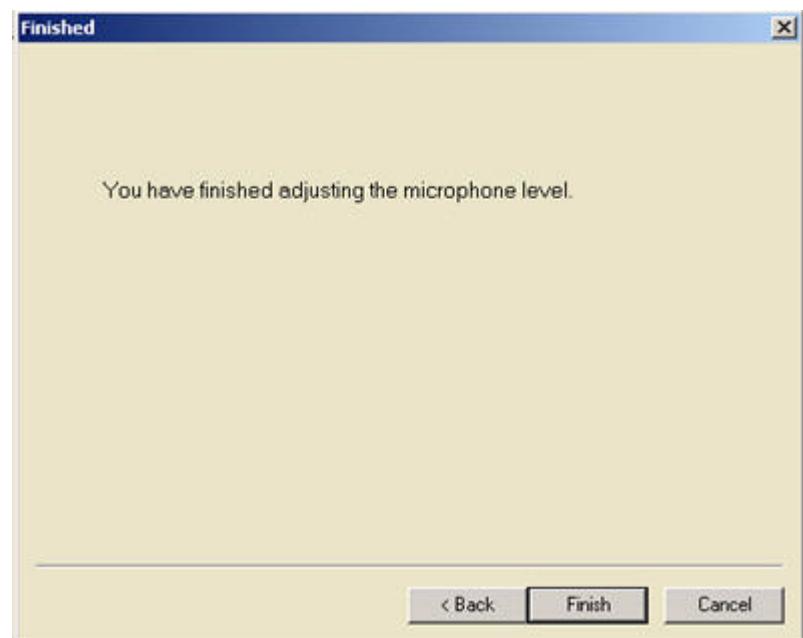


After the user clicks **Next**, the last wizard dialog box appears and indicates the adjustment of the microphone is complete.

The tuning of the microphone remains in force until the user logs out or shuts down that session of the application, unless your application stores the information and restores it on initialization.

Caution: If you have set the *EnableGainWizard* property of the *PowerscribeSDK* object to *False*, the tuning wizard does not appear at startup.

The next screen presents instructions, a **Start** button, a recording progress bar, and a vertical slider that shows the volume level. After clicking the **Start** button, while the user reads the text, the wizard tunes the microphone.



Requesting That User Set Up Foot Pedals and Microphones

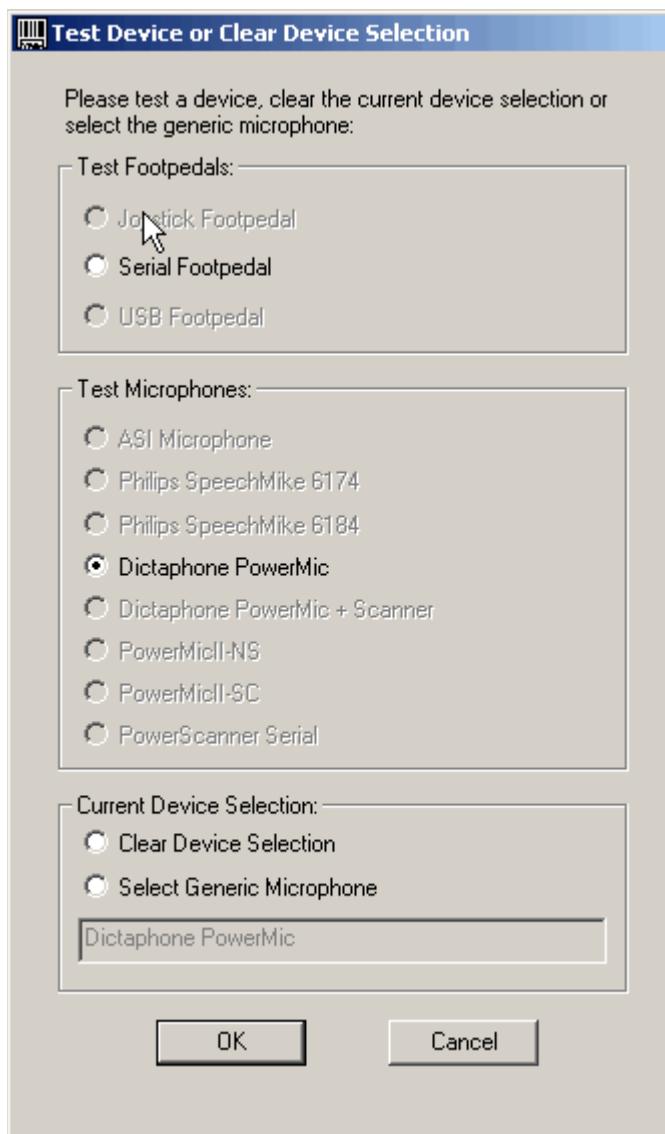
When you initially installed the *SDK*, you installed the **Micwiz.exe** file in the **C:\Windows\System32** directory on all client machines. Your application can check each workstation's type of microphone or foot pedal, and, if it finds any device other than a *PowerMic* or *PowerMic II*, run this wizard.

To check for the type of foot pedal or microphone equipment installed on the workstation, your application can call the read-only **Type** property of the **Microphone** object (see table).

Dictation Receiving Device Types

AudioDeviceType	Value
psAudioDeviceUnknown	0
psAudioDevicePhillipsSpeechMic	1
psAudioDeviceASIMic	2
psAudioDevicePowerSerialMic	3
psAudioDevicePowerMicI	4
psAudioDeviceBoomerangMic	5
psAudioDeviceGenericMic	6
psAudioDevicePowerMicII	7
psAudioDeviceBluetooth	8
psAudioDeviceArrayMic	9
psAudioDevicePocketPC	10
psAudioDevicePalmPilot	11
psAudioDeviceDROlympus	12
psAudioDeviceOlympusDRec	13
psAudioDeviceOlympusDS330	14
psAudioDeviceOlympusDS2	15
psAudioDevicePC	16
psAudioDevicePCDynamic	17
psAudioDeviceTelephonyServer	18
psAudioDeviceTelephonyDictaphone	19
psAudioDeviceTelephonyDVIPS	20
psAudioDeviceDVI	21
psAudioDeviceLanier	22

Window Popped Up by the
Microphone.Setup() Method



The *PowerMic* or *PowerMic II* USB microphones are the only ones that are automatically configured. To have the application configure another type of microphone, call the **Setup()** method of the **Microphone** object:

```
if ((objMicrophone.Type != psAudioDevicePowerMicI) ||
    (objMicrophone.Type != psAudioDevicePowerMicII))
{
    objMicrophone.Setup();
}
```

The wizard then pops up the window shown here, where the user can set up the foot pedal or non-*PowerMic* microphone used on the workstation.

 **Note:** If more than one device is connected, the USB PowerMic becomes the default device.

Bluetooth device functionality was tested with the *Plantronics Audio 910*. Array microphone device functionality was tested with *SoundMAX Superbeam*. For a list of manufacturers who support these models, refer to the Nuance Communications web site.

Configuring Audio/Orientation Settings, Default Button Actions, Mic Light

There are several properties of the microphone that are automatically set to default settings, but that users frequently prefer to modify. Those settings include:

- Setting the level for:
 - Audio playback volume on a scale of 0 to 10
 - Audio playback speed on a scale of 0 to 10
 - Audio rewinding speed on a scale of 0 to 10
- Turning on and off:
 - Monkey chatter when fast forwarding or rewinding the audio
 - Voice activated recording (stops when you stop talking)
 - Removing audio distortion caused by voice modulation during dictation
 - Playing an audio tone when ready to dictate
- Turn on/off and set the color of the light on the microphone
- Choose an option for:
 - User orientation—Left- or right-handed settings for microphone buttons
 - **DICTATE** and **PLAYBACK** button actions: Either **On/Off**, **Deadman**, or **Always On**



Caution: After you change them, microphone property settings do not take effect until you call the **Update()** method of the **Microphone** object. They then remain in force until you change them and call the **Update()** method again. If you do not call **Update()**, the changes are not

maintained when the user logs out or shuts down that session of the application, unless your application saves those settings and restores them on initialization.

Configuring Microphone Settings

You can set microphone settings using a series of **Microphone** object properties.

Setting Audio Volume, Rewind, and Playback Speeds

The audio volume can be set on a scale of 0 to 10, with 10 being the loudest. You can adjust the volume by setting the **PlaybackVolume** property:

```
objMicrophone.PlaybackVolume = 5;
```

You can set the speed that the audio rewinds at by setting the **WindingSpeed** property on a scale of 0 to 10, with 0 the slowest and 10 the fastest available:

```
objMicrophone.WindingSpeed = 5;
```

You can set the speed that the audio plays at by setting the **PlaybackSpeed** property on a scale of 0 to 10, with 0 the slowest and 10 the fastest available:

```
objMicrophone.PlaybackSpeed = 5;
```

To have changes to microphone settings take effect, be sure to call the **Update()** method of the **Microphone** object. Refer to [Ensuring Microphone Settings Take Effect](#) below.

Turning On/Off Monkey Chatter

You can choose to turn on or off monkey chatter, the vocalization that occurs when you rewind or fast forward the audio, by setting the **MonkeyChatter** property to **True/False**:

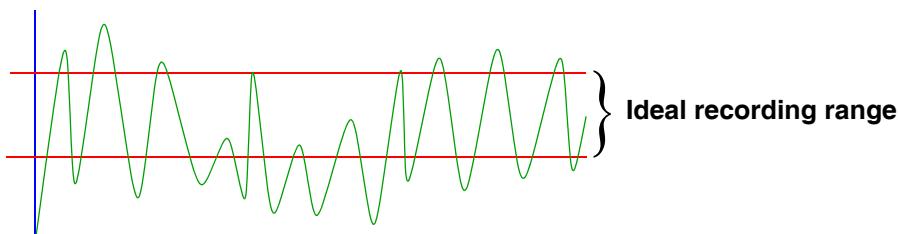
```
objMicrophone.MonkeyChatter = True;
```

After changing any microphone settings, for them to take effect, call the **Update()** method of the **Microphone** object ([Ensuring Microphone Settings Take Effect on page 227](#)).

Reducing Distortion of Audio

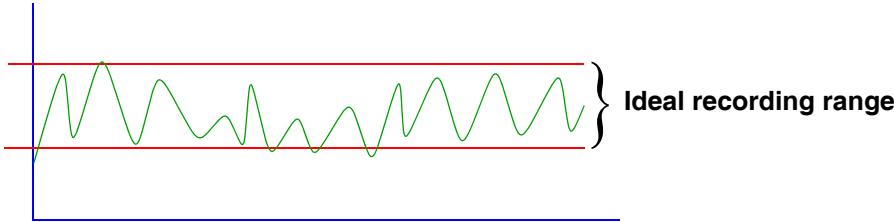
During dictation the volume of the user's voice can modulate up and down, moving outside the ideal range of the microphone, which can lead to distortion of the audio during playback (see illustration below):

Voice Modulation above/below Ideal Recording Range



You can have your *SDK* application reduce that distortion by the highs and lows of the voice modulation in the played back audio, called *normalizing the audio playback*, as shown in the next illustration:

Audio Normalized During Playback for Clarity



The property you set to normalize the audio's *volume* during playback is called **NormalizePlaybackVolume**, a Boolean property that you set to **True** to normalize the audio or **False** to not normalize it (the default is **False**):

```
objMicrophone.NormalizePlaybackVolume = True;
```

One of the risks in correcting audio distortion with most products is their tendency to *clip* (cut off) portions of the recording. This problem does not occur with the *PowerScribe SDK*. The **NormalizePlaybackVolume** property of the **Microphone** object corrects distortion without clipping the audio.

To have changes to microphone settings take effect, be sure to call the **Update()** method of the **Microphone** object. Refer to [Ensuring Microphone Settings Take Effect](#) below.

Playing Audio Tone Signal

You can also choose to have the microphone play a tone when it is ready to receive dictation. To turn this feature on or off, you set the **PlayAudioTone** property to **True** or **False**:

```
objMicrophone.PlayAudioTone = False;
```

To have changes to microphone settings take effect, be sure to call the **Update()** method of the **Microphone** object. Refer to [Ensuring Microphone Settings Take Effect](#) below.

Setting Microphone for Left- or Right-Handed Usage

If you do not set it otherwise, the microphone's orientation is set to right-handed use. To set the microphone for left- or right- handed usage, you set the **Orientation** property of the **Microphone** object to either **psMicOrientationLeft** (1) or **psMicOrientationRight** (0):

```
var psMicOrientationRight = 0;
var psMicOrientationLeft = 1;
objMicrophone.Orientation = psMicOrientationLeft;
```



Note: Remember that in JavaScript, you must define the constant to use it; otherwise you must use the numeral to set the property.

To have changes to microphone settings take effect, be sure to call the **Update()** method of the **Microphone** object. Refer to [Ensuring Microphone Settings Take Effect](#) below.

Turning On and Setting Color of the Microphone Light

Your application can manipulate the color of the light on the microphone that displays during any action or state. You can also manipulate whether it flashes or remains steadily on. For instance, you might turn the light on red and flashing during dictation, turn it on green and flashing during audio playback, and set it to a green that is steadily on when the microphone is in an idle state.

For instance, to have the microphone light turn red and flash during dictation, you set the **LightColor** property of the **Microphone** object to **psMicLightColorRedFlashing**:

```
objMicrophone.LightColor = psMicLightColorRedFlashing;
```

Or you can set it to one of the other settings in the table that follows.

psMicLightColorState	Value	Explanation
psMicLightColorOff	0	Turn off the light.
psMicLightColorRedOn	1	Set the color to red.
psMicLightColorRedFlashing	2	Set the color to flashing red.
psMicLightColorGreenOn	3	Set the color to green.
psMicLightColorGreenFlashing	4	Set the color to flashing green.

To have changes to microphone settings take effect, be sure to call the **Update()** method of the **Microphone** object. Refer to [Ensuring Microphone Settings Take Effect](#) below.

Choosing Dictate and Playback Button Actions

You can determine how the **DICTATE** and **PLAYBACK** buttons on the microphone should behave in your application. The ways that these buttons can operate are possible values of the *MicButtonRecordType* argument shown in the table below.

MicButtonRecordType	Value	Explanation
psMicButtonTypeOnOff	0	Pressing the DICTATE button toggles on or off recording; pressing the PLAY/STOP button toggles on or off playback. The user does not have to hold down the button.
psMicButtonTypeDeadMan	1	DICTATE button must be held down during recording, PLAY/STOP button must be held down during playing; releasing the button turns the function off.
psMicButtonTypeAlwaysOn	2	Microphone starts recording action when the DICTATE button is pressed once (not held down). Recording continues and remains on until the PLAY/STOP is pressed once (not held down) to stop the action.

Set the **MicButtonRecord** property of the **Microphone** object to one of the *MicButtonRecordTypes* in the table. For instance, to have the microphone record while the user holds down the **DICTATE** button, you would set the property to **psMicButtonTypeDeadMan**:

```
objMicrophone.MicButtonRecord = psMicButtonTypeDeadMan;
```

Another dictation setting that applies regardless of the **DICTATE** or **PLAYBACK** button action you choose is *voice activated recording*, which stops recording when the user stops speaking and resumes recording when the user starts speaking again. This feature is especially useful if periods of silence between dictated statements are common. You can turn this feature on by setting the **VoiceActivatedRecording** property to **True**:

```
objMicrophone.VoiceActivatedRecording = True;
```

To have changes to microphone settings take effect, be sure to call the **Update()** method of the **Microphone** object. Refer to [Ensuring Microphone Settings Take Effect](#) below.

Ensuring Microphone Settings Take Effect

After you change them, the microphone property settings do not take effect until you call the **Update()** method of the **Microphone** object:

```
objMicrophone.Update();
```

This method updates all **Microphone** property settings in the database, so you need only call it once for all settings. Updating all the settings at once like this prevents the drag on application performance that multiple updates to the database would otherwise create.

Setting Transcribe and Navigate Preferences

You can have the user choose options for transcribing and navigating with the microphone and set the application to behave accordingly. Among the choices you can offer the user are:

- Transcribe Preferences

Select a default response that should occur when the user presses the **TRANSCRIBE** button on the microphone:

- Transcribe text and select all the text as long as it does not exceed a set length
- Transcribe text and place cursor at the begining of the transcribed text
- Transcribe text and place cursor at the end of the transcribed text

- Navigation Preferences

A report can contain portions of paragraph text that are enclosed in square brackets. This text is treated like the name of a data field. This feature applies to both plain and structured reports.

The adjacent illustration shows a structured report that contains square bracketed fields, with one bracketed field selected.

For navigation preferences, you can choose a default response that should occur when the user presses the **TAB** or **NAVIGATE FORWARD** button on the microphone:

- Navigate to the next field in square brackets (plain or structured reports) or next section (structured reports only)
- Navigate to successive fields in square brackets (plain or structured reports) and skip all sections (structured reports only)
- Navigate to successive report sections and skip all fields in square brackets (structured reports only)

When the cursor finds square brackets, it selects the brackets and the field name within them; when it moves to a section, the cursor positions itself, but does not select text.

If navigating to the next section, place the cursor at the beginning or end of the section

- Auto advance to first shortcut field in a structured report



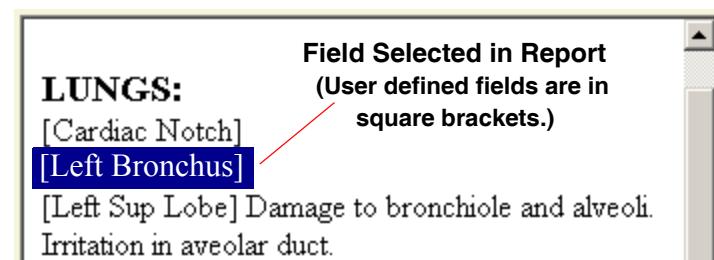
Caution: After you change them, microphone property settings do not take effect until you call the **Update()** method of the **Microphone** object. They then remain in force until you change them and call the **Update()** method again or until the user logs out or shuts down, unless your application saves those settings and restores them on initialization.

Setting Transcribe Preferences

You can set how the application should act when the user transcribes dictated text by pressing the **TRANSCRIBE** button on the microphone. You determine the action taken by setting two properties of the **Microphone** object, **TranscribePreference** and **TranscribeSelectCount**.

If you want the application to select the text after it is transcribed, you set the **TranscribeSelectCount** property to the maximum number of characters you want selected, starting at the beginning of the text just transcribed. If you want all the text selected, be sure the number of characters to select exceeds the number of characters in the transcribed text:

```
var psTranscribePreferenceSelect = 0;
objMicrophone.TranscribeSelectCount = 200;
objMicrophone.TranscribePreference = psTranscribePreferenceSelect;
```



The **TranscribeSelectCount** property does not work alone; for it to work, you also set the **TranscribePreference** property to **psTranscribePreferenceSelect**.

If you do not want to select the text after it is transcribed, you do not have to set the **TranscribeSelectCount** property; just set the **TranscribePreference** property to a value that places the cursor at either the beginning or end of the text just transcribed, **psTranscribePreferenceCursorBegin (2)** or **psTranscribePreferenceCursorEnd (1)**. See the next table for a summary of the **TranscribePreference** property values.

To put the cursor at the end of the transcribed text without selecting it, set the properties:

```
var psTranscribePreferenceCursorBegin = 2;
objMicrophone.TranscribePreferenceCount = 0;
objMicrophone.TranscribePreference = psTranscribePreferenceCursorBegin;
```

To have changes to microphone settings take effect, be sure to call the **Update()** method of the **Microphone** object. Refer to [Ensuring Microphone Settings Take Effect on page 227](#).

TranscribePreference	Value	Explanation	TranscribeSelectCount Dependency
psTranscribePreferenceSelect	0	Selects all transcribed text after a transcribe action. Default.	TranscribeSelectCount set to 0.
psTranscribePreferenceCursorEnd	1	Places the cursor at the end of the transcribed text after a transcribe action.	Or selects the number of characters in TranscribeSelectCount as long as the characters in transcribed text do not exceed value of TranscribeSelectCount property.
psTranscribePreferenceCursorBegin	2	Places the cursor at the beginning of the transcribed text after a transcribe action.	

SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

Setting Navigate Preferences

You can set how the application should navigate through the report when the user presses the **TAB** or **NAVIGATE FORWARD** button on the microphone by setting two properties of the **Microphone** object, **NavigatePreference** and **NavigateSectionPreference**.

You set the **NavigatePreference** property to one of three values, shown in the next table. For instance, to navigate to either the next square bracketed field or the next section, whichever the application finds first, you can set the property to **psNavigateTypeAll**:

```
var psNavigateTypeAll = 0;
objMicrophone.NavigatePreference = psNavigateTypeAll;
```

NavigatePreference	Value	Explanation
psNavigateTypeAll	0	Navigates through both sections and square bracketed field names in structured reports or through only field names in plain reports. When it moves to a field name, selects both the field name and the square brackets enclosing it. Default.
psNavigateTypeNextField	1	Navigates through square bracketed fields and, in structured reports, skips report sections. Selects the next field in square brackets.
psNavigateTypeNextSection	2	In structured reports, navigates the cursor to the next section and ignores all square bracketed fields. In plain reports, where there are no sections, this preference behaves the same as psNavigateTypeNextField .

You set the **NavigateSectionPreference** property to where the application should place the cursor when it navigates to the next section, either the beginning or the end of the section. This property applies only to structured reports.

NavigateSectionPreference	Value	Explanation
psNavigateSectionPreferenceCursorEnd	1	Move the cursor to the end of the section after the navigation.
psNavigateSectionPreferenceCursorBegin	2	Move the cursor to the beginning of the section after the navigation.
<i>SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.</i>		

The application ignores this property setting if it is not navigating sections (when the **NavigatePreference** property is **psNavigateTypeNextField**).

After you have changed any microphone settings, be sure to call the **Update()** method of the **Microphone** object for them to take effect. Refer to [Ensuring Microphone Settings Take Effect on page 227](#).

Beeping on Field or Section Navigation

To have the microphone emit a beep each time the user navigates to a new field or section in the report, you can set its **BeepOnNavigate** property to one of the values shown in the next table.

MicBeepOption	Value	Explanation
psMicBeepOptionOff	0	Turn off beeping on the microphone.
psMicBeepOptionAll	1	Turn on beeping of the microphone when navigating to either a new field or a new section in the report. In plain reports, this option has the same effect as psMicBeepOptionField .
psMicBeepOptionField	2	Turn on beeping of the microphone when navigating to a new field.
psMicBeepOptionSection	3	Turn on beeping of the microphone when navigating to a new section in the report.
<i>SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.</i>		

For instance, to have the microphone beep only when the user navigates to a new section, you would set the property to **psMicBeepOptionSection**:

```
objMicrophone.BeepOnNavigate = psMicBeepOptionSection;
```

After you have changed any microphone settings, be sure to call the **Update()** method of the **Microphone** object for them to take effect. Refer to [Ensuring Microphone Settings Take Effect on page 227](#).

Automatically Advancing to Next Shortcut Field

Another property of the **Microphone** object automatically moves the cursor to the next shortcut field. That property is **AutoAdvanceToNextShortcutField**, discussed in more detail in [Chapter 12, Working with Shortcuts, Categories, and Words on page 279](#).

To have changed microphone settings take effect, be sure to call the **Update()** method of the **Microphone** object. Refer to [Ensuring Microphone Settings Take Effect on page 227](#).

Mimicking Microphone with GUI Elements

Your application can use GUI elements that you create to mimic the microphone.

Before you begin, you should be sure to have instantiated a **PowerscribeSDK** object and a **PlayerEditorCtl** or **WordPlayerEditorCtl**. Throughout this chapter, the **PlayerEditor** mentioned refers to both types, unless otherwise specified.

Recording and Transcribing Audio

Before you can record or transcribe any audio, you first create a report, then associate it with the **PlayerEditor** where the report text should be transcribed.



Caution: If you record audio without having created a report to receive it, subsequently creating a report wipes that recorded audio out of the buffer, and you can never retrieve that audio. Similarly, if you exit the application without transcribing audio that is in the buffer, you lose the audio and cannot retrieve it.

```
try
{
    objReport = PowerscribeSDK.NewReport("ORD-120406");
    objReport.SetPlayerEditor(objRptEditor);
}
catch(error)
{
    alert("Create Report:" + error.description);
}
```

To begin recording audio in response to a user clicking a **DICTATE** button in your GUI, you call the **Record()** method of the **Microphone** object:

```
objMicrophone.Record();
```

Recording continues until your application calls the **Stop()** method. Recognition occurs in the background alongside the dictation process.

You might have another button in your GUI that the user clicks to end dictation and transcribe audio. When the user clicks that button, your application can call the **Stop()** method in response to that button:

```
objMicrophone.Stop();
```

To immediately transcribe the recognized audio into a **PlayerEditor**, call the **Transcribe()** method of the **Microphone** object:

```
objMicrophone.Transcribe();
```

You can also transcribe the recognized audio by calling the **Left()** method, which simulates pressing the upper left button (A) on the *PowerMic* or the  button on the *PowerMic II* microphone:

```
objMicrophone.Left();
```

If there is audio waiting in the buffer to be transcribed, the **Left()** method transcribes it; otherwise, the method initiates the action the **Left** button on the microphone would take. That action differs from *PowerMic* to *PowerMic II*. For details on the buttons of these two microphones, refer to [Appendix E](#).

Once the audio is transcribed and appears in the **PlayerEditor**, the cursor is positioned based on the transcribe preference settings.

Advancing, Rewinding, Playing, and Stopping Audio

To rewind the audio or advance to a particular position in it (and see the cursor follow along with the audio), you use these methods in your button click handlers:

- **Rewind()**
- **Play()**
- **Stop()**
- **Forward()**
- **GoToEnd()**
- **GoToStart()**

For instance, to rewind the audio in response to the user clicking a **REWIND** button in your GUI, you can call the **Rewind()** method:

```
objMicrophone.Rewind();
```

This method rewinds the entire audio to its beginning, highlighting the corresponding text in the **PlayerEditor**.

To play back the audio, you call the **Play()** method:

```
objMicrophone.Play();
```

This method plays the audio starting where the cursor is located in the **PlayerEditor** and highlights the corresponding text in the **PlayerEditor**.

If the user wants to interrupt the audio while it is playing or rewinding by pressing a **STOP** button in your GUI, you call the **Stop()** method:

```
objMicrophone.Stop();
```

To move the audio from any location to the end of the text so that further dictation can ensue, you call the **GoToEnd()** method, then call the **Record()** method again:

```
objMicrophone.GoToEnd();
objMicrophone.Record();
```

You can also return to the start of the audio by calling the **GoToStart()** method:

```
objMicrophone.GoToStart();
```

You can fast forward the audio by calling the **Forward()** method:

```
objMicrophone.Forward();
```

Handling Microphone Events

You are not required to handle microphone events in your application; however, handling them gives you the option of taking appropriate action in response to them.

Before you can handle any **Microphone** events, you should map them, then be sure receipt of them from the microphone is enabled (see [Mapping Microphone Events](#) below and [Turning On Events from Microphone on page 235](#)).

You can then handle several types of events when the microphone sends notifications:

- Any button pressed down on the microphone—**ButtonDown** event
- Any button released on the microphone—**ButtonUp** event
- A change of the location in the audio during recording, rewinding, or playing—**WavePosition** event
- Disconnecting of the microphone from the computer—**MicConnectionChange** event
- Text has been scanned with the scanner on the microphone—**ScanText** event

Mapping Microphone Events

Before you can receive microphone events, you need to map them. You map them in JavaScript by instantiating the **EventMapper** object as follows:

```
MicSink = new ActiveXObject("PowerscribeSDK.EventMapper");
```

You then map the **Microphone** object events:

```
MicSink.ButtonDown = HandleButtonDown;
MicSink.ButtonUp = HandleButtonUp;
MicSink.ScanText = HandleScanText;
MicSink.WavePosition = HandleWavePosition;
MicSink.MicConnectionChange = HandleMicConnectionChange;
```

Later, after you create the **Microphone** object, be sure to call **Advise()** on the **EventMapper** object to have the application receive the events:

```
try
{
    ...
    MicSink.Advise(objMicrophone);
}
catch(error)
{
    alert("Activating Mic Events: " + error.number + error.description);
}
```

Turning On Events from Microphone

Another step you might need to take to receive event notifications from the microphone applies only if you have previously taken over the **Microphone** object events; if you have, then you must return the events to normal functioning by calling the **ReceiveEvents()** method and passing it **False**:

```
objMicrophone.ReceiveEvents(false);
```

To take over the microphone, you disable receipt of events. For more information on taking over the microphone, refer to [Chapter 17, Taking Over Control of the Microphone on page 427](#).

Handling ButtonDown Event (Button Pressed)

Your application receives the **ButtonDown** event whenever a button on the microphone or foot pedal has been pressed. In a handler for this event, you can determine the particular button that has been pressed using the *MicButtonType* constants (see table) and, if warranted, take action in response. The handler could start out like this:

```
function HandleButtonDown(btnPressed)
{
    // Define the constants for use in JavaScript
}
```

MicButtonType Constants Sent by ButtonDown and ButtonUp Events

MicButtonType	Device	Button Label/Name or Graphic	Value
psMicButtonNone	PowerMic or PowerMic II	<i>None</i>	0
psMicButtonRecord	PowerMic or PowerMic II	DICTATE or 	1
psMicButtonTranscribe	PowerMic or PowerMic II	Button A (upper left) or 	2
psMicButtonForward	PowerMic or PowerMic II	FF or 	3
psMicButtonTabForward	PowerMic or PowerMic II	Button B (upper right) or 	4
psMicButtonPlayStop	PowerMic or PowerMic II	STOP/PLAY or 	5
psMicButtonBottomLeft	PowerMic or PowerMic II	Button C (lower left) or left 	6
psMicButtonRewind	PowerMic or PowerMic II	REW or 	7
psMicButtonBottomRight	PowerMic or PowerMic II	Button D (lower right) or right 	8
psMicButtonScan	Scan	SCAN  or 	13
psMicButtonTabBackward	PowerMic II		14
psMicButtonEnter	PowerMic II		15
psFootButtonPlay	Foot Pedal	Play	9
psFootButtonRewind	Foot Pedal	Rewind	10

MicButtonType	Device	Button Label/Name or Graphic	Value
psFootButtonForward	Foot Pedal	Forward	11
psFootButtonStop	Foot Pedal	Stop	12

For instance, if the button pressed is the **STOP/PLAY** button on a *PowerMic* microphone, you might want to distinguish the action that is intended (stop or play) and modify an indicator in your application:

```
if (btnPressed == psMicButtonPlayStop)
{
}
```

To determine the action being taken with the **STOP/PLAY** button, you can find out the current action taking place on the microphone by calling the **GetStatus()** method:

```
var btnStatus = objMicrophone.GetStatus();
```

The **MicStatus** values that the **GetStatus()** method can return are listed in the table below.

If the microphone is in the process of recording, playing, rewinding, or fast forwarding, the receipt of **psMicButtonPlayStop** in the **ButtonDown** event indicates that the button was pressed to stop the action, rather than to play the audio. If the microphone is idle, then the button was pressed to play the audio. You can use this information to take appropriate action in your application:

```
if (btnStatus != psMicStatusIdle)
{
    // Set light color to green to indicate Mic is idle
    objMicrophone.LightColor = psMicLightColorGreenOn;
}
else
{
    // Set light color to flashing green to indicate audio is playing
    objMicrophone.LightColor = psMicLightColorGreenFlashing;
}
```

You do not have to call the **Stop()** or **Play()** methods of the **Microphone** in your application, as the *SDK* automatically initiates the actual button functions.

Your application can take similar action for any of the buttons pressed.

MicStatus Values Returned by GetStatus()

MicStatus	Value
psMicStatusIdle	0
psMicStatusPlaying	1
psMicStatusRewinding	2
psMicStatusForwarding	3
psMicStatusRecording	4

SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

The application can continue the same response (highlighting of the indicator) until a **ButtonUp** event communicates that the action has stopped. Next, let's look at handling the **ButtonUp** event.

Handling ButtonUp Event (Button Released)

Your application receives the **ButtonUp** event whenever a button on the microphone or foot pedal has been released after being pressed. In a handler for this event, you can determine the particular button that has been released using the *MicButtonType* constants (see [MicButtonType Constants Sent by ButtonDown and ButtonUp Events on page 235](#)) and, if warranted, take action in response. The handler could start out like this:

```
function HandleButtonUp(btnPressed)
{
    // Define the constants for use in JavaScript
    // Modify Application mode indicators
}
```

In this handler, you can take actions similar to those you took in the **ButtonDown** event handler. On release of a microphone or foot pedal button, your application could check the status of the microphone and take appropriate action.

Tracking Audio Position by Handling WavePosition Event

While a person is recording, rewinding, or playing audio, your application can track the numeric location of the current sound and associated word in the audio. Each time the audio moves during these actions, the **WavePosition** event is fired, providing the current wave position in milliseconds.



Note: The wave position value tends to differ for the same position when you switch from recording to playing back the audio, because the engine removes silences from the recording. As a result, the playback audio length is shorter than the recording length.

Your **WavePosition** event handler could start out like this:

```
function WavePosition(currPosition)
{
    // Update the number of milliseconds displayed.
}
```

You might choose to display the number of milliseconds passed to the event handler.

Handling MicConnectionChange Event

Whenever a USB device, whether microphone or foot pedal, is connected to or disconnected from the hardware, the application receives a **MicConnectionChange** event.

The event sends a *MicConnectionStatus* to its event handler that indicates whether the USB device connected or disconnected.

The handler starts out looking like this:

```
function HandleMicConnectChange (MicConnectStatus)
{
    // Define the constants for
    // use in JavaScript
    var psMicConnected = 0;
    var psMicDisconnected = 1;
}
```

MicConnectionStatus Constants Sent by Mic-ConnectionChange Event()

MicConnectionStatus	Val
psMicConnected	0
psMicDisconnected	1

SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

You might want to take action in response to this event, such as displaying a status popup when the microphone or foot pedal has been connected and displaying a warning when the device has been disconnected:

```
if (MicConnectionStatus == psMicConnected)
    // Display Popup
else if (MicConnectionStatus == psMicDisconnected)
    // Display warning
```

Handling ScanText Event

Your application need only handle the **ScanText** event if the microphone has a scanner. Since the user can plug in a different device each time he or she dictates, you can have your application first determine whether or not the microphone has a scanner.

Determining Microphone Has Scanner

While working with the microphone, your application may need to determine whether or not the device has a scanner. To find out, you retrieve the Boolean value of the **IsScanner** property of the **Microphone** object:

```
if (objMicrophone.IsScanner)
{
    // Scan the bar code
}
```

Handling ScanText Event

The **ScanText** event fires whenever a user finishes scanning a bar code with the microphone scanner. The event sends the scanned bar code to its handler. The handler could start as follows, receiving the bar code in the single *scanText* argument:

```
function HandleScanText(scanText)
{
}
```

Once the handler receives the bar code in the *scanText* string, your application can take appropriate action on it. Typically, the bar code is an MRN or accession number that your application would use to create a report, assigning the bar code input as the report ID:

```
objReport = pscribeSDK.NewReport(scanText);
```

Alternatively, if a report is already currently active, you might want to insert the bar code input into the report by calling the **InsertText()** method of the **Report** object:

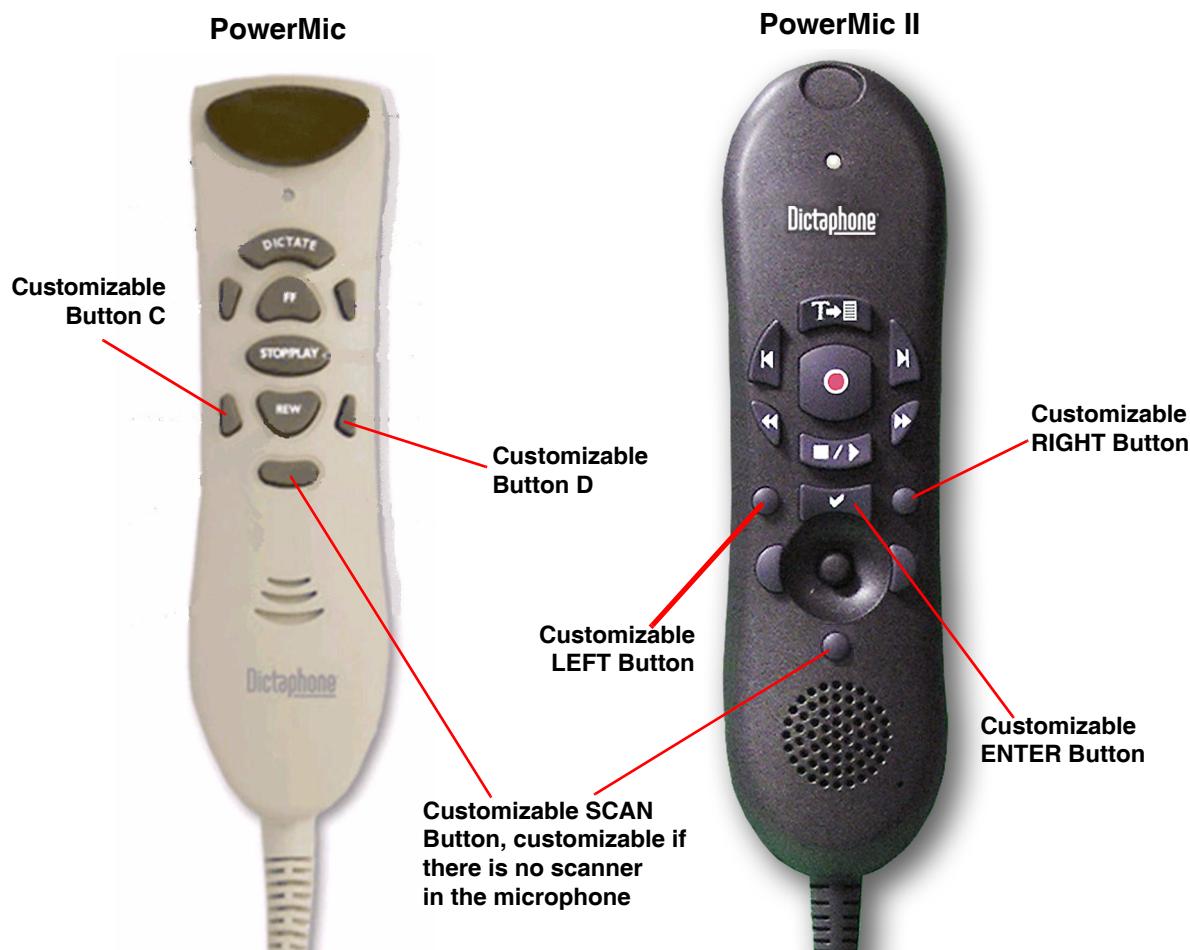
```
objReport.InsertText(scanText);
```

Customizing Programmable Microphone Buttons

Before you begin to customize the programmable microphone buttons, you should be sure to have a **PowerscribeSDK** object and instantiated **PlayerEditorCtl** or **WordPlayerEditorCtl**. Throughout the remainder of this chapter, the **PlayerEditor** mentioned refers to both types, unless otherwise specified.

In addition, you should have mapped **Microphone** object events and be sure that the application is receiving them (refer to [Mapping Microphone Events on page 234](#) and [Turning On Events from Microphone on page 235](#)).

The illustration below indicates where the customizable buttons are located on each *PowerMic* microphone model.



The next table lists the *MicButtonType* constants that the **ButtonDown** and **ButtonUp** events receive and how they correspond to customizable buttons on *PowerMic* microphones.

PowerMic Customizable Buttons and Corresponding MicButtonType Constants

Mic Model	Btn Name	MicButtonType	Value
PowerMic	Customizable Button C	psMicButtonBottomLeft	6
	Customizable Button D	psMicButtonBottomRight	8
	Customizable Scan Button	psMicButtonScan	13
PowerMic II	Customizable ENTER Button	psMicButtonEnter	15
	Customizable LEFT Button	psMicButtonBottomLeft	6
	Customizable RIGHT Button	psMicButtonBottomRight	8
	Customizable Scan Button	psMicButtonScan	13
SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.			

You would start your custom action in the **ButtonDown** and **ButtonUp** event handlers, where your application receives the button event notifications. For instance, in the **ButtonDown** event handler, when the user is working with a *PowerMic II* microphone, you could call an appropriate custom function in response to each customizable button pressed:

```
function HandleButtonDown(btnPressed)
{
    // Define the constants for use in JavaScript
    switch btnPressed
    {
        case psMicButtonBottomLeft:
            // Call custom text search function
            break;
        case psMicButtonBottomRight:
            // Call custom text replace function
            break;
        case psMicButtonEnter:
            // Call custom save report function
            break;
        case psMicButtonScan:
            // Call custom erase section text function
            break;
        default:
            exit;
    }
}
```

When your application receives non-customizable button events, you can ignore them and let the *SDK* manage them automatically.

Working with Headsets for Audio in Conjunction with Microphone or Foot Pedal Buttons

You can work with a headset for dictating and still use the buttons on either a microphone or a foot pedal. Your code does not have to change to work this way, as long as you have set up the devices in the control panel.

Code Summary

```
<HTML> <OBJECT ID="pscribeSDK">
</OBJECT>
<HEAD>
<TITLE>Powerscribe SDK</TITLE>
<!-- This code summary is not a complete program; it assumes
inclusion of required code from the previous chapters and requires
HTML elements not delineated -->
<SCRIPT type="text/javascript">

    // Define MicStatus constants for use in JavaScript
    var psMicStatusIdle = 0;
    var psMicStatusPlaying = 1;
    var psMicStatusRewinding = 2;
    var psMicStatusForwarding = 3;
    var psMicStatusRecording = 4;

    // Define MicButtonType constants for use in JavaScript
    var psMicButtonPlayStop = 5;
    var psMicButtonBottomLeft = 6;
    var psMicButtonBottomRight = 8;
    var psMicButtonEnter = 15;
    var psMicButtonScan = 13;

function InitializePS()
{
    //Bold statements in this function differ from earlier chapters
    try
    {
        // Instantiate PowerScribe SDK Object
        pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");
        // Initialize PowerScribe SDK Server
        var url = "http://server2/pscribesdk/";
        pscribeSDK.Initialize(url, "JavaScript_Demo");
        // The disableRealtimeRecognition argument of Initialize()
        // defaults to False, turning on realtime recognition

        // Event Mapper Object for Microphone in Bold
        MicSink = new ActiveXObject("PowerscribeSDK.EventMapper");

        MicSink.ButtonDown = HandleButtonDown;
        MicSink.ButtonUp = HandleButtonUp;
        MicSink.ScanText = HandleScanText
        MicSink.WavePosition = HandleWavePosition;
        MicSink.MicConnectionChange = HandleMicConnectionChange;
        ...
        try
        {
            objMicrophone = pscribeSDK.Microphone;
```

```
        MicSink.Advise(objMicrophone);
        objMicrophone.Tuning();
        objMicrophone.ReceiveEvents(false);
        objMicrophone.CustomMicTestingFunction();
        objMicrophone.CustomMicPropertiesSettings();
    }
    catch(error)
    {
        alert("Activating Mic Events: " + error.description);
    }
}
catch(error)
{
    alert("InitializePowerscribe: " + error.number + error.description);
}
}

function CustomMicTestingFunction
{
if ((objMicrophone.Type != psAudioDevicePowerMicI) ||
    (objMicrophone.Type != psAudioDevicePowerMicII))
{
    objMicrophone.Setup();
}
}

function CustomMicPropertiesSettings(Vol, RWspd, PLYspd, Mky, Norm, Tone)
{
    // Set these properties based on user input
    objMicrophone.PlaybackVolume = Vol;
    objMicrophone.WindingSpeed = RWspd;
    objMicrophone.PlaybackSpeed = PLYspd;
    objMicrophone.MonkeyChatter = Mky;
    objMicrophone.NormalizePlaybackVolume = Norm;
    objMicrophone.PlayAudioTone = Tone;

    // Set remaining properties to predefined settings
    var psMicOrientationRight = 0;
    var psMicOrientationLeft = 1;
    objMicrophone.Orientation = psMicOrientationRight;
    objMicrophone.MicButtonRecord = psMicButtonTypeDeadMan;
    objMicrophone.VoiceActivatedRecording = True;

    var psTranscribePreferenceSelect = 0;
    var psTranscribePreferenceCursorEnd = 1;
    var psTranscribePreferenceCursorBegin = 2;
```

```
objMicrophone.TranscribeSelectCount = 80;
objMicrophone.TranscribePreference = psTranscribePreferenceCursorEnd;

var psNavigateTypeAll = 0;
var psNavigateTypeNextField = 1;
var psNavigateTypeNextSection = 2;

objMicrophone.NavigatePreference = psNavigateTypeNextField;

var psMicBeepOptionField = 2;
var psMicBeepOptionOff = 0;

objMicrophone.BeepOnNavigate = psMicBeepOptionField;

return;
}

function HandleScanText(scanText)
{
    var strReportID = scanText;
    CreateReportWithScannedID(strReportID);
}

function CreateReportWithScannedID(strReportID)
{
    try
    {
        objReport = PowerscribeSDK.NewReport(strReportID);
        objReport.SetPlayerEditor(objRptEditor);
        ReptSink.Advise(objReport);
    }
    catch(error)
    {
        alert("Create Report:" + error.description);
    }
}

function HandleButtonDown(btnPressed)
{
    if (btnPressed == psMicButtonPlayStop)
    {
        var psMicLightColorOff = 0;
        var psMicLightColorRedOn = 1;
        var psMicLightColorRedFlashing = 2;
        var psMicLightColorGreenOn = 3;
        var psMicLightColorGreenFlashing = 4;

        var btnStatus = objMicrophone.GetStatus();

        if (btnStatus != psMicStatusIdle)
        {
            switch (btnStatus)
            {
```

```
        case psMicStatusPlaying:
            objMicrophone.LightColor = psMicLightColorGreenFlashing;
            break;
        case psMicStatusRewinding:
            objMicrophone.LightColor = psMicLightColorGreen;
            break;
        case psMicStatusForwarding:
            ObjMicrophone.LightColor = psMicLightColorRedFlashing;
            break;
        case psMicStatusRecording:
            ObjMicrophone.LightColor = psMicLightColorRed;
            break;
        default:
            break;
    }
}
else
{
    // Turn Microphone Light Off to indicate it is idle
    objMicrophone.LightColor = psMicLightColorOff;
}
else
{
    // Handle only custom buttons; otherwise,
    // Let SDK take default actions
    switch (btnPressed)
    {
        case psMicButtonBottomLeft:
            // Call custom text search function
            break;

        case psMicButtonBottomRight:
            // Call custom text replace function
            break;

        case psMicButtonEnter:
            // Call custom save report function
            break;

        case psMicButtonScan:
            // Call custom erase section text function
            break;

        default:
            exit;
    }
}
```

```
function HandleButtonUp(btnPressed)
{
    // Define the constants for use in JavaScript
    // Modify Application mode indicators
}

function WavePosition(currPosition)
{
    // Update the number of milliseconds displayed.
}

function HandleMicConnectionChange(MicConnectionStatus)
{
    // Define the constants for use in JavaScript
    var psMicConnected = 0;
    var psMicDisconnected = 1;

    if (MicConnectionStatus == psMicConnected)
        // Display Popup
    else if (MicConnectionStatus == psMicDisconnected)
        // Display warning
}

</SCRIPT>
</HEAD>

<BODY bgcolor="#fffff" language="javascript" onload="InitializePS()"
onunload="UninitializePS()">

<form>
...
<div id="objRptEditor">
    <object id="PlayerEditor" style="WIDTH: 605px; HEIGHT: 265px; BACK-
    GROUND-COLOR: #EFE7CE" ...
        classid="clsid:A470D150-C3FF-4584-9B0B-7D31AF4621C1" viewastext>
    </object>
</div>

<div id="objLevelMeter">
    <object id="LevelMeter" height="20" width="150" ...
        classid="clsid:492FFC0D-0A4A-4EAD-BC52-966AE1568FCD">
    </object>
</div>

...
    <!-- Set Properties using values from text boxes ->
    <input language="javascript" type="button" name="btnSetProperties"
        value="Set Properties"
        onClick="CustomMicPropertiesSettings(Vol.value,RWspd.value,
        PLYspd.value, Mky.value, Norm.value, Tone.value)"><br>
    ...
</form>
</body>
```

Using Command and Control

Objectives

This chapter introduces the **SDK Grammar** and **Rules** objects and covers how to use them to create custom voice commands and have your application respond to them. It contains the following topics:

- [What Is an SDK Grammar?](#)
- [Steps to Defining and Using a Grammar](#)
- [Creating XML Grammar Template](#)
- [Creating a Grammar Object](#)
- [Creating a Rules Object](#)
- [Preparing to Handle Grammar Events](#)
- [Switching to Command Mode](#)
- [Handling Command Events](#)

What Is an SDK Grammar?

A grammar is a collection of specialized custom voice commands that you define in an XML file and can use only in **Command** mode. These commands are always separate from the default grammar voice commands that are embedded in the *SDK*, such as **Next Field**, **End of Line**, **Next Section**, and **Finish Report**, all operational in **Dictation** mode.

You might want to have a set of voice commands, for instance, for a group of users who cannot use the mouse because they need to have their hands free. For example, a pathologist doing an autopsy might work from a headset and never be able to click a mouse or keyboard key while dictating. In such a situation, you might want to have voice commands to take such basic actions as displaying shortcuts and creating or editing a report. The commands you would need might be:

- Display Shortcuts
- Edit Report
- Create Report

You could also create a set of voice commands for checking off options on a form, like the one shown in the adjacent illustration.

In this case, the voice commands would call the heading name first, then the selection under that heading; some of the commands would be:

- Cardiac Normal Left Ventricle
- Cardiac Blocked Left Ventricle
- Pulmonary Visceral Pleura
- Renal Artery
- Renal Pelvis

You can create several sets of voice commands by creating a **Grammar** object for each one.

Your application can then deploy one or more **Grammar** objects, depending on the situation of the person who is dictating.

Cardiac

- Normal Left Ventricle
- Blocked Left Ventricle
- Normal Right Ventrical
- Blocked Right Ventrical

Pulmonary

- Visceral Pleura
- Mediastinal Pleura
- Costal Pleura
- Pleural Cavity

Renal

- Artery
- Pelvis
- Medulla
- Cortex

Steps to Defining and Using a Grammar

To create your custom set of commands (a grammar):

- Create an **EventMapper** for the **Grammar** object **Command** event and link it to the handler.
- Create an XML grammar template that contains the rules of the grammar, using the voice XML standard outlined at <http://www.w3.org/TR/2002/CR-speech-grammar-20020626/>.

- Create the **Grammar** object and activate all rules.
- Create a **Rules** object to retrieve the rules for the grammar.
- Set up a mechanism to switch the application between **Dictation** mode and **Command** mode, or to have the end user dictate default grammar commands to change the mode.
- Create a **Command** handler to handle the commands when the application receives them.

Creating XML Grammar Template

You define each grammar in an XML file. The grammar is made up of a collection of rules. The rules contain the words that the speech recognition engine should interpret as commands.

You start the grammar with a mandatory opening line of XML code, then at the top level you have a **<grammar>** section that defines the contents of the file as a grammar. In the **<grammar>** tag, you assign the following attributes of the grammar:

- **version**—Always “**1.0**”
- **mode**—Set to “**voice**” to indicate it is a spoken command grammar
- **root**—Assign a unique name to the root
- **xmlns**—Assign the namespace for the grammar

```
<grammar version="1.0" mode="voice" root="Lungs" xmlns="Lungs">
... </grammar>
```

Inside the **<grammar>** section of the XML file, you create a **<rule>** section for each word that should be interpreted as a command in **Command** mode, for example:

```
<rule id="LungsRule" scope="public">
<one-of>
    <item output="LNG"> Lungs </item>
</one-of>
</rule>
```

You put each word or phrase that in a **<one-of>** section and define the individual item in an **<item>** section with an **output** attribute set to an identifier for the item. The actual content of the **<item>** section is the word or phrase to be recognized as a command and passed as an argument to the **Command** event handler. For the example shown here, when the *SDK* engine recognizes the word it *hears* is **Lungs**, it triggers a **Command** event and passes **LNG** to the event handler as an argument.

Grammar with Single Level

A simple grammar would have only one level of <one-of> and <item> sections, like the [Single Level Rules for Form Grammar](#) shown below, where you would dictate a single word, either **Lungs**, **Neck**, or **Brain** and the application could be programmed to, for instance, select the corresponding check box.

Each rule contains all elements that a single dictated statement can contain. Because each word is a command by itself in this case, each has its own rule.

Single Level Rules for Form Grammar

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar version="1.0" mode="voice"
    root="Options" xmlns="UpperBody">

    <rule id="LungsRule" scope="public">
        <one-of>
            <item output="LNG" > Lungs </item>
        </one-of>
    </rule>

    <rule id="NeckRule" scope="public">
        <one-of>
            <item output="NCK" > Neck </item>
        </one-of>
    </rule>

    <rule id="BrainRule" scope="public">
        <one-of>
            <item output="BRN" > Brain </item>
        </one-of>
    </rule>
</grammar>
```

The **output** attribute's value for each command (circled in the listing) in the grammar becomes the argument sent to the **Command** event handler when the corresponding word is dictated.

- Lungs
- Neck
- Brain

Commands Grammar Defines & Corresponding Args Sent to Command Event Handler

Dictated Phrase Event Handler Responds To...	When Arg Sent to Event Handler Is...
Lungs	LNG
Neck	NCK
Brain	BRN

Grammar with Nested Item Levels

You might want to have commands made of multiple words where you can choose a starting word and then choose one of several words to follow it.

For instance, you could choose an item from a form with several items (see adjacent illustration), where **Cardiac** is one of the top items, and it has several subitems you might want to dictate. The combinations of commands you could use in this example would be:

- Cardiac, Normal Left Ventricle
- Cardiac, Blocked Left Ventricle
- Cardiac, Normal Right Ventricle
- Cardiac, Blocked Right Ventricle

Cardiac

- Normal Left Ventricle
- Blocked Left Ventricle
- Normal Right Ventricle
- Blocked Right Ventricle

The grammar for this form would contain a very clear structure, as shown in the XML listing of the [Two-Level Rules for Form Grammar](#).

Two-Level Rules for Form Grammar

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar version="1.0" mode="voice" root="MedForm" xmlns="MedForm">
<rule id="Cardiac" scope="public">

<item>Cardiac</item>
<one-of>
    <item output="norm">Normal</item>
    <item output="blocked">Blocked</item>
<one-of>
    <one-of>
        <item output="Crd-Left">Left</item>
        <item output="Crd-Right">Right</item>
    </one-of>
    <item>Ventricle</item>
</one-of>
</rule>
</grammar>
```

Commands Grammar Defines &
Corresponding Arguments Sent to
Command Event Handler

Dictated Phrase Event Handler Responds To...	When Argument Sent to Event Handler Is...
Cardiac Normal Left Ventricle	normCrd-Left
Cardiac Blocked Left Ventricle	blockedCrd-Left
Cardiac Normal Right Ventricle	normCrd-Right
Cardiac Blocked Right Ventricle	blockedCrd-Right

Note that there is a separate rule for each phrase. A single rule contains all elements that a single dictated statement can contain. Each distinct element in the multi-level grammar has a unique **output** attribute value that identifies that particular dictated phrase. The **output attribute** values of the dictated words are concatenated to form the argument that the **Command** event sends to its handler.

Grammar with Optional Phrases

You might want to have commands of several words where you can choose to start with one of several verbs, such as **Display**, **Open**, or **Create** and always act on a single noun, like a **Report**. In addition, you could have optional phrases in between **Create** and **Report**, like **XML**, which refines the type of report or **New**, which is extraneous, but the dictation user might say. The combinations of commands you could dictate would be:

- Display Report
- Open Report
- Create Report
- Create New Report
- Create XML Report
- Create New XML Report

Because a single rule in a grammar contains all elements that a single dictated statement can contain, this rule would contain much more information than rules for simpler grammars (see [Multi-Level Rules for Grammar with Optional Phrases, Rule for Action1](#)).

The **Action1** rule, shown below, *understands* when the user dictates **Display** or **Open** followed by **Report**. The **output** attribute determines the argument that the **Command** event sends to its handler. Dictating **Display Report** sends an argument of **DisplayCmd** to the **Command** event handler, whereas dictating **Open Report** sends an argument of **OpenCmd**.

Multi-Level Rules for Grammar with Optional Phrases, Rule for Action1

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar version="1.0" mode="voice" root="Pathology" xmlns="Pathology">
<rule id="Action1" scope="public">
<item>
  <one-of>
    <item output="DisplayCmd">Display</item>
    or
    <item output="OpenCmd">Open</item>
  </one-of> 2
  <item>Report</item>
</item>
</rule>
```

Commands Action1 Rule of Grammar Defines & Corresponding Arguments Sent to Command Event Handler

Dictated Phrase Event Responds To...	When Argument Sent to Event Handler Is...
Display Report	DisplayCmd
Open Report	OpenCmd

(The Action2 rule is in the continuation of this grammar on next page.)

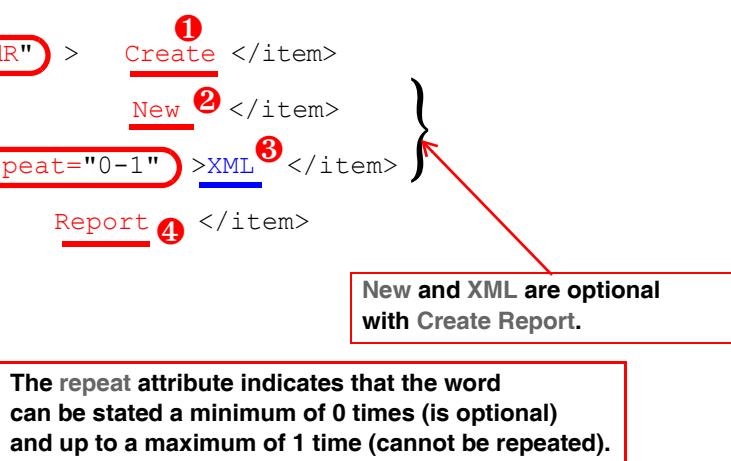
In the continuation of the multi-level grammar with optional phrases shown in the next listing, the **Action2** rule *understands* more combinations of dictated phrases.

The **output** attributes of the various words in the dictated phrase are concatenated to form the argument that the **Command** event sends to its handler. Notice, however, that the optional word **New** does not have an **output** attribute. Why? Because dictating the word **New** actually has no effect. Because **Create Report** and **Create New Report** actually initiate the same action, they send the same argument string (**CreateCmdR**) to the event handler. By contrast, adding the optional term **XML** to the dictated statement changes the ultimate action taken, creating an entirely different type of report, so the word **XML** has an **output** attribute. The **Command** event concatenates the **X** onto the argument it sends to its handler when the phrase dictated includes **XML** (**CreateCmdRX**).

Multi-Level Rules for Grammar with Optional Phrases, Rule for Action2

```
<rule id="Action2" scope="public">
<item>
  <item output="CreateCmdR" > Create ① </item>
  <item repeat="0-1" > New ② </item>
  <item output="X" repeat="0-1" > XML ③ </item>
  <item> Report ④ </item>
</item>
</rule>
</grammar>
```

The word **New** does not require an **output** attribute, because it has no affect on the command—**Create XML Report** and **Create New XML Report** result in the same action.



Commands Action2 Rule of Grammar Defines & Corresponding Arguments Sent to Command Event Handler

Dictated Phrase Event Responds To...	When Argument Sent to Event Handler Is...
Create Report Create New Report	CreateCmdR
Create XML Report Create New XML Report	CreateCmdRX

The **repeat** attribute indicates how many times the word can be stated in the dictated phrase. If **repeat** equals **0**, then the word is optional; if **repeat** equals **1**, the word cannot be repeated. You can provide a range for the value of **repeat**, just as the example shown here provides a range of **0-1**.

The complete listing of this grammar is shown next.

Complete Listing of Multi-Level Rules for Grammar with Optional Phrases

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar version="1.0" mode="voice" root="Pathology"
xmlns="Pathology"><rule id="Action1" scope="public">
<item>
  <one-of>
    <item output="DisplayCmd">Display</item>
    <item output="OpenCmd">Open</item>
  </one-of>
  <item>Report</item>
</item>
</rule>

<rule id="Action2" scope="public">
<item>
  <item output="CreateCmdR">Create</item>
  <item repeat="0-1">New</item>
  <item output="X" repeat="0-1">XML</item>
  <item>Report</item>
</item>
</rule>
</grammar>
```

Grammar with Multiple Verbs for Same Action

You might want to have more than one command that takes the same action and/or choose from more than one object of that action, creating, for instance, these command phrases:

- Show Shortcuts
- Display Shortcuts
- Show Words
- Display Words

Because **Show** and **Display** take the same action, you need only differentiate between the objects of the actions, **Shortcuts** and **Words**, by assigning each a distinct **output** attribute value.

Multi-Level Rules for Grammar with Multiple Verbs for Same Action

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar version="1.0" mode="voice" root="Pathology" xmlns="Pathology">

<rule id="Action" scope="public">
  <item>
    <one-of>
      <item> Show </item>
      <item> Display </item>
    </one-of>
    <one-of>
      <item output="Shcts"> Shortcuts </item>
      <item output="Wrds"> Words </item>
    </one-of>
  </item>
</rule>
</grammar>
```

Either Show or Display (to take the same action) can be followed by Shortcuts (one action) or Words (diff action).

Commands Grammar Defines That Send "Shcts" to Command Event Handler

Dictated Phrase Event Handler Responds To...	When Argument Sent to Event Handler Is...
Show Shortcuts	Shcts
Display Shortcuts	Shcts

Commands Grammar Defines That Send "Wrds" to Command Event Handler

Dictated Phrase Event Handler Responds To...	When Argument Sent to Event Handler Is...
Show Words	Wrds
Display Words	Wrds

Creating a Grammar Object

To use the grammar template you just created in your application, you create a **Grammar** object using the **Grammar** property of the **PowerscribeSDK** object and pass it the XML string (also referred to as an XML *template*):

```
var sMainGrTemplate = "<?xml version='1.0' encoding='UTF-8'?> <grammar  
version='1.0' mode='voice' root='Pathology' xmlns='Pathology'> <rule  
id='Action' scope='public'>..."  
  
objGrammar = pscribeSDK.Grammar(sMainGrTemplate);
```

Once your application has the **Grammar** object, it needs to activate the rules of the grammar using the **ActivateAllRules()** method of the **Grammar** object:

```
objGrammar.ActivateAllRules();
```

You might activate the rules of the grammar in the initialization phase of your application. Later, on uninitialization, you deactivate those rules using the **DeactiveAllRules()** method:

```
objGrammar.DeactivateAllRules();
```

To have more than one **Grammar** available in the application, you can create another **Grammar** object using the **Grammar** property and pass it the XML string for that grammar:

```
var form2Template = "<?xml version='1.0' encoding='UTF-8'?> <grammar  
version='1.0' mode='voice' root='ExamForm' xmlns='ExamForm'> <rule  
id='Exam' scope='public'>..."  
  
objForm2Grammar = pscribeSDK.Grammar(form2Template);
```

You then can activate the rules of this grammar to use it alongside the original grammar.

You can have up to 256 active grammars in a single application.

Once the rules are active, you should also be sure to create an **EventMapper** object for the **Grammar**, link it to the **Command** event handler, and call the **Advise()** method to begin receiving the events. Refer to [Preparing to Handle Grammar Events on page 258](#) for more information.

Creating a Rules Object

To display any information about the rules in the grammar, your application can retrieve the **Rules** object, which enumerates all the **Rules** in the grammar, and take appropriate action using them. You retrieve the **Rules** object using the **Rules** property of the **Grammar** object:

```
var objGrRules = objGrammar.Rules;  
var countRules = objGrRules.Count;  
// Display rules information for the user
```

Working with Rules

When you retrieve the **Rules** collection of the **Grammar** object, you retrieve all the individual rules of that grammar. You retrieve the collection using the **Rules** property:

```
objRulesCollect = objGrammar.Rules;
```

The collection is a standard COM collection, so you can retrieve individual **Rule** object from the collection using the standard Microsoft **Item()** method:

```
objRule = objRulesCollect.Item(1);
```

Each **Rule** object in the **Rules** collection has an individual **Name** that is set by the **id** attribute in the **<rule>** section of the grammar:

```
<rule id="Action" scope="public">
```

You can test the **Name** property of the **Rule** object to identify the rule:

```
if (objRule.Name == "Action")  
{  
}
```

When working with the **Rule** object, you can use its **State** property to determine whether or not it has been activated. The **State** property value is either **GrammarRuleStateActive** or **GrammarRuleStateInactive**:

```
var GrammarRuleStateInactive = 0;  
var GrammarRuleStateActive = 1;  
  
if (Rules.Item(i).State == GrammarRuleStateActive)  
{  
}
```

Or you can set the **Rule** object's **State** property to **GrammarRuleStateActive** or **GrammarRuleStateInactive** to activate or deactivate the individual rule. For instance, if a rule corresponds to a checked off selection on a form, then you might want to activate that rule, whereas if it has not been checked off, you might want to deactivate that rule:

```
for (var i = 1; i <= Rules.Count; i++)  
{  
    // If the check box for the item is checked  
    Rules.Item(i).State = GrammarRuleStateActive  
    else  
        Rules.Item(i).State = GrammarRuleStateInactive  
}
```

Once the application is in **Command** mode, the *Speech Recognition* engine responds to the commands of rules that have been activated and ignores those that have been deactivated.

Preparing to Handle Grammar Events

As with other objects that fire events, you need to handle the **Command** event of the **Grammar** object.

Mapping Grammar Object Events

To prepare to handle **Command** events that the **Grammar** object fires, you can map the events with the **EventMapper** object:

```
GrmrSink = new ActiveXObject("PowerscribeSDK.EventMapper");
```

You should then link the **Command** event handler to the **Command** event:

```
GrmrSink.Command = HandleCommand;
```

And, finally, be sure to call the **Advise()** method of the **PowerscribeSDK** object and pass it the **Grammar** object:

```
GrmrSink.Advise(objGrammar);
```

Switching to Command Mode

For the rules associated with the grammar to take effect, your application must be in **Command** mode. You might have the user choose to switch to **Command** mode or you might set your application up to automatically switch to **Command** mode during initialization.

To put the application in **Command** mode during initialization, you set the **Mode** property of the **PowerscribeSDK** object to **psSystemModeCommand**:

```
var psSystemModeDictateOnly = 0;  
var psSystemModeDictate = 1;  
var psSystemModeCommand = 2;  
var psSystemModeNumber = 3;  
var psSystemModeLetter = 4;  
  
pscribeSDK.Mode = psSystemModeCommand;
```

You can also activate existing default commands and use these commands to switch modes:

Switch to Command

Switch to Dictate

Switch to Number

Switch to Spell

To activate these commands and other default commands, refer to [Deploying DefaultGrammar Commands on page 262](#).

Handling Command Events

Once your application is in **Command** mode, when the speech recognition engine *hears* and recognizes a word from the grammar that it should interpret as a command, the engine triggers that word's **Rule** and the **Grammar** object fires a **Command** event.

The **Command** event handler needs to receive two arguments:

- Word or phrase from the grammar that was recognized as a command
- The *OutConstruction* argument, the **output** attribute setting of the item in the **Rule** of the rule that was triggered; for instance, **LNG** for **Lungs** in the rule item from the grammar shown above.

```
<rule id="LungsRule" scope="public">
  <one-of>
    <item output="LNG"> Lungs </item>
  </one-of>
</rule>
```

The handler starts out as follows:

```
function HandleCommand(strPhrase, strOutConstruction)
{
}
```

You use the **Command** handler to handle any and all commands in the grammar. To determine the action to take, the handler uses the **output** attribute of the **Rule** item to successfully identify the rule that was triggered:

```
if (strOutConstruction == "LNG")
{
    // Check Lung option in the form
}
```

Handling Multi-Level Rules in a Grammar

In a grammar with multi-level rules, your **Command** event handler receives an event with the phrase containing all the words from all levels of items in the rule; for example, if the application is using the grammar shown in the adjacent block, when the dictated phrase is **Cardiac Left Ventricle**, your handler receives **Cardiac Left Ventricle** as the phrase and **Crd-Left** as the *OutConstruction* argument.

For this particular grammar, when the **Grammar** object fires the **Command** event and the handler receives either **Crd-Left** or **Crd-Right** as the *OutConstruction* argument, the application can take appropriate action based on which value the argument contains:

Excerpt from Grammar with Rules for Form Commands

```
<rule id="Cardiac" scope="public">
    <one-of>
        <item output="Crd-Left">Cardiac</item>
        <one-of>
            <item>Left Ventricle </item>
        </one-of>
    </one-of>
</rule>

<rule id="Cardiac2" scope="public">
    <one-of>
        <item output="Crd-Right">Cardiac</item>
        <one-of>
            <item>Right Ventricle </item>
        </one-of>
    </one-of>
</rule>
```

```
function HandleCommand(strPhrase, strOutConstruction)
{
    if (strOutConstruction == "Crd-Left")
    {
        // Check Left Ventricle in the form
    }
    else if (strOutConstruction == "Crd-Right")
    {
        // Check Right Ventricle in the form
    }
}
```

Handling Multi-Level Rules with Optional Phrases

For a grammar with both multiple levels and optional phrases (see partial listing below), your handler can have several cases:

```
switch strOutConstruction
{
    case "DisplayCmd":
        // Display Report
        break;
    case "OpenCmd":
        // Edit Report
        break;
    case "CreateCmdP":
        // Create New Report
        // or Create Report
        break;
    case "CreateCmdX":
        // Create New XML Report
        // or Create XML Report
        break;
    default
        break;
}
```

Excerpt from Grammar with Rules for Report Commands

```
<one-of>
    <item output="DisplayCmd">Display</item>
    <item output="OpenCmd">Open</item>
    <item>
        <item output="CreateCmdP">Create
        </item>
        <item repeat="0-1"> New </item>
    </item>
    <item>
        <item output="CreateCmdX">Create
        </item>
        <item repeat="0-1">New</item>
        <item>XML</item>
    </item>
</one-of>
<one-of>
    <item>Report</item>
</one-of>
```

Handling Rules with Multiple Verbs for One Action

For a grammar with multiple verbs for the same action on multiple nouns (see partial listing in the adjacent block), your handler can have multiple cases:

```
switch strOutConstruction
{
    case "Shcts":
        // Display Pulldown List
        // of Shortcuts
        break;
    case "Wrds":
        // Display Pulldown List
        // of Words
        break;
    default
        break;
}
```

Excerpt from Grammar with Rules to Show Shortcuts/Words

```
<one-of>
    <item>
        <one-of>
            <item>Show</item>
            <item>Display</item>
        </one-of>
        <one-of>
            <item output="Shcts">Shortcuts
            </item>
            <item output="Wrds">Words</item>
        </one-of>
    </item>
</one-of>
```

Deploying DefaultGrammar Commands

As mentioned earlier under [Switching to Command Mode on page 258](#), the *SDK* provides some commands that take predefined actions (see the table below). These commands are delineated in a default grammar.

Your application can deploy these commands in one of two ways:

- Have the commands take the predefined actions listed here.
- Modify the commands to take custom actions defined in a customized default grammar.

DefaultGrammar Commands and Their Predefined Actions

Command		Arguments Passed to Command Event Handler	Default Action (formatting is saved for XML reports only)
Category	How Dictated		
General Actions	Switch to Dictate	DictateMode	Switch the mode to Dictation .
	Switch to Command	CommandMode	Switch the mode to Command .
	Switch to Spell	SpellMode	Switch the mode to Letter/Spell .
	Switch to Number	NumberMode	Switch the mode to Number .
Editor Actions	Bold That	Bold	Bold current selected text in PlayerEditor.
	Italicize That	Italic	Italicize current selected text in PlayerEditor.
	Underline That	Underline	Underline current selected text in PlayerEditor.
	Capitalize That	Capitalize	Capitalize current selected text in PlayerEditor.
	Beginning of Document	BeginDocument	Go to beginning of the current editor text.
	End of Document	EndDocument	Go to end of the current editor text.
	Begin of Line	BeginLine	Place cursor at beginning of current line.
	End of Line	EndLine	Place cursor at end of current line.
	Unselect That	Unselect	Place cursor at beginning of current selection.
	Next Field	NextField	Navigate to the next field in square brackets [] in the PlayerEditor text (simulate right push).
	Previous Field	PreviousField	Navigate to the previous field in square brackets [] in the PlayerEditor text.
	Next Section	NextSection	Navigate to next section. Positions cursor at the end of the current section of XML report.
	Previous Section	PreviousSection	Navigate to previous section. Positions cursor at the end of the current section of XML report.

Command		Arguments Passed to Command Event Handler	Default Action (formatting is saved for XML reports only)
Category	How Dictated		
Numbered List Actions	Number one	StartNumList	Begins a new numbered list.
	One Period	StartNumList	Begins a new numbered list.
	Begin List	StartList	Begins a new list.
	Begin Numbered List	StartNumList	Begins a new numbered list.
	Beginning of list	StartList	Begins a new list.
	Beginning of Numbered List	StartNumList	Begins a new numbered list.
	Start List	StartList	Begins a new list.
	Start Numbered List	StartNumList	Begins a new numbered list.
	New List	StartList	Begins a new list.
	New Numbered List	StartNumList	Begins a new numbered list.
	End list	EndList	Ends the current list.
	End Numbered List	EndList	Ends the current numbered list.
	Stop List	EndList	Ends the current list.
	Stop Numbered List	EndList	Ends the current numbered list.
	End of list	EndList	Ends the current list.
	End of Numbered List	EndList	Ends the current numbered list.
Dictation Actions	Transcribe	Transcribe	No default implementation.
	Sign Report	Sign	No default implementation.
	Finish Report	Done	No default implementation.

Implementing Predefined Actions

To include the default grammar commands in your application, you create a **DefaultGrammar** object using the **DefaultGrammar** property of the **PowerscribeSDK** object and pass **True** to the property:

```
objDefGrammar = pscriveSDK.DefaultGrammar(True);
```

The property takes an argument of **True** to implement the predefined actions of the commands. To then activate the rules of the default grammar, as with any other grammar, you call **ActivateAllRules()** on the object:

```
objDefGrammar.ActivateAllRules();
```

Your application now accepts the default grammar commands (shown in the previous table) when they are spoken in **Dictation** mode and takes the default actions for those commands, with the exceptions of the **Transcribe**, **Sign Report**, and **Finish Report**, which take no default action.

Implementing Custom Actions

You can, alternatively, choose to selectively modify the standard actions that default grammar commands take in your application by working with the **DefaultGrammar** object.

You create the default grammar object using the **DefaultGrammar** property of the **PowerscribeSDK** object. To use the existing commands in the grammar but override the default actions of the commands, you pass the property an argument of **False**:

```
objDefGrammar = objPscribeSDK.DefaultGrammar(False);
```

You then implement your alternative to the default grammar by coding the action to be taken when one of the rules triggers a **DefaultGrammar Command** event (with the same prototype as the **Grammar Command** event) and the event handler receives an *OutConstruction* argument that indicates the particular command that was dictated. For instance, the **DefaultGrammar Command** event handler would receive the string “**Done**” when the **Finish Report** command has been dictated, the string “**Sign**” for the **Sign Report** command, and the string “**Transcribe**” for the **Transcribe** command. The handler might implement all three of these commands, as they otherwise would take no action:

```
function HandleDefGrammarCommand(strPhrase, strOutConstruction)
{
    if (strOutConstruction == "Done")
    {
        objReport.Save();
        ...
    }

    if (strOutConstruction == "Sign")
    {
        objReport.Save();
        objReport.MarkforAdaptation();
        ...
    }

    if (strOutConstruction == "Transcribe")
    {
        objMicrophone.Transcribe();
        objReport.Save();
        ...
    }
}
```

The start of a possible custom response to each command is shown here, but your application could take other or additional actions.

Code Summary

```

<HTML>
<OBJECT ID="pscribeSDK">
</OBJECT>
<HEAD>
<TITLE>Powerscribe SDK</TITLE>
<!-- This code summary is not a complete program; it assumes
inclusion of required code from the previous chapters -->
<SCRIPT type="text/javascript">
...
function InitializePS()
{
    //Bold statements in this function differ from earlier chapters
    try
    {
        // Instantiate PowerScribe SDK Object
        pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");
        // Initialize PowerScribe SDK Server
        var url = "http://server2/pscribesdk/";
        pscribeSDK.Initialize(url, "JavaScript_Demo");
        // The disableRealtimeRecognition argument of Initialize()
        // defaults to False, turning on realtime recognition
        ...
        // Event Mapper Object for Grammar and DefaultGrammar
        GrmrSink = new ActiveXObject("PowerscribeSDK.EventMapper");
        GrmrSink.Command = HandleCommand;

        DefGrmrSink = new ActiveXObject("PowerscribeSDK.EventMapper");
        DefGrmrSink.Command = HandleDefGrammarCommand;
        ...
        CreateGrammarObjects();
    }
    catch(error)
    {
        alert("InitializePowerscribe: " + error.number + error.description);
    }
}

function CreateGrammarObjects()
{
    // Assign Main Grammar XML to String and Create Main Grammar Object
    var sMainGrTemplate = "<?xml version='1.0' encoding='UTF-8'?>
    <grammar version='1.0' mode='voice' root='Pathology'
    xmlns='Pathology'> <rule id='Action' scope='public'>...""
    objGrammar = pscribeSDK.Grammar(sMainGrTemplate);
}

```

```
// Assign Exam Grammar XML to String, Create Exam Form2 Grammar Object
var sForm2GrTemplate = "<?xml version="1.0" encoding="UTF-8"?>
<grammar version="1.0" mode="voice" root="ExamForm2"
xmlns="ExamForm2"> <rule id="Examinations" scope="public">...""

objGrammar2 = pscribeSDK.Grammar(sForm2GrTemplate);

// Create Default Grammar Object
objDefGrammar = pscribeSDK.DefaultGrammar(true);

// Activate Rules for All Grammar Objects
objGrammar.ActivateAllRules();
objGrammar2.ActivateAllRules();
objDefGrammar.ActiveAllRules();

// Activate Events for All Grammar Objects:
GrmrSink.Advise(objGrammar);
GrmrSink.Advise(objGrammar2);
DefGrmrSink.Advise(objDefGrammar);
}

function SwitchMode()
{
    var SystemModeDictate = 1;
    var SystemModeCommand = 2;

    // if user has selected Dictate mode, if in Command mode, switch
    {
        if (pscribeSDK.Mode == SystemModeCommand)
            pscribeSDK.Mode = SystemModeDictate;
    }

    // if user has selected Command Mode; if in Dictate mode, switch:
    {
        if (pscribeSDK.Mode == SystemModeDictate)
            pscribeSDK.Mode = SystemModeCommand;
    }
}

function ListActiveRules(objGrammarObject)
{
    var objGrammarRulesCollection = objGrammarObject.Rules;
    var countRules = objGrammarRulesCollection.Count;

    // Define RuleState constants for use in application
    var GrammarRuleStateInactive = 0;
    var GrammarRuleStateActive = 1;

    for (var i = 1; i <= objGrammarRulesCollection.Count; i++)
    {
        var objRule = objRulesCollect.Item(i);

        // If Rule's state is Active, include rule in list
        if (objRule.State == GrammarRuleStateActive)
```

```
        // Print the rule to the list
        elseif objRule.State == GrammarRuleStateInactive
            // Skip this Rule and check the next Rule
    }
}

function HandleCommand(strPhrase, strOutConstruction)
{
    // Check strOutConstruction to see which command received
    // Handle commands from more than one grammar

    if (strOutConstruction == "LNG")
    {
        // Check Lung option in the form
    }
    else if (strOutConstruction == "NCK")
    {
        // Check Neck option in the form
    }
    else if (strOutConstruction == "BRN")
    {
        // Check Brain option in the form
    }

    switch strOutConstruction
    {
        case "DisplayCmd":
            // Display Report
            break;

        case "OpenCmd":
            // Edit Report
            break;

        case "CreateCmdP":
            // Create New Report or Create Report
            break;

        case "CreateCmdX"
            // Create New XML Report or Create XML Report
            break;

        default
            break;
    }
}

function HandleDefGrammarCommand(strPhrase, strOutConstruction)
{
    // Take actions for those commands with no predefined implementation
    if (strOutConstruction == "Done")
    {
        objReport.Save();
    }
}
```

```
if (strOutConstruction == "Sign")
{
    objReport.Save();
    objReport.MarkforAdaptation();
}

if (strOutConstruction == "Transcribe")
{
    objMicrophone.Transcribe();
    objReport.Save();
}

}

function DoLogoff()
{
try
{
    objGrammar.DeactivateAllRules();
    objGrammar2.DeactivateAllRules();
    objDefGrammar.DeactivateAllRules();

    pscribeSDK.Logoff();
}
catch(error)
{
    alert(error.description);
}
}

...
</SCRIPT>
</HEAD>

<BODY bgcolor="#fffff" language="javascript" onload="InitializePS()"
onunload="UninitializePS()">

<form>
...
<input language="javascript" type="button" name="btnChangeMode"
      value="Login" onClick="SwitchMode()"><br>
<input language="javascript" type="button" name="btnListActiveRules"
      value="Login" onClick="ListActiveRules()"><br>
<input language="javascript" type="button" name="btnLogout"
      value="Logout" onClick="DoLogoff()"><br>

...
</form>
...
</BODY>
</HTML>
```

Chapter 11

Getting Started Developing SDK Administrator Application

Objectives

The *PowerScribe® SDK* provides you with a standard *SDK Administrator* application, but it also provides you with tools to create your own custom *SDK Administrator* program. The first section of this chapter summarizes the features available in the *SDK Administrator* program provided:

- [SDK Administrator Provided vs. Custom Administrator Application](#)

The remainder of this chapter uses JavaScript to present fundamental steps to begin developing an *SDK Administrator* application:

- [Instantiating the PSAdminSDK Object](#)
- [Initializing Administrator Application](#)
- [Configuring Administrator Application to Listen to Events](#)
- [Releasing Event Maps Before Exiting](#)
- [Logging Out of Application](#)
- [Code Summary](#)

SDK Administrator Provided vs. Custom Administrator Application

Before you bother to develop your own *SDK Administrator* application, you should become acquainted with the *SDK Administrator* that Nuance provides with the product. If you then decide you would prefer to have a custom application, you can proceed to develop one.

Running the SDK Administrator

To run the *SDK Administrator*, open a web browser and enter the following for the URL, using the name of your server: <http://<servername>/pscribesdk/admin>

Understanding Features of SDK Administrator

After you log in to the product using the **admin** login and password provided for your site, you then can use it to take several actions under the tabs shown in the illustration below:



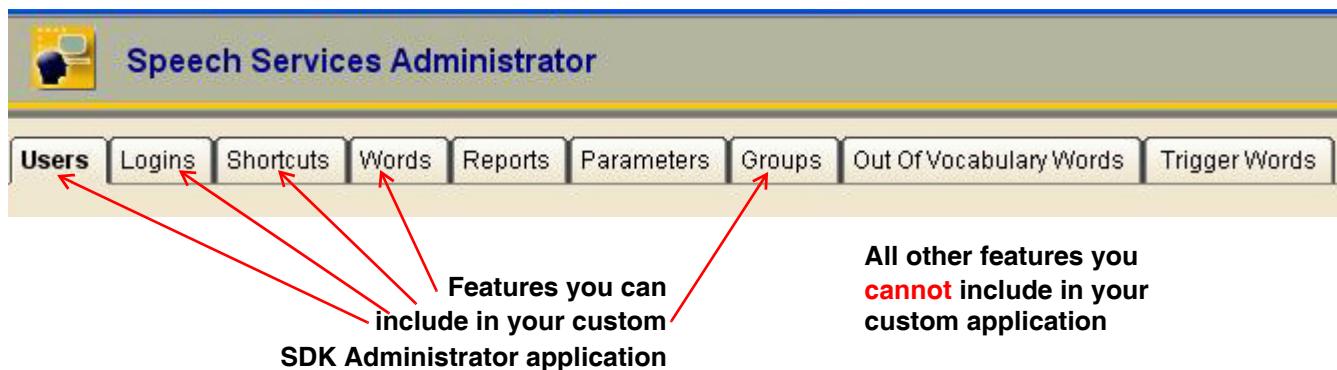
- **Users**
 - Create, edit, disable, and delete users for *PowerScribe SDK* applications
 - Create other administrator level users
 - Set the initial values that apply to a new user by default
- **Logins** — Logout users, especially dangling logged in users
- **Shortcuts and Words**
 - Create global shortcuts or words to be available to all users
 - Create categories and assign them to shortcutsIf you are not sure what a shortcut is or need more information on shortcuts/words, refer to [Chapter 12, Working with Shortcuts, Categories, and Words on page 279](#).
- **Reports**
 - View a list of all reports in process, including who originally dictated the report, who last edited it, when it was originally dictated, the processing state of the report, and whether or not the report is locked
 - Edit the properties of the report, execute recognition on the report, export the report, unlock the report, and delete or purge the report

- **Parameters** — Enable and disable PS parameters, such as those that determine the maximum number of days before audio is purged from the system
- **Groups** — Create groups and add shortcuts to those groups
- **OutOfVocabularyWords** — Find a list of out-of-vocabulary-words that the recognition engine automatically creates (refers to words that are frequently dictated but not in the language model); add these words to the language model or remove them from the list; export the entire list to a file
- **Trigger Words** — Work with custom trigger words—words that you dictate to indicate custom commands follow (an example of a trigger word is *Dictaphone*)

For more information on these features, refer to the **Help** available in the product GUI.

Features You Can Include in Custom Administrator

Your custom *SDK Administrator* application can include only some of the features of the *SDK Administrator* provided. The features you can include are under the five tabs indicated below:



- **Users**
- **Logins**
- **Shortcuts** (Categories are also under this tab)
- **Words**
- **Groups**

Your custom application cannot include the features available under the **Reports**, **Parameters**, **OutOfVocabularyWords**, or **Trigger Words** tabs. To use these capabilities, you must work with the *SDK Administrator* provided.

If you decide that you want to develop your own application, proceed with the remainder of this chapter, then go through the appropriate subsequent chapters for information on:

- Creating users—[Chapter 13](#)
- Creating shortcuts and words—[Chapter 12](#)
- Creating groups—[Chapter 14](#)
- Carrying out advanced administrator tasks—[Chapter 16](#)

Instantiating the PSAdminSDK Object

To begin interacting with the *SDK Administrator Web Server*, your application must instantiate a **PowerscribeSDK** object, the main object in the object model, then instantiate a **PSAdminSDK** object, a parallel object in the model that you use to take administrative level actions. You instantiate the two objects in JavaScript as follows:

```
pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");
objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");
```

Both **PSAdminSDK** and **PowerscribeSDK** objects are required in your *SDK Administrator* application. You need not call the **Initialize()** method of the **PowerscribeSDK** object in this type of application; instantiating the **PowerscribeSDK** object is sufficient.

You cannot use methods and properties of any objects under the **PSAdminSDK** object in the object model until you have initialized the **PSAdminSDK** object and logged in to the *SDK Administrator* application.

Initializing Administrator Application

Now that you have the **PSAdminSDK** object, you initialize the object using its **Initialize()** method. This method has two arguments, the URL to your *SDK Administrator* application on the server and a unique application name that identifies the context for logging in and out:

```
var url = "http://server2/myadminApp/";
objAdmin.Initialize(url, "UserConfigs");
```

You normally take this action in a login function or script, after an administrative level user with access to the application has entered a login and password in a startup dialog box. The function or script can then continue the process of logging in that administrator by calling the **Login()** method of the **PSAdminSDK** object and passing it the login and password:

```
objAdmin.Login(login.text, password.text);
```

Logging Out Dangling Logged In User



Note: The login for a particular user never expires. As a result, if the user exits from the application without logging off, the user remains logged in until your application logs that user out.

In an *SDK Administrator* application, just as in a *PowerScribe SDK* application, if the administrator has shut down the application without first logging out, he or she remains logged in indefinitely and cannot log in again unless the application logs him or her out first. Your application should handle this *dangling logged in user* (error **-2146778148**, the same error code for both the **PowerscribeSDK** and the **PSAdminSDK** objects) by:

- Logging in another administrator user

- Logging out the dangling logged in user
- Logging out the other administrator user
- Relogging in the no-longer-dangling user

Logging Out Dangling Logged In User Programmatically

To log out a dangling logged in user inside your application:¹

1. To log in another administrator user, you log in again, this time passing another login name to the **Login()** method of the **PSAdminSDK** object:

```
objAdmin.Login("admin", admPassword);
```
2. To then log off the dangling logged in administrator user, you would call the **LogoffUser()** method of the **PSAdminSDK** object.

```
objAdmin.LogoffUser("userJ", strPasswd);
```
3. To log out the second administrator user, who is now the only user logged in, you would call **Logoff()** method of the **PSAdminSDK** object.

```
objAdmin.Logoff();
```
4. Finally, now that you have successfully logged out the no-longer-dangling user, you can log that user back in using the **Login()** method of the **PSAdminSDK** object:

```
objAdmin.Login("userJ", strPasswd);
```
5. On successful login, you might open the main application window:

```
window.open("main.htm", "_self", "height=100, width=50, status=yes", true);
```

The entire block of code might look like this:

```
try
{
    objAdmin.Login(user.text, passwd.text);
}
catch(error)
{
    if (error == -2146778148)
    {
        objAdmin.login("admin", "");
        objAdmin.LogoffUser(user.text, passwd.text);
        objAdmin.Logoff();
        objAdmin.Login(user.text, passwd.text, "UserConfigs");
        window.open("main.htm", "_self", "height=100, width=50, status=yes",
                    true);
    }
}
```

1. You can log out a dangling logged in user by logging in as the original **admin** user to the *SDK Administrator* application provided and, from there, logging the dangling user out. Although this action quickly resolves the situation, manual intervention of this kind is not as efficient as programmatic control of the situation.

```
    else
    {
        // Handle Other Errors
    }
}
```



Note: Note that the error code for **No user is logged in** is the same for the **LogoffUser()** methods of both the **PowerscribeSDK** and the **PSAdminSDK** objects: -2146798283.

Configuring Administrator Application to Listen to Events

After you have initialized the *SDK Administrator* application, first determine what kinds of actions you plan to take with your application. The only actions that result in **PSAdminSDK** object events are exporting and importing user voice models. If you plan to take those types of actions with this application, you need to set up the application to listen to events.

In JavaScript, you set up an *SDK Administrator* application to listen to events much the way you set up a *PowerScribe SDK* application to listen to events, as covered in [Chapter 3](#), under [Configuring the Application to Listen to Events on page 27](#). Some of the differences for an *SDK Administrator* application are delineated below.

Instantiating EventMapper for PSAdminSDK Objects

In JavaScript applications, you map events with the **Map Creation** facility of the product. To use **Map Creation**, you must instantiate an **EventMapper** object to catch events that **PSAdminSDK** objects trigger. To indicate that the **EventMapper** is catching events fired by objects in the **PSAdminSDK** branch of the object model, let's name it **AdminSink**:

```
AdminSink = new ActiveXObject("PSAdminSDK.EventMapper");
```

There are only a few events that objects in the **PSAdminSDK** branch of the object model fire. Two of them are shown below, both events that the **User** object fires when the application has exported voice models to be copied or moved to another system or has imported voice models from another system. If you plan to work with those events, you map the event handlers to the events using the **EventMapper** object, just as you did for **PowerscribeSDK** object events:

```
AdminSink.EndExport = HandleEndExport;
AdminSink.EndImport = HandleEndImport;
```

Once you have the events mapped, you can call the **Advise()** method of the **EventMapper** object and pass it the **PSAdminSDK** object:

```
AdminSink.Advise(objAdmin);
```

Once you call the **Advise()** method, the application starts receiving events in response to the associated methods.

Creating Event Delegates in C#

In C#, you create the event delegates using the **PSAdminSDK** object. For instance, if the object is named **m_adminSDK**, you create the **EndImport** and **EndExport** delegates as follows:

```
m_adminSDK.EndExport += new PSAdminSDKLib._IPSAdminSDKEvents  
    _EndExportHandler(Handle_EndExport);  
  
m_adminSDK.EndImport += new PSAdminSDKLib._IPSAdminSDKEvents  
    _EndImportHandler(Handle_EndImport);
```

When you create the actual handler for the **EndExport** event, you pass it the arguments for the file location string, error number, and error message from the **PSAdminSDKLib**:

```
private void Handle_EndExport(string FileLocation, int ErrNum, string ErrMessage)  
{ // Take appropriate action }
```



*Note: For samples showing how to set up events in Visual Basic 6.0, Visual Basic .NET, C++, and Java, refer to the sample code in the **Samples** directory under the root on the CD.*

Releasing Event Maps Before Exiting

Before you log off the administrator user of the application, you release the event maps by calling **Unadvise()** on the **EventMapper** object and passing it the name of the object whose events you want to release:

```
AdminSink.Unadvise(objUser);
```

You usually call **Unadvise()** in your user defined shutdown procedure, inside a **Try/Catch** block:

```
function cleanup()  
{  
    try  
    {  
        if (AdminSink)  
            AdminSink.Unadvise();  
    }  
    catch(error)  
    {  
        alert("Cleanup " + error.description);  
    }  
}
```

Logging Out of Application

You usually log out the logged in user as part of a sequence of actions you take during a shutdown of the application. To log out the user, you use the **Logoff()** method instead of the **LogoffUser()** method (discussed earlier under *Initializing Administrator Application on page 272*).

Code Summary

```
<HTML>
<HEAD>

<TITLE>SDK Administrator</TITLE>
<SCRIPT type="text/javascript">

function InitializeAdminApp()
{
    try
    {
        // Instantiate Admin SDK Object
        pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");
        objAdmin = pscribeSDK.AdminSDK;
        objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");

        // Initialize Admin SDK SDK Server
        var url = "http://server2/adminsdk/";
        objAdmin.Initialize(url, "UserConfigs");

        // Create EventMapper Object for Admin SDK Object
        AdminSink = new ActiveXObject("PSAdminSDK.EventMapper");

        // Map Handlers for Each Event to SDK EventMapper Object
        AdminSink.EndExport = HandleEndExport;
        AdminSink.EndImport = HandleEndImport;
        SDKsink.Advise(objAdmin);
    }
    catch(error)
    {
        alert("InitializeAdmin: " + error.number + error.description);
    }
    finally
    {
        DoLogin()
    }
}
```

```
function DoLogin()
{
    try
    {
        objAdmin.Login(user.text, passwd.text);
        var lastUsername = user.text;
        var lastPassword = passwd.text;
    }
    catch(error)
    {
        // If the user is already logged in (is dangling logged in user)
        if (error == -2146778148)
        {
            objAdmin.login(alt_user.text, alt_passwd.text);
            objAdmin.LogoffUser(lastUsername, lastPassword, "UserConfigs");
            objAdmin.Logoff();
            objAdmin.Login(lastUsername, lastPassword);
        }
        else
        {
            // Handle Other Errors
        }
    }
    finally
    {
        window.open("main.htm", "_self", "height=100, width=50,
                    status=yes", true);
    }
}

...

```

```
function DoLogoff()
{
    try
    {
        objAdmin.Logoff();
        Cleanup();
    }
    catch(error)
    {
        alert(error.description);
    }
}
```

```
function cleanup()
{
    try
```

```
{  
    if (AdminSink)  
    {  
        AdminSink.Unadvise();  
    }  
}  
catch(error)  
{  
    alert("Cleanup Error: " + error.description);  
}  
}  
  
</SCRIPT>  
</HEAD>  
  
<BODY bgcolor="#fffff" language="javascript" onload="InitializeAdminApp()"  
onunload="DoLogoff()">  
  
<form>  
...  
    <input language="javascript" type="button" name="btnLogin"  
          value="Login" onClick="DoLogin()"><br>  
    <input language="javascript" type="button" name="btnLogout"  
          value="Logout" onClick="DoLogoff()"><br>  
...  
</form>  
...  
</BODY>  
</HTML>
```

Working with Shortcuts, Categories, and Words

Objectives

In this chapter, you learn how to facilitate creating custom shortcuts and words for dictation:

Shortcuts:

- [Understanding Shortcuts and How They Are Used](#)
- [Creating and Editing Shortcuts / Deploying Shortcuts](#)
- [Creating Categories for Shortcuts](#)
- [Selecting, Inserting and Auto Expanding Shortcuts in PlayerEditor—JavaScript](#)
- [Auto Expanding Shortcuts After Drag and Drop into PlayerEditor—C#](#)

Words:

- [Understanding Words / Creating and Editing Words / Deploying Words](#)

Training Shortcuts or Words:

- [Overview of Training Shortcuts, Words, or Punctuation Marks](#)
- [Preparing to Train Voice Shortcuts, Words, or Punctuation Marks](#)
- [Retrieving Pronunciation While Training Voice Shortcuts, Words, or Punctuation Marks](#)
- [Adding Trained Shortcut to Collection, Loading into Memory](#)
- [Adding Trained Word or Punctuation Mark to Collection, Loading into Memory](#)

Understanding Shortcuts and How They Are Used

To save time in dictation, people can deploy *shortcuts*. Shortcuts are single words or short phrases that the dictation application replaces with a longer phrase, a process called *expanding the shortcut*. For example, when physicians dictate reports, they often vocalize the phrase **Normal Chest** rather than stating a longer phrase **This patient's chest is normal**.

Types of Shortcuts: Voice and Text

Although shortcuts can be dictated words or phrases (called *voice* shortcuts), they can also be typed words or groups of letters (called *text* shortcuts). For instance, instead of typing out a paragraph of standard input, you could type a series of letters like **S3 LMP** that the application would automatically expand into a paragraph or complete section of the report.

Shortcut Categories and Groups

Shortcuts can be organized by user-defined **Categories**, created by an administrator level user. You can also associate related or similar shortcuts with each other by organizing them into user-defined **Groups**. Each shortcut can belong to only a single **Category**. By contrast, **Groups** can overlap, so a single shortcut can belong to more than one **Group**.

For information, refer to [Assigning the Shortcut a Category on page 286](#) and/or [Chapter 14, Creating Groups in Your Application on page 371](#).

How SDK Deploys Shortcuts and Words

When you create shortcuts, you add them to a collection. Any shortcut in that collection can be for the currently logged in user only or globally available to all users.

The shortcuts are in the collection, but do not affect the *PowerScribe SDK* application until the application loads the shortcuts using the **LoadShortcuts()** method. Your application might call this method once during a session, perhaps immediately after a user creates a set of shortcuts or in response to a user request to load all new shortcuts. It might also call this method during initialization of the application to ensure previously existing shortcuts are available to the application.

The application loads shortcuts from the collection into the in-memory language model. **LoadShortcuts()** always loads shortcuts created by or for the currently logged in user, and either loads or does not load shortcuts available to all users, based on arguments passed to the method; it ignores shortcuts created exclusively for other users.

The same scenario applies to **Words**. After you add the **Word** to the **Words** collection, they do not affect the *PowerScribe SDK* application until it later calls **LoadWords()**.

Overview of Creating and Using Shortcuts

Your application needs to handle multiple phases of action to include shortcuts:

- Create Shortcuts—Develop a way for the user to create and edit shortcuts in your *SDK Administrator* application or using *Administrator* functionality within your *PowerScribe SDK* application, optionally including creating categories and assigning them to shortcuts
- Deploy Shortcuts—Load shortcuts into the in-memory language model that your *PowerScribe SDK* application is using and enable the shortcuts to activate them



Caution: *If you do not load and enable the shortcuts after a user or administrator creates them, they are not available for use. That is why you usually load and enable shortcuts immediately after the application initializes and the user has successfully logged in (see [Taking Startup Actions after Successful Login on page 36](#)). If you create any new shortcuts after the application is already running, you must load those shortcuts before trying to use them (see [Loading Shortcuts Collection into the In-Memory Language Model on page 289](#)). You might also load a single new shortcut (see [Loading Single Shortcut into the In-Memory Language Model on page 289](#)). Shortcuts must also be enabled to be available, as explained under [Disabling and Re-Enabling Shortcuts on page 290](#).*

In addition, your application can optionally provide users the ability to:

- Create categories and assign each shortcut a category
- Train shortcuts (see [Overview of Training Shortcuts, Words, or Punctuation Marks on page 301](#))
- Modify existing shortcuts (during the creation stage or in a subsequent editing stage)
- Enable or disable the shortcuts during the deployment process
- Check whether or not the shortcuts are enabled before executing code that deploys them

Creating and Editing Shortcuts

Steps to Creating Shortcuts

You take several steps to create shortcuts for users of your application:

- Create a **PSAdminSDK** object for later use in creating and working with **Shortcuts** objects (for a brief introduction to the **PSAdminSDK** object, refer to [Types of Applications You Can Develop on page 3](#))
- Create a **Shortcuts** collection to contain the shortcuts
- Add shortcuts to the collection

Later you load the shortcuts before you can use them.

Creating the PSAdminSDK Object

In [Chapter 1](#) you were introduced to the types of applications you can develop with *SDK*:

- *PowerScribe SDK* applications, for dictation users, using the **PowerscribeSDK** object
- *SDK Administrator* applications, for administrators, using the **PSAdminSDK** object

For a high-level overview of the types of applications you can create with *SDK*, refer to [Chapter 1](#), under [Types of Applications You Can Develop on page 3](#).

You have your application manage shortcuts and words through the *SDK Administrator* portion of the *PowerScribe SDK* product. However, you do not have to create an entire *SDK Administrator* application to deploy the functionality—you can instead create a **PSAdminSDK** object inside your *PowerScribe SDK* application and utilize shortcut and word functionality there.

(Alternatively, you could create a separate *SDK Administrator* application whose purpose is to facilitate the act of creating shortcuts and custom words and phrases that your *PowerScribe SDK* application could later load into the local in-memory language model on initialization.)

When you want to use the *SDK Administrator* functionality in a *PowerScribe SDK* application for dictating reports, you create a **PSAdminSDK** object using the **AdminSDK** property of the **PowerscribeSDK** object:

```
objAdmin = pscriveSDK.AdminSDK;
```

After you create the object, the *PowerScribe SDK* application can utilize it to allow dictating users to add shortcuts and/or words to their own local versions of the currently deployed language model and later delete any shortcuts or words they added. It can also facilitate use of the shortcuts and/or words that have been added.

Creating a Shortcuts Collection Object

To create a **Shortcuts** collection object, you call the **GetShortcuts()** method of the **PSAdminSDK** object and pass it several arguments:

- *enType*—Type of shortcuts the collection should contain, **psShortcutTypeVoice** (dictated), **psShortcutTypeText** (typed), **psShortcutTypeAll** (both), **psShortcutTypeUser** (user-defined), or **psShortcutTypeTemplate** (only templates for formatting). Only voice and text shortcuts are expanded automatically by the *PowerScribe SDK*. Others require you write custom code.

Values for enType Argument

Type	Value	Explanation
psShortcutTypeAll	0	Includes both spoken and typed shortcuts from the collection.
psShortcutTypeVoice	1	Includes only spoken shortcuts from the collection.

Type	Value	Explanation
psShortcutTypeText	2	Includes only typed shortcuts from the collection.
psShortcutTypeUser	4	Includes only user-defined shortcuts from the collection.
psShortcutTypeTemplate	5	Includes only templates for formatting.

- *enIncludeGlobals*—Whether or not to include global shortcuts (available to all users)—to include all shortcuts, exclude or include global shortcuts, or get only global shortcuts.

Values for the **enIncludeGlobals** Argument

IncludeGlobals	Value	Explanation
psGlobalTypeAll	0	Includes both all global shortcuts (available to all users) and all individual user shortcuts for all users.
psGlobalTypeExclude	1	Includes only individual shortcuts for the logged in user, not those created strictly for other individual users and not global shortcuts that are available to all users.
psGlobalTypeInclude	2	Includes global shortcuts in the collection alongside individual shortcuts for the logged in user.
psGlobalTypeOnly	3	Creates a collection of global shortcuts only.

- *filterByUser*—If you want to include shortcuts that belong to a particular user, a string containing the login name of that user. If this argument is empty, it becomes the logged in user by default.
- *filterByCategory*—If you want to filter the shortcuts returned by their category, indicate the type of filter information to use, **psFilterTypeByAll** to include shortcuts from all categories, **psFilterTypeName** to include shortcuts from a category with a particular name, or **psFilterTypeNone** to include shortcuts that have no category.

Values for the **filterByCategory** Argument

IncludeGlobals	Value	Explanation
psFilterTypeByAll	0	Includes all shortcuts, regardless of their category.
psFilterTypeName	1	Includes only those shortcuts that belong to a particular category, named in the next argument.
psFilterTypeNone	2	Include shortcuts that have no category.

- *bsCategoryName*—If you pass **psFilterTypeName** for the previous argument, this argument should be a string containing the name of the category; otherwise, pass NULL.
- *filterByGroup*—If you want to filter the shortcuts returned by group, indicate the type of filter information to use, **psFilterTypeByAll** to include shortcuts from all groups, **psFilterTypeName** to include shortcuts from a group with a particular name,

psFilterTypeGlobalName to include all shortcuts that belong to groups available to all users, or **psFilterTypeNone** to include all shortcuts that do not belong to any groups.

Values for the **filterByGroup** Argument

IncludeGlobals	Value	Explanation
psFilterTypeByAll	0	Includes all shortcuts, regardless of their group, that belong to groups.
psFilterTypeName	1	Includes only those shortcuts that belong to a particular group, named in the next argument.
psFilterTypeGlobalName	2	Includes shortcuts that belong to global groups, groups that are available to all users.
psFilterTypeNone	3	Include shortcuts that do not belong to any groups.

- *bsGroupName*—If you pass **psFilterTypeName** for the previous argument, this argument should be a string containing the name of the group; otherwise, it should be NULL.
- *filterByShortName*—If you want to filter the shortcuts by the *short* name of a **Shortcut**, pass that *short* name for this argument.

The simplest form of this method call might look as follows, using the first two arguments only, because they are the only ones required:

```
objShctsColl = objAdmin.GetShortcuts(psShortcutTypeVoice, psGlobalTypeAll);
```

When the user of the *PowerScribe SDK* application retrieves the shortcuts (perhaps by pressing a button that calls the **GetShortcuts()** method), if your application passes the **psGlobalTypeExclude** or **psGlobalTypeInclude** values for the *enIncludeGlobals* argument, the method includes in the returned **Shortcuts** collection the individual shortcuts that the logged in user created or that were assigned his or her login ID as the *author*, and either excludes or includes shortcuts created for all users (global shortcuts).

On the first call of **GetShortcuts()**, before you have any shortcuts, the object returned is an empty collection container. Once the collection has been populated, a call to **GetShortcuts()** retrieves the types of shortcuts you indicate (using filtering arguments) from the database and returns them in a **Shortcuts** collection.

Adding Shortcuts to the Collection

To maximize the potential of shortcuts, when you create one, you indicate its type—voice, text, user-defined or template and whether you want only the logged in user to use the shortcut or whether it is *Global* to all users.

Your application can use this information to let the person doing the dictating choose the shortcuts needed based on the type and author (user who created them) before loading them.

To add shortcuts to the collection returned by the **GetShortcuts()** method, you use the **Add()** method of the **Shortcuts** object and pass it several pieces of information about the shortcut:

- *Type*—Either voice, text, or all shortcut types (see the [Values for enType Argument on page 282](#)).
- *ShortText*—String of actual shortcut text that is spoken or typed.
- *LongText*—String of text to replace the shortcut after expanding it. It can be either plain text or structured text using an XML template to define the format.
- You can build the *LongText* of a plain text shortcut in any plain text editor. If you use an editor that applies formats, those formats are stripped out when you pass the text to this method.
- You can build a structured XML template that creates formatted shortcuts, with sections, paragraphs, and specified styles, as detailed in [Chapter 15, on Creating Templates for Structured Reports and Shortcuts on page 385](#).
- To pass the *LongText* for an existing shortcut, you can use its **Shortcut** object to retrieve the **LongText** property of the shortcut.
- *Transcription*—If you have trained the shortcut, the string containing the transcribed audio pronunciation of the shortcut returned by the **GetTranscription()** method of the **PhoneticTranscriber** object (see [Retrieving Pronunciation While Training Voice Shortcuts, Words, or Punctuation Marks on page 303](#)). If you have not trained the shortcut, you do not need to pass this argument, because it is optional and you can pass an empty string for it.

To pass the phonetic transcription for an existing shortcut, you can use its **Shortcut** object to retrieve the **Transcription** property of the shortcut and pass that for this argument.

- *Global*—**True** adds the shortcut to the global collection available to all users, rather than to the collection available for only the currently logged in user.
- *AddLocal*—**False** ensures the shortcut is saved to the database.



Caution: If you set the *AddLocal* argument to **True**, the shortcut is stored only locally and only in the current instance of the collection—never saved to the database.

For example, to add a voice shortcut of **normal foot** to the individual collection for the currently logged in user, but not to the global collection for all users, you would make this call:

```
var psShortcutTypeVoice = 1;  
  
objShctsColl.Add(psShortcutTypeVoice, "normal foot", "This patient's  
foot is normal.", "", False, False);
```

To add the same shortcut to the global collection and not the individual user shortcut collection, you would make this call:

```
objShctsColl.Add(psShortcutTypeVoice, "normal foot", "This patient's  
foot is normal.", "", True, False);
```

Modifying Short Phrase and Expanded Text of Existing Shortcuts

After you have established a **Shortcuts** collection, you can retrieve a single shortcut from the collection in a **Shortcut** (singular) object using the **Item()** method:

```
objShcut = objShctsColl.Item(1);
```

You can then modify the short phrase of the shortcut by changing the setting of its **ShortText** property:

```
for (i = 1; i <= objShctsColl.Count; i++)
{
    objShcut = objShctsColl.Item(i);
    if (objShcut.ShortText == txtOldShortText.text)
    {
        objShcut.ShortText = txtNewShortText.text;
    }
}
objShcut.Update();
```

You can change the expanded text that results from using the shortcut by changing the setting of its **LongText** property in the same way.

You can set also some other **Shortcut** object properties—**CategoryName**, **Transcription**, and **Type**. After you have changed the setting of one or more **Shortcut** object properties, to have the new settings take effect, call the **Update()** method of the **Shortcut** object:

```
objShcut.Update();
```

Assigning the Shortcut a Category

Before you can assign a shortcut a particular category, you must first create **Category** objects as explained in [Creating a Collection of Categories on page 291](#).

Once you have a collection of **Categories** with some **Category** objects in it, to assign a category to a shortcut, you set the **CategoryName** property of the **Shortcut** object to the value of the **Name** property for an existing **Category** object or to a string that you know is the name of a **Category** object:

```
objShcut.CategoryName = objCategory.Name;
objShcut.CategoryName = "Chest";
objShcut.Update();
```

Be sure to call the **Update()** method to have all new property settings take effect:

Retrieving Author, Global Status, and LastDateModified of Shortcut

Once you have a **Shortcuts** collection, you can retrieve information about each shortcut in the collection by looping through the collection and determining the values of its **Shortcut** object properties—**Author**, **Type**, **ShortText**, **LongText**, **Transcription**, and/or **Global** (True if the shortcut is available to all users, False if not). For instance, to retrieve the information about all voice shortcuts with an author **jjones**, you could test the **Type** and **Author** properties, then for those that match, print the **ShortText** and **LongText** properties:

```
for (i = 1; i <= objShctsColl.Count; i++)
{
    objShcut = objShctsColl.Item(i);
    if ((objShcut.Type == psShcutTypeVoice) && (objShcut.Author == "jjones"))
    {
        document.write(objShcut.ShortText+ ":" + (objShcut.LongText));
    }
}
objShcut.Update();
```

You can return the values of all **Shortcut** object properties. The read-only properties of the **Shortcut** object that you might want to retrieve the values of include the **Author**, **Global**, and **LastModifiedDate** properties.

To change the author of a shortcut, you must delete the shortcut and add a new shortcut to the collection to replace it, passing new value for the read-only **Author** property.

Removing Shortcuts from the Collection

To remove a shortcut from the collection, you first retrieve the particular shortcut to remove from the collection in a single **Shortcut** object:

```
objShctToRemove = objShctsColl.Item(20);
```

Then call the **Remove()** method of the **Shortcuts** collection object and pass it the **Shortcut** object for the shortcut to remove:

```
objShctsColl.Remove(objShctToRemove);
```

Creating and Merging Structured Shortcuts

A shortcut can expand into as much text as needed; for instance, one shortcut could expand into one or more sections of a structured report, each containing formatted paragraphs of text. To have the shortcut expand this way, you create a structured shortcut using an XML template. For more information on creating and merging these shortcuts into a report, refer to [Chapter 15](#), on [Creating Templates for Structured Reports and Shortcuts on page 385](#).

Deploying Shortcuts

After you have added shortcuts to the **Shortcuts** collection, you need to make the shortcuts available to be used. You might take that action in one of these ways:

- Have the user deploy one application to create the shortcuts (an *SDK Administrator* or hybrid *PowerScribe SDK* application) and another to dictate the reports using the shortcuts
- Have the user execute the same application to both create the shortcuts and use them, but load the shortcuts into memory after the user creates them

For either of these approaches, you take the same steps to deploy the shortcuts.

Steps to Deploying Shortcuts

To use the shortcuts you created, your application needs to have loaded them into memory; you can load them into memory by taking these steps:

- Load the shortcuts collection into the local in-memory language model (see [Loading Shortcuts Collection into the In-Memory Language Model on page 289](#))
- Optionally, load a single additional shortcut into the in-memory language model (see [Loading Single Shortcut into the In-Memory Language Model on page 289](#))
- Before using the shortcuts, check to see if the shortcuts are enabled; if they have been disabled, enable the shortcuts (see [Disabling and Re-Enabling Shortcuts on page 290](#))



Note: Shortcuts are enabled by default, and as long as no one has disabled them, you need not enable them to use them.

Loading Shortcuts Collection into the In-Memory Language Model

To load shortcuts into the in-memory language model of the machine where the user is running the *PowerScribe SDK* application, you call the **LoadShortcuts()** method of the **PowerscribeSDK** object and pass it **True** to include global shortcuts created for all users or pass it **False** to include only the shortcuts for the individual user who has just logged in:

```
pscribeSDK.LoadShortcuts(true);
```

This method call loads all shortcuts from all collections its find in the database. It determines the particular shortcuts to load based on who is logged in and on whether or not to include the global shortcuts available to all users along with those for that individual user.

To load the shortcuts of a particular user, you can pass this method a second argument that is the login name of a particular user whose shortcuts should be loaded and a third argument that indicates the type of shortcuts, either voice, text, or both (see the [Values for enType Argument for ShortcutTypes](#) you can pass to **LoadShortcuts()** as the third argument):

```
pscribeSDK.LoadShortcuts(false, "jsmith", psShortcutTypeVoice);
```

If you have not created shortcuts, calling this method does not raise an exception; instead, the *SDK* ignores the method call.

Loading Single Shortcut into the In-Memory Language Model

After loading the entire collection of shortcuts, you might want to load a single shortcut into the in-memory language model, perhaps immediately after it has been created, using the **LoadShortcut()** method of the **PowerscribeSDK** object, which is more efficient than calling **LoadShortcuts()** for a single shortcut. You might take this action when a user has taken or would like to take one of these actions while running the application:

- Has added a new shortcut
- Has trained or retrained a shortcut
- Would like to modify the *LongText* that results from expanding the shortcut

You pass the **LoadShortcut()** method details about the shortcut, similar to the details you passed to the **Add()** method of the **Shortcuts** object when you were adding a shortcut to the collection. Among the details you pass to this method is the *LongText* of the shortcut, which can be revised by passing a new string in the *LongText* argument. Here is a complete list of the arguments this method takes:

- *ShortText*—String of actual shortcut text.

- *LongText*—String of text to replace the shortcut after expanding it. It can be either plain text or structured text using an XML template to define the format.
You can build the *LongText* of a plain text shortcut in any plain text editor. If you use an editor that applies formats, those formats are stripped out when you pass the text to this method.
You can build a structured XML template that creates formatted shortcuts, with sections, paragraphs, and specified styles, as detailed in [Chapter 15](#), on [Creating Templates for Structured Reports and Shortcuts on page 385](#).
- *Type*—Either voice, text, or all shortcut types (see the [Values for enType Argument on page 282](#)).
- *Transcription*—String containing the transcribed audio pronunciation of the shortcut returned by the **GetTranscription()** method of the **PhoneticTranscriber** object (see [Retrieving Pronunciation While Training Voice Shortcuts, Words, or Punctuation Marks on page 303](#)). This argument is optional and you usually pass an empty string for it.
- To pass the phonetic transcription for an existing shortcut, you use its **Shortcut** object to retrieve the **Transcription** property of the shortcut and pass that for this argument.

For example, to modify the *LongText* produced by the **normal chest** shortcut, you can load the shortcut into the in-memory language model with a new string for the *LongText* argument, as follows:

```
var psShortcutTypeVoice = 1;
pscribeSDK.LoadShortcut("normal chest", "This patient's series of chest
x-rays indicate a normal chest.", psShortcutTypeVoice, "");
```

Disabling and Re-Enabling Shortcuts

After you load shortcuts into the in-memory language model, they are automatically enabled and ready to use. Under certain circumstances, however, you might need to disable the shortcuts and, once they are disabled, to use them again you must re-enable them. To either enable or disable shortcuts, you use **EnableShortcuts()** method of the **PowerscribeSDK** object.

When you call **EnableShortcuts()**, you pass it the shortcut type for the first argument (voice is **1**, text is **2**, and both are **3**, the same as in the [Values for enType Argument](#) table). You can call this method to either enable the shortcuts or disable them, so you also pass it a second parameter of either **True** to enable them or **False** to disable them.

To disable voice shortcuts, pass **1** or the equivalent constant, **psShortcutTypeVoice**, for the first argument and **False** for the second:

```
var psShortcutTypeVoice = 1;
pscribeSDK.EnableShortcuts(psShortcutTypeVoice, false);
```

If you want to disable both voice (dictated) and text (typed) shortcuts, you can skip the first argument, and the method chooses **psShortcutTypeAll** (**3**) by default:

```
pscribeSDK.EnableShortcuts(false);
```

When you are ready to re-enable the shortcuts, you call the method again and pass it **True**:

```
pscribeSDK.EnableShortcuts(true);
```

If you have not created shortcuts, calling the method does not raise an exception; instead, the **SDK** ignores the method call.

Verifying That Shortcuts Are Enabled

To verify that shortcuts are enabled before proceeding to activate code that should use them, you call the **IsShortcutsEnabled()** method of the **PowerscribeSDK** object and pass it the type of shortcuts to check—voice, text, or both—from the [Values for enType Argument](#) table:

```
if (pscribeSDK.IsShortcutsEnabled(psShortcutTypeText) != True)
{
    pscribeSDK.EnableShortcuts(psShorcutTypeText, True);
}
```

Creating Categories for Shortcuts

Before you can have categories for **Shortcut** objects, you need to create **Category** objects.

Creating a Collection of Categories

To create **Category** objects, you first create a **Categories** collection by calling the **GetCategories()** method of the **PSAdminSDK** object:

```
var objCatsCollection;
objCatsCollection = objAdmin.GetCategories();
```

The method returns a **Categories** collection that is initially empty. This is the only **Categories** collection in the entire *PowerScribe SDK* system, as all category objects are available to apply to any and all **Shortcut** objects across the entire system. **Categories** are not applicable to particular users or to all users, but to particular shortcuts that can belong to any individual user or all users.

Adding a Category to the Collection

To add a **Category** object to the **Categories** collection, you call the **Add()** method of the collection object and pass it first a string containing the name of the category, then a string containing the description of the category:

```
objCatsCollection.Add("Chest", "Chest-related shortcuts.");
objCatsCollection.Add("Foot", "Foot-related shortcuts.");
```

When you create the category, it is required to have a name and its name must be unique across the entire *PowerScribe SDK* system; the method returns an error if the name is not unique. The name can use alphabetic and numeric characters but no special characters, such as question marks or exclamation points.

Removing a Category from the Collection

To remove a category from the **Categories** collection, you first loop through the collection until you find the particular **Category** object whose **Name** property matches the name of the category to remove:

```
for (var i = objCatsCollection.count; i >= 1; i--)  
{  
    if objCatsCollection.Item(i).Name == txtCategoryToRemove.value  
    {  
        objCategoryToRemove = objCatsCollection.Item(i);  
    }  
}
```

You then call the **Remove()** method of the **Categories** object and pass it the object for the **Category** to remove, then **False** to remove only the one **Category** from the collection:

```
    objCatsCollection.Remove(objCategoryToRemove, False);  
}  
}
```

To remove all categories from the collection, pass **True** for the second argument.

Assigning a Category to a Shortcut

Once you have a collection of **Categories**, you can assign a category to a shortcut by setting the **CategoryName** property of the **Shortcut** object to the value of the **Name** property for an existing **Category** object:

```
objShcut.CategoryName = objCategory.Name;
```

Or set the **CategoryName** to a string that you know is the name of a **Category** object:

```
objShcut.CategoryName = "Chest";
```

To have property changes take effect, call the **Update()** method of the **Shortcut** object:

```
objShortcut.Update();
```

Changing the Category Name or Description

After you create a category, you are not forced to keep the original name. Before you can change any of the properties of the **Category**, you retrieve it from the collection by calling the **GetCategories()** method of the **PSAdminSDK** object and passing it a string containing the name of the category:

```
objCategory = objAdmin.GetCategories("Clinical Steps");
```

You can change the name by setting the **Name** property of the **Category** object. You can also modify the **Description** property for the object:

```
objCategory.Name = "Clinical Procedures";  
objCategory.Description = "Step-by-step Clinical Procedures";
```

You then call the **Update()** method of the **Category** object: `objCategory.Update()`

Understanding Words

To expand the knowledge base of the dictation engine, people can add specific *words* to their language models. **Word** objects are single words that the application can recognize when the user dictates them in audio. For example, when physicians dictate reports, they often have an existing language model that understands words in a particular field, such as *Radiology*, but they might also need words that are not already in the language model, such as the names of new diagnoses, treatments, and medications.

A person might also want to improve the general ability of the speech recognition to interpret his or her particular pronunciation of a word. In that case, the user might also want to record audio of speaking that word or phrase and add that audio and the correct transcription of it to his or her own version of the language model.

Overview of Creating and Using Words

Your application needs to handle two phases of action to include words:

- Create Words—Develop a way to create words in your *SDK Administrator* application or using *Administrator* functionality within your *PowerScribe SDK* application
- Deploy Words—Load words into the in-memory language model so that users of the *PowerScribe SDK* application can use them



Caution: If you do not load words after a user or administrator creates them, they are not available for use. That is why you usually load words immediately after the application initializes and the user has successfully logged in (see [Taking Startup Actions after Successful Login on page 36](#)). If you create any new words after the application is already running, you must load those words before trying to use them (see [Loading Words into the In-Memory Language Model on page 299](#)). You might also load a single new word (see [Loading Single Word into the In-Memory Language Model on page 299](#)).

In addition, your application can additionally provide users the ability to:

- Train words during the creation process (see [Overview of Training Shortcuts, Words, or Punctuation Marks on page 301](#))
- Retrieve, modify, or remove a particular word from the collection

Creating and Editing Words

You take several steps to create custom words for users of your application.

Steps to Creating Words

To create and edit words, you take the following major steps:

- Create a **PSAdminSDK** object for later use in creating and working with **Words** objects (see [Creating the PSAdminSDK Object on page 282](#)) (for an introduction to the **PSAdminSDK** object, refer to [Types of Applications You Can Develop on page 3](#))
- Create a **Words** collection to contain the words
- Add the word or words to the collection
- Retrieve information about words in the collection
- Remove a word from the collection



*Note: Whenever the spell checker finds a word it believes is misspelled, it offers the user the option of adding it to the dictionary, which also adds the word to the **Words** collection.*

Creating a Words Collection Object

If you have not already created the **PSAdminSDK** object, create one now:

```
objAdmin = pscribeSDK.AdminSDK
```

To create a **Words** collection object, you call the **GetWords()** method of the **PSAdminSDK** object that you created earlier and pass it several values:

- *WordCategoryCode* of the words the collection should contain (see table).



*Caution: In every language, you must define constants for **WordCategoryCode**, **WordStateCode**, and **WordSourceCode** to use them with **GetWords()**; otherwise, you must use the numeric value for a single code or bitwise numeric value for more than one code.*

Values for WordCategoryCode Argument

WordCategoryCode	Value	WordCategoryCode	Value
psWordCategoryGeneral	1	psWordCategoryBrandNameDrug	32
psWordCategoryName	2	psWordCategoryHeading	64
psWordCategoryPlace	4	psWordCategoryDepartmentName	128
psWordCategoryHospital	8	psWordCategoryRemoved	256
psWordCategoryGenericDrug	16		

- *WordStateCode* of the words the collection should include—These states are approval states such as waiting for approval (either **psWordStateWaitingApproval** or **psWordStateAuto**, both considered the same), already approved, or rejected. To include all words regardless of their approval state, use **psWordStateAll**.

Values for WordStateCode Argument

WordStateCode	Value	WordStateCode	Value
psWordStateAll	0	psWordStateRejected	4
psWordStateWaitingApproval	1	psWordStateAuto	8
psWordStateApproved	2		

- *WordSourceCode* of the word—the spell checker, training, added by an end user, added by an administrator, imported from another workstation or server, or ALL.

Values for WordSourceCode Argument

WordSourceCode	Value	WordSourceCode	Value
psWordSourceAll	0	psWordSourceImport	4
psWordSourceSpell	1	psWordSourceAdmin	8
psWordSourceTrain	2	psWordSourceUser	16

- *IncludeGlobals*—Optional argument for the method; indicates whether to retrieve all words in the system for the collection or only those the logged in user created, or both.

Values for IncludeGlobals Argument

IncludeGlobals	Value	IncludeGlobals	Value
psGlobalTypeAll	0	psGlobalTypeInclude	2
psGlobalTypeExclude	1	psGlobalTypeOnly	3

To create a collection of words for generic drugs that have been approved and have been added by the user, you would call the **GetWords()** method as follows:

```
objWordsColl = objAdmin.GetWords(psWordCategoryGenericDrug,
                                psWordStateApproved, psWordSourceUser);
```

Using numeric bitwise values, you can combine the options for the first three arguments of this method. For instance to retrieve a collection of words that includes all generic and brand name drugs, you would OR the two options by passing **psWordCategoryGenericDrug | psWordCategoryBrandNameDrug** or **48** for the *WordCategoryCode* argument.

```
objWordsColl = objAdmin.GetWords(48, psWordStateApproved, psWordSourceUser);
```

You could also retrieve all the words that are either approved or waiting approval, but not those that have been rejected by passing **3** for the *WordStateCode* argument:

```
objWordsColl = objAdmin.GetWords(48, 3, psWordSourceUser);
```

You could also retrieve all the words that have been added by either the logged in user or the administrator by passing **psWordSourceAdmin** | **psWordSourceUser** or **24** for the *WordSourceCode* argument:

```
objWordsColl = objAdmin.GetWords(48, 3, 24);
```

When the user of the *PowerScribe SDK* application retrieves the words (perhaps by pressing a button that calls the **GetWords()** method), if your application passes the **psGlobalTypeExclude** or **psGlobalTypeInclude** values for the *IncludeGlobals* argument, the method includes in the returned **Words** collection all words that the logged in user created, either excluding or including words created for all users (global words).

On the first call of **GetWords()**, before you have any words, the object returned is an empty collection container. Once the collection has been populated, a call to **GetWords()** retrieves the types of words indicated by the arguments from the database and returns them in a **Words** collection.

Adding Words to the Collection

To add words to the collection returned by the **GetWords()** method, you use the **Add()** method of the **Words** collection object and pass it several arguments about the word:

- *Word*—Text of the word or phrase to add.
- *WordCategoryCode*—See the [Values for WordCategoryCode Argument on page 294](#)). With this method, you can use the constants in C# and Visual Basic, but must define them in JavaScript.
- *Transcription*—String containing the transcribed audio returned by the **GetTranscription()** method of the **PhoneticTranscriber** object (see [Retrieving Pronunciation While Training Voice Shortcuts, Words, or Punctuation Marks on page 303](#)). This argument is optional and you usually pass an empty string for it.

To pass the phonetic transcription for an existing word, you can use its **Word** object to retrieve the **Transcription** property of the word and pass that for this argument.

- *WordStateCode*—Approval status; see the [Values for WordStateCode Argument on page 295](#)). With this method, you can use the constants in C# and Visual Basic, but must define them in JavaScript.
- *WordSourceCode*—Who entered the word (see the [Values for WordSourceCode Argument on page 295](#)). With this method, you can use the constants in C# and Visual Basic, but must define them in JavaScript.
- *Global*—Optional. **True** adds the word to the global collection available to all users; default is **False**.

- *AddLocal*—**False** ensures the word is saved to the database.



Caution: If you set the *AddLocal* argument to **True**, the word is stored only locally and only in the current instance of the collection—never saved to the database.

For example, to add **glycemic index** to the collection of **Words** for the currently logged in user as a general word, added by the user, that has been approved, call **Add()** as follows:

```
var psWordCategoryGeneral = 1;
var psWordStateApproved = 2;
var psWordSourceUser = 16;

objWordsColl.Add("glycemic index", psWordCategoryGeneral, "",
    psWordStateApproved, psWordSourceUser, False, False);
```

To add the same word to the global **Words** collection available to all users and not the individual user **Words** collection, you would make this call:

```
objWordsColl.Add("glycemic index", psWordCategoryGeneral, "",
    psWordStateApproved, psWordSourceUser, True, False);
```

This word is now permanently flagged as global, so when you later retrieve a global **Words** collection, this word is included in the collection.

Retrieving and Modifying Particular Words

After you have established a collection of words, you can retrieve a single word from the collection in a **Word** (singular) object using the **Item()** method:

```
objWord = objWordsColl.Item(1);
```

You can retrieve information about each **Word** in the collection by looping through the collection and determining the values of its **Word** object properties—**Author**, **Word** (text of the word), **Category**, **Transcription**, **ReadOnly**, **State**, **Source**, and/or **Global**. For instance, to retrieve a list of all words added by a user with a login of **jjones**, you'd search the collection for words with an **Author** equal to **jjones**, then print their **Word** property value:

```
for (i = 1; i <= objWordsColl.Count; i++)
{
    objWord = objWordsColl.Item(i);
    if (objWord.Author == "jjones")
    {
        document.write("Word Item #"+ i +": "+ (objWord.Word));
    }
}
```

Although you can return the values of all the **Word** object properties, you can set only the **Transcription**, **State**, **Source**, and **Word** properties.

To modify the **State** of the word, you can set the property:

```
objWord.State = psWordStateApproved;
```

Changing the **Word** property actually changes the word itself:

```
objWord.Word = "St. Vincent's Hospital";
```

After you change the appropriate properties of the **Word** object, the changes occur only in the current login session, unless you call the **Update()** method of the **Word** object to save the changes:

```
objWord.Update();
```

The **ReadOnly** property of the word is a Boolean that indicates whether or not the word is from a read-only collection retrieved with the **PowerscribeSDK.GetLMWords()** method (see [Retrieving a Read-Only Collection of Words on page 300](#)).

To change any of the read-only properties of the word—**Author**, **Category**, **Global**, or **ReadOnly** properties—you can delete the word and add a new word to the collection to replace it. To delete a word, refer to [Removing Words from the Collection on page 298](#).

Removing Words from the Collection

To remove a word from the collection, first retrieve the **Word** object of the particular word:

```
objWordToRemove = objWordsColl.Item(26);
```

Then call the **Remove()** method of the **Words** collection object and pass it the **Word** object for the word to remove:

```
objWordsColl.Remove(objWordToRemove);
```

Deploying Words

After you have added one or more words to the **Words** collection, the application cannot recognize the new word unless you add it to the in-memory language model.

Steps to Deploying Words

To use the words you created, your application needs to take these actions:

- Load the words into the local in-memory language model
- Optionally, load a single additional word into the in-memory language model

Loading Words into the In-Memory Language Model

To load an entire collection of words into the in-memory language model of the machine where the user is running the *PowerScribe SDK* application, during the initialization of your application, you could call the **LoadWords()** method of the **PowerscribeSDK** object. You pass this method **True** to include global words created for all users or **False** to include only the words for the individual user who has just logged in:

```
pscribeSDK.LoadWords(False);
```

Loading Single Word into the In-Memory Language Model

During initialization of the *PowerScribe SDK* application, instead of loading the entire collection of words, you can load a single word using the **LoadWord()** method of the **PowerscribeSDK** object. You might take this action when you want to load a word that you do not want added to the collection, but want loaded under particular conditions. You pass the **LoadWord()** method details about the word, similar to the details you passed to the **Add()** method of the **Words** object when you were adding a word to the collection, using these arguments:

- *Word*—String of actual word text.
- *Category*—Category of the word, either general, a name, a place, an institution, a generic or brand name drug, a heading or department name (see the [Values for WordCategoryCode Argument on page 294](#)).
- *Transcription*—String containing the transcribed audio pronunciation of the word returned by the **GetTranscription()** method of the **PhoneticTranscriber** object (see [Retrieving Pronunciation While Training Voice Shortcuts, Words, or Punctuation Marks on page 303](#)). This argument is optional and you usually pass an empty string for it.
- To pass the phonetic transcription for an existing word, you can use its **Word** object to retrieve the **Transcription** property of the word and pass that for this argument.

For example, to load a word that is not part of the usual **Words** collection into the in-memory language model, you call the method as follows:

```
var psWordCategoryPlace = 4;  
pscribeSDK.LoadWord("Concord Hospital", psWordCategoryPlace, "");
```

Removing Single Word from the In-Memory Language Model

You again use the **LoadWords()** method to either change the category of a word already in the language model or to remove a single word from the in-memory language model. You remove a word by calling the method and passing it **psWordCategoryRemoved** for its category:

```
var psWordCategoryPlace = 4;  
pscribeSDK.LoadWord("Dr. Tarajanamak", psWordCategoryRemoved, "");
```

Retrieving a Read-Only Collection of Words

To display a collection of all **Words** in the language model, you can retrieve a read-only collection from the language model currently residing in memory.

To retrieve the read-only collection of **Words**, you call the **GetLMWords()** method of the **PowerscribeSDK** object and pass it the *GlobalType* (see [Values for IncludeGlobals Argument on page 295](#) and [Values for the enIncludeGlobals Argument on page 283](#)) to indicate the type of words you want in the collection and an empty string for the second argument:

```
var psGlobalTypeAll = 0;  
objRowWordsColl = pscribeSDK.GetLMWords(psGlobalTypeAll, "");
```

The method returns a pointer to a read-only **Words** collection. You can then retrieve values of properties of that **Words** collection and of each individual **Word** object in the collection, but you cannot modify any of those properties.

Retrieving Words Collection Containing Forms of a Punctuation Mark

To retrieve of collection of **Words** that contains all the possible written forms you might dictate for a particular **Word** object that represents a dictated punctuation mark or other non-word **Word** object, you call the **GetLMWords()** method of the **PowerscribeSDK** object, but you pass it a different set of arguments from those you would pass to retrieve a read-only collection of words.

For instance, to create the punctuation mark at the end of a sentence, you might speak *period* while to create a dot between a file name and its extension you might speak *dot*.

To retrieve a collection of all the ways that a particular punctuation mark occurs:

1. For the first argument (*includeGlobals*) of the **GetLMWords()** method, you can pass any value, because as soon as you pass a string that is not empty for the second argument, the first argument is ignored.
2. To retrieve all written forms dictated for *period*, you would call the method as follows:

```
var psGlobalTypeAll = 0;
objROWordsColl = pscribeSDK.GetLMWords(psGlobalTypeAll, ".");
```
3. The collection **GetLMWords()** returns would include the following **Word** strings:
.\\period, .\\dot

Other types of non-word **Word** objects a collection that **GetLMWords()** returns can include:

, \\comma	\\New-Line	\\Next-Paragraph
: \\colon	\\New-Paragraph	\\Paragraph
~; \\semicolon	\\Next-Line	

Overview of Training Shortcuts, Words, or Punctuation Marks

When you create voice shortcuts for **Speech Recognition** users of your application, you supply only the text, because the *Speech Recognition* engine already knows the standard way that the phrase is spoken. However, users who speak with an accent or regional pronunciation might find that recognition of their particular pronunciation tends to be less accurate than they would like. To improve the *Speech Recognition* engine's interpretation of his or her particular pronunciation, such a user can record audio of speaking a shortcut, word, or phrase, then add that audio and the correct transcription of it to his or her language model. This action is referred to as *training the shortcut* or *training the word*, because it trains the *Speech Recognition* engine in a person's speech patterns.

Steps to Training Shortcuts, Words, or Punctuation Marks

To train the speech recognition engine to recognize custom shortcuts:



Caution: Before your user trains shortcuts, if the application is processing a report, it should save the report using the **Save()** method of the **Report** object; otherwise, either the entire report or the most recent changes made to the report will be lost.

- Create a **PhoneticTranscriber** object to record audio for shortcuts, words, or punctuation marks.
- Secure use of the microphone and the **LevelMeter**, if applicable, for the **PhoneticTranscriber**.
- Set the **Word** property of the **PhoneticTranscriber** object to the shortcut, word, or punctuation mark being trained.
- Use the microphone to record the shortcut or word audio.
- Release the microphone.
- Add the shortcut to the collection of shortcuts; add the word or punctuation mark to the collection of words.

Preparing to Train Voice Shortcuts, Words, or Punctuation Marks

In the sections that follow, you start the process of training voice shortcuts or words. After you carry out these actions, you then retrieve the phonetic transcription for the shortcut or word.

Creating PhoneticTranscriber Object

To train voice shortcuts or the sound of words for speech recognition users of your application, you create a **PhoneticTranscriber** object to receive the dictation for the voice shortcut or word using the **PhoneticTranscriber** property of the **PowerscribeSDK** object:

```
objPhoneTrnscriber = pscribeSDK.PhoneticTranscriber;
```

Securing Microphone for PhoneticTranscriber

To secure the **Microphone** object in the application strictly for use by **PhoneticTranscriber** while training shortcuts or words, you call the **SetMicrophone()** method of the **PhoneticTranscriber** object and pass it the **Microphone** object:

```
objPhoneTrnscriber.SetMicrophone(objMicrophone);
```

Securing LevelMeter for PhoneticTranscriber

If your application contains a **LevelMeter** that shows action is taking place while the user speaks into the microphone, you need to secure the **LevelMeter** object in the application strictly for use by **PhoneticTranscriber** while training shortcuts (or words); otherwise, the **LevelMeter** does not react during the training process. To secure the **LevelMeter**, you

call the **SetLevelMeter()** method of the **PhoneticTranscriber** object and pass it the instance of the **LevelMeter** ActiveX control:

```
objPhoneTrnscriber.SetLevelMeter(objLevelMeter);
```

Retrieving Pronunciation While Training Voice Shortcuts, Words, or Punctuation Marks

After you create the **PhoneticTranscriber** object and have secured the microphone and (optionally) **LevelMeter** for the training process, the user then speaks into the microphone to actually record the shortcut or word.

Although the user must speak into the microphone or other audio device when training shortcuts, words, or punctuation marks, you can choose one of two ways to receive the resulting phonetic transcription of the audio from the end user in your application:

- Use your own custom GUI buttons that call **PhoneticTranscriber** methods to start and stop recording, then retrieve the audio.
- Use the microphone's **DICTATE** button to record and **TRANSCRIBE** (or **LEFT**) button to transcribe. Then receive the audio in an event handler argument.

Let's look at how to use each of these techniques, first for a shortcut, then a word, and finally a punctuation mark.

Training Audio of Shortcut or Word Using Custom GUI Buttons

To record/transcribe the audio when training shortcuts or words using custom GUI buttons:

1. To indicate the shortcut or word you want to train, set the **Word** property of the **PhoneticTranscriber** object to a string containing the spelled form of the word:

```
objPhoneTrnscriber.Word = "arteriography";
```

2. To begin training (start recording audio of) the shortcut or word, you call the **Start()** method of the **PhoneticTranscriber** object and then, to end recording, you call the **Stop()** method of the same object. You can call these methods in response to corresponding GUI buttons:

```
objPhoneTrnscriber.Start();
```

```
objPhoneTrnscriber.Stop();
```

3. Immediately after you stop recording with the **PhoneticTranscriber** object's **Stop()** method, you call its **GetTranscription()** method to force recognition of the audio and put the transcribed text into a string called the *phonetic transcriber*:

```
strShctAudioTranscr = objPhoneTrnscriber.GetTranscription();
```

For a voice shortcut, the text spoken to create the *phonetic transcriber* is the equivalent of the *shortText* of the shortcut, rather than the *longText*. For instance for the shortcut **normal chest**, the text dictated is **normal chest** rather than **This patient's chest is normal**.

When you call **GetTranscription()**, you can put it into a **try** block to smoothly handle the types of errors that the user can correct on the spot; for instance, if the error returned indicates no speech was heard or background noise interfered, your application can request that the user redictate the phrase:

```
try
{
    strShctAudioTranscr = objPhoneTrnscriber.GetTranscription()
}
catch recordgError
{
    switch recordgError
    {
        case 0x800a4e35:
            alert("Please speak at a louder volume and re-record
                  the shortcut, word, or phrase.");
        case 0x80042335:
            alert("Background noise or a long pause detected. Please
                  re-record the shortcut, word, or phrase without pausing.");
        ...
    }
}
```

4. When you have finished training (both recording and transcribing) the audio, you are ready to release the microphone and **LevelMeter** from exclusive use of the **PhoneticTranscribe** object (see [Releasing the Microphone and LevelMeter on page 308](#)).

Training Audio of Shortcut or Word Using Microphone Buttons

To record/transcribe the audio when training shortcuts or words using Microphone buttons:

If you want the user to deploy the microphone's **DICTATE** button to record and its **TRANSCRIBE** (or **LEFT**) button to transcribe the audio for a shortcut or word being added to a language model, you need to:

1. To indicate the shortcut or word you want to train, set the **Word** property of the **PhoneticTranscriber** object to a string containing the spelled form of the word:
`objPhoneTrnscriber.Word = "arteriography";`
2. Create an **EventMapper** object and link the **TranscriptionResult** event handler to the **TranscriptionResult** event for that object.
3. Be sure to call the **Advise()** method of the **EventMapper** object so that the application begins receiving the events.
4. Create an event handler function to handle the **TranscriptionResult** event that the **PhoneticTranscriber** object fires. Inside the handler, release the microphone and **LevelMeter** from exclusive use of the **PhoneticTranscribe** object.

Handling the TranscriptionResult Event

The **TranscriptionResult** event handler would start as follows:

```
function HandleTranscriptionResult(strShctAudioTranscr, errNum, errMsg)
{
}
```

The first argument the handler receives is the phonetic transcription string. If an error occurs, the handler also receives the associated error number and error message.

When you have finished recording and transcribing the audio, you are ready to release the microphone (see [Releasing the Microphone and LevelMeter on page 308](#)). In this handler, your application can initiate all actions it needs to take in response to the audio being transcribed—first releasing the microphone, then using the phonetic transcription string to add the shortcut or word to the collection. It can also take any additional action you want.

Training Audio of Punctuation Marks Using Custom GUI Buttons

To record/transcribe the audio when training punctuation marks using custom GUI buttons:

1. To retrieve a collection of the different strings for a punctuation mark, call the **GetLMWords()** method of the PowerscribeSDK object, and pass any value for the first argument, then the symbol for the punctuation mark for the second argument. For instance, to retrieve all written forms dictated for *period*, you would call the method as follows:

```
var psGlobalTypeAll = 0;
objRQWordsColl = pscribeSDK.GetLMWords(psGlobalTypeAll, ".");
```

The collection **GetLMWords()** returns would include the following **Word** strings:

```
.\\period, .\\dot
```

2. To indicate the punctuation mark you want to train, you would set the **Word** property to the symbol followed by a backslash, then the textual name of the symbol; for example:

```
PhoneticTranscriber.Word = ".\\period";
```

3. To begin training (start recording audio of) the punctuation mark, you call the **Start()** method of the **PhoneticTranscriber** object and then, to end recording, you call the **Stop()** method of the same object. You can call these methods in response to corresponding GUI buttons:

```
objPhoneTrnscriber.Start();
```

```
objPhoneTrnscriber.Stop();
```

4. Immediately after you stop recording with the **PhoneticTranscriber** object's **Stop()** method, you call its **GetTranscription()** method to force recognition of the audio and put the transcribed text into a string called the *phonetic transcriber*:

```
strShctAudioTranscr = objPhoneTrnscriber.GetTranscription();
```

When you call **GetTranscription()**, you can put it into a **try** block to smoothly handle the types of errors that the user can correct on the spot; for instance, if the error returned indicates no speech was heard or background noise interfered, your application can request that the user redictate the phrase:

```
try
{
    strShctAudioTranscr = objPhoneTrnscriber.GetTranscription()
}
catch recordgError
{
    switch recordgError
    {
        case 0x800a4e35:
```

```
        alert("Please speak at a louder volume and re-record  
        the shortcut, word, or phrase.");  
  
    case 0x80042335:  
        alert("Background noise or a long pause detected. Please  
        re-record the shortcut, word, or phrase without pausing.");  
        ...  
    }  
}
```

5. When you have finished training (both recording and transcribing) the audio, you are ready to release the microphone and **LevelMeter** from exclusive use of the **PhoneticTranscribe** object (see [Releasing the Microphone and LevelMeter on page 308](#)).

Training Audio of Punctuation Marks Using Microphone Buttons

To record/transcribe the audio when training punctuation marks using microphone buttons:

If you want the user to deploy the microphone's **DICTATE** button to record and its **TRANSCRIBE** (or **LEFT**) button to transcribe the audio for a punctuation mark being added to a language model, you need to:

1. To retrieve a collection of the different strings for a punctuation mark, call the **GetLMWords()** method of the PowerscribeSDK object, and pass any value for the first argument, then the symbol for the punctuation mark for the second argument. For instance, to retrieve all written forms dictated for *period*, you would call the method as follows:

```
var psGlobalTypeAll = 0;  
objRowWordsColl = pscribeSDK.GetLMWords(psGlobalTypeAll, ".");
```

The collection **GetLMWords()** returns would include the following **Word** strings:

.\\period, .\\dot

2. To indicate the punctuation mark you want to train, you would set the **Word** property to the symbol followed by a backslash, then the textual name of the symbol, such as:

```
PhoneticTranscriber.Word = ".\\period";
```
3. Create an **EventMapper** object and link the **TranscriptionResult** event handler to the **TranscriptionResult** event for that object.
4. Be sure to call the **Advise()** method of the **EventMapper** object so that the application begins receiving the events.
5. Create an event handler function to handle the **TranscriptionResult** event that the **PhoneticTranscriber** object fires. Inside the handler, release the microphone and **LevelMeter** from exclusive use of the **PhoneticTranscribe** object.

Handling the TranscriptionResult Event

The **TranscriptionResult** event handler would start as follows:

```
function HandleTranscriptionResult(strShctAudioTranscr, errNum, errMsg)
{
}
```

The first argument the handler receives is the phonetic transcription string. If an error occurs, the handler also receives the associated error number and error message.

When you have finished recording and transcribing the audio, you are ready to release the microphone (see [Releasing the Microphone and LevelMeter on page 308](#)). In this handler, your application can initiate all actions it needs to take in response to the audio being transcribed—first releasing the microphone, then using the phonetic transcription string to add the shortcut or word to the collection. It can also take any additional action you want.

Releasing the Microphone and LevelMeter

After the recording of the shortcut is complete, the application can release the **Microphone** from exclusive use by the **PhoneticTranscriber** by calling the **ReleaseMicrophone()** method of the **PhoneticTranscriber** object:

```
objPhoneTrnscriber.ReleaseMicrophone();
```

This action automatically releases the **LevelMeter** if it has also been secured for use of the **PhoneticTranscriber**.

Adding Trained Shortcut to Collection, Loading into Memory

Before you can use a trained shortcut, you must add it to the **Shortcuts** collection, then load the shortcut into the in-memory language model.

You use the phonetic transcription string you obtained from the microphone to add the trained shortcut to the collection.

Using the Phonetic Transcription String to Add Shortcut to Collection

To add a trained shortcut to the Shortcuts collection:

1. Call the **Add()** method of the **Shortcuts** collection object, just as you did with non-trained shortcuts, only this time you pass as the *Transcription* argument the phonetic transcription string retrieved from either the **GetTranscription()** method or the **TranscriptionResult** event handler:

- ```
objShctsColl.Add(psShortcutTypeVoice, "normal foot", "This patient's
foot is normal.", strShctAudioTranscr, True, False);
```
2. Load the shortcut into the in-memory language model for immediate use:  

```
pscribeSDK.LoadShortcut("normal foot", "This patient's
foot is normal.", strShctAudioTranscr)
```

## Revising Training for Existing Shortcut

To revise the training for an existing Shortcut, you could:

1. Retrieve that existing **Shortcut** object from the collection and set its **Transcription** property using the new phonetic transcription string:

```
objShortcut = objShctsColl.Item(4);
objShortcut.Transcription = strShctAudioTranscr;
```

Or you could add the new **Shortcut** as shown in [Adding New Shortcut with Existing Phonetic Transcriber](#).

2. Load the shortcut into the in-memory language model for immediate use:

```
pscribeSDK.LoadShortcut("normal foot", "This patient's
foot is normal.", strShctAudioTranscr)
```

## Adding New Shortcut with Existing Phonetic Transcriber

To add a new Shortcut using an existing phonetic transcriber:

1. Call the **Add()** method of the **Shortcuts** collection object and pass as the *Transcription* argument the existing phonetic transcription string from an existing shortcut object's **Transcription** property:

```
objShortcut = objShctsColl.Item(4);
objShctsColl.Add(psShortcutTypeVoice, "normal foot", "This patient's
foot appears normal.", objShortcut.Transcription, True, False);
```

2. Load the shortcut into the in-memory language model for immediate use:

```
pscribeSDK.LoadShortcut("normal foot", "This patient's
foot is normal.", objShortcut.Transcription)
```

# Adding Trained Word or Punctuation Mark to Collection, Loading into Memory

Before you can use a trained word or punctuation mark, you must add it to the **Words** collection, then load the word or punctuation mark into the in-memory language model.

You use the phonetic transcription string you obtained from the microphone to add the trained word or punctuation mark to the collection.

## Using the Phonetic Transcription String to Add Word to Words Collection

To add a trained word to the Words collection:

1. Call the **Add()** method of the **Words** collection object just as you would to add a non-trained word, only also pass the phonetic transcription string as the *Transcription* argument of the method:

```
objWordsColl.Add("glycemic index", psWordCategoryGeneral,
 strWordAudioTrascr, psWordStateApproved, psWordSourceUser, False,
 True);
```

2. Load the word into the in-memory language model for immediate use:

```
pscribeSDK.LoadWord("glycemic index", psWordCategoryGeneral,
 strWordAudioTrascr)
```

## Using the Phonetic Transcription String to Add Punctuation Mark to Words Collection

To add a trained punctuation mark to the Words collection:

1. Call the **Add()** method of the **Words** collection object just as you would to add a non-trained word, only you also pass the phonetic transcription string as the *Transcription* argument of the method:

```
objWordsColl.Add(".\period", psWordCategoryGeneral,
 strWordAudioTrascr, psWordStateApproved, psWordSourceUser,
 False, True);
```

2. Load the word into the in-memory language model for immediate use:

```
pscribeSDK.LoadWord(".\period", psWordCategoryGeneral,
 strWordAudioTrascr)
```

## Revising Training for Existing Word

To revise the training for an existing Word:

1. Retrieve that existing **Word** object from the collection, then set its **Transcription** property using the new phonetic transcription string:

```
objWord = objWordsColl.Item(9);
objWord.Transcription = strWordAudioTranscr;
```

Or you could add the new **Word** as shown in [Adding New Word with Existing Phonetic Transcriber](#).

2. Load the word into the in-memory language model for immediate use:

```
pscribeSDK.LoadWord("glycemic index", psWordCategoryGeneral,
strWordAudioTrascr)
```

## Adding New Word with Existing Phonetic Transcriber

To add a new **Word** using an existing phonetic transcriber:

1. Call the **Add()** method of the **Words** collection object and pass as the *Transcription* argument the existing phonetic transcription string from an existing shortcut object's **Transcription** property:

```
objWord = objWordsColl.Item(9);
objWordsColl.Add("glycemic index", psWordCategoryGeneral, objWord.
Transcription, psWordStateApproved, psWordSourceUser, True,
False);
```

2. Load the word into the in-memory language model for immediate use:

```
pscribeSDK.LoadWord("glycemic index", psWordCategoryGeneral,
objWord.Transcription)
```

## Importing and Training Trigger Words

Trigger words are custom words that you can dictate to indicate you are switching to a custom command mode. For instance, **Dictaphone** is a trigger word that you could dictate before you dictate a custom healthcare command.

You can create commands that are specific to your installation and trigger them using a custom trigger word that the *PowerScribe SDK* development team at Nuance creates for you. The team sends you an XML file that you use to import the words through the *SDK Administrator* **Trigger Words** tab.

To train a trigger word:

1. Import the trigger words into the *SDK Administrator* by opening the *SDK Administrator* client and clicking on the **Trigger Words** tab. To open the *SDK*

*Administrator* client, go to a browser and enter this URL: <http://<servername>/pscribesdk/admin>

2. Find the **Import** button at the bottom of the page and click it. Browse until you find the XML file of trigger words that the *PowerScribe SDK* development team at Nuance provided. Select the file and follow the prompts to finish importing it.
3. Create a **PhoneticTranscriber** object. For details refer to [Creating PhoneticTranscriber Object on page 302](#).
4. Secure the microphone and **LevelMeter** for exclusive use of the **PhoneticTranscriber** object. for details, refer to [Securing Microphone for PhoneticTranscriber on page 302](#) and [Securing LevelMeter for PhoneticTranscriber on page 302](#).
5. Using a **PhoneticTranscriber** object, train the trigger word and retrieve the phonetic transcription string that contains its pronunciation. For details refer to [Training Audio of Shortcut or Word Using Custom GUI Buttons on page 303](#) or [Training Audio of Shortcut or Word Using Microphone Buttons on page 305](#).
6. Add each trigger word to the **Words** collection to store it in the database; then load each **Word** into the in-memory language model. For details, refer to [Adding Trained Word or Punctuation Mark to Collection, Loading into Memory on page 310](#).

## Selecting, Inserting and Auto Expanding Shortcuts in PlayerEditor—JavaScript

You can select a shortcut from a list view created in JavaScript and insert it into a **PlayerEditorCtl** and have the shortcut expand automatically in the report that is displaying in the editor.

For instance, in the list view shown here, you could select the shortcut you want to insert into the report, then click the **Insert** button to insert the expanded text of the shortcut.

The function that triggers on a click of the **Insert** button could deploy this functionality by first storing the **ShortText** property of the shortcut in the first column, and the

associated **Shortcut** object in the **Tag** property of the list view item. The function could then retrieve the selected item from the list view (named **Listview1**) and retrieve the **Shortcut** object whose **ShortText** property value matches the item selected using the **Tag** property:

```
var psMergeSrcTypeShortcut = 3;

function btnInsert_onclick()
{
 try
 {
 var currentReoprt = window.dialogArguments[0].Report;

 if (currentReport == null)
 return;

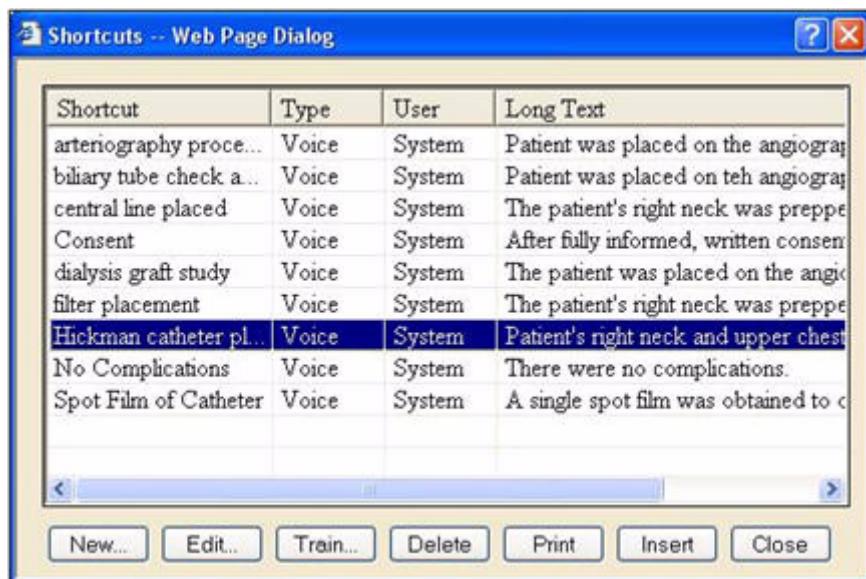
 var selItem = Listview1.SelectedItem;
 var shObj = selItem.Tag;
 }
}
```

The function could then call the **MergeTemplate()** method of the **Report** object and pass it the **Shortcut** object's **LongText** property to merge into the report at the cursor location:

```
 currentReport.MergeTemplate(shObj.LongText, psMergeSrcTypeShortcut);
}
catch (error)
{
 alert(error.description);
}
}
```

For more information on the **MergeTemplate()** method, refer to [Chapter 6, Working with Shortcuts, Categories, and Words on page 279](#).

The sample code shown here is part of a JavaScript sample supplied with the product. Portions of the code shown in the [JavaScript Code Summary on page 324](#) are from that sample.



# Auto Expanding Shortcuts After Drag and Drop into PlayerEditor—C#

You can have your application automatically expand shortcuts when users drag and drop them into the **PlayerEditor**. You drag those shortcuts from the source *SDK* application where the shortcut is defined and drop them in the destination application where the **PlayerEditorCtl** expands and integrates them.

The only place you need to write special code to implement this drag and drop functionality is in the source that you drag the shortcuts or template from. Let's see one way to implement it.

## Implementing Drag/Drop Feature in Drag Source

One way the application that contains the shortcuts or template to be dragged can implement drag and drop expansion of those shortcuts is to take these steps:

1. Create an **IDataObject** using the MSDN Library:

```
public class Form1: System.Windows.Forms.Form, IDataObject
{
 ...
}
```

2. Develop code for these methods methods of the **IDataObject**:

- **GetFormats()**
- **GetDataPresent()**
- **GetData()**

3. Before you work with these methods, you need to declare a **PowerscribeSDK** object that you later instantiate in the **Form1()** routine:

```
public POWERSCRIBESDKLib.PowerscribeSDK pscriveSDK;
...
objSDK = new POWERSCRIBESDKLib.PowerscribeSDK();
```

4. In the **GetFormats()** method, retrieve a list of formats that data of this **IDataObject** are stored in (or can be converted to). The data formats should include “**pssdk shortcut**” for *SDK* shortcuts and “**pssdk template**” for *SDK* templates:

```
public string[] GetFormats()
{
 System.Diagnostics.Debug.WriteLine("GetFormats");
 string [] formats = new string[3];
 formats[0] = "pssdk shortcut";
 formats[1] = DataFormats.Text.ToString();
 return formats;
}
```



**Note:** You would create this same type of routine for the "pssdk template" format required to drag and drop a template into a report.

5. In a **GetDataPresent()** method that receives a **format** string as an argument, check to see whether data is available in the “**pssdk shortcut**” format for *SDK* shortcuts that should be expanded (see code below) or “**pssdk template**” format for *SDK* templates that should be applied (only the first format is illustrated in this example):

```
public bool GetDataPresent(string format)
{
 System.Diagnostics.Debug.WriteLine("GetDataPresent2:" + format);
 bool bRet = false;
 if ((format == "pssdk shortcut") || (format == DataFormats.Text))
 {
 bRet = true;
 }
 return bRet;
}
```

6. In addition to a **GetDataPresent()** method to test the format of the data, your application also needs to have an **ItemDrag()** method to take action when the user selects and drags a shortcut from the listview:

```
private void listView1_ItemDrag(object sender, System.Windows.Forms.
 ItemDragEventArgs e)
{
 System.Diagnostics.Debug.WriteLine("dragging");
 listView1.DoDragDrop(this, DragDropEffects.Copy);
}
```

7. In the **GetData()** method, you set up the data object in the data format you want sent to the **PlayerEditor**. If the format type of the string passed to this method is **pssdk shortcut**, then you retrieve the selected **ListView** item:

```
public object GetData(string format, bool autoConvert)
{
 if (format == "pssdk shortcut")
 {
 ListViewitem lvi = listView1.SelectedItems[0];
```

8. You then create a string to contain the **LongText** property of the shortcut in the **GetData()** method. And you create a unicode encoded string to contain the **LongText** of the **Shortcut** that you want to expanded in the **PlayerEditor**. You must null terminate the string by adding **00** to the end of it using code like this:

```
string strShortcutLongText = lvi.Tag.ToString();
System.Text.UnicodeEncoding ue = new System.Text.UnicodeEncoding();
int nBytes = ue.GetByteCount(strShortcutLongText);
int nBytesPerChar = ue.GetByteCount("1");
// Add nBytesPerChar for correct null-termination of the
// string on the PlayerEditor side.
byte [] byteArray = new byte[nBytes+nBytesPerChar];
```

```
ue.GetBytes(strShortcutLongText, 0, strShortcutLongText.Length,
ByteArray, 0);
```

9. Finally in **GetData()**, you create a **MemoryStream** object to return the **LongText** in and pass it the **ByteArray** you created with **GetBytes()**:

```
// Return the memory stream containing the LongText.
System.IO.MemoryStream ms = new System.IO.MemoryStream(ByteArray);
return ms;
}
```

Notice that in **GetData()** you retrieve the **LongText** of the shortcut from the **Tag** property of the **lvi ListView**. Let's look at how the **LongText** gets into the **Tag** property in the next step.

10. The application sets that **Tag** property as part of the action it takes when the user clicks the **Login** button in the GUI. This routine also takes several steps not related to logging in, including:

- Retrieves a **PSAdminSDK** object. Because it is in C#, the application casts the object as a **PSADMINSDKLib.IPSAdminSDK** type:

```
PSADMINSDKLib.IPSAdminSDK objAdminSDK =
(PSADMINSDKLib.IPSAdminSDK)objSDK.AdminSDK;
```

- Uses the **PSAdminSDK** object to retrieve a collection of the logged in user's shortcuts by calling its **GetShortcuts()** method:

```
PSADMINSDKLib.Shortcuts objShortcuts =
objAdminSDK.GetShortcuts(
0, PSADMINSDKLib.GlobalType.psGlobalTypeAll,
txtUserName.Text,
PSADMINSDKLib.FilterType.psFilterTypeNone, null,
PSADMINSDKLib.FilterType.psFilterTypeNone, null,
null);
```

- Adds the **ShortText** of each **Shortcut** to the **ListView** using an **IEnumerator** object and calling the **GetEnumerator()** method on the **Shortcuts** collection:

```
IEnumerator enumerator = objShortcuts.GetEnumerator();
```

- Create a **Shortcut** object called **objSC**. Then for each **Shortcut** object in the collection, add the **ShortText** property value to the **ListView**, populating the **ListView** with **ShortText** values, and set the **Tag** property of the **ListView** item to the **LongText** of that **Shortcut** object:

```
PSADMINSDKLib.Shortcut objSC = null;
while (enumerator.MoveNext())
{
 objSC = (PSADMINSDKLib.Shortcut)enumerator.Current;
 ListViewItem lvi = listView1.Items.Add(new ListViewItem(objSC.
 ShortText));
 lvi.Tag = objSC.LongText;
}
```

## Using the Drag Source Application

After you have completed, compiled, and run code using the **IDataObject**, when you drag a shortcut's **ShortText** from the listview to a **PlayerEditorCtl** in an active *SDK* application, the **PlayerEditorCtl** queries the object to see whether it contains either a “**pssdk shortcut**” or “**pssdk template**” drop data format. The **PlayerEditorCtl** takes the following action on these drop data formats:

- **pssdk shortcut** format—**PlayerEditorCtl** integrates the **LongText** (from the **Tag** property of the **ListView** item dragged).
- **pssdk template** format—**PlayerEditorCtl** integrates the **templatetext** from the source application into the text displaying.

The source application that you dragged the shortcuts from then fires the **shortcut dropped** event, passing the **IDataObject** object as the event's first argument.

The **Report** displaying in the **PlayerEditorCtl** that is the destination for the dropped shortcut or template receives the **shortcut dropped** event notification and the **PlayerEditorCtl** integrates the long text or template into the report content at the location of the cursor when the drop occurs.

See the drag and drop code summary in [C# Drag and Drop Code Summary on page 317](#).

## C# Drag and Drop Code Summary

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Runtime.InteropServices;
using POWERSCRIBESDKLib;
using PSADMINSDKLib;

namespace DragDrop
{
 /// Summary description for Form1.
 public class Form1 : System.Windows.Forms.Form, IDataObject
 {
 private System.Windows.Forms.ListView listView1;
 private System.Windows.Forms.Label label1;
 private System.Windows.Forms.Label label2;
 private System.Windows.Forms.Label label3;
 /// Required designer variable.
 private System.ComponentModel.Container components = null;
 private System.Windows.Forms.Label label4;
 private System.Windows.Forms.TextBox txtUserName;
 private System.Windows.Forms.TextBox txtPassword;
```

```
private System.Windows.Forms.Button btnLogin;
private System.Windows.Forms.TextBox txtSDKServerURL;

private POWERSCRIBESDKLib.PowerscribeSDK objSDK;

public Form1()
{
 InitializeComponent();

 objSDK = new POWERSCRIBESDKLib.PowerscribeSDK();
}

/// Clean up any resources being used.
protected override void Dispose(bool disposing)
{
 if(disposing)
 {
 if (components != null)
 {
 components.Dispose();
 }
 }
 base.Dispose(disposing);
}
#region Windows Form Designer generated code
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
private void InitializeComponent()
{
 this.listView1 = new System.Windows.Forms.ListView();
 this.label1 = new System.Windows.Forms.Label();
 this.label2 = new System.Windows.Forms.Label();
 this.txtUserName = new System.Windows.Forms.TextBox();
 this.txtPassword = new System.Windows.Forms.TextBox();
 this.btnLogin = new System.Windows.Forms.Button();
 this.label3 = new System.Windows.Forms.Label();
 this.label4 = new System.Windows.Forms.Label();
 this.txtSDKServerURL = new System.Windows.Forms.TextBox();
 this.SuspendLayout();
 //
 // listView1
 //
 this.listView1.Location = new System.Drawing.Point(8, 160);
 this.listView1.Name = "listView1";
 this.listView1.Size = new System.Drawing.Size(264, 152);
 this.listView1.TabIndex = 4;
 this.listView1.DoubleClick += new System.EventHandler
 (this.listView1_DoubleClick);
 this.listView1.ItemDrag += new System.Windows.Forms.
 ItemDragEventHandler(this.listView1_ItemDrag);
 //
 // label1
 //
 this.label1.Location = new System.Drawing.Point(8, 64);
 this.label1.Name = "label1";
 this.label1.Size = new System.Drawing.Size(100, 16);
}
```

```
this.label1.TabIndex = 1;
this.label1.Text = "User name:";
//
// label2
this.label2.Location = new System.Drawing.Point(120, 64);
this.label2.Name = "label2";
this.label2.Size = new System.Drawing.Size(100, 16);
this.label2.TabIndex = 2;
this.label2.Text = "Password:";
//
// txtUserName
this.txtUserName.Location = new System.Drawing.Point(8, 80);
this.txtUserName.Name = "txtUserName";
this.txtUserName.TabIndex = 1;
this.txtUserName.Text = "textBox1";
//
// txtPassword
this.txtPassword.Location = new System.Drawing.Point(120, 80);
this.txtPassword.Name = "txtPassword";
this.txtPassword.PasswordChar = '*';
this.txtPassword.TabIndex = 2;
this.txtPassword.Text = "textBox2";
//
// btnLogin
this.btnLogin.Location = new System.Drawing.Point(224, 80);
this.btnLogin.Name = "btnLogin";
this.btnLogin.Size = new System.Drawing.Size(48, 20);
this.btnLogin.TabIndex = 3;
this.btnLogin.Text = "Login";
this.btnLogin.Click += new System.EventHandler
 (this.button1_Click);
//
// label3
this.label3.Location = new System.Drawing.Point(8, 112);
this.label3.Name = "label3";
this.label3.Size = new System.Drawing.Size(264, 40);
this.label3.TabIndex = 6;
this.label3.Text = "Drag a shortcut from this list into an
active PlayerEditorCtl instance. " +
"The shortcut LongText will appear in the PlayerEditorCtl.";
this.label3.Click += new System.EventHandler(this.label3_Click);
//
// label4
this.label4.Location = new System.Drawing.Point(8, 8);
this.label4.Name = "label4";
this.label4.Size = new System.Drawing.Size(160, 16);
this.label4.TabIndex = 7;
this.label4.Text = "Powerscribe SDK Server URL:";
this.label4.Click += new System.EventHandler(this.label4_Click);
//
// txtSDKServerURL
this.txtSDKServerURL.Location = new System.Drawing.Point(8, 24);
this.txtSDKServerURL.Name = "txtSDKServerURL";
this.txtSDKServerURL.Size = new System.Drawing.Size(264, 20);
```

```
 this.txtSDKServerURL.TabIndex = 0;
 this.txtSDKServerURL.Text = "textBox3";
 //
 // Form1
 this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
 this.ClientSize = new System.Drawing.Size(280, 318);
 this.Controls.Add(this.txtSDKServerURL);
 this.Controls.Add(this.label4);
 this.Controls.Add(this.label3);
 this.Controls.Add(this.btnLogin);
 this.Controls.Add(this.txtPassword);
 this.Controls.Add(this.txtUserName);
 this.Controls.Add(this.label2);
 this.Controls.Add(this.label1);
 this.Controls.Add(this.listView1);
 this.Name = "Form1";
 this.Text = "DragDrop Sample";
 this.Closing += new System.ComponentModel.CancelEventHandler
 (this.Form1_Closing);
 this.Load += new System.EventHandler(this.Form1_Load);
 this.ResumeLayout(false);
 }
}

/// The main entry point for the application.
[STAThread]
static void Main()
{
 Application.Run(new Form1());
}

private void Form1_Load(object sender, System.EventArgs e)
{
 txtSDKServerURL.Text = "";
 txtUserName.Text = "";
 txtPassword.Text = "";
}

private void listView1_ItemDrag(object sender, System.Windows.Forms.
 ItemDragEventArgs e)
{
 System.Diagnostics.Debug.WriteLine("dragging");
 listView1.DoDragDrop(this, DragDropEffects.Copy);
}

public bool GetDataPresent(Type format)
{
 System.Diagnostics.Debug.WriteLine("GetDataPresent: " +
 format.ToString());
 // TODO: Add CMyDataObject.GetDataPresent implementation
 return false;
}
```

```
public bool GetDataPresent(string format)
{
 System.Diagnostics.Debug.WriteLine("GetDataPresent2: " + format);
 bool bRet = false;
 if (format == "pssdk shortcut" || format == DataFormats.Text)
 {
 bRet = true;
 }
 return bRet;
}
public bool GetDataPresent(string format, bool autoConvert)
{
 System.Diagnostics.Debug.WriteLine("GetDataPresent1: " + format);
 return GetDataPresent(format);
}

public object GetData(Type format)
{
 System.Diagnostics.Debug.WriteLine("GetData1: " +
 format.ToString());
 return null;
}
public object GetData(string format)
{
 System.Diagnostics.Debug.WriteLine("GetData2: " + format);
 return null;
}

public object GetData(string format, bool autoConvert)
{
 if (format == "pssdk shortcut")
 {
 ListViewItem lvi = listView1.SelectedItems[0];
 // Create a shortcut via the PowerScribe SDK admin
 string strShortcutLongText = lvi.Tag.ToString();
 System.Text.UnicodeEncoding ue = new System.Text._
 UnicodeEncoding();
 int nBytes = ue.GetByteCount(strShortcutLongText);
 int nBytesPerChar = ue.GetByteCount("1");
 // Here you add nBytesPerChar to correctly null-terminate
 // the string on the PlayerEditor side
 byte [] byteArray = new byte[nBytes+nBytesPerChar];
 ue.GetBytes(strShortcutLongText, 0, strShortcutLongText.Length,
 byteArray, 0);
 System.IO.MemoryStream ms = new System.IO.MemoryStream
 (byteArray);
 return ms;
 }
 else
 {
```

```
 return "DragDrop application";
 }
}
public string[] GetFormats()
{
 System.Diagnostics.Debug.WriteLine("GetFormats");
 string [] formats = new string[2];
 formats[0] = "pssdk shortcut";
 formats[1] = DataFormats.Text.ToString();
 return formats;
}
public string[] GetFormats(bool autoConvert)
{
 return GetFormats();
}

public void SetData(object data)
{
 // TODO: Add CMyDataObject.SetData implementation
}

private void Form1_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
 if (btnLogin.Text == "Logoff")
 {
 objSDK.Logoff(1);
 objSDK.Uninitialize();
 }
}

private void button1_Click(object sender, System.EventArgs e)
{
 bool bOK = false;
 bool bLogin = false;

 try
 {
 if (btnLogin.Text == "Login")
 {
 bLogin = true;
 btnLogin.Text = "Wait...";
 DoLoginForm dlgDoLogin = new DoLoginForm();
 dlgDoLogin.objSDK = objSDK;
 dlgDoLogin.strUserName = txtUserName.Text;
 dlgDoLogin.strPassword = txtPassword.Text;
 dlgDoLogin.strSDKServerURL = txtSDKServerURL.Text;
 dlgDoLogin.ShowDialog();

 // Get the PSAdminSDK object using the AdminSDK property
 PSADMINSDKLib.IPSAdminSDK objAdminSDK =
 (PSADMINSDKLib.IPSAdminSDK) objSDK.AdminSDK;
 }
 }
}
```

```
// Get all of the user's shortcuts in a collection
PSADMINSDKLib.Shortcuts objShortcuts =
objAdminSDK.GetShortcuts(
 0, PSADMINSDKLib.GlobalType.psGlobalTypeAll,
 txtUserName.Text,
 PSADMINSDKLib.FilterType.psFilterTypeNone, null,
 PSADMINSDKLib.FilterType.psFilterTypeNone, null,
 null);

// Add the shortcuts to the list view
IEnumerator enumerator = objShortcuts.GetEnumerator();
PSADMINSDKLib.Shortcut objSC = null;
while (enumerator.MoveNext())
{
 // Get the shortcut and store in objSC Shortcut object
 objSC = (PSADMINSDKLib.Shortcut)enumerator.Current;

 // Add the Shortcut's ShortText as an item to the list view
 ListViewItem lvi = listView1.Items.Add(new ListViewItem
 (objSC.ShortText));

 // Associate the shortcut's LongText with the list view item
 // using the item's Tag property.
 // When a user drops a list view item into a PlayerEditorCtl, the
 // PlayerEditorCtl calls back into the GetData() method here to
 // retrieve the shortcut's LongText, stored in the item's Tag.
 lvi.Tag = objSC.LongText;
}
else
{
 objSDK.Logoff(1);
 objSDK.Uninitialize();
}
bOK = true;
}
catch (COMException ce)
{
 MessageBox.Show(ce.ToString());
}
finally
{
 if (bOK == true)
 {
 btnLogin.Text = bLogin ? "Logoff" : "Login";
 }
 else
 {
 btnLogin.Text = bLogin ? "Login" : "Logoff";
 }
}
```

```
 }
 }

 public void SetData(Type format, object data)
 {
 // TODO: Add CMyDataObject.System.Windows.Forms.IDataObject.SetData
 // implementation
 }

 public void SetData(string format, object data)
 {
 // TODO: Add CMyDataObject.System.Windows.Forms.IDataObject.
 // SetData implementation
 }

 public void SetData(string format, bool autoConvert, object data)
 {
 // TODO: Add CMyDataObject.System.Windows.Forms.IDataObject.
 // SetData implementation
 }
}

}
```

## JavaScript Code Summary

### Shortcuts ListView

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
 <head>
 <title></title>
 <!--LINK href="../css/common.css" type="text/css" rel="stylesheet"-->
 <meta content="False" name="vs_showGrid">
 <meta content="http://schemas.microsoft.com/intellisense/ie5"
 name="vs_targetSchema">
 <script language="javascript" id="Utils" src="../scripts/utils.js">
 </script>

 <script language="javascript" id="Shortcuts" src="../scripts/Shortcuts.js">
 </script>

 <script>
 function btnPrint()
 {
 window.print();
 }
 </script>
 </head>

 <body language="javascript" bgcolor="#EFE7CE" onload="return
window.onload()">
```

```
 ms_positioning="GridLayout">
<P>
<div id="ListViewObj">
<OBJECT id="Listview1" style="Z-INDEX: 100; LEFT: 16px; WIDTH: 95%;
POSITION: absolute; TOP: 16px; HEIGHT: 384px"
classid="clsid:BDD1F04B-858B-11D1-B16A-00C0F0283628" VIEWASTEXT>

 <PARAM NAME="_ExtentX" VALUE="24500">
 <PARAM NAME="_ExtentY" VALUE="10160">
 <PARAM NAME="SortKey" VALUE="0">
 <PARAM NAME="View" VALUE="3">
 <PARAM NAME="Arrange" VALUE="0">
 <PARAM NAME="LabelEdit" VALUE="1">
 <PARAM NAME="SortOrder" VALUE="0">
 <PARAM NAME="Sorted" VALUE="0">
 <PARAM NAME="MultiSelect" VALUE="0">
 <PARAM NAME="LabelWrap" VALUE="-1">
 <PARAM NAME="HideSelection" VALUE="0">
 <PARAM NAME="HideColumnHeaders" VALUE="0">
 <PARAM NAME="OLEDragMode" VALUE="0">
 <PARAM NAME="OLEDropMode" VALUE="0">
 <PARAM NAME="AllowReorder" VALUE="0">
 <PARAM NAME="Checkboxes" VALUE="0">
 <PARAM NAME="FlatScrollBar" VALUE="0">
 <PARAM NAME="FullRowSelect" VALUE="-1">
 <PARAM NAME="GridLines" VALUE="-1">
 <PARAM NAME="HotTracking" VALUE="0">
 <PARAM NAME="HoverSelection" VALUE="-1">
 <PARAM NAME="PictureAlignment" VALUE="0">
 <PARAM NAME="TextBackground" VALUE="0">
 <PARAM NAME="_Version" VALUE="393217">
 <PARAM NAME="ForeColor" VALUE="-2147483640">
 <PARAM NAME="BackColor" VALUE="16777215">
 <PARAM NAME="BorderStyle" VALUE="1">
 <PARAM NAME="Appearance" VALUE="1">
 <PARAM NAME="MousePointer" VALUE="0">
 <PARAM NAME="Enabled" VALUE="1">
 <PARAM NAME="NumItems" VALUE="0">
 <param name="OLEDragMode" value="0">
 <param name="OLEDropMode" value="0">
</OBJECT>
</div>
<script>
CreateObjectCtl("ListViewObj");
</script>

<table border="0" style="Z-INDEX: 101; LEFT: 16px; POSITION:
absolute; TOP: 408px;">
```

```
<tr>
 <td align="left" nowrap>
 <INPUT language="javascript" class="clsFlatBtn" id="btnNew"
 onclick="return btnNew_onclick()" type="button" value="New..." name="btnNew" style="WIDTH: 64px; HEIGHT: 24px">
 <INPUT language="javascript" class="clsFlatBtn" id="btnEdit"
 onclick="return btnEdit_onclick()" type="button" value="Edit..." name="btnEdit" style="margin-left:3px; WIDTH: 64px; HEIGHT: 24px">
 <INPUT language="javascript" class="clsFlatBtn" id="TrainBtn"
 onclick="return TrainBtn_onclick()" type="button" value="Train..." name="TrainBtn" style="margin-left:3px; WIDTH: 64px; HEIGHT: 24px">
 <INPUT language="javascript" class="clsFlatBtn" id="btnDelete"
 onclick="return btnDelete_onclick()" type="button" value="Delete" name="btnDelete" style="margin-left:3px; WIDTH: 64px; HEIGHT: 24px">
 <INPUT language="javascript" class="clsFlatBtn" id="btnPrint"
 onclick="return btnPrint()" type="button" value="Print" name="btnPrint" style="margin-left:3px; WIDTH: 64px; HEIGHT: 24px">
 <INPUT language="javascript" class="clsFlatBtn" id="btnInsert"
 onclick="return btnInsert_onclick()" type="button" value="Insert" name="btnInsert" style="display:none; position:inherit; margin-left:3px; WIDTH: 64px; HEIGHT: 24px">
 <INPUT language="javascript" id="btnClose" type="button" value="Close" name="btnClose" onclick="return btnClose_onclick()" style="margin-left:3px; WIDTH: 64px; HEIGHT: 24px">
 </td>
</tr>
</table>
</P>
</body>
</html>
```

## Shortcuts Scripts

```
//SDK Administrator, Shortcuts.js File
// Global Variables
var gShortcutsObj = null;
var gScript = null;
var gPSDK = null;
var gAdminSDK = null;

var psShortcutTypeAll = 0;
var psShortcutTypeVoice = 1;
var psShortcutTypeText = 2;
var psShortcutTypeUser = 4;
var psShortcutTypeTemplate = 5;

var psMergeSrcTypeShortcut = 3;
```

```
var psGlobalTypeAll = 0;
var psGlobalTypeExclude = 1;
var psGlobalTypeInclude = 2;
var psGlobalTypeOnly = 3;

var psFilterTypeAll = 0;

var bUsePlainTextEditor = window.dialogArguments[1];
var gEditArg = null;

// Object passed into Edit dialog
var xmlDoc = new ActiveXObject("Msxml2.DOMDocument.4.0");

// List view column
var gColLongText = 3;
document.title = "Shortcuts";

function btnNew_onclick()
{
 NewShortcut();
}

function btnEdit_onclick()
{
 EditShortcut();
}

function window_onload()
{
 if (window.dialogArguments[2] == true)
 btnInsert.style.display = "inline";

 InitParams(window.dialogArguments[0].top.script);
}

function btnDelete_onclick()
{
 DeleteShortcut();
}

function btnClose_onclick()
{
 window.close();
}

function InitParams(script)
{
 try
 {
```

```
 if (script == null)
 return;
 gScript = script;
 gPSDK = gScript.pscribeSDK;
 gAdminSDK = gScript.adminSDK;
 // for XML Report sample...
 if (gAdminSDK == null)
 gAdminSDK = gScript.pscribeSDK.AdminSDK;
 gShortcutsObj = BuildListItem(Listview1, psShortcutTypeAll,
 psGlobalTypeInclude);
}
catch(e)
{
 alert(e.description);
}
}

function InitEditArg()
{
 // Initialize gEditArg object
 if (gEditArg == null)
 {
 gEditArg = new Object();
 }
 gEditArg.gScript = gScript;
 gEditArg.sTitle = "New Shortcut";
 gEditArg.bSave = false;
 gEditArg.sName = "";
 gEditArg.sUser = "Personal";
 gEditArg.sType = "Voice";
 gEditArg.sLong = "";
 gEditArg.sTranscription = "";
}

function NewShortcut()
{
 try
 {
 InitEditArg();
 if (bUsePlainTextEditor)
 var dlg = doModalDialogSizeable('ShortcutEdit.htm', gEditArg, 700, 530);
 else
 var dlg = doModalDialogSizeable('ShortcutEditEx.htm', gEditArg, 700, 530);
 if (gEditArg.bSave == true)
 { // Determine type and owner (author) of Shortcut
 var bGlobal = (gEditArg.sUser == "System")? true : false;
 var sTrans = "";
 var type = (gEditArg.sType == "Voice" ? psShortcutTypeVoice :
 psShortcutTypeText);

```

```
// Add Shortcut to collection using Add() Method of Shortcut Object
var newShortcut = gShortcutsObj.Add(type, gEditArg.sName,
 gEditArg.sLong, gEditArg.sTranscription, bGlobal);
gPSDK.LoadShortcut(gEditArg.sName, gEditArg.sLong, type,
 gEditArg.sTranscription);
// Add the new item into the list view control
var nItem = Listview1.ListItems.Add();
// First Column = ShortText
nItem.Text = newShortcut.ShortText;
// Second Column = Shortcut Type
nItem.SubItems(1) = GetTypeString(newShortcut.Type);
// Third Column = Shortcut Author
var sUser = new String(newShortcut.Author);
if (sUser.length == 0)
 sUser = "System";
nItem.SubItems(2) = sUser;
// Fourth Column = LongText (formatted by formatLongText function)
nItem.SubItems(gColLongText) = formatLongText(newShortcut.LongText);
 nItem.Tag = newShortcut;
}
}
catch(err)
{
 alert(err.description);
}
}

function EditShortcut()
{
try
{
 var selItem = Listview1.SelectedItem;
 var shObj = selItem.Tag;
 if (shObj.Global) // If the Global property of Shortcut is True
 {
 alert("System shortcuts can only be edited by an
 administrator.");
 return;
 }
 if (shObj.LongText.search("<?xml") > 0)
 {
 var sure = window.confirm("This is the XML Shortcut. By editing it
 you will lose the original formatting. Do you want to continue?");
 if (!sure)
 return;
 }
 InitEditArg();
 gEditArg.sTitle = "Edit Shortcut";
 gEditArg.sName = shObj.ShortText;
 gEditArg.sType = GetTypeString(shObj.Type);
```

```
var sUser = shObj.Author;
if (sUser != "System")
 sUser = "Personal";
gEditArg.sUser = sUser;
// Retrieve expanded text using LongText property of Shortcut
gEditArg.sLong = shObj.LongText;

if (bUsePlainTextEditor)
 var dlg = doModalDialogSizeable('ShortcutEdit.htm', gEditArg, 700, 530);
else
 var dlg = doModalDialogSizeable('ShortcutEditEx.htm', gEditArg, 700, 530);
if (gEditArg.bSave == true)
{
 // Update the LongText property of Shortcut with edited text
 shObj.LongText = gEditArg.sLong;
 selItem.SubItems(gColLongText) = formatLongText(shObj.LongText);
 // Load the shortcut using LoadShortcut() of PowerscribeSDK object
 gPSDK.LoadShortcut(shObj.ShortText, shObj.LongText, shObj.Type,
 shObj.Transcription);
 return;
}
catch(e)
{
 alert(e.description);
}
}

function DeleteShortcut()
{
try
{
 var selItem = Listview1.SelectedItem;
 var shObj = selItem.Tag;

 if (shObj.Global)
 {
 alert("System shortcuts can only be deleted by an administrator.");
 }
 else
 {
 var sure = window.confirm("Are you sure want to delete this shortcut?");
 if (!sure)
 return;
 // Remove the Shortcut from the collection using Remove() method
 gShortcutsObj.Remove(shObj);
 selItem.Tag = null;
 Listview1.ListItems.Remove(selItem.Index);
 }
}
```

```
 }
 }
 catch(e)
 {
 alert(e.description);
 // if this shortcut was deleted
 if (e.number == -2146776281)
 {
 selItem.Tag = null;
 Listview1.ListItems.Remove(selItem.Index);
 }
 }
}

function BuildListItem(listCtrl, type, globalType)
{
 try
 {
 var retShortcuts;
 // Add columns: Name, Type, Author, Long Text
 var nCol = 0;
 var nWidth = 70;
 listCtrl.ColumnHeaders.Clear();
 newCol = listCtrl.ColumnHeaders.Add();
 newCol.Text = "Shortcut"
 newCol.Width = 2 * nWidth;
 nCol++;
 newCol = listCtrl.ColumnHeaders.Add();
 newCol.Text = "Type"
 newCol.Width = nWidth;
 nCol++;
 newCol = listCtrl.ColumnHeaders.Add();
 newCol.Text = "User"
 newCol.Width = nWidth;
 nCol++;
 newCol = listCtrl.ColumnHeaders.Add();
 newCol.Text = "Long Text"
 newCol.Width = 5 * nWidth;
 gColLongText = nCol;
 // Add items from PSAAdminSDK
 listCtrl.ListItems.Clear();
 // Retrieve Shortcuts Collection using GetShortcuts() method
 retShortcuts = gAdminSDK.GetShortcuts(type, globalType, "",
 psFilterTypeByAll, "", "");
 var nCount = retShortcuts.Count;
 xmlDoc.async = false;
 for (var i = 1; i <= nCount; i++)
 {
 var shortcut = retShortcuts.Item(i);
```

```
nItem = listCtrl.ListItems.Add();
nItem.Text = shortcut.ShortText;
nItem.SubItems(1) = GetTypeString(shortcut.Type);
var sUser = new String(shortcut.Author);
if (sUser.length == 0)
 sUser = "System";
nItem.SubItems(2) = sUser;
// this is not an xml just show the text
nItem.SubItems(gColLongText) = formatLongText(shortcut.LongText);
nItem.Tag = shortcut;
}
return retShortcuts;
}
catch(e)
{
 alert(e.description);
}
}

function GetTypeString(iType)
{
 var sRet = "All";
 if (iType == psShortcutTypeVoice)
 sRet = "Voice";
 else if (iType == psShortcutTypeText)
 sRet = "Text";
 return sRet;
}

function TrainBtn_onclick()
{
try
{
 var selItem = Listview1.SelectedItem;
 var dlgParams = new Array();
 var shObj = selItem.Tag;
 if (shObj.Type == psShortcutTypeText)
 {
 alert("Text shortcuts cannot be trained.");
 return;
 }
 if (shObj.Global)
 {
 alert("System shortcuts cannot be trained.");
 }
 else
 {
 InitEditArg();
 dlgParams[0] = gPSDK;
```

```
 dlgParams[1] = 1;
 dlgParams[2] = shObj.ShortText;
 dlgParams[3] = false; // disable word
 dlgParams[4] = false; // disable category
gEditArg.gScript.newModalDialog('../dialogs/PhoneticTrans.htm', dlgParams, 280, 170);
 // Check if returned transcription is not empty
 if ('' != dlgParams[2])
 {
 shObj.Transcription = dlgParams[2];
gPSDK.LoadShortcut(shObj.ShortText, shObj.LongText, shObj.Type, shObj.Transcription);
 }
 }
 return;
}
catch(e)
{
 alert(e.description);
}
}

function btnInsert_onclick()
{
try
{
 var currentReport = window.dialogArguments[0].Report;
 if (currentReport == null)
 return;
 var selItem = Listview1.SelectedItem;
 var shObj = selItem.Tag;
 currentReport.MergeTemplate(shObj.LongText, psMergeSrcTypeShortcut);
}
catch(error)
{
 alert(error.description);
}
}

function formatLongText(inText)
{
try
{
 if (inText.search("<?xml") > 0)
 {
 var xmlDocLongText = new ActiveXObject("MSXML2.DOMDocument");
 xmlDocLongText.async = false;
 xmlDocLongText.loadXML(inText);
 var el = xmlDocLongText.documentElement;
 var text = el.text;
 return text;
 }
}
```

```
 }
 else
 {
 return inText;
 }
 }
 catch(e)
 {
 return inText;
 }
}
```

## Additional Shortcuts Functions

```
<script>

function AddShortcutCategory(catName)
{
 var catsCollectn = gAdminSDK.GetCategories();
 catsCollectn.Add(txtCategory.value, "New Category");
}

function AssignCategory(catName, shctName)
{
 var shctCollectn = gAdminSDK.GetShortcuts();
 for (var i = 1; i <= shctCollectn.Count; i++)
 {
 if (shctCollectn.Item(i).Name == shctName)
 {
 shctCollectn.Item(i).CategoryName = txtCategory.value;
 }
 }
}

...
</script>

<table>
<tr>
 <td nowrap>Shortcut Category:</td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="txtCategory">
 </td>
</tr>
<tr>

 <input language="javascript" type="button" name="btnAddCategory"
 value="Add Shortcut Category"
 onClick="AddShortcutCategory(txtCategory.value)">
```

```
<input language="javascript" type="button" name="btnAssignCategory"
 value="Assign Category to Shortcut"
 onClick="AssignCategory(txtCategory.value, txtShortText.value)">
</td>

 </tr>
</table>

...
</BODY>
</HTML>
```

## Words ListView

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>

 <head>
 <title>Words.htm</title>
 <meta content="Microsoft Visual Studio .NET 7.1" name="GENERATOR">
 <meta content="http://schemas.microsoft.com/intellisense/ie5"
 name="vs_targetSchema">
 <script language="javascript" id="Words" src="../scripts/Words.js">
 </script>
 <script language="javascript" id="Utils" src="../scripts/utils.js">
 </script>
 </head>

 <body language="javascript" style="BACKGROUND-COLOR: #EFE7CE"
 onload="return window_onload()" ms_positioning="GridLayout"
 bgcolor="#000000">

 <p>
 <div id="ListViewObj">
 <object id="Listview1" style="Z-INDEX: 100; LEFT: 16px; WIDTH: 95%;
 POSITION: absolute; TOP: 16px; HEIGHT: 384px"
 classid="clsid:BDD1F04B-858B-11D1-B16A-00C0F0283628" viewastext>
 <param name="_ExtentX" value="24500">
 <param name="_ExtentY" value="10160">
 <param name="SortKey" value="0">
 <param name="View" value="3">
 <param name="Arrange" value="0">
 <param name="LabelEdit" value="1">
 <param name="SortOrder" value="0">
 <param name="Sorted" value="0">
 <param name="MultiSelect" value="0">
 <param name="LabelWrap" value="-1">
 <param name="HideSelection" value="0">
 <param name="HideColumnHeaders" value="0">
 <param name="OLEDragMode" value="0">
 <param name="OLEDropMode" value="0">
 </object>
 </div>
 </p>
 </body>
</html>
```

```
<param name="AllowReorder" value="0">
<param name="Checkboxes" value="0">
<param name="FlatScrollBar" value="0">
<param name="FullRowSelect" value="-1">
<param name="GridLines" value="-1">
<param name="HotTracking" value="0">
<param name="HoverSelection" value="0">
<param name="PictureAlignment" value="0">
<param name="TextBackground" value="0">
<param name="_Version" value="393217">
<param name="ForeColor" value="-2147483640">
<param name="BackColor" value="-2147483643">
<param name="BorderStyle" value="1">
<param name="Appearance" value="1">
<param name="MousePointer" value="0">
<param name="Enabled" value="1">
<param name="NumItems" value="0">
<param name="OLEDragMode" value="0">
<param name="OLEDropMode" value="0">
</object>
</div>
<script>
 CreateObjectCtl("ListViewObj");
</script>
<input language="javascript" id="btnNew" style="Z-INDEX: 101;
LEFT: 16px; WIDTH: 64px; POSITION: absolute; TOP: 408px;
HEIGHT: 24px" onclick="return btnNew_onclick()" type="button"
value="New..." name="btnNew">
<input language="javascript" id="btnTrain" style="Z-INDEX: 106;
LEFT: 96px; WIDTH: 64px; POSITION: absolute; TOP: 408px; HEIGHT: 24px"
onclick="return btnTrain_onclick()" type="button" value="Train..."
name="btnTrain">
<input language="javascript" id="btnDelete" style="Z-INDEX: 105;
LEFT: 176px; WIDTH: 64px; POSITION: absolute; TOP: 408px; HEIGHT: 24px"
onclick="return btnDelete_onclick()" type="button" value="Delete"
name="btnDelete">
<input language="javascript" id="btnClose" style="Z-INDEX: 104;
LEFT: 256px; WIDTH: 64px; POSITION: absolute; TOP: 408px; HEIGHT: 24px"
type="button" value="Close" name="btnClose" onclick="return
btnClose_onclick()">
</p>
</body>

</html>
```

## Words Script

```
//SDK Administrator, Words.js File
// Global Variables
var gWordsObj = null;
var gScript = null;
```

```
var gPSDK = null;
var gAdminSDK = null;

// Category
var psWordCategoryAll=0;
var psWordCategoryGeneral=1;
var psWordCategoryName=2;
var psWordCategoryPlace=4;
var psWordCategoryHospital=8;
var psWordCategoryGenericDrug=16;
var psWordCategoryBrandNameDrug =32;
var psWordCategoryHeading =64;
var psWordCategoryDepartmentName =128;
var psWordCategoryRemoved = 256;

// State
var psWordStateAll= 0;
var psWordStateWaitingApproval= 1;
var psWordStateApproved = 2;
var psWordStateRejected = 4;
var psWordStateAuto = 8;

// Source
var psWordSourceAll= 0;
var psWordSourceSpell= 1;
var psWordSourceTrain= 2;
var psWordSourceImport= 4;
var psWordSourceAdmin= 8;
var psWordSourceUser= 16;

// IncludeGlobals
var psGlobalTypeAll = 0;
var psGlobalTypeExclude = 1;
var psGlobalTypeInclude = 2;
var psGlobalTypeOnly = 3;

var gEditArg = null; // Object passed into Edit dialog
var xmlDoc = new ActiveXObject("Msxml2.DOMDocument.4.0");

// List view column
var gColLongText = 3;
var gColSource = 4;

document.title = "Words";

function btnNew_onclick()
{
 NewWord();
}
```

```
}

function btnTrain_onclick()
{
 TrainWord()
}

function window_onload()
{
 InitParams(window.dialogArguments.top.script);
}

function btnDelete_onclick()
{
 DeleteWord();
}

function btnClose_onclick()
{
 window.close();
}

function InitParams(script)
{
 try
 {
 if (script == null)
 return;

 gScript = script;
 gPSDK = gScript.pscribeSDK;
 gAdminSDK = gScript.adminSDK;
 gWordsObj = BuildListItem(Listview1, psWordCategoryAll,
 psWordStateAll, psWordSourceAll, psGlobalTypeInclude);
 }
 catch (e)
 {
 alert(e.description);
 }
}

function InitEditArg()
{
 // Initialize gEditArg object
 if(gEditArg == null)
 {
 gEditArg = new Object();
 }
 gEditArg.gScript = gScript;
 gEditArg.sTitle = "New Word";
```

```
gEditArg.bSave = false;
gEditArg.sName = "";
gEditArg.iCategory = psWordCategoryGeneral;
}

function NewWord()
{
try
{
InitEditArg();
var dlg = doModalDialogSizeable('WordEdit.htm',gEditArg,350,130);

// Check if Edit dialog mark for save
if(gEditArg.bSave == true)
{
 // Save into database
 var sTranscription = "";
 var bGlobal = false;

 // Add Word to Words Collection using Add() Method
 var newWord = gWordsObj.Add(gEditArg.sName, gEditArg.iCategory,
 sTranscription, psWordStateApproved, psWordSourceUser,
 bGlobal);
 // Load Word into Memory using LoadWord() method of
 // PowerscribeSDK object
 gPSDK.LoadWord(gEditArg.sName, gEditArg.iCategory,
 sTranscription);

 // Add the new item into the list view control
 var nItem = Listview1.ListItems.Add();
 // First Column = Word
 nItem.Text = newWord.Word;
 // Second Column = Category of Word based on psWordCategory
 nItem.SubItems(1) = GetTypeString(newWord.Category);
 // Third Colulmn = Author of Word
 var sUser = new String(newWord.Author);
 if(sUser.length == 0)
 sUser = "System";
 nItem.SubItems(2) = sUser;
 nItem.Tag = newWord;
}
}
catch(err)
{
 alert(err.description);
}
}

function DeleteWord()
{
```

```
try
{
 var selItem = Listview1.SelectedItem;
 var selIndex = selItem.Index;
 var selObj = selItem.Tag;

 if (selObj.Global)
 {
 alert("System words can only be deleted by an administrator.");
 }
 else
 {
 var sure = window.confirm("Are you sure want to delete this word?");
 if (!sure)
 return;
 // Delete word from collection using Remove() method of Words object
 gWordsObj.Remove(selObj);
 selItem.Tag = null;
 Listview1.ListItems.Remove(selIndex);
 }
 return;
}
catch(e)
{
 alert(e.description);
// if this word has been deleted in a previous change to collection
 if (e.number == -2146778277)
 {
 selItem.Tag = null;
 Listview1.ListItems.Remove(selIndex);
 }
}
}

function TrainWord()
{
try
{
 var selItem = Listview1.SelectedItem;
 var selObj = selItem.Tag;
 var selIndex = selItem.Index;
 var dlgParams = new Array();

 if (selObj.Global)
 {
 alert("System words cannot be trained.");
 }
 else
 {
 InitEditArg();
 }
}
```

```
gEditArg.sName = selObj.Word;
dlgParams[0] = gPSDK;
dlgParams[1] = selObj.Category;
dlgParams[2] = selObj.Word;
dlgParams[3] = false; // disable word
dlgParams[4] = false; // disable category

gEditArg.gScript.newModalDialog('../dialogs/PhoneticTrans.htm',
 dlgParams, 280, 170);

// Check if returned transcription is not empty
if ("") != dlgParams[2])
{
 selObj.Transcription = dlgParams[2];
 // Load word into memory w LoadWord() method
 // of PowerscribeSDK object
 gPSDK.LoadWord(selObj.Word, selObj.Category, selObj.Transcription);
}
}

return;
}

catch(e)
{
 alert(e.description);
}

}

function BuildListItem(listCtrl, category, state, source, globalType)
{
try
{
 var retVal;

 // Add columns: Name, Type, Author, Long Text
 var nCol = 0;
 var nWidth = 70;
 listCtrl.ColumnHeaders.Clear();

 newCol = listCtrl.ColumnHeaders.Add();
 newCol.Text = "Word"
 newCol.Width = 3 * nWidth;
 nCol++;
 newCol = listCtrl.ColumnHeaders.Add();
 newCol.Text = "Category"
 newCol.Width = 2 * nWidth;
 nCol++;
 newCol = listCtrl.ColumnHeaders.Add();
 newCol.Text = "User"
 newCol.Width = 5 * nWidth;
 gColLongText = nCol;
}
```

```
// Add items from PSAdminSDK
listCtrl.ListItems.Clear();
retVal = gAdminSDK.GetWords(category, state, source, globalType);
var nCount = retVal.Count;
xmlDoc.async = false;
for (var i = 1; i <= nCount; i++)
{
 var word = retVal.Item(i);
 nItem = listCtrl.ListItems.Add();
 nItem.Text = word.Word;
 nItem.SubItems(1) = GetTypeString(word.Category);
 var sUser = new String(word.Author);
 if (sUser.length == 0)
 sUser = "System";
 nItem.SubItems(2) = sUser;
 nItem.Tag = word;
}
return retVal;
}
catch(e)
{
 alert(e.description);
}
}

function GetTypeString(iType)
{
 var sRet = "All";
 if(iType == psWordCategoryGeneral)
 sRet = "General";
 else if (iType == psWordCategoryName)
 sRet = "Name";
 else if (iType == psWordCategoryPlace)
 sRet = "Place";
 else if (iType == psWordCategoryHospital)
 sRet = "Hospital";
 else if (iType == psWordCategoryGenericDrug)
 sRet = "Generic Drug";
 else if (iType == psWordCategoryBrandNameDrug)
 sRet = "Brand Name Drug";
 else if (iType == psWordCategoryHeading)
 sRet = "Heading";
 else if (iType == psWordCategoryDepartmentName)
 sRet = "Department Name";
 else if (iType == psWordCategoryRemoved)
 sRet = "Removed";

 return sRet;
}
```

# *Creating Users and Assigning Access Levels*

## **Objectives**

This chapter introduces users, language models, and other related objects you might use in a *PowerScribe SDK* or *SDK Administrator* application. This chapter focuses on actions you might take in an *SDK Administrator* application, where you usually create users (see [Chapter 1](#) for more on the types of applications you can develop with *SDK*):

- [Understanding Language Models and Users](#)
- [Retrieving Language Models](#)
- [Creating a User Object](#)
- [Assigning User Privilege Levels](#)
- [Setting Up and Assigning Format Profiles](#)
- [Assigning the User an Accent](#)
- [Enabling and Disabling Logging for User](#)
- [Enabling or Disabling Users](#)
- [Working with Assigned Language Model](#)
- [Working with User Acoustic Models](#)
- [Retrieving and Modifying User Information](#)
- [Removing a User](#)

# Understanding Language Models and Users

A user is an individual person who is allowed to log in to an *SDK* application. All users are automatically able to dictate reports and have the application's speech recognition engine transcribe the text, unless you assign the particular user a lower level of report access. In addition, some users can be administrators, who might have various administrative privileges to carry out tasks such as creating other users, deleting reports, and taking similar additional actions.

If the user is employing speech recognition, a system administrator usually assigns that user a particular language model. Each language model is a vocabulary for a particular specialty that the *Speech Recognition* engine understands. For instance, the *Radiology* model or the *Pathology* model would be for specific fields in medicine.

The language models are all stored on the server. When a speech recognition user logs in to a *PowerScribe SDK* application, the *SDK* loads the language model assigned to the user into the memory of the workstation running the application.

## Understanding Types of User Privileges

Each *SDK* user can be assigned two possible types of access privileges:

- Report privileges
- Administrative privileges

When you first create a user, he or she by default is a **Speech Recognition** user, with privileges to create reports, then dictate, transcribe, and edit them. All users have these capabilities unless you change their *report privileges* to restrict the user to one of two other options:

- Only creating and dictating reports (**Dictate Only**)
- Only transcribing and editing reports (**Transcriptionist**)

You can also choose to assign each user administrative privileges, making that user an **Administrator** level user. Usually, an **Administrator** level user can both create new users and modify all types of users. The **Admin** login account that is already established when you install *SDK* is the highest **Administrator** level user, called a **System Administrator**. However, you can assign particular users of any **Administrator** level access to individual capabilities, so that you are the one who ultimately determines what each administrator can control.



**Note:** The original **Admin** login account always remains capable of carrying out all tasks; you cannot modify the original account.

## Report Privileges

You can assign a user *report privileges* that determine actions the user can take on reports.

### Report Actions

- Create reports
- Dictate into reports
- Transcribe dictated audio into text
- Listen to reports
- Correct/edit reports

## Administrative Privileges

You give users access to particular types of administrative tasks by assigning *administrative privileges*. Administrative tasks fall into these categories: creating users, setting system parameters and using utilities, managing reports, and managing shortcuts and custom words.

### Creating Users

- Create and modify users
- Assign report and administrative privileges

### Setting System Parameters and Using Utilities

- Create and modify autoformat profiles using the *AutoformatProfile Customization* utility
- Run the *Recognition Monitor* utility
- Modify PS Parameter settings in the *SDK Administrator* application provided

### Managing Reports

- Delete and purge reports or change their status

### Managing Global Shortcuts and Words

- Delete or otherwise manage shortcuts available to all users
- Delete or otherwise manage custom words available to all users

# Steps to Creating Users

As the original **Admin** user or as a user with appropriate privileges, you can create new users. If you have even more privileges, you can also selectively assign administrative privileges to other users you create. To create a user, you take several major steps:

- Create a **PSAdminSDK** object
- Retrieve the collection of **LanguageModels** installed with the *SDK*
- Retrieve the **Users** collection object (already contains the original **Admin** users)
- Add the **User** to the collection

While adding the user to the collection, you:

- Assign login and password
- Set the first, middle, and last name of the user
- Assign report access rights, if applicable, to the user
- Assign administrative privileges, if applicable, to the user
- Assign a language model to the user
- Assign an **AutoformatProfile** to the user
- Assign use of realtime recognition or server based recognition
- Enable the user

## Creating the PSAdminSDK Object

Your application can manage users and language models (and the related **AdminPrivileges** and **AutoformatProfiles** objects as well) through the *SDK Administrator* objects of the *PowerScribe SDK* product. To use your *SDK Administrator* application's functionality inside your non-administrator *PowerScribe SDK* application, after you create a **PowerscribeSDK** object, you can create a **PSAdminSDK** object using the **AdminSDK** property of the **PowerscribeSDK** object:

```
objAdmin = pscribeSDK.AdminSDK;
```

This property returns an initialized and logged in **PSAdminSDK** object instance, so that you can immediately begin using the objects under the **PSAdminSDK** object in the product's object model, including all their methods, properties, and events.

## Retrieving Language Models

Usually whoever installs the *SDK* chooses to install particular language models for users. Before you can assign a language model to a user, you need to retrieve a collection of those language models installed and available to assign to users.



*Note: You assign language models only to speech recognition users.*

### Retrieving Language Models Collection

To retrieve the **LanguageModels** collection, call the **GetLanguageModels()** method of the **PSAdminSDK** object:

```
var objLangModelColl = objAdmin.GetLanguageModels();
```

To then display a list of all language models in your GUI, you can use the **Name** property to retrieve each and add it to the list while you loop through the collection using **Item()**:

```
for (i = 1; i <= objLangModelColl.Count; i++)
{
 strOptionName = objLangModelColl.Item(i).Name;
 document.form.cbLgModels.options[i]=new Option(strOptionName);
}
```

## Selecting a Language Model from the Collection

You can select a language model from the collection using the **Item()** method. The language model is returned in a single **LanguageModel** object:

```
objLangModel = objLangModelColl.Item(i);
```

You then retrieve the **LanguageID** property of the **LanguageModel** object:

```
var LangID = objLangModel.LanguageID;
```

To assign the language model to a user, you can pass this **LanguageID** to the **Add()** method of the **Users** collection object (see [Adding User to Collection and Assigning Login](#) below). You can also use this **LanguageID** property to explicitly set a **User** object's **LanguageModelID** property (see [Working with Assigned Language Model on page 355](#)).

To show a textual description of the particular language model selected for the user, you can display the **Description** property of the **LanguageModel** object:

```
alert(objLangModel.Description);
```

## Creating a User Object

To create any **User** objects, you first need a collection of **Users** to contain them. To retrieve a **Users** collection object, you call the **GetUsers()** method of the **PSAdminSDK** object. The method takes no arguments:

```
objUsersCollection = objAdmin.GetUsers();
```

## Adding User to Collection and Assigning Login

If you have not previously created any users, the **Users** collection is empty. You add users to the collection using the **Add()** method of the **Users** collection object. You pass the method all the information about the user in several arguments:

- *Name*—String containing first name of the user.
- *MiddleName*—String containing middle name or initial of the user; an empty string if the user has no middle name.
- *LastName*—String containing last name of the user.

- *LoginName*—String containing user name that the user should enter to log in.
- *Password*—String containing password the user should enter to log in.
- *BadgeID*—String containing a company badge ID; an empty string if not applicable.
- *Enabled*—Boolean. **True** if the user should have access to the application immediately. **True** by default.
- *UseRealtimeRecognition*—Boolean. **True** if the local *Speech Recognition* engine should transcribe dictation from this user immediately, **False** if dictated audio should be sent to the *Speech Recognition* server. **True** by default.
- *LanguageID*—**LanguageID** property of the **LanguageModel** object (retrieved from the **LanguageModels** collection) for the language model being assigned to this user. If you do not pass this argument, it is **0** by default.
- *ReportRights*—Type of report user; if the user is strictly an **Administrator** and cannot work with report content at all, set this argument to **psReportRightsNone**. See the table below for other options.

#### Values for ReportRights Argument or Property

ReportRights Constant	Value	User Type and Explanation
psReportRightsNone	0	Administrator not working with report content. Default.
psReportRightsVoiceRecognition	1	Speech Recognition user. Can create, transcribe, and correct reports.
psReportRightsDictateOnly	2	Dictation Only user. Can create and dictate reports only. Usually sends the audio to a transcriptionist.
psReportRightsTranscription	3	Transcriptionist. Cannot create reports. Can play back, transcribe, and correct reports.

- *AdminPrivilege*—**AdminPrivilege** object, whose property settings assign administrative privileges. For more on this object, see [Granting Administrative Privileges on page 349](#). Set to **0** by default.
- *AutoFormatProfileID*—ID of an **AutoformatProfile** object that defines how to automatically format all reports this user creates. For more information, see [Setting Up and Assigning Format Profiles on page 351](#). Set to **-1** by default.
- *EnableLogging*—**True** to turn on logging for this user, **False** to turn it off. **False** by default.

A call of the **Add()** method that adds a user named *John Q. Public* with no badge ID, who starts out able to log in (enabled), has **Speech Recognition** report rights with speech recognized in real time, might look like this:

```
var cEnableUser = true;
var cRealTime = true;
```

```
objUsersCollection.Add("John", "Q", "Public", "jqpublic", "jqp987", "",
cEnableUser, cRealTime, LangID, psReportRightsVoiceRecognition,
objAdminPrivlg, objAutoformatProfiles.Item(6).ProfileID, True);
```

Next, let's find out more about constants you can pass to the **Add()** method to assign report rights, and objects you can pass to the method to assign administrative privilege levels ([Assigning User Privilege Levels on page 349](#)) and automatic report formatting profiles ([Setting Up and Assigning Format Profiles on page 351](#)).

## Assigning User Privilege Levels

Each *SDK* user can be assigned report privileges and/or specific administrative privileges.

### Granting Report Access Privileges

Each user has only one level of access to reporting capabilities in any *PowerScribe SDK* dictation application (see the table below). To set the user's level of reporting privileges, you set the **ReportRights** property of the **User** object to the most appropriate value in the table of [Values for ReportRights Argument or Property on page 348](#).

To give the user the *right* to create reports by dictating and using speech recognition, you set the **ReportRights** property of the **User** object as follows:

```
var psReportRightsVoiceRecognition = 1;
objUser.ReportRights = psReportRightsVoiceRecognition;
```

To set up a user restricted to playing back dictation of a report and revising the text, you set the **ReportRights** property of that **User** object to **psReportRightsTranscription**:

```
var psReportRightsTranscription = 3;
objUser.ReportRights = psReportRightsTranscription;
```

If you have users that you want to restrict to only dictating and never editing or correcting their own reports, you can set the **ReportRights** property of the **User** object as follows:

```
var psReportRightsDictateOnly = 2;
objUser.ReportRights = psReportRightsDictateOnly;
```

To deny the user any right to create or modify reports, you set the **ReportRights** property of the **User** object to **psReportRightsNone**:

```
var psReportRightsNone = 0;
objUser.ReportRights = psReportRightsNone;
```

### Granting Administrative Privileges

If you are creating an application used mainly for dictation, many users might not need any administrative privileges or you might grant them somewhat limited administrative

privileges. The table that follows shows all administrative capabilities in the *SDK*. You can grant or deny a user access to any or all of these capabilities. You grant access by setting the corresponding property of the **AdminPrivileges** object to **True** (all are **False** by default).

### AdminPrivileges Object Properties

Property Name	Capabilities To
AccessRecognitionMonitor	Monitor tasks being performed by the <i>Speech Recognition Server</i> , accessed at this URL: <b>http://&lt;servername&gt;/pscribesdk/RecMonitor/</b>
AutoformatProfileCustomization	Access the <i>Speech Recognition AutoformatProfile Customization</i> tool, accessed at this URL: <b>http://&lt;servername&gt;/pscribesdk/AFCustom/</b>
CreateModifyAdmins	Create, modify, or delete <b>Administrator</b> level users, who can create, modify, or delete other <b>Administrator</b> level users as well as users with report rights, but cannot create, modify, or delete <b>System Administration</b> users. Can also take other administrative actions if assigned access to those capabilities through associated properties. (No user can delete the original <b>Admin</b> login account.)
CreateModifySystemAdmins	Create, modify, or delete <b>System Administrator</b> users.
CreateModifyUsers	Create, modify, or delete <i>PowerScribe SDK</i> application users who have report rights.
ManageParameters	Modify the settings of the <i>PS Parameters</i> through an <i>SDK Administrator</i> application.
ManageReports	Take actions on reports, such as deleting them, through an <i>SDK Administrator</i> application. Otherwise, only the user who creates the report can delete it.
ManageShortcuts	Take actions on shortcuts through an <i>SDK Administrator</i> application. Those actions include creating and deleting shortcuts.
ManageWords	Take actions on custom words through an <i>SDK Administrator</i> application. Those actions include creating and deleting custom words.

To set any of these **AdminPrivileges** properties, you first create an **AdminPrivileges** object using the **AdminPrivileges** property of the **User** object:

```
objAdminPriv = objUser.AdminPrivileges;
```

You can then allow the user access to any capability by setting the associated **AdminPrivileges** property in the table to **True**. For instance, to allow the user access to

the *Recognition Server Monitor*, you set the **AccessRecognitionMonitor** property of the **AdminPrivileges** object to **True**:

```
objAdminPriv.AccessRecognitionMonitor = true;
```

To allow the user to modify other users, you would set the **CreateModifyUsers** property of the **AdminPrivileges** object to **True**:

```
objAdminPriv.CreateModifyUsers = true;
```

## Setting Up and Assigning Format Profiles

Format profiles are unique sets of predefined formatting rules that the *SDK* automatically applies to reports created by any user assigned that format profile. In *SDK* these formatting profiles are called *AutoformatProfiles* or *PostProfiles* because the product applies them automatically and applies them after the report is complete.

### Creating Custom Format Profiles

You create custom report formatting profiles by running the *Speech Services AutoformatProfile Customization* tool of the *SDK*, available at the following URL:

<http://<servername>/pscribesdk/AFCustom/>

The aspects of the report that an **AutoformatProfile (PostProfile)** can format are:

- Number, date, and time formats (such as European and North American date and time)
- Numbered list formats (number style and punctuation) and heading or department name capitalization (title style or all caps)
- Drug name capitalization (standard, initial caps, or all caps with optional of all caps after *Allergies*)
- Capitalization and expansion of special words and phrases (CODE BLUE in all caps or expansion of DNI to DO NOT INTUBATE) and of abbreviations like cc and q.d.
- General format options, such as type of symbol for the word *dash*; abbreviation of feet, inches, and other measurements; Roman numerals for types of diabetes, and so on

### Retrieving Collection of Format Profiles

To retrieve an object for a collection of all the **AutoformatProfiles** that have been created, you call the **GetPostProfiles()** method of the **PSAdminSDK** object:

```
objFmtProfilesColl = objAdmin.GetPostProfiles();
```

If no **AutoformatProfiles** are defined, the method returns an empty collection.

## Assigning Format Profile to User

After you retrieve the collection, you can retrieve the object for a specific format profile by looping through the collection and finding the one whose **Name** property matches the one you want to assign:

```
for (i = 1; i <= objFmtProfilesColl.Count; i++)
{
 if (objFmtProfilesColl.Item(i).Name == strFormatRequested)
 objFmtProfile = objFmtProfilesColl.Item(i);
}
```

You could then pass that profile object's **ProfileID** property as the *AutoFormatProfileID* argument of the **Add()** method of the **User** object (bolded argument below):

```
var cEnableUser = true;
var cRealTime = true;

objUsersCollection.Add("John", "Q", "Public", "jqppublic", "jqp987", "",
cEnableUser, cRealTime, LangID, psReportRightsVoiceRecognition,
objAdminPrivlg, objFmtProfile.ProfileID, True);
```

Or you could explicitly set the **AutoFormatProfileID** property of a particular **User** object using the **ProfileID** property of the profile, then call the **Update()** method of the **User** object to force the property value to take effect immediately:

```
objUser.AutoFormatProfileID = objFmtProfile.ProfileID
objUser.Update();
```

## Assigning the User an Accent

If the user speaks with an accent, such as **Indian English**, you can assign the user the appropriate accent from the read-only collection of accents provided by *PowerScribe SDK*.

### To assign the user an accent:

1. Retrieve the **UserAccents** collection object by calling the  **GetUserAccents()** method of the **PSAdminSDK** object:

```
objUserAccentsCollectn = objAdmin.GetUserAccents();
```

2. Loop through the collection looking for the **Name** property for the particular **UserAccent** from the collection that matches the name of the accent requested:

```
for(i = 0, i >= objUserAccentsCollectn.count; i++)
{
 if(objUserAccentCollectn.Item(i).Name == strAccentNameRequested)
 {
```

3. When you find the matching accent, retrieve the **UserAccent** object for the **UserAccent** associated with that name:

```
objUserAccent = objUserAccentCollectn.Item(i);
```

4. Assign the **User** object's **AccentID** property the value of the **UserAccent** object's **ID** property, the long numeric identifier for that accent:

```
objUser.AccentID = objUserAccent.ID;
break;
}
}
```

5. You can also display the name of the **UserAccent** using the object's **Name** property:

```
alert(objUserAccent.Name);
```

## Enabling and Disabling Logging for User

If you enable logging for a particular user, the *SDK* produces a log file that shows all the API calls the application has made for each login session and stores them in the following location:

**C:\Documents and Settings\<username>\Local Settings\Application Data\Dictaphone\HSG\Trace\PowerScribeSDK\**

To enable or disable logging for the user, you first retrieve the **User** object.

### To retrieve the User object for a particular user:

1. Retrieve the **Users** collection from the database by calling the **GetUsers()** method of the **PSAdminSDK** object:

```
objUsersColl = objAdmin.GetUsers();
```

2. Retrieve the **User** object for a user by looping through the **Users** collection with the **Item()** method of the **Users** object and finding the user with a matching **LoginName** property setting (you might also check **Name** and **LastName** properties):

```
for (i = 1; i <= objUsersColl.Count, i++)
{
 if (currUserLogin == objUsersColl.Item(i).LoginName)
 objUser = objUsersColl.Item(i);
}
```

**Once you have the User object, logging is disabled for that user until you enable it:**

1. Enable logging for the user by setting the **EnableLogging** property of that **User** object to **True**, then calling the **Update()** method to make the change take effect:

```
objUser.EnableLogging = true;
objUser.Update();
```

2. After you have enabled logging for a user, you can later disable logging for that user by setting the **EnableLogging** property of that **User** object to **False**, then calling the **Update()** method to make the change take effect:

```
objUser.EnableLogging = false;
objUser.Update();
```



**Note:** If you set the **EnableLogging** property of the **PowerscribeSDK** object to **True**, it overrides the setting of the **EnableLogging** property of the **User** object.

## Enabling or Disabling Users

Any user with administrative privileges to modify other users can lock particular users in the collection out of any *PowerScribe SDK* applications that they can currently access.

When you first create the user with the **Add()** method, you usually pass **True** for the **Enabled** argument, although you could also pass **False** for that argument and enable the user later by setting the **Enabled** property of the **User** object to **True**:

```
objUser.Enabled = True;
```

The change to any **User** object property takes effect only after you call the **Update()** method:

```
objUser.Update();
```

To lock a user out of any *PowerScribe SDK* applications he or she can currently access, you set the **Enabled** property of the user to **False**, then update the **User** object with **Update()**:

```
objUser.Enabled = False;
objUser.Update();
```

Now, the user can no longer log in to any *SDK* applications unless you change the **Enabled** property back to **True**.

# Working with Assigned Language Model

You can work with the language model assigned to the user. For instance, you might want to modify the language model of the currently logged in user or display the name of the language model assigned to the currently logged in user.

## Modifying Language Model Assigned

To change the language model assigned to the user, you first select the particular model from the collection (here it's retrieved based on its index in the collection):

```
var selectedIndex;
var objSelectedLangModel = objLangModelColl.Item(selectedIndex);
```

You can then set the **LanguageModelID** property of the **User** object to the **LanguageID** property of the **LanguageModel** object from the collection:

```
objUser.LanguageModelID = objSelectedLangModel.LanguageID;
```

After you change any property setting for a user, the change does not take effect until you call the **Update()** method of the **User** object:

```
objUser.Update();
```

## Displaying Language Model Assigned

You can retrieve and perhaps display the name of the language model that has been assigned to the logged in user by first retrieving that user's **LanguageModelID**, then find a model in the collection whose **LanguageID** property matches it by looping through the **LanguageModels** collection.

You could retrieve that language model's **LanguageID** and **Description** properties and display that information:

```
for (i = 1; i <= objLangModelColl.Count; i++)
{
 if (objUser.LanguageModelID == objLangModelColl.Item(i).LanguageID)
 {
 strLangModelInUse = objLangModelColl.Item(i).LanguageID
 strLMdescription = objLangModelColl.Item(i).Description
 }
}
// Pass the strLangModelInUse, strLMdescription to GUI for display
```

# Working with User Acoustic Models

*PowerScribe SDK* automatically adapts a standard voice model to the speech of each individual user, producing what is called an *acoustic model*. The model also adjusts for the particular type of audio device the user speaks into, as the sound varies from device to device. Each individual user has a different acoustic model for each audio device he or she uses. For instance, a BlueTooth device might detect the user's voice differently from how a *PowerMic II* does. You can access each acoustic model using the **AcousticModel** object.

You start by retrieving an **AcousticModels** collection for a particular user, returned by the **AcousticModels** property of that **User** object:

```
objAcousticModsColl = objUser.AcousticModels;
```

You can then retrieve a particular **AcousticModel** object from the collection based on its **Type** property, which indicates whether the type of audio device associated with the acoustic model is hand held, telephony, BlueTooth, or Array (see table of equivalent **AcousticModelType** constants):

```
for (i = 0; i >=
objAcousticModsColl.count; i++)
{
 if (objAcousticModsColl.item(i).Type = psAcousticModelTypeBlueTooth)
 objAcousticMod = objAcousticModsColl.item(i);
}
```

AcousticModelType Constant	Value
psAcousticModelTypeHandHeld	1
psAcousticModelTypeTelephony	2
psAcousticModelTypeBlueTooth	3
psAcousticModelTypeArray	4

Once you have the **AcousticModel** object, you can find the values of its properties to determine:

- Name and description of the acoustic model
- Last date the acoustic model was modified
- How much training the user has completed with the associated audio device

To determine the name and description of the acoustic model, you can use the **Name** and **Description** properties of the **AcousticModel** object:

```
var strAMname = objAcousticMod.Name;
var strAMdesc = objAcousticMod.Description;
```

The **AcousticModel** object's **Name** property can return one of several US English model names, shown in the table that follows.

<b>AcousticModel Name String</b>	<b>Language and Device That This Model Uses to Retrieve Greatest Accuracy</b>
BestMatch III Medical	For US English speakers dictating into a 11 KHz to 99 KHz headset or handheld microphone/recording device using medical vocabularies, then having the audio transcribed on the server.
Bluetooth 8kHz	For US English speakers dictating into an 8 KHz Bluetooth device and transcribing the audio from it.
Medical Telephony	For US English speakers dictating into an 8 KHz to 99 KHz hand-held digital recorder or dictating text over the telephone using medical vocabularies, then having the audio transcribed on the server.
BestMatch III Medical Indian	For US English speakers with an Indian accent dictating into a 11 KHz to 99 KHz headset or handheld microphone/recording device using medical vocabularies, then having the audio transcribed on the server.
Indian Medical Telephony	For US English speakers with an Indian accent dictating into an 8 KHz to 99 KHz hand-held digital recorder or dictating text over the telephone using medical vocabularies, then having the audio transcribed on the server.
BestMatch Array	For US English speakers dictating into a 11 KHz to 99 KHz Array microphone, then having the audio transcribed on the server.

To find out the last date that the acoustic model was modified, you can use the **LastModDate** property of the **AcousticModel** object:

```
var strAMdatemodified = objAcousticMod.LastModDate;
```

To find out how much training the user has completed with the associated audio device, you can use the **TrainingState** property of that particular **AcousticModel** object of that user:

```
var strAMtrainState = objAcousticMod.TrainingState;
```

In your custom *SDK* administrator application, you might display the name/description of the acoustic model, the last date and time it was modified, and the level of training the user has completed for that model, for the administrator's information. When you display the information, using the strings the corresponding properties return, the data would appear in the following formats:

- **AcousticModel.Name** = “BestMatch III Medical”
- **AcousticModel.Description** string = “Enroll Voice Model for User: 2”
- **AcousticModel.LastModDate** string = “07/07/2007 14:42:39”
- **AcousticModel.TrainingState** integer = 6

For more on how you can use the **TrainingState** property, refer to [Chapter 13, Creating Custom Training Module](#). For information on the strings associated with numeric values of the **AcousticModel.TrainingState** property, see the **UserProfile.TrainingState** property in

the *PowerScribe SDK API Help*. For a list of the strings that the *SDK* sets the **AcousticModel.Description** property to, refer to the **UserProfile.TrainingData** property values in [Determining Other Privileges of User on page 42](#).

# Retrieving and Modifying User Information

In your *PowerScribe SDK* application, you might want to check the value of certain user properties before taking particular actions. You can check the values of those properties in your application, and if they are not read-only, you can change the settings of the properties. You can also let users with appropriate access privileges modify some **User** object properties.

## Retrieving the User Object

You might want to change one or more properties of the **User** object. Before you can set any properties of the user, you need to retrieve the **User** object for the currently logged in user.

### To retrieve the User object for the currently logged in user:

1. Retrieve the **Users** collection from the database by calling the **GetUsers()** method of the **PSAdminSDK** object:

```
objUsersColl = objAdmin.GetUsers();
```

2. Retrieve the **User** object for the user by looping through the **Users** collection with the **Item()** method and finding the user with a matching **LoginName** property setting:

```
for (i = 1; i <= objUsersColl.Count, i++)
{
 if (strUserToChange == objUsersColl.Item(i).LoginName)
 objUser = objUsersColl.Item(i);
}
```

## Changing User Password

### Once you have the User object, to change the password for the user:

Set the **Password** property of the **User** object, then call the **Update()** method of the **User** object to make the change take effect immediately:

```
objUser.Password = newpassword.text;
objUser.Update();
```

## Changing User Name and Badge Information

**Once you have the User object, to change the name or badge ID of the user:**

Set the **Name**, **MiddleName**, **LastName**, and **BadgeID** properties of that **User** object, then call the **Update()** method to have the change take effect immediately:

```
objUser.Name = "Jonathan"
objUser.MiddleName = "T."
objUser.LastName = "Pubman"
objUser.BadgeID = "JTP-00879"
objUser.Update();
```

## Changing User Report Capabilities

**Once you have the User object, to assign the user a different type of report capability:**

Set the **ReportRights** property of the **User** object, then call the **Update()** method of the **User** object to force the change to take effect immediately:

```
objUser.ReportRights = psReportRightsDictateOnly;
objUser.Update();
```

## Changing User Administrative Capabilities

**Once you have the User object, to grant or take away an administrative privilege:**

1. Use the **AdminPrivilege** property of the **User** object to retrieve its associated **AdminPrivileges** object:  

```
objAdminPrivileges = objUser.AdminPrivilege;
```
2. Set the **AdminPrivileges** object property that corresponds to the type of administrative privilege you want to grant to the user to **True**; then call the **Update()** method of the **User** object. For instance, to allow the user to purge reports, you would set the **ManageReports** property to **True**:  

```
objAdminPrivileges.ManageReports = true;
objUser.Update();
```

## Switching to Batch Speech Recognition (on Server)

**Once you have the User object, to switch from realtime speech recognition to batch processing of report audio on the Recognition Server:**

Set the **UseRealTimeRecognition** property of the **User** object to **False**; then call the **Update()** method of the **User** object:

```
objUser.UseRealTimeRecognition = false;
objUser.Update();
```

# Removing a User

To remove a user from the database who no longer needs access to your application, you remove the user from the **Users** collection:

1. Retrieve the the **Users** collection object by calling the **GetUsers()** method of the **PSAdminSDK** object:  
`objUsersColl = objAdmin.GetUsers();`
2. If you have the login name of the user to remove, you can call the  **GetUser()** (singular user) method of the **PSAdminSDK** object and pass it the login name of the user:  
`objUserToRmv = objAdmin.GetUser(strLogin);`
3. Call the **Remove()** method of the **Users** collection object. You pass the method the **User** object of the individual user to remove from the collection:  
`objUsersColl.Remove(objUserToRmv);`
4. Be sure to handle any errors that occur, shown in the table below:

#### Errors Returned by Remove() Method

Error Code	Explanation
0x800A73C	The user does not exist in the system.
0x800AC73C	Error removing user.

# Code Summary

```
<HTML>
<HEAD>
<TITLE>SDK Administrator</TITLE>
<!-- This code summary is not a complete program; it assumes
inclusion of required code from the previous chapter -->
<SCRIPT type="text/javascript">

var strLoginNameOfUserBeingDefined;
var LangID;

var psReportRightsNone = 0;
var psReportRightsVoiceRecognition = 1;
var psReportRightsDictateOnly = 2;
var psReportRightsTranscription = 3;
```

```
function InitializeAdminApp()
{
 try
 {
 // Instantiate PowerscribeSDK Object
 pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");

 // Instantiate Admin SDK Object
 objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");

 // Initialize Admin SDK SDK Server
 var url = "http://server2/myAdminSDK/";
 objAdmin.Initialize(url, "UserConfigs");

 // Create EventMapper Object for Admin SDK Object
 AdminSink = new ActiveXObject("PSAdminSDK.EventMapper");

 // Map Handlers for Each Event to SDK EventMapper Object
 AdminSink.EndExport = HandleEndExport;
 AdminSink.EndImport = HandleEndImport;
 SDKsink.Advise(objAdmin);
 }
 catch(error)
 {
 alert("InitializeAdmin: " + error.number + error.description);
 }
 finally
 {
 DoLogin();
 }
}

function DoLogin()
{
 try
 {
 objAdmin.Login(user.text, passwd.text);
 }
 catch(error)
 {
 // If the user is already logged in (is dangling logged in user)
 if (error == -2146778148)
 {
 objAdmin.login("admin", "");
 objAdmin.LogoffUser(user.text, passwd.text);
 objAdmin.Logoff();
 objAdmin.Login(user.text, passwd.text);
 }
 else
 {
 // Handle Other Errors
 }
 }
}
```

```
 }
 finally
 {
 window.open("main.htm", "_self", "height=100, width=50,
 status=yes", true);
 }
 }

// Function to Build list of Language Models for GUI
function BuildLangModelsList()
{
 objLangModelsColl = objAdmin.LanguageModels;
 for (i = 1; i <= objLangModelsColl.Count; i++)
 {
 strLangModOption = objLangModelColl.Item(i).LanguageID;
 document.form.cbLangModels.Options[i] = new Option(strLangModOption);
 }
}

// Function to Add User to Database on click of Add User button
function AddUserToDatabase()
{
 objUsersColl = objAdmin.GetUsers();

 objUser = objUsersColl.Add(txtFirst.value, txtMid.value, txtLast.value,
 txtLogin.value, txtPasswd.value, txtBadge.value);

 strLoginNameOfUserBeingDefined = txtLogin.value;
 BuildUsersList();
}

// Function to Build list of Users for GUI
function BuildUsersList()
{
 objUsersColl = objAdmin.Users;
 for (i = 1; i <= objUsersColl.Count; i++)
 {
 strUserOption = objUsersColl.Item(i).Name;
 document.form.cbUsersList.Options[i] = new Option(strUserOption);
 }
}

function AssignUserAccess()
{
 RetrieveAndAssignLangModel();
 AssignAdminAndReportPrivils();
 SetOtherUserProperties();
}
```

```
// Function to retrieve LanguageModel selected & assign to user
function RetrieveAndAssignLangModel()
{
 for (i = 1; i <= objLangModelsColl.Count; i++)
 {
 if (cbLangModels.options.value == objLangModelColl.Item(i).LanguageID)
 LangID = objLangModelColl.Item(i).LanguageID;
 alert(objLangModelColl.Item(i).Description);
 }

 objUser = objAdmin.GetUser(strLoginNameOfUserBeingDefined);
 objUser.LanguageModelID = LangID;
}

// Function to Create Report Access Privileges Option List
function BuildReportPrivList()
{
 document.form.cbRptPrivs.Options[0] = new Option("None");
 document.form.cbRptPrivs.Options[1] = new Option("Speech Recognition");
 document.form.cbRptPrivs.Options[2] = new Option("Dictate Only");
 document.form.cbRptPrivs.Options[3] = new Option("Transcription Only");
}

// Function to Build AutoformatProfile(PostProfile) Option List
function BuildPostProfileList()
{
 objFmtProfilesColl = objAdmin.GetPostProfiles();
 for (i = 1; i <= objFmtProfilesColl.Count; i++)
 {
 document.form.cbPProfs.Options[i] =
 new Option(objFmtProfilesColl.Item(i).Name);
 }
}

// Function to Assign User Privileges Based on GUI Selections
function AssignAdminAndReportPrivils()
{
 // Retrieve an AdminPrivileges object for the user
 objUser = objAdmin.GetUser(strLoginNameOfUserBeingDefined);
 objAdminPriv = objUser.AdminPrivilege;

 // Set the properties of the AdminPrivileges object for this user
 objAdminPriv = objAdmin.AdminPrivileges;
 objAdminPriv.AccessRecognitionMonitor = chkbxRecMon.value;
 objAdminPriv.AutoformatProfileCustomization = txtPProfs.value;
 objAdminPriv.CreateModifyAdmins = chkbxAddAdmins.value;
 objAdminPriv.CreateModifySystemAdmins = chkbxAddSysAdmins.value;
 objAdminPriv.CreateModifyUsers = chkbxAddUsers.value;
 objAdminPriv.ManageParameters = chkbxParams.value;
 objAdminPriv.ManageReports = chkbxManageRpts.value;
}
```

```
objAdminPriv.ManageShortcuts = chkbxManageShcts.value;
objAdminPriv.ManageWords = chkbxManageWrds.value;
// Assign Speech Recognition privileges to this user
objUser.ReportRights = cbRptPrivs.options.value;
return;
}

// function to Assign Other User Properties
function SetOtherUserProperties()
{
 objUser.Enabled = chkbxEnabled.value;
 objUser.UseRealTimeRecognition = chkbxRealTime.value;
 objUser.LoggingEnabled = chkbxLoggingEnabled.value;
 objUser.AutoFormatProfileID = SetPostProfID(cbPProfs.option.value);
}

function SetPostProfID(profileName)
{
 var objFmtProfsColl = objAdmin.GetPostProfiles();
 for (i = 0; i <= objFmtProfsColl.Count; i++)
 {
 if (objFmtProfsColl.Item(i).Name == profileName)
 var objProfileID = objFmtProfsColl.Item(i).ProfileID;
 }
 return ProfileID;
}

// Function to Present List of Accents
function PresentAccentList()
{
 objUserAccentsCollectn = objAdmin.GetUserAccents();
 for (i = 1; i <= objUserAccentsCollectn.count; i++)
 {
 strAccent = objUserAccentsCollectn.Item(i).Name;
 document.form.cbAccents.Options[i] = new Option(strAccent);
 }
}

// Function to Assign Accent to user
function FindMatchingAccent()
{
 for(i = 0, i <= objUserAccentsCollectn.count; i++)
 {
 if(objUserAccentCollectn.Item(i).Name == cbAccents)
 {
 objUserAccent = objUserAccentCollectn.Item(i);
 objUser.AccentID = objUserAccent.ID;
 break;
 }
 }
}
```

```
// Function to return the acoustic model name
// for a particular user's particular acoustic model number
GetUserAcousticModel(givenUser, givenModNum)
{
 objUsersColl = objAdmin.GetUsers();
 for (i = 0; i <= ObjUsersColl.count; i++)
 {
 if (objUsersColl.Item(i).LoginName == givenUser
 {
 objUser == objUsersColl.Item(i)
 break;
 }
 var objAMcoll = (objUser.AcousticModels);
 var objAM = objAMcoll.Item.(givenModNum - 1);
 alert("Acoustic Model Name: " + objAM);
 }

 // Event Handlers

 function HandleEndExport(strFileLoc, intErrNum, strErrMsg)
 {
 if (strFileLoc != "")
 // Show location of file in status bar
 window.status = "Exported Voice Models File Location: " + strFileLoc;
 else
 // Show error message in status bar
 window.status = "Export of Voice Models Failed: " + strErrMsg;
 }

 function HandleEndImport(intErrNum, strErrMsg)
 {
 if (intErrNum == 0)
 // Show location of file in status bar
 window.status = "Imported Voice Models Successfully";
 else
 // Show error message in status bar
 window.status = "Import of Voice Models Failed: " + strErrMsg;
 }

 function DoLogoff()
 {
 try
 {
 objAdmin.Logoff();
 Cleanup();
 }
 catch(error)
 {
 alert(error.description);
 }
 }
}
```

```
function cleanup()
{
 try
 {
 if (AdminSink)
 {
 AdminSink.Unadvise();
 }
 }
 catch(error)
 {
 alert("Cleanup Error: " + error.description);
 }
}

</SCRIPT>
</HEAD>

<BODY bgcolor="#fffff" language="javascript" onload="InitializeAdminApp()"
onunload="DoLogoff()">

<form>

<!-- Examples of text boxes to retrieve input/buttons to take actions -->
...
<p><center>Enter Basic User Information</center></p>

<!-- Text boxes to retrieve basic User Settings -->
<tr>
 <td nowrap>First Name:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="txtFirst">
 </td>
 <td nowrap>Middle Name:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="txtMid">
 </td>
 <td nowrap>Last Name:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="txtLast">
 </td>
 <td nowrap>Login Name:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"</pre>
```

```
 name="txtLogin">
 </td>

 <td nowrap>Password:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="txtPasswd">
 </td>

 <td nowrap>Badge Number:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="txtBadge">
 </td>
</tr>

<tr>
 <input language="javascript" type="button" name="btnAddUser"
 value="Add User" onClick="AddUserToDatabase () ">

</tr>

...
<p><center>Assign Language Model & Access Privileges</center></p>
<tr>
 <td nowrap>Language Model:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="cbLangModels">
 </td>

 <td nowrap>Report Privileges:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="cbRptPrivils">
 </td>
</tr>

<p><center>Assign Accent</center></p>
<tr>
 <td nowrap>Accent:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="cbAccents">
 </td>
</tr>
```

```
<p><center>Display Acoustic Model of User</center></p>
<tr>
 <td nowrap>Login Name:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="txtUserAM">
 </td>
 <td nowrap>Acoustic Model Number:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="txtAMnumber">
 </td>
</tr>
<tr>
 <td colspan="4" style="text-align: center;>
 <input language="javascript" type="button" name="btnGetAcousticModel"
 value="Get Acoustic Model Name"
 onClick=" GetUserAcousticModel(txtUserAM, txtAMnumber)">

 </td>
</tr>

<tr>
 <td>
 <input type=checkbox name="chkbxRecMon" value="Access Recognition
 Monitor">Access Recognition Monitor
 </td>
 <td>
 <input type=checkbox name="chkbxAddUsers" value=
 "Add Speech Recognition & Report Users ">Add Speech Recognition
 & Report Users
 </td>
 <td>
 <input type=checkbox name="chkbxAddAdmins" value="Add Admin Users">
 Add Admin Users
 </td>
 <td>
 <input type=checkbox name="chkbxAdSysAdmins" value="Add System Admin
 Users">Add System Admin Users
 </td>
 <td>
 <input type=checkbox name="chkbxParams" value=
 "Modify Parameter Settings">Modify Parameter Settings
 </td>
 <td>
 <input type=checkbox name="chkbxManageRpts" value="Manage Reports">
 Manage Reports
 </td>
</tr>
```

```
<td>
 <input type=checkbox name="chkbxManageShcts" value="Manage Shortcuts">
 Manage Shortcuts
</td>

<td>
 <input type=checkbox name="chkbxManageWrds" value="Manage Words">
 Manage Words
</td>

</tr>
...
<p><center>Assign Other User Property Values</center></p>

<tr>
 <td>
 <input type=checkbox name="chkbxEnabled" value="Enable User">
 Enable User
 </td>

 <td>
 <input type=checkbox name="chkbxRealTime" value=
 "Realtime Speech Recognition">Realtime Speech Recognition
 </td>

 <td nowrap>Autoformat Profile for Report Format:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="cbPProfs">
 </td>

 <td>
 <input type=checkbox name="chkbxEnableLogging" value=
 "Enable Logging">Enable Logging
 </td>
</tr>

<tr>
 <td>
 <input language="javascript" type="button" name="btnAssignAccent"
 value="Assign User Accent"
 onClick="FindMatchingAccent()">

 </td>
</tr>

<tr>
 <td>
 <input language="javascript" type="button" name="btnAssignUserAccess"
 value="Assign User Access & Property Values"
 onClick="AssignUserAccess()">

 </td>
</tr>
```

```
<input language="javascript" type="button" name="btnLogin"
 value="Login" onClick="DoLogin()">

<input language="javascript" type="button" name="btnLogout"
 value="Logout" onClick="DoLogoff()">

...
</form>
...
</BODY>
</HTML>
```

# *Creating Groups in Your Application*

## **Objectives**

In this chapter, you learn how to create groups to associate related shortcuts with:

- [Understanding Groups](#)
- [Overview of Creating Groups](#)
- [Creating and Populating Groups](#)
- [Creating the PSAdminSDK Object](#)
- [Creating a Groups Collection Object](#)
- [Adding Group Objects to the Groups Collection](#)
- [Adding Shortcuts to a Group](#)
- [Retrieving a Particular Group](#)
- [Modifying Name/Description of the Group](#)
- [Determining Who Owns the Group](#)
- [Determining Whether Group Contains Shortcuts Available to All Users](#)
- [Removing Shortcuts from a Group](#)
- [Removing a Group from the Collection](#)

# Understanding Groups

You can associate related or similar shortcuts with each other by organizing them into **Groups**. Although each shortcut can belong to only a single **Category**, that same shortcut can belong to more than one **Group**.

For instance, if you have groups named **Head**, **Neck**, **Pelvis**, **Skull**, **Abdomen**, and **Chest**, a shortcut that belongs to **Pelvis** could also belong to **Abdomen** and a shortcut that belongs to **Head** could also belong to **Skull**.

## Overview of Creating Groups

Your application needs to handle multiple phases of action to include groups:

- Create Groups—Develop a way for the user to create and edit groups
- Add Shortcuts to a Group—Add shortcuts to the appropriate groups

In addition, you might want the application to provide users the ability to:

- Modify the name of the group
- Modify the description of the group
- View the owner/creator of the group
- Determine whether or not a group is available to all users
- Remove shortcuts from a group, but keep the shortcuts in the system

## Creating and Populating Groups

### Steps to Creating Groups

You take several steps to create shortcuts for users of your application:

- Create a **PSAdminSDK** object for later use in creating and working with **Group** and **Groups** objects (for a brief introduction to the **PSAdminSDK** object, refer to [Types of Applications You Can Develop on page 3](#))
- Create a **Groups** collection to contain all individual **Group** objects
- Add **Group** objects to the collection
- Add **Shortcut** objects to each **Group**

# Creating the PSAdminSDK Object

In [Chapter 1](#) you were introduced to the types of applications you can develop with *SDK*:

- *PowerScribe SDK* applications, for dictation users, using the **PowerscribeSDK** object
- *SDK Administrator* applications, for administrators, using the **PSAdminSDK** object

For a high-level overview of the types of applications you can create with *SDK*, refer to [Chapter 1](#), under [Types of Applications You Can Develop on page 3](#).

If your application manages shortcuts, either as an *SDK Administrator* application based on the **PSAdminSDK** object or as *PowerScribe SDK* application based on the **PowerscribeSDK** object, it can also manage **Groups** for that help organize those shortcuts.

In an *SDK Administrator* application based on the **PSAdminSDK** object, you retrieve the **PSAdminSDK** object using the **new** command in JavaScript:

```
objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");
```

In a *PowerScribe SDK* application deploying *SDK Administrator* functionality, you create a **PSAdminSDK** object using the **AdminSDK** property of the **PowerscribeSDK** object:

```
objAdmin = pscribeSDK.AdminSDK;
```

# Creating a Groups Collection Object

To create a **Groups** collection object, you call the **GetGroups()** method of the **PSAdminSDK** object and pass it two values:

- *includeGlobals*—Whether or not to include global groups (available to all users)—to include all groups, exclude or include global groups, or get only global **Groups**.

## Values for the IncludeGlobals Argument

IncludeGlobals	Value	Explanation
psGlobalTypeAll	0	Includes both all global groups (available to all users) and all individual user groups belonging to <i>all</i> users.
psGlobalTypeExclude	1	Includes only individual groups for the logged in user, not those created strictly for other individual users and not global groups that are available to all users.
psGlobalTypeInclude	2	Includes global groups in the collection alongside individual groups for the logged in user.
psGlobalTypeOnly	3	Creates a collection of global groups only.

- *filterByGroupName*—String containing the name of a particular **Group** to retrieve in the collection. An empty string by default.

- *filterByShortcut*—Pointer to a **IDispatch** object that represents a **Shortcut** you want the group to contain; you pass this argument to select groups that have this shortcut as a member. The argument defaults to NULL, not filtering by any shortcut. C++ automatically casts the **Shortcut** object you pass for this argument; in JavaScript no casting is required; in C#, you need to cast the **Shortcut** to an **object** type.

On the first call of **GetGroups()**, before you have any groups, the object returned is an empty collection container. Once the collection has been populated, a call to **GetGroups()** retrieves groups of the type you indicate (using filtering arguments) from the database and returns them in a **Groups** collection.

The method call might look as follows to obtain a collection that includes all **Groups**:

```
objGroupsColl = objAdmin.GetGroups(psGlobalTypeAll, "", NULL);
```

On the first call of **GetGroups()**, before you have any groups, the object returned is an empty collection container. Once the collection has been populated, a call to **GetGroups()** retrieves the types of groups from the database that are indicated by the arguments passed and returns them in a **Groups** collection.

To return a collection of only groups that contain a particular shortcut, you pass its **Shortcut** object as the third argument:

```
objGroupsColl = objAdmin.GetGroups(psGlobalTypeAll, "", objShortcut);
```

## Adding Group Objects to the Groups Collection

To create the actual **Group** objects, you add them to the **Groups** collection using the **Add()** method of the **Groups** collection object. You pass this method three arguments:

- *Name*—String containing the name of the group. This name is required and must not already exist in the collection.
- *Description*—String containing a short description of the purpose of the group. The description is not required, so you can pass an empty string for it.
- *isGlobal*—Boolean indicating whether the group is available to all users or not.

To create a group of shortcuts for procedures, you might call the **Add()** method this way:

```
objGroupsColl.Add("Procedures", "Shortcuts that expand into full
descriptions of procedures.", true);
```

# Adding Shortcuts to a Group

Once you have established **Group** objects, you can add the individual shortcuts to a **Group**.

## To add shortcuts to each Group:

1. Retrieve the **Group** object from the **Groups** collection by looping through the **Groups** collection searching for a group that has the name you are looking for:

```
for(var i = 1; i <= objGroupsColl.Count; i++)
{
 if (objGroupsColl.Item(i).Name = strGroupToFind)
 {
 objGroup = objGroupsColl.Item(i);
 break;
 }
}
```

2. Retrieve the **Shortcut** object for the particular shortcut you want to add to the **Group** by searching for the shortcut by its **ShortText** property value:

```
for(var i = 1; i <= objShctColl.Count; i++)
{
 if (objShctColl.Item(i).ShortText = strShctToFind)
 {
 objShortcut = objShctColl.Item(i);
 break;
 }
}
```

3. Call the **AddShortcut()** method of the **Group** (singular) object and pass it the **Shortcut** object for the shortcut to add to the group:

```
objGroup.AddShortcut(objShortcut);
```

# Retrieving a Particular Group

After you have established several **Group** objects in the **Groups** collection, you can retrieve a single **Group** from the collection in a **Group** (singular) object using the **Item()** method:

```
for (i = 1; i <= objGroupsColl.Count; i++)
{
 if (objGroupsColl.Item(i).Name == "Procedures")
 {
 objGroup = objGroupsColl.Item(i);
 break;
 }
}
```

Once you have the **Group** object, you can use that object to work with the group's properties.

# Modifying Name/Description of the Group

Using the **Group** object, you can modify the name of the group by changing the setting of its **Name** property:

```
objGroup.Name = "Surgical Procedures";
```

You can modify the description of the group by modifying its **Description** property:

```
objGroup.Description = "Surgical Procedures for St. Vincent's Hospital";
```

After you modify one or more of the properties of the **Group** object, to have those changes take effect immediately and remain in effect even when the user logs in again, you call the **Update()** method of the **Group** object, which saves the changes to the database:

```
objGroup.Update();
```

# Determining Who Owns the Group

Using the **Group** object, you can determine the owner of the group, the person who created the group, by retrieving the value of its **Author** property:

```
alert("Group Owner:" + objGroup.Author);
```

# Determining Whether Group Contains Shortcuts Available to All Users

Using the **Group** object, you can determine whether the group is *global*, that is, contains shortcuts that are available to all users, by retrieving the value of its **Global** property:

```
if objGroup.Global = True
{
 alert("Shortcuts in " + objGroup.Name + "Group for All Users");
}
else
{
 alert("Shortcuts in " + objGroup.Name + "Group for Logged in User Only")
}
```

## Removing Shortcuts from a Group

You can remove an individual shortcut from a group.

### To remove a shortcut from the group:

1. Retrieve the particular shortcut to remove from the **Shortcuts** collection in a single **Shortcut** object:

```
for(var i = objShctColl.Count; i >= 1; i--)
{
 if (objShctColl.Item(i).ShortText == strShctToRemove)
 {
 objShortcutToRemove = objShctColl.Item(i);
 break;
 }
}
```

2. Call the **RemoveShortcut()** method of the **Group** object and pass it the **Shortcut** object for the shortcut you want to remove:

```
objGroup.RemoveShortcut(objShortcutToRemove);
```

## Removing a Group from the Collection

You can remove a particular group from the **Groups** collection.

### To remove a Group object from the Groups collection:

1. Retrieve the particular group to remove from the collection in a single **Group** object:

```
for (var i = objGroupsColl.Count; i >= 1; i--)
{
 if (objGroupsColl.Item(i).Name == strGroupToRemove)
 {
 objGroupToRemove = objGroupsColl.Item(i);
 break;
 }
}
```

2. Call the **Remove()** method of the **Groups** collection object and pass it the **Group** object for the group to remove:

```
objGroupsColl.Remove(objGroupToRemove);
```

# Code Summary

```
<HTML> <HEAD>
<TITLE>SDK Administrator, Add or Modify Groups</TITLE>
<object ID="pscribeSDK">
</object>

<META HTTP-EQUIV="Content-Type" content="text/html; charset=iso-8859-1">
...
</HEAD>

<BODY bgcolor="#FFCC99" language="javascript" onload="InitializeAdmin()">
<form>

<table WIDTH="1000" valign="top">
<COL SPAN=1>
<COL WIDTH="500px">

<tr>
<td>
Create New or Update Existing Group
</td>
</tr>

<tr>
<td>Type of Groups in Collection:
<select id="cbGlobal" name="cbGlobal">
<option value="0">All Global and All User-Specific</option>
<option value="1">Logged in User's Groups Only</option>
<option value="2">Global and Logged in User's Groups</option>
<option value="3">Global Groups Only</option>
</select>
</td>
</tr>

<tr>
<td>
<input id="btnGetGroupCollection" language="javascript" type="button"
 name="btnGetGroupCollection"
 value="Get Groups Collection" onclick="RetrieveGroupsCollection()">
</td>
</tr>

<tr>
<td>Collection Retrieved:
<input id="txtCollectionType" language="javascript" style="WIDTH:
205, HEIGHT: 22" enabled
 value="Global Groups Collection" maxlength="56" size="42"
 name="txtCollectionType">
</td>
</tr>
```

```
<tr>
 <td>

 Add or Remove a Group
 </td>
</tr>

<tr>
 <td nowrap>Owner of Group (Login Name):
 <input id="cbOwner" style="WIDTH: 205, HEIGHT: 22" enabled
 maxlength="24" size="24"
 name="cbOwner">

 </td>
</tr>

<tr>
 <td>Name of Group:

 <input id="txtGroupName" style="WIDTH: 205, HEIGHT: 22" enabled
 maxlength="64" size="48"
 name="txtGroupName">
 </td>
</tr>

<tr>
 <td>Description of Group:
 <input id="txtGroupDesc" style="WIDTH: 205, HEIGHT: 22" enabled
 maxlength="64" size="48"
 name="txtGroupDesc">
 </td>
</tr>

<tr>
 <td>Owner of New Group:
 <select id="cbGroupOwner" name="cbGroupOwner">
 <option value="true">System (For All Users)</option>
 <option value="false">Logged in User</option>
 </select>
 </td>
</tr>

<tr>
 <td>
 <input language="javascript" type="button" name="btnAddGroup"
 value="Add Group to Collection" onclick="AddGroupToCollection() ">

 </td>
</tr>
```

```
<tr>
 <td>
 <input language="javascript" type="button" name="btnRemoveGroup"
 value="Remove Group from Collection"
 onClick="RemoveGroupFromCollection()">

 </td>
</tr>

<tr>
 <td>

 Add Shortcut/Remove Shortcut from Group

 </td>
</tr>

<tr>
 <td nowrap>Name of Shortcut (ShortText):
 <input style="WIDTH: 205, HEIGHT: 22" enabled maxlength="24" size="24"
 name="txtShortcut">
 </td>
</tr>

<tr>

 <td>
 <input language="javascript" type="button" name="btnAddShortcutToGroup"
 value="Add Shortcut to Group" onClick="AddShortcutToGroup">
 </td>

</tr>

<tr>

 <td>
 <input language="javascript" type="button"
 name="btnRemoveShortcutFromGroup"
 value="Remove Shortcut from Group" onClick="RemoveShortcutFromGroup">
 </td>

</tr>

</table>
</form>
</BODY>

<SCRIPT type="text/javascript">
// Global Variables for Application
var psGlobalTypeAll = 0;
var psGlobalTypeExclude = 1;
var psGlobalTypeInclude = 2;
var psGlobalTypeOnly = 3;

var objUserProfile;
var objGroupsColl;
```

```
var objGroup;
var objAdmin;

function InitializeAdmin()
{
 try
 {
 // Instantiate PowerscribeSDK Object
 pscriveSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");
 // Instantiate Admin SDK Object
 objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");
 objAdmin = top.scripts.pscriveSDK.PSAdminSDK;
 // Initialize Admin SDK Server
 var url = "http://server2/myAdminSDK/";
 objAdmin.Initialize(url, "UserConfigs");
 objAdmin.Login("admin", "");
 ...
 }
 catch(error)
 {
 alert("InitializeAdmin: " + error.number + " " + error.description);
 }
}

function RetrieveGroupsCollection()
{
 // First check that all appropriate fields contain values
 // Test to see the type of groups the collection should contain
 // psGlobalTypeAll, psGlobalTypeExclude, psGlobalTypeInclude,
 // psGlobalTypeOnly

 var strName = "";

 objGroupsColl = objAdmin.GetGroups(psGlobalTypeAll, "", 0);
 txtCollectionType.value = "All System and User-Specific Groups";

 if (cbGlobal.Options.value == psGlobalTypeAll)
 {
 objGroupsColl = objAdmin.GetGroups(cbGlobal.Options.value, "", 0);
 txtCollectionType.value = "All System and User-Specific Groups";
 }
 else if (cbGlobal.Options.value == psGlobalTypeExclude)
 {
 objGroupsColl = objAdmin.GetGroups(cbGlobal.Options.value, "", 0);
 txtCollectionType.value = "All " + objUserProfile.LoginName + " Groups";
 }
 else if (cbGlobal.Options.value == psGlobalTypeInclude)
 {
 objGroupsColl = objAdmin.GetGroups(cbGlobal.Options.value, "", 0);
 txtCollectionType.value = "All System and " +
 objUserProfile.LoginName + " Groups";
 }
}
```

```
else if (cbGlobal.Options.value == psGlobalTypeOnly)
{
 objGroupsColl = objAdmin.GetGroups(cbGlobal.Options.value, "", 0);
 txtCollectionType.value = "System Groups Only";
}
}

function AddGroupToCollection()
{
 // First check that all appropriate fields contain values
 var groupName = txtGroupName.value;
 objGroupsColl.Add(txtGroupName.value, txtGroupDesc.value,
 cbGroupOwner.value);
 alert(groupName + " added to the collection.")
}

function RemoveGroupFromCollection()
{
 var groupName = txtGroupName.value;
 var objGroupToRemove;
 for (var i=objGroupsColl.Count; i>=0; i--)
 {
 if (objGroupsColl.Item(i).Name == groupName)
 {
 objGroupToRemove = objGroupsColl.Item(i);
 break;
 }
 }

 objGroupsColl.Remove(objGroupToRemove);
 alert(groupName + " removed from the collection.")
}

function AddShortcutToGroup()
{
 var objShortcutsColl = objAdmin.GetShortcuts();
 var objShortcut;

 for (var i=0; i<=objShortcutsColl.count; i++)
 {
 if (objShortcutsColl.Item(i).Name == txtShortcut.value)
 {
 objShortcut = objShortcutsColl.Item(i);
 }
 break;
 }

 objGroup.AddShortcut(objShortcut);
 alert(objShortcut.ShortText + " added to group.");
}
```

```
function RemoveShortcutFromGroup()
{
 var objShortcutsColl = objAdmin.GetShortcuts();
 var objShortcutToRemove;
 for (var i = objShortcutsColl.Count; i >= 1; i--)
 {
 if (objShortcutsColl.Item(i).Name == txtShortcut.value)
 {
 objShortcutToRemove = objShortcutsColl.Item(i);
 }
 break;
 }

 objGroup.RemoveShortcut(objShortcutToRemove);
 alert(objShortcutToRemove.ShortText + " removed from group.");
}

</script>
</HTML>
```



# *Creating Templates for Structured Reports and Shortcuts*

## **Objectives**

This chapter covers how to include in your application a template builder to create and edit templates for structured reports and shortcuts. You and/or your end users can deploy this template builder to simplify the process of creating XML report templates. This chapter introduces the **XmTemplateEdit** template builder object and covers steps to using the object:

- [How Application Uses Template Object](#)
- [Steps to Using XmTemplateEdit Object to Develop Report Templates](#)
- [Steps to Using XmTemplateEdit Object to Develop and Edit Structured Shortcuts](#)
- [Developing Graphical Element That Reflects Template Structure](#)
- [Correlating Template with Graphical Element](#)

# How Application Uses Template Object

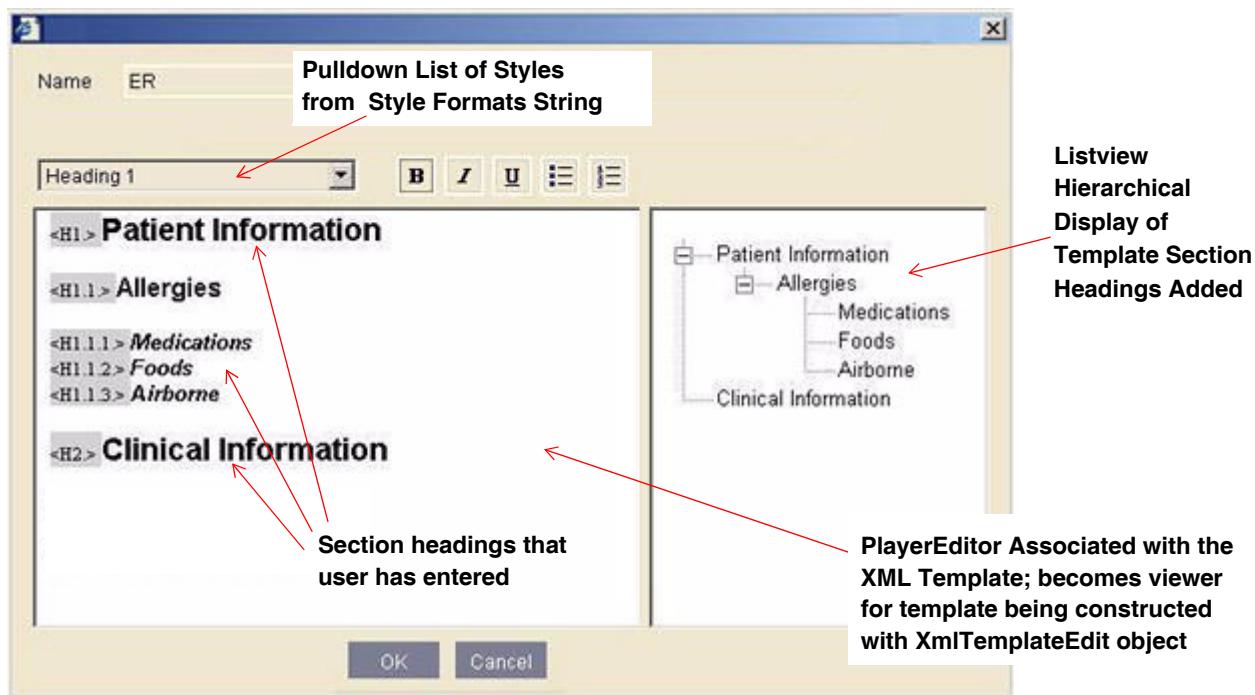
You saw in [Chapter 6](#) how you can use an XML template to provide a *structure* to a report, creating sections and multiple subsections and/or applying formats to headings and paragraphs.

Generally, it is considerable work to create report structures in XML files. To develop an easier way to create a report template, you can embed an interactive structured template builder in your application by taking these fundamental steps:

- Add a template builder object (**XmLEdit** object) to your application
- Associate a **PlayerEditor** with the **XmLEdit** object
- Develop a structured XML report template or shortcut XML template in the **PlayerEditor** associated with the **XmLEdit** template builder object

The illustration below shows an *SDK* application that presents an interactive structured template builder. In this case, the application uses an embedded **PlayerEditor** (to the left) and a listview hierarchical display of the section headings added to the template (to the right). Although all applications that deploy a template builder object (**XmLEdit** object) also deploy a **PlayerEditor**, the listview or a similar graphical element that displays the structure of the report is optional and you would create it yourself.

In the background, the application uses the **XmLEdit** template builder to create an XML report or shortcut template, based on the input entered in the **PlayerEditor**, which forms a window through which you access the **XmLEdit** object.



In an application like the one illustrated here, you indicate each XML report section to add to the template by entering its heading line, such as **Patient Information**, in the **PlayerEditor**.

You then select the text and choose a style format to assign to that text. The style formats available for the template are those you supply to the application when you create the **XmLEditTemplate** template builder object. Your application can make a list of style formats available to the end user in any way you'd like. Here, the style formats are presented in a pulldown list.

Including a graphical element in your GUI can help the user visualize the structure of the template as it evolves. It is ideal to display the headings in a listview-like structure. In the example shown, for instance, you can see that **Patient Information** and **Clinical Information** are both Level 1 sections and **Allergies** is a subsection under **Patient Information**, at Level 2. **Medications**, **Foods**, and **Airborne** are all subsections of **Allergies**, at Level 3. Each report or shortcut template can contain up to nine heading (section) levels.

The report structure might also contain specific paragraph content that should appear in every report using the template.

You might also want to create structured shortcuts. Typically, a single structured shortcut contains both headings and paragraph text that later replace the associated shortcuts in the structured report.

## Steps to Using XmLEditTemplate Object to Develop Report Templates

To use the **XmLEditTemplate** object to develop a report template, you take several steps:

- Create a string containing the empty XML template provided
- Create a series of *style formats* in XML, following the designated schema that indicates:
  - All levels of headings for the report and their styles (font, size, color, and so on)
  - All paragraph styles
- Create a string to contain the *style formats*
- Instantiate a COM object called the **XmLEditTemplate** object in your application, using the empty XML template and the style format
- Initialize the **XmLEditTemplate** object
- Create a **PlayerEditor** to associate with the **XmLEditTemplate** object
- (Optional) Create a GUI element (such as a pulldown list) for accessing heading and paragraph styles
- (Optional) Create a GUI element (such as a listview) to show sections of the report or shortcut template
- Retrieve text displaying in the **PlayerEditor** associated with the **XmLEditTemplate** template builder object (actually this text is in the template builder) and store it in a string to use as the template for a report

Now, let's see how to execute these steps in the pages that follow.

# Creating Strings to Initialize XmlTemplateEdit Object

You need to create two strings before you can initialize the **XmlTemplateEdit** object:

- Empty XML template string
- Styles format string

## Creating Empty XML Template String

For your application to use an XML template object, it needs to have the empty XML template available for the application to start with. You put the content of that template into a string, so that you can later pass the string as an argument.

The empty structured report template, shown below, is an XML template that has an empty **<body>** tag. As users create sections, your application uses the **XmlTemplateEdit** template builder object to add them to the **<body>** section:

```
<clu xmlns="http://www.dictaphone.com/HSG/CLU/Extraction/2002-12-02"
 xmlns:t="http://www.dictaphone.com/HSG/CLU/Template/2002-12-02"
 xmlns:html="http://www.w3.org/1999/xhtml" >
 <doc>
 <body>
 <!-- You add <Section> tags using template.xsd format -->
 </body>
 </doc>
</clu>
```

Refer to [Appendix C](#) for the schema used to create this template. Be sure to create a string and set it to the content of this template:

```
var templateXml = "<clu xmlns=\"http://www.dictaphone.com/...</clu>";
```

## Creating XML Style Format String

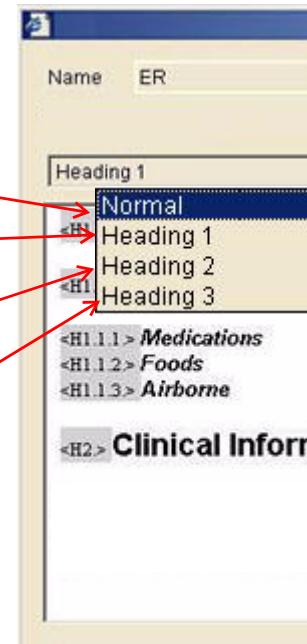
For your application to use an XML template, it needs an XML string that defines the hierarchy of the XML structure by defining all the headings in template. This string defines each heading in a **<styleFormat>** tag and assigns it a **Name**, **Type**, **Level**, and **html:style**. Refer to [Appendix D](#) for the schema to follow in creating this template.

Below is a sample XML *style formats* template that defines styles for report paragraphs and three levels of headings, applying the schema found in [Appendix D](#). Your *style formats* template would contain similar details:

```

<pssdk xmlns="http://www.dictaphone.com/HSG/PSSDK/2004-11-30" xmlns:
 html="http://www.w3.org/1999/xhtml">
 <template>
 <styleFormats>
 <styleFormat Name="P1" Type="Para" Level="0"
 html:style="font-family:Times;font-size:
 10pt;font-weight:normal;font-style:
 normal;"> Normal </styleFormat>
 <styleFormat Name="H1" Type="Head"
 Level="1" html:style="font-family:
 Arial;font-size:14pt;font-weight:
 bold;"> Heading 1 </styleFormat>
 <styleFormat Name="H2" Type="Head"
 Level="2" html:style="font-family:
 Arial;font-size:12pt;
 font-weight:bold;"> Heading 2 </styleFormat>
 <styleFormat Name="H3" Type="Head"
 Level="3" html:style="font-family:
 Arial;font-size:10pt;font-style:italic;
 font-weight:bold;
 tab-count:2;"> Heading 3 </styleFormat>
 </styleFormats>
 </template>
</pssdk>
```

You cannot change the settings  
of the Name parameters for the tags



You can change content  
of each style tag, using  
any term you want instead  
of Normal or Heading #

The settings of the **Name** parameter for a paragraph and nine heading levels are predefined and you cannot change them. They are always **P1** for the paragraph, and **H1**, **H2**, **H3**, and so on for the headings.

You can, however, modify the content of each tag; for example, your tags are not required to be **Normal**, **Heading 1**, **Heading 2**, **Heading 3**, but can be whatever text you choose.

Once you have the *style formats* template, you create a string to contain it:

```
var styleXml = "<pssdk xmlns=\"http://www.dictaphone.com/HSG/PSSDK/2004-11-30\"><template><styleFormats><styleFormat Name='P1' Type='Para' Level='0' html:style='font-family:Times;font-size:10pt;font-weight:normal;font-style: normal;'>Normal</styleFormat><styleFormat Name='H1' Type='Head' Level='1' html:style='font-family:Arial;font-size:14pt;font-weight:bold;'>Heading 1</styleFormat><styleFormat Name='H2' Type='Head' Level='2' html:style='font-family:Arial;font-size:12pt;font-weight:bold;'>Heading 2</styleFormat><styleFormat Name='H3' Type='Head' Level='3' html:style='font-family:Arial;font-size:10pt;font-style:italic;font-weight:bold;tab-count:2;'>Heading 3</styleFormat></styleFormats></template></pssdk>";
```

When your application presents options for formatting text in the template, those options should match the headings and paragraphs from the *style formats* string (see the parallels shown in the illustration above).

# Creating and Intializing the XmlTemplateEdit Object

Once you have created the report or shortcut template structure in an XML string and styles in a *style formats* string, you can create an XML report template object that uses that structure. To add such an object, you use an **XmlTemplateEdit** ActiveX object.

## To create an **XmlTemplateEdit** ActiveX object in your program:

1. Select the **XmlTemplateEdit** template builder ActiveX object in the toolbox; then drag and drop the object into your Visual Studio application form.
2. Assign the **XmlTemplateEdit** template builder ActiveX object a name. In the code shown here, the control has been named **axobjXmlTemplt**.
3. Create the **XmlTemplateEdit** ActiveX object in JavaScript as follows:

```
var axobjXmlTemplt;
axobjXmlTemplt = new ActiveXObject("PowerscribeSDK.XmlTemplateEdit");
```

4. Once you have the **XmlTemplateEdit** template builder object, you intialize that object by calling its **Initialize()** method. You pass the method two arguments that are the strings of formatting information to use—the first the string that contains the empty XML report template and the second the string that contains the XML *style format*:

```
axobjXmlTemplt.Initialize(templateXml, styleXml);
```

# Associating Editor with **XmlTemplateEdit** Template Builder Object

If you have not already created a **PlayerEditor** to associate with the **XmlTemplateEdit** template builder object, create one now (for details on how, refer to [Embedding PlayerEditor Control in Your Application on page 59](#)).

You need the **PlayerEditor** for the **XmlTemplateEdit** template builder object so that you or another user can enter information for the template into that editor. After you create the **PlayerEditor**, you associate the **XmlTemplateEdit** template builder object with it by calling the **SetEditor()** method of the **XmlTemplateEdit** object and passing the method the name of the **PlayerEditor** object:

```
axobjXmlTemplt.SetEditor(objTmpEditor);
```

After you take this action, you or another user can enter information for the template into the **PlayerEditor**. You employ the **PlayerEditor** as a viewer into the template you are building

with the **XmlTemplateEdit** object. The **XmlTemplateEdit** template builder object is in control of and effectively *drives* the **PlayerEditor**.

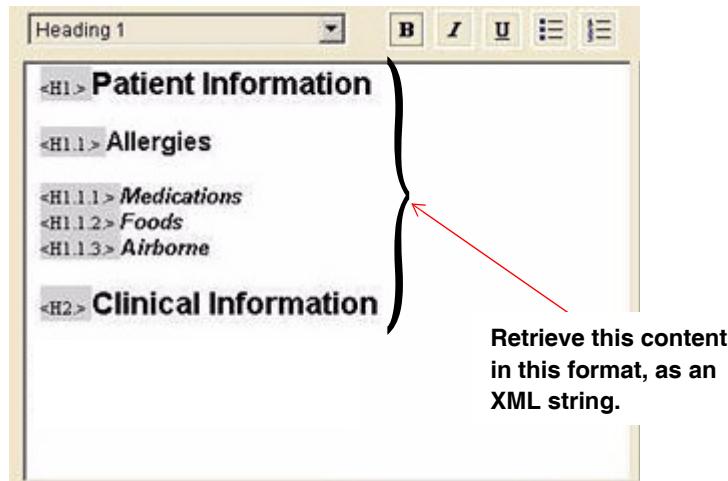
When you execute methods of the **XmlTemplateEdit** object, you see the results of those methods in the **PlayerEditor** window.

## Creating Template with Content from Associated PlayerEditor

Once you have associated a **PlayerEditor** with the **XmlTemplateEdit** template builder, you can enter text into the **PlayerEditor** window and assign the text formats. When you have finished, you then take an action (such as clicking a button) to initiate creating the XML template from the content of the **XmlTemplateEdit** object that you see displaying in the **PlayerEditor**.

To retrieve an XML template string that becomes the template (see the adjacent illustration), call the **GetText()** method of the **XmlTemplateEdit** object and pass it a constant (or its numeric equivalent) for the type of string to create. In this case, you want to create an XML formatted string, so you pass **psFormatTypeXml** or its numeric equivalent, **2**:

```
var psFormatTypeXml = 2;
strXmlTemplate = axobjXmlTemplt.GetText(psFormatTypeXml);
```



Retrieve this content in this format, as an XML string.

This method call retrieves all text you see in the **PlayerEditor** associated with the **XmlTemplateEdit** object in a single XML formatted template string.

You can then use the XML formatted template string to create a report with the **NewReport()** method of the **PowerscribeSDK** object:

```
objXmlReport = pscrbeSDK.NewReport("ORD-030407", strXmlTemplate);
```

You could, alternatively, pass the XML string to the **MergeTemplate()** method of a **Report** object to apply that template to the report:

```
objReport.MergeTemplate(strXmlTemplate, psMergeSrcTypeDict);
```

For the first argument, you pass the XML template string. The second argument is optional and defaults to **psMergeSrcTypeReuse** or **2** unless you pass **psMergeSrcTypeDict** or its numeric equivalent of **1**. For more information about **MergeTemplate()**, refer to [Chapter 6](#).

# Steps to Using XmlTemplateEdit Object to Develop and Edit Structured Shortcuts

To use the **XmlTemplateEdit** object to create or edit the expanded text (**LongText**) for structured shortcuts, you take the following major steps in your application:

- Take all of the steps outlined under [Steps to Using XmlTemplateEdit Object to Develop Report Templates on page 387](#), except the last step.
- Retrieve an existing shortcut from a **Shortcuts** collection and put the text of its **LongText** property into the **PlayerEditor** associated with the **XmlTemplateEdit** object.
- Modify the **LongText** of the **Shortcut** object in the **PlayerEditor** associated with the **XmlTemplateEdit** object.
- Retrieve text from the the **PlayerEditor** associated with the **XmlTemplateEdit** object and set the **Shortcut** object's **LongText** property using the text.

Let's look at these extra steps for working with a structured shortcut in the pages that follow.

## Using XmlTemplateEdit Object to Create or Modify Structured Shortcuts

When you place your cursor in **PlayerEditor** associated with the **XmlTemplateEdit** object, you are working on the **XmlTemplateEdit** template. In this **XmlTemplateEdit** template, you can create new or modify existing structured shortcuts.

### To create a shortcut or modify an existing shortcut:

1. Create or retrieve a collection of shortcuts using the **GetShortcuts()** method of the **PSAdminSDK** object:  

```
objShortcutsColl = objAdmin.GetShortcuts;
```
2. Retrieve a specific shortcut in the collection using the **Item()** method:  

```
objShortcut = objShortcutsColl.Item(1);
```
3. Retrieve the string contained in the **LongText** property of the **Shortcut** object, which is the expanded and formatted text of the shortcut:  

```
strExpandedShctText = objShortcut.LongText;
```
4. Put the expanded and formatted text of the shortcut into the **XmlTemplateEdit** object to be edited by calling the **SetText()** method of the **XmlTemplateEdit** object and passing the method the expanded text:  

```
axobjXmlTemplt.SetText(strExpandedShctText);
```

The expanded shortcut text should now appear in the **PlayerEditor** associated with the **XmLEdit** object.

5. Edit the shortcut text and format displaying in the **PlayerEditor**.

**To retrieve the shortcut from the editor associated with the XmLEdit object and use it:**

When you have finished revising the shortcut, retrieve its text using the **GetText()** method of the **XmLEdit** object; then use the retrieved text to set the **LongText** property of the shortcut:

```
objShortcut.LongText = axobjXmlTemplt.GetText();
```

Or

You can also retrieve the text of the shortcut (an XML string) and pass that string to the **MergeTemplate()** method of the **Report** object. This method merges the shortcut into the currently open report. Be sure to pass the **psMergeSrcTypeShortcut** constant or its numeric equivalent (3) as the second argument to **MergeTemplate()**:

```
var strXmlTemplate = axobjXmlTemplt.GetText();
objReport.MergeTemplate(strXmlTemplate, psMergeSrcTypeShortcut);
```



*Note: **MergeTemplate()** merges the template with the current report, the one that is open and displaying in a **PlayerEditor**. That report displays in a **PlayerEditor** other than the one associated with the **XmLEdit** object.*

## Changing Text Style in XmLEdit Template

When you enter a heading and/or paragraph text into the **PlayerEditor** associated with the **XmLEdit** object, the text starts out as plain until you assign it a style.

After you (or another user) modify the format of the text in the **XmLEdit** object, your application can set that style in the XML template by calling the **SetSelTextStyle()**

method of the  
**XmLEdit**  
object and passing it  
the value of the

Heading 1
Heading 2
Heading 3

```
<styleFormat Name="H1" Type="Head"
Level="1" html:style="font-family:
Arial; font-size:14pt; font-weight:
bold;">Heading 1</styleFormat>
```

**Name** attribute from the corresponding **<styleFormat>** tag in the *style formats* template (see illustration). You pass the name in a string.

For instance, if the style format of the selected text is **Heading 1**, you could set that style format in the template by passing a string containing **H1** to the **SetSelTextStyle()** method:

```
axobjXmlTemplt.SetSelTextStyle("H1");
```

# Preparing for XmlTemplateEdit Events

When the cursor moves to a new line or the structure of the template changes, the **XmlTemplateEdit** object fires events that your application can handle. You could use the information about the structure of the template to update the listview or graphical element in your application so that it reflects the changes to the template.

## To prepare to handle XmlTemplateEdit object events:

1. Create an **EventMapper** object and link the **XmlTemplateEdit** event handlers to the events for that object:
  - **EditorCursorMoveToNewLine**
  - **SectionsChanged**
  - **SectionNameChanged**

The code might look something like this:

```
XmlSink = new ActiveXObject("PowerscribeSDK.EventMapper");

XmlSink.EditorCursorMoveToNewLine = HandleEditCursorMove;
XmlSink.SectionsChanged = HandleTemplateSectionsChanged;
XmlSink.SectionNameChanged = HandleTemplateSectionNameChanged;
```

2. After you create the **XmlTemplateEdit** object, call the **Advise()** method of the **EventMapper** object so that the application begins receiving the events. You pass the method the **XmlTemplateEdit** object:

```
axobjXmlTemplt = new ActiveXObject("PowerscribeSDK.XmlTemplateEdit");
XmlSink.Advise(axobjXmlTemplt);
```

3. Create event handler functions to handle each event that the **XmpTemplateEdit** object fires:

```
function HandleEditCursorMove(numbOfLine, strStyle)
{
 ...
}

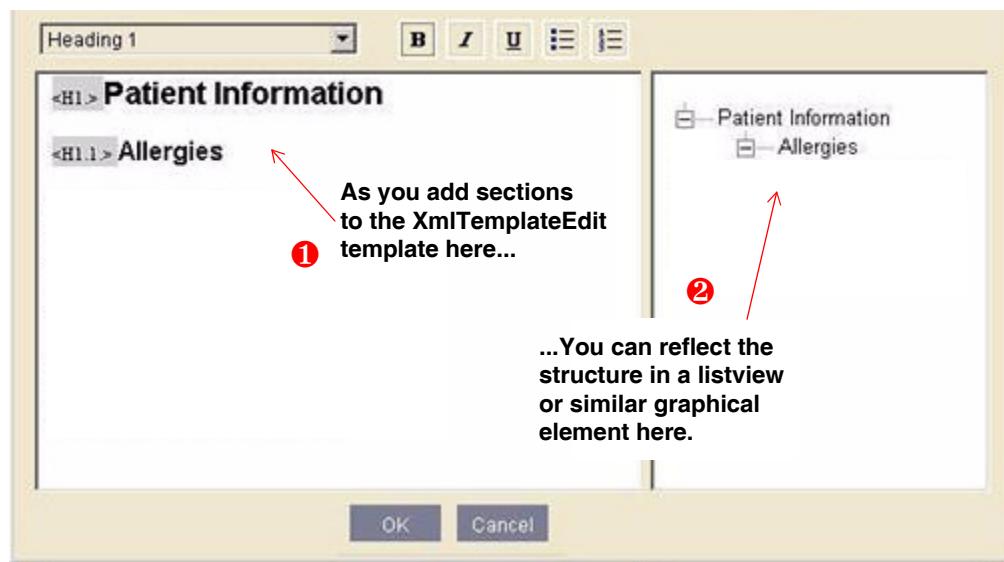
function HandleTemplateSectionsChanged()
{
 ...
}

function HandleTemplateSectionNameChanged
 (oldHeadingPath, newHeadingName)
{
 ...
}
```

# Developing Graphical Element That Reflects Template Structure

To create a listview or similar graphical element in your application that displays the text of each heading in the template, you can retrieve the section names from the **XmlTemplateEdit** object whenever a section has been added to the template.

Each time you (or another user) add a section to the template, the **XmlTemplateEdit** object fires the **SectionsChanged** event. In response to that event, inside the event handler, you can retrieve the section names from the **XmlTemplateEdit** object so that you can display them in the listview or other graphical element.



The event handler usually starts as follows:

```
function HandleTemplateSectionsChanged()
{
}
```

In the handler, you retrieve the sections added to the **XmlTemplateEdit** template by calling the **GetSections()** method of the **XmlTemplateEdit** object:

```
objTemplSectionsColl = axobjXmlTempl.GetSections();
```

This method returns the **Sections** collection associated with the template you are developing in the **XmlTemplateEdit** template.

Once you have the collection, you can add the section names to the listview or other graphical element in the application so that it reflects the structure of the template, by retrieving the **Section** object for each successive section and adding it to the listview. You might take this action using a custom function that updates the listview, by calling it from inside the **SectionsChanged** event handler:

```
UpdateListview(objTemplSectionsColl, 1)
```

The function might be like the one shown below that uses the **Sections** collection to retrieve individual **Section** objects and feed their heading names into the listview through another custom function called **AddToListview()** (code not shown):

```
function UpdateListview(objTmSectionsColl, secLvl)
{
 if (objTmSectionsColl.Count > 0)
 {
 for (i = 1; i <= objTmSectionsColl.Count; i++)
 {
 if (objTmSectionsColl.Item(i).HeadingName != "")
 {
 // add section heading name to listview
 // using secLvl to determine
 AddToListview(objTmSectionsColl.Item(i).HeadingName)
 }

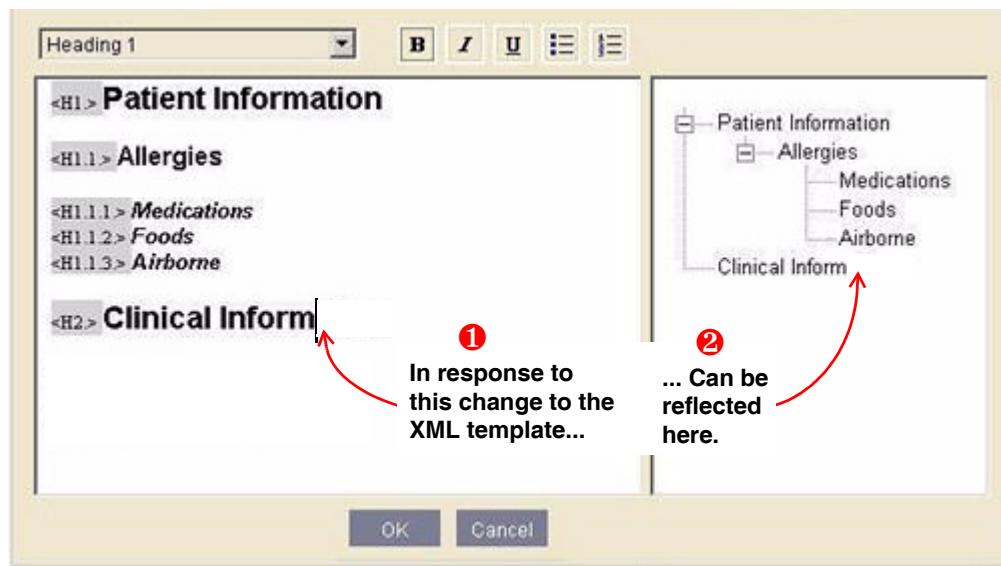
 if (objTmSectionsColl.Item(i).Subsections.Count > 0)
 {
 objSBsections = objTmSectionsColl.Item(i).Subsections;
 UpdateListview(objSBsections, secLvl+1); //Call recursively
 }
 }
 }
}
```

# Handling Change to Section Name in Template

When a section name has been changed, the **XmlTemplateEdit** object fires the **SectionNameChanged** event. The event's handler would start as follows:

```
function HandleTemplateSectionNameChanged(oldHeadingPath, newHeadingName)
{
}
```

You might want to update the listview or other graphical element in your application so that it shows the new section name as it is changed (see illustration below).



You can retrieve the text from the section of the template where the cursor has been placed in the **XmlTemplateEdit** template by using the **GetCurrentSection()** method of the object:

```
objTmSection = axobjXmlTemplt.GetCurrentSection();
```

You can then have the handler reflect the **XmlTemplateEdit** template heading change in the listview or other graphical element of your application.

You might parse the *oldHeadingPath* for the old heading name, then traverse the listview using the *oldHeadingPath* and replace the old heading name with the *newHeadingName*.

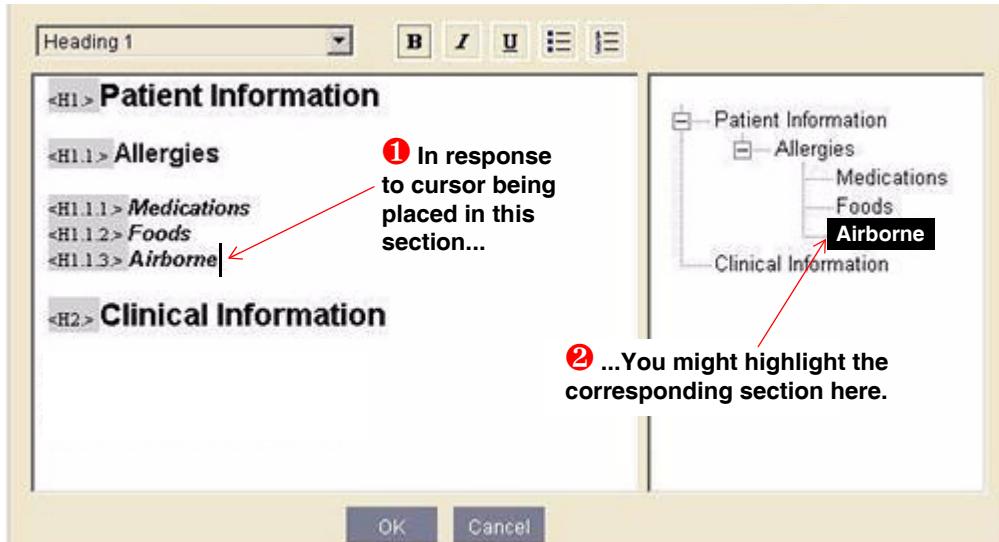
# Correlating Template with Graphical Element

To fully utilize a listview or similar graphical element that reflects the structure of the template being developed in the **XmlTemplateEdit** template, you can:

- Highlight corresponding branch of the listview when cursor moves to a section of template
- Position cursor in the section of the template that corresponds to branch clicked in listview
- In a combo box list (or other indicator), highlight the style of the text that has been selected in the **XmlTemplateEdit** template

## Positioning Cursor in Graphical Element Based on Click in XmlTemplateEdit Template

When the user clicks in a section in the **XmlTemplateEdit** template (shown to the left in the illustration below), you might want to put the cursor in the corresponding section of the listview or other graphical element of your application, like the one shown in the area to the right in the illustration below. To indicate when to respond, you might fire a custom event when the click occurs.



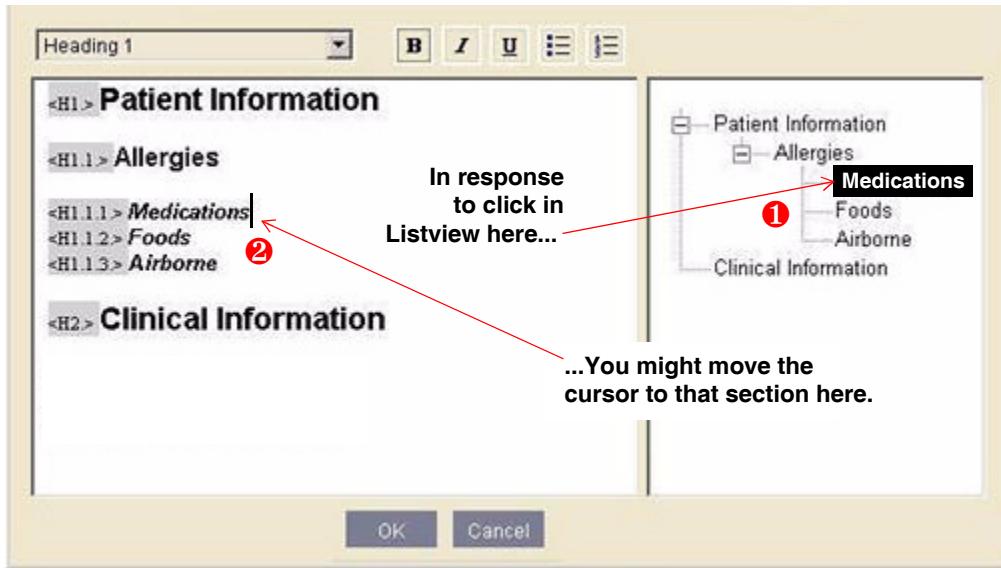
Then, in your custom event's handler, you could retrieve the section of the template where the cursor is positioned by calling the **GetCurrentSection()** method of the **XmlTemplateEdit** object:

```
objSectionEdited = axobjXmlTemplt.GetCurrentSection();
```

Once it identifies the section of the template that is being worked on, your application can highlight the corresponding section name in its listview or other graphical element.

## Positioning Cursor in XmlTemplateEdit Template Based on Click in Graphical Element

When the user clicks on a section in the listview or other graphical element of your application (in the area to the right in the illustration below), you might want to fire a custom event and in response, in the handler for that event, put the cursor at the end of the text in the corresponding section of the **XmlTemplateEdit** template, as shown to the left in the illustration below.



You position the cursor at the end of the text in the section of the **XmlTemplateEdit** object by calling its **SetCurrentSection()** method and passing it the section :

```

function CustomHandlerListViewClick(treeBranchClick)
{
 // Determine the branch of the listview clicked
 // Retrieve the corresponding section in the template
 // Pass that section to an XmlTemplateEdit object method
 axobjXmplt.SetCurrentSection(objSection);
}

```

## Indicating the Font of Text Selected in XmlTemplateEdit Template

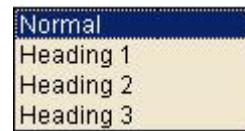
After you move the cursor in the editor to a new line of the template, the **XmlTemplateEdit** object fires an event called **EditorCursorMoveToNewLine**. The handler for the event starts out as follows:

```
function HandleEditCursorMove(numbOfLine, strStyle)
{
}
```

In this event handler, you can parse the *strStyle* argument that the event passes to the handler. The *strStyle* argument contains a string that holds the style of the text selected in the **XmlTemplateEdit** template. That style includes the entire HTML string that defines the text format, such as:

```
"font-family:Times;font-size:10pt;font-weight:normal;
font-style:normal;"
```

Once your application has determined the style of the selected text, it can reflect that style by highlighting a matching style button or combo box option, just as **Normal** is highlighted in the adjacent illustration.



Your code might compare the HTML style in the *strStyle* string to each combo box list element; when you find a match, you can retrieve the index of the matching combo list option and use that index to highlight the option that reflects the style:

```
var selStyle = cbStyleFormat.options(cbStyleFormat.selectedIndex);
if(selStyle.value != strStyle)
{
 var options = cbStyleFormat.options;
 for(var i=0; i<options.length; i++)
 {
 if(options(i).value == strStyle)
 {
 cbStyleFormat.selectedIndex = i;
 break;
 }
 }
}
```

## Code Summary

```
<HTML>
<HEAD>
<TITLE>XML Template Builder -- Structured Reports & Shortcuts</TITLE>
<!-- This sample is not complete; it requires additional
 code from previous chapters -->
<SCRIPT type="text/javascript"> ...
// Global Variables

var axobjXmlTemplt;
var objXmlReport;
var objRptEditor;
var objTmpEditor;
var templateXml;
var styleXml;
var strXmlTemplate;
var objAdmin;
var strShortText;

function InitializePS()
{
 try
 {
 // Instantiate PowerScribe SDK Object
 pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");
 objAdmin = pscribeSDK.AdminSDK;

 // Initialize PowerScribe SDK Server
 var url = "http://server2/pscribesdk/";
 pscribeSDK.Initialize(url, "JavaScript_Demo");
 ...

 ReptSink = new ActiveXObject("PowerscribeSDK.EventMapper");
 RptEditorSink = new ActiveXObject("PowerscribeSDK.EventMapper");
 TemplEditorSink = new ActiveXObject("PowerscribeSDK.EventMapper");

 // Create Event Mapper Object for XmlTemplateEdit Object
 XmlSink = new ActiveXObject("PowerscribeSDK.EventMapper");

 // Map Event Handlers for XmlTemplateEdit Object
 XmlSink.EditorCursorMoveToNewLine = HandleEditCursorMoveToNewLine;
 XmlSink.SectionsChanged = HandleTemplateSectionsChanged;
 XmlSink.SectionNameChanged = HandleTemplateSectionNameChanged;
 // Map Event Handler for Other objects...
 ...

 // Create 2 PlayerEditors: 1st for Template Builder, 2nd for Report
 CreateObjectCtl("objRptEditor")
 RptEditorSink.Advise(objRptEditor);
```

```
 CreateObjectCtl("objTmpEditor")
 TemplEditorSink.Advise(objTmpEditor);

 // Create XmlTemplateEdit Object and Initialize
 var axobjXmlTemplt;
 axobjXmlTemplt = new ActiveXObject("PowerscribeSDK.XmlTemplateEdit");
 InitializeTemplateBuilder();

 // Initiate receipt of Events from XmlTemplateEdit Object
 XmlSink.Advise(axobjXmlTemplt);
 }
 catch(error)
 {
 alert("InitializePowerscribe: " + error.number + error.description);
 }
} ...
```

```
function InitializeTemplateBuilder()
{
 // Set XML Template and XML Style Strings
 var templateXml = txtXmlTemplate.value;
 var styleXml = txtXmlStyle.value;

 // Use XML Template & XML Style Strings to Initialize XmlTemplateEdit
 axobjXmlTemplt.Initialize(templateXml, styleXml);

 // Associate Editor with XmlTemplateEdit Object
 axobjXmlTemplt.SetEditor(objTmpEditor);
}
```

```
function GetStringFromTemplateBuilder()
{
 // Retrieve Template String from PlayerEditor Window
 var psFormatTypeXml = 2;
 strXmlTemplate = axobjXmlTemplt.GetText(psFormatTypeXml);
}
```

```
function CreateNewXmlReport()
{
 if (strXmlTemplate == "")
 {
 alert("Creating Report using Original Template");
 objXmlReport = pscribeSDK.NewReport(txtReportID.value,
 templateXml);
 }
 else
 {
 alert("Creating Report using Displayed Template");
 objXmlReport = pscribeSDK.NewReport(txtReportID.value,
```

```
 strXmlTemplate);
 }

 objXmlReport.SetPlayerEditor(objPlayerEditor);
 objXmlReport.ExclusiveLock(true);
 pscribeSDK.ActivateReport(objXmlReport);
 objRptEditor.EnablePlayer(true);
 ReptSink.Advise(objXmlReport);
}

function MergeXmlTemplateIntoCurrReport()
{
 // First check to be sure there is a current report in Report editor
 // Check that XML string is not empty
 if (strXmlTemplate != "")
 {
 objXmlReport.MergeTemplate(strXmlTemplate, psMergeSrcTypeDict);
 }
 else
 {
 alert("You must press Get Template to retrieve template first.");
 }
}

function HandleEditCursorMove(numbofLine, strStyle)
{
 // If style selected in combo box does not equal style of
 // selected text, change the style selected in the list
 // to reflect style of the selected text

 var selStyle = cbStyleFormat.options(cbStyleFormat.selectedIndex);

 if(selStyle.value != strStyle)
 {
 var options = cbStyleFormat.options;
 for(var i=0; i<options.length; i++)
 {
 if(options(i).value == strStyle)
 {
 cbStyleFormat.selectedIndex = i;
 break;
 }
 }
 }
}
```

```
function HandleTemplateSectionsChanged()
{
 objTemplSectionsColl = axobjXmlTemplt.GetSections();
 UpdateListview(objTemplSectionsColl, 1)
}

function UpdateListview(objTmSectionsColl, secLvl)
{
 if (objTmSectionsColl.Count > 0)
 {
 for (i = 1; i <= objTmSectionsColl.Count; i++)
 {
 if (objTmSectionsColl.Item(i).HeadingName != "")
 {
 // add section heading name to listview
 // using secLvl to determine
 AddToListview(objTmSectionsColl.Item(i).HeadingName)
 }

 if (objTmSectionsColl.Item(i).Subsections.Count > 0)
 {
 objSBsections = objTmSectionsColl.Item(i).Subsections;
 UpdateListview(objSBsections, secLvl+1); //Call recursively
 }
 }
 }
}

function HandleTemplateSectionNameChanged
 (oldHeadingPath, newHeadingName)
{
 objSectionEdited = axobjXmlTemplt.GetCurrentSection();
 // Replace oldHeadingPath with newHeadingName in
 // corresponding section of Listview
}

function CustomHandlerListViewClick(treeBranchClick)
{
 // Determine the branch of the listview clicked
 // Retrieve the corresponding section in the template
 // Pass that section to an XmlTemplateEdit object method
 axobjXmlTemplt.SetCurrentSection(objSection);
}
```

```
// Retrieve a shortcut and put its longtext into the editor associated
// with the XmlTemplateEdit template builder.

// Remember that Shortcuts require a PSAdminSDK object, in this case
// created during initialization (objAdmin)

function GetShortcutToEdit()
{
 objShortcutsColl = objAdmin.GetShortcuts;
 for (i = 0; i <= objShortcutsColl.Count; i++)
 {
 if (strShortText == txtShortText.value)
 if (strShortText == objShortcutsColl.Item(i).ShortText)
 objShortcut = objShortcutsColl.Item(i);
 }
 strExpandedShctText = objShortcut.LongText;
 axobjXmlTemplt.SetText(strExpandedShctText);
}

function ReplaceLongTextForShortcut(shortText)
{
 objShortcutsColl = objAdmin.GetShortcuts;
 for (i = 0; i <= objShortcutsColl.Count; i++)
 {
 if (shortText == objShortcutsColl.Item(i).ShortText)
 objShorcut = objShortcutsColl.Item(i);
 }
 objShortcut.LongText = axobjXmlTemplt.GetText();
}

function MergeNewShortcutIntoReport()
{
 strXmlTemplate = axobjXmlTemplt.GetText();
 objReport.MergeTemplate(strXmlTemplate, psMergeSrcTypeShortcut);
}

function CreateObjectCtl(divID)
{
 var divRef = document.getElementById(divID);
 if (divRef)
 {
 divRef.innerHTML = divRef.innerHTML;
 }
}

...
</SCRIPT> </HEAD>

<BODY bgcolor="#fffff" language="javascript" onload="InitializePS()"
onunload="UninitializePS()"> <form>

...
<div id="objRptEditor">
 <object id="RptPlayerEditor" ...>
```

```
 classid="clsid:9D6DE7B5-9D53-4E79-8564-2DD8783B1ADA">
 </object>
</div>
<div id="objTmpEditor">
 <object id="TmpPlayerEditor" ...
 classid="clsid:9D6DE7B5-9D53-4E79-8564-2DD8783B1ADA">
 </object>
</div>

<tr>
 <td nowrap>Report ID:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="txtReportID">
 </td>
</tr>

<tr>
 <input language="javascript" type="button" name="btnCreateXmlRpt"
 value="Create New XML Report" onClick="CreateNewXmlReport() ">
</tr>

<tr>
 <input language="javascript" type="button"
 name="btnMergeXmlTemplt" value="Merge Template into Report"
 onClick="MergeXmlTemplateIntoCurrReport() ">
</tr>

<tr>
 <input language="javascript" type="button" name="btnGetTemplate"
 value="Get Template" onClick="GetStringFromTemplateBuilder() ">
</tr>

<tr>
 <td nowrap>Shortcut to Edit:</td>
 <td>
 <input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
 name="txtReportID">
 </td>
</tr>

<tr>
 <input language="javascript" type="button" name="btnReplaceShortcut"
 value="Replace Expansion for Shortcut"
 onClick="ReplaceLongTextForShortcut(txtShortText.value) ">
</tr>

<tr>
 <input language="javascript" type="button" name="btnMergeShortcut"
 value="Merge Shortcut into Report"
 onClick="MergeNewShortcutIntoReport() ">
</tr>

</form> </BODY> </HTML>
```

# *Executing Advanced SDK Administrator Tasks*

## **Objectives**

This chapter takes you through adding some advanced tasks to your own custom *SDK Administrator* program.



*Note: You can carry out all of these tasks using the *SDK Administrator* provided with the product without developing your own application.*

In your own *SDK Administrator* you can take these actions:

- [Copying Users from One SDK System to Another](#)
- [Exporting and Importing User Voice Models](#)
- [Importing Users from Non-SDK System](#)
- [Copying Words from One SDK System to Another](#)
- [Importing Words from Non-SDK System](#)
- [Copying Shortcuts from One SDK System to Another](#)
- [Importing Shortcuts from Non-SDK System](#)
- [Copying Categories from One SDK System to Another](#)
- [Importing Categories from Non-SDK System](#)
- [Copying Groups from One SDK System to Another](#)
- [Importing Groups from Non-SDK System](#)
- [Managing Reports](#)

# Copying Users from One SDK System to Another

To copy users to a new *SDK* system, you first export the users from the current (old) system, then import them into the new system. This action exports information stored in the **User** object, such as the **Name**, **MiddleName**, **LastName**, **LoginName**, and other property settings.

You might also want to export other information about the user. For instance, when the user executes training on an *SDK* system, the *SDK* forms a voice model for the user. To avoid forcing the user to go through training again on the new *SDK* system, you could redeploy the user's voice model from the old *SDK* system. You can copy or move the voice model to the new system by exporting it from the old system, then importing it into the new system. Taking this action ensures the user a smooth transition to the new system.

## Exporting Users

**To export all users in the **Users** collection from a particular *SDK* system:**

1. Obtain a **PSAdminSDK** object by instantiating the object:

```
objAdmin = new ActiveXObject ("PSAdminSDK.PSAdminSDK");
```

2. Call the **ExportUsers()** method of the **PSAdminSDK** object.

```
var strXML_List = objAdmin.ExportUsers();
```

This method exports the users and all their property settings into an XML formatted string that you utilize to import the **Users** to another *SDK* system.

3. Export the voice models of all users in the collection (see [Exporting and Importing User Voice Models on page 409](#)).

## Importing Users

**To import all the users in a **Users** collection that you exported from another *SDK* system:**

1. Obtain a **PSAdminSDK** object by instantiating the object:

```
objAdmin = new ActiveXObject ("PSAdminSDK.PSAdminSDK");
```

2. Call the **ImportUsers()** method of the **PSAdminSDK** object and pass it the XML formatted string that the **ExportUsers()** method returned:

```
objAdmin.ImportUsers(strXML_List);
```

3. Import the voice models for the users (see [Exporting and Importing User Voice Models on page 409](#)).

# Exporting and Importing User Voice Models

To ensure each user a smooth transition to a new *SDK* system that you plan to import user information into, you also export the voice model for each user from the old system and import that voice model into the new system.

## Preparing to Handle Export and Import Events

As with events of other objects, you should prepare to handle the **EndExport** and **EndImport** events of the **User** object in your JavaScript application.

### To prepare to handle User object events:

1. Instantiate an **EventMapper** object to catch events that the **User** object, part of the **PSAdminSDK** family of objects, triggers:

```
UserSink = new ActiveXObject("PSAdminSDK.EventMapper");
```

2. Link the **User** event handlers to the events for that object:

- **EndExport**
- **EndImport**

```
UserSink.EndExport = HandleEndExport;
UserSink.EndImport = HandleEndImport;
```

3. Call the **Advise()** method of the **EventMapper** object so that the application begins receiving the events. You pass the method the **User** object:

```
UserSink.Advise(objUser);
```

4. Create event handler functions to handle each event that the **User** object fires.

## Exporting User Voice Model



**Note:** Exporting a voice model is an action you take on a single **User** object, rather than on a **Users** collection; you might want to loop through the collection of **Users** you are exporting to be sure you export each voice model.

To export the voice model for a particular user, you first obtain that **User** object from the **Users** collection, then call the **ExportVoiceModel()** method of the **User** object:

```
objUser.ExportVoiceModel();
```

The method takes no arguments. When it executes, it starts a background process and returns immediately.

If the export process succeeds, the method places the voice model into a **.zip** file that the **User** object sends to the **EndExport** event handler when it fires the event.

If the export process fails, the **User** object still fires the **EndExport** event, but in this situation the object sends the error number and associated message to the handler.

## Handling EndExport Event

The **User** object fires the **EndExport** event after either succeeding in or attempting to export a voice model.

If the exporting process succeeds, the **User** object sends the event handler a URL that indicates the location of the **.zip** file containing the voice model. If the process fails, the **User** object sends the event handler the error number and associated message that indicate why it failed.

The handler might manage the arguments it receives as follows:

```
function HandleEndExport(strFileLoc, intErrNumber, strErrMsg)
{
 if (strFileLoc != "")
 {
 // Copy the file from the strFileLoc to another disk
 }
 else
 {
 alert("Export of voice model for " +objUser.LoginName+ " failed.")
 }
}
```

If the process succeeds, you can find the file containing the exported voice model, copy it to another system disk, and import it into another *SDK* system.

## Importing User Voice Model



*Note: Importing a voice model is an action you take on a single **User** object, rather than on a **Users** collection; you might want to loop through the collection of **Users** you have imported to be sure you import each voice model.*

To import a voice model that you exported from another *SDK* system, you first obtain the **User** object associated with the voice model from the **Users** collection, then call the **ImportVoiceModel()** method of the **User** object. You pass the method the URL to the location of the **.zip** file containing the voice model:

```
strVoiceURL = "http://server1/pscribesdk/Models/Voice/UserVoice_1.zip";
userObject.ImportVoiceModel(strVoiceURL);
```

When the process of importing the voice model completes, the **User** object fires the **EndImport** event and sends its handler the error number and associated error message of any error that occurs. If the process succeeds, the **User** object sends **0** to the event's handler for the error number and an empty string for the error message.

## Handling the EndImport Event

The **User** object fires the **EndImport** event after either succeeding in or attempting to import a voice model.

If the importing process succeeds, the user model becomes available to the application and is deployed the next time the user logs in. If the importing process fails, the **User** object sends the event handler the error number and associated message that indicate why it failed.

The handler might manage the results of an import as follows:

```
function HandleEndImport(ErrNumber, ErrMessage)
{
 if (ErrNumber != 0)
 {
 alert("Import of voice model for " + objUser.login + " succeeded.")
 }
 else
 {
 alert("Import of voice model for " + objUser.login + " failed.")
 }
}
```

## Importing Users from Non-SDK System

You can also import users from a non-SDK system.

After you retrieve the users from the non-SDK system, put the information about the users into an XML formatted string using the following format:

```
<?xml version="1.0" encoding="UTF-8"?>

<ns1:UserList
 xsi:type="ns1:UserList" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
 instance"; xmlns:ns1="user.beans.middleware.powerscribe.dictaphone.com">
 <array xsi:type="soapenc:Array"
 soapenc:arrayType="ns1:UserBean [1]" Number of users in list
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/";>
 <item>
 <userID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">10</userID>
 <loginName xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">zzzzz</loginName>
 <firstName xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">Zoe</firstName>
 <middleName xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">Z</middleName>
 <lastName xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">Zerta</lastName>
 </item>
 </array>
</ns1:UserList>
```

```
<password xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">x</password>
<badgeID xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">n/a</badgeID>
<enabled xsi:type="xsd:boolean" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">true</enabled>
<useRealtime xsi:type="xsd:boolean" xmlns:xsd="http://www.w3.org/
 2001/XMLSchema">true</useRealtime>
<languageID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">1</languageID>
<voiceID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">0</voiceID>
<voiceName xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">Microphone</voiceName>
<trainingState xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">0</trainingState>
<userType xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">1</userType>
<adminPrivs xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">0</adminPrivs>
<sysAdminPrivs xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">0</sysAdminPrivs>
<ppID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">0</ppID>
<loggingEnabled xsi:type="xsd:boolean"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">false
 </loggingEnabled>
</item>
 ... repeat for each user
</array>
</ns1:UserList>
```

The XML format contains tags that correspond to the properties of the **User** object; most of the tags have the same name as the equivalent property, with a few exceptions:

- The **<userType>** tag is equivalent to the **ReportRights** property
- The **<adminPrivs>** and **<sysAdminPrivs>** tags should be set to **0**; you can assign privileges after you have successfully imported the user

Most tags are optional. The only tags required are:

- **<userID>**
- **<firstName>**
- **<password>**
- **<lastName>**
- **<loginName>**

Once you have created the string, call the **ImportUsers()** method of the **PSAdminSDK** object and pass it the string:

```
objAdmin.ImportUsers(strNonSDKuserlist);
```

You can also import an XML file of users through the GUI of the *SDK Administrator* provided with the product.

# Copying Words from One SDK System to Another

You can copy custom words from one *SDK* system to another by using methods of the **PSAdminSDK** object. This process copies all words in the collection, both global words and words specific to particular users.

## Exporting Words

To export custom words in the **Words** collection from a particular *SDK* system:

1. Obtain a **PSAdminSDK** object by instantiating the object:

```
objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");
```

2. Call the **ExportWords()** method of the **PSAdminSDK** object.

```
var strWords_List = objAdmin.ExportWords();
```

This method exports all words in the **Words** collection and all their property settings into an XML formatted string that you utilize to import the **Words** to another *SDK* system.

## Importing Words

To import all the words in a **Words** collection that you exported from another *SDK* system:

1. Obtain a **PSAdminSDK** object by instantiating the object:

```
objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");
```

2. Call the **ImportWords()** method of the **PSAdminSDK** object and pass it the XML formatted string that the **ExportWords()** method returned:

```
objAdmin.ImportWords(strWords_List);
```

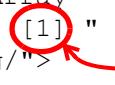
## Importing Words from Non-SDK System

You can also import words from a non-*SDK* system.

After you retrieve the words from the non-*SDK* system, put the information about the words into an XML formatted string using the following format:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ns1:WordList xsi:type="ns1:WordList" xmlns:xsi="http://www.w3.org/2001/
```

```
XMLSchema-instance" xmlns:ns1="word.beans.middleware.powerscribe.
dictaphone.com">

 <ArrayOfWordBean xsi:type="soapenc:Array"
 soapenc:arrayType="ns1:WordBean [1]" xmlns:soapenc="http://
schemas.xmlsoap.org/soap/encoding/">
 
 Number of words in list

 <item>
 <ID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">10</ID>
 <author xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">System</author>
 <authorID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">0</authorID>
 <wordName xsi:type="xsd:string" xmlns:xsd="http://
www.w3.org/2001/XMLSchema">Methamphetamine</wordName>
 <category xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">16</category>
 <transcription xsi:type="xsd:string" xmlns:xsd=
 "http://www.w3.org/2001/XMLSchema"> </transcription>
 <state xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">2</state>
 <source xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">8</source>
 <categoryName xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/
2001/XMLSchema">Generic Drug</categoryName>
 <lastModifiedDate xsi:type="xsd:dateTime" xmlns:xsd=
 "http://www.w3.org/2001/XMLSchema">10/13/2006 10:50:00 AM
 </lastModifiedDate>
 <authorLongName xsi:type="xsd:string" xmlns:xsd="http://
www.w3.org/2001/XMLSchema">System</authorLongName>
 </item>
 ... repeat for each word
 </ArrayOfWordBean>
</ns1:WordList>
```

The XML format contains tags that correspond to the properties of a **Word** object, with a few exceptions:

- The **<wordName>** tag is equivalent to the **Word** property
- No tag is equivalent to the **Global** property, because an empty **<author>** tag, or one that contains **System**, indicates that the word is a global word

Most of the tags are required, but a few are optional:

- **<transcription>**
- **<lastModifiedDate>**
- **<authorLongName>**

When you use any tag, you must be sure to include the information in the format shown here. For instance, for <lastModifiedDate> you must include the full date and time, using the four digit year in the date and the full time—hours, minutes, seconds, with AM or PM at the end.

Once you have created the string, call the **ImportWords()** method of the **PSAdminSDK** object and pass it the string:

```
objAdmin.ImportWords(strNonSDKwordlist);
```

## Copying Shortcuts from One SDK System to Another

You can copy shortcuts from one *SDK* system to another by using methods of the **PSAdminSDK** object. This process copies all shortcuts in the collection, both global shortcuts and shortcuts specific to particular users.

### Exporting Shortcuts

**To export shortcuts in the Shortcuts collection from a particular *SDK* system:**

1. Obtain a **PSAdminSDK** object by instantiating the object:

```
objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");
```

2. Call the **ExportShortcuts()** method of the **PSAdminSDK** object.

```
var strShortcuts_List = objAdmin.ExportShortcuts();
```

This method exports all shortcuts in the **Shortcuts** collection in the database along with all their property settings into an XML formatted string that you utilize to import the **Shortcuts** to another *SDK* system.

### Importing Shortcuts

**To import all the shortcuts in a Shortcuts collection that you exported from another *SDK* system:**

1. Obtain a **PSAdminSDK** object by instantiating the object:

```
objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");
```

2. Call the **ImportShortcuts()** method of the **PSAdminSDK** object and pass it the XML formatted string that the **ExportShortcuts()** method returned:

```
objAdmin.ImportShortcuts(strShortcuts_List);
```

You can also generate your own string of shortcuts to import, using the **XmlTemplateEdit** object to create them and the **XmlTemplateEdit GetText()**

method to retrieve the string, then pass that string to the **ImportShortcuts()** method of the **PSAdminSDK** object:

```
strXMLradShortcuts = "<?xml version=\"1.0\" encoding=\"utf-8\" ?>
<pssdk xmlns=\"http://www.dictaphone.com/HSG/PSSDK/2004-11-30\"
xmlns:html=\"http://www.w3.org/1999/xhtml\">...</pssds>

objAdmin.ImportShortcuts(strXMLradShortcuts);
```

## Importing Shortcuts from Non-SDK System

You can also import shortcuts from a non-*SDK* system. After you retrieve the shortcuts from the non-*SDK* system, put the information into an XML formatted string using the following format:

```
<?xml version="1.0" encoding="UTF-8" ?>

<ns1:ShortcutList xsi:type="ns1:ShortcutList" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
xmlns:ns1="shortcut.beans.middleware.powerscribe.dictaphone.com">

<ArrayOfShortcutBean xsi:type="soapenc:Array"
soapenc:arrayType="ns1:ShortcutBean [1]" xmlns:soapenc="http://
schemas.xmlsoap.org/soap/encoding/">  Number of shortcuts in list

<item>
 <ID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">40</ID>
 <author xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">rs</author>
 <authorID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">2</authorID>
 <type xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">1</type>
 <shortText xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">Lower Respiratory Clear</shortText>
 <longText xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/
 XMLSchema">Lower respiratory study shows left bronchial
 passageways clear, right bronchial passageways clear.</longText>
 <transcription xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/
 2001/XMLSchema"> </transcription>
 <lastModifiedDate xsi:type="xsd:dateTime" xmlns:xsd="http://
 www.w3.org/2001/XMLSchema">09/22/2006 16:03:29 PM
 </lastModifiedDate>
 <authorLongName xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/
 2001/XMLSchema">Smith, John</authorLongName>
</item>
```

```
... repeat for each shortcut
</ArrayOfShortcutBean>
</ns1:ShortcutList>
```

The XML format contains tags that correspond to the properties of a **Shortcut** object; however, there is no tag equivalent to the **Global** property, because an **<author>** tag that contains **System** indicates that the shortcut is a global shortcut.

Most tags are required, but several are optional:

- **<transcription>**
- **<lastModifiedDate>**
- **<authorLongName>**

Once you have created the string, call the **ImportShortcuts()** method of the **PSAdminSDK** object and pass it the string:

```
objAdmin.ImportShortcuts(strNonSDKshctList);
```

## Copying Categories from One SDK System to Another

You can copy shortcuts from one *SDK* system to another by using methods of the **PSAdminSDK** object. This process copies all shortcuts in the collection, both global shortcuts and shortcuts specific to particular users.

## Exporting Categories

**To export all categories in the Categories collection from a particular *SDK* system:**

1. Obtain a **PSAdminSDK** object by instantiating the object:

```
objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");
```

2. Call the **ExportCategories()** method of the **PSAdminSDK** object.

```
var strXMLCat_List = objAdmin.ExportCategories();
```

This method exports the categories and all their property settings into an XML formatted string that you utilize to import the **Categories** to another *SDK* system.

## Importing Categories

To import all the categories in a Categories collection that you exported from another *SDK* system:

1. Obtain a **PSAdminSDK** object by instantiating the object:

```
objAdmin = new ActiveXObject ("PSAdminSDK.PSAdminSDK");
```

2. Call the **ImportCategories()** method of the **PSAdminSDK** object and pass it the XML formatted string that the **ExportCategories()** method returned:

```
objAdmin.ImportUsers(strXMLCat_List);
```

## Importing Categories from Non-SDK System

You can also import categories from a non-*SDK* system. After you retrieve the categories from the non-*SDK* system, put the information into an XML formatted string using the following format:

```
<?xml version="1.0" encoding="UTF-8" ?>

<ns1:CategoryList xsi:type="ns1:CategoryList" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ns1="shortcutcategory.beans.powerscribe.dictaphone.com">

 <maxRows xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/XMLSchema">1</maxRows>

 <pageIndex xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/XMLSchema">0</pageIndex>

 <pageSize xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/XMLSchema">-1</pageSize>

 <ArrayOfCategoryBean xsi:type="soapenc:Array" soapenc:
 arrayType="ns1:CategoryBean[1]" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
 
 
 Number of categories in list
 <item>
 <ID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/XMLSchema">16</ID>
 <categoryName xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/XMLSchema">Arterial Blockage</categoryName>
 <description xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/XMLSchema">Procedure to discover or treat arterial
 blockage.
 </description>
 <lastModifiedDate xsi:type="xsd:dateTime" xmlns:xsd="http://www.w3.org/2001/XMLSchema">06/01/2007 16:11:10 PM</lastModifiedDate>
 </item>
 </ArrayOfCategoryBean>
</CategoryList>
```

```
</lastModifiedDate>
</item>
</ArrayOfCategoryBean>
</ns1:CategoryList>
```

## Copying Groups from One SDK System to Another

You can copy groups from one *SDK* system to another by using methods of the **PSAdminSDK** object. This process copies all groups in the collection, both global groups and groups specific to particular users.

### Exporting Groups

**To export all groups in the Groups collection from a particular *SDK* system:**

1. Obtain a **PSAdminSDK** object by instantiating the object:

```
objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");
```

2. Call the **ExportGroups()** method of the **PSAdminSDK** object.

```
var strXMLGrp_List = objAdmin.ExportGroups();
```

This method exports the groups and all their property settings into an XML formatted string that you utilize to import the **Groups** to another *SDK* system.

### Importing Groups

**To import all the groups in a Groups collection that you exported from another *SDK* system:**

1. Obtain a **PSAdminSDK** object by instantiating the object:

```
objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");
```

2. Call the **ImportGroups()** method of the **PSAdminSDK** object and pass it the XML formatted string that the **ExportGroups()** method returned:

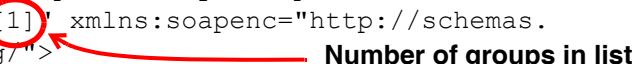
```
objAdmin.ImportGroups(strXMLGrp_List);
```

# Importing Groups from Non-SDK System

You can also import groups from a non-*SDK* system. After you retrieve the groups from the non-*SDK* system, put the information into an XML formatted string using the following format:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ns1:GroupList xsi:type="ns1:GroupList" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ns1="group.beans.powerscribe.dictaphone.com">

 <maxRows xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/XMLSchema">1</maxRows>
 <pageIndex xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/XMLSchema">0</pageIndex>
 <pageSize xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/XMLSchema">-1</pageSize>

 <ArrayOfGroupBean xsi:type="soapenc:Array" soapenc:
 arrayType="ns1:GroupBean[1]" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
 Number of groups in list

 <item>
 <ID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/XMLSchema">6</ID>
 <groupName xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/XMLSchema">Abdomen</groupName>
 <authorID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/XMLSchema">0</authorID>
 <author xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/XMLSchema">System</author>
 <authorLongName xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/XMLSchema"><None>, <None></authorLongName>
 <description xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/XMLSchema">Of or relating to the abdominal cavity or one of the organs in the abdominal cavity.</description>
 <lastModifiedDate xsi:type="xsd:dateTime" xmlns:xsd="http://www.w3.org/2001/XMLSchema">06/01/2007 17:00:23 PM
 </lastModifiedDate>
 <Shortcuts xsi:type="soapenc:Array" soapenc:arrayType="ns2:ShortcutBean[5]"
 xmlns:ns2="shortcut.beans.powerscribe.dictaphone.com">

 <item>
 <ID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/XMLSchema">11</ID>
 <author xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/2001/XMLSchema">System</author>
 <authorID xsi:type="xsd:int" xmlns:xsd="http://www.w3.org/2001/XMLSchema">0</authorID>
 <type xsi:type="ns2:shortcutType">Voice</type>
 <shortText xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/
```

```
2001/XMLSchema">biliary tube check and change</shortText>
<longText xsi:type="xsd:string" xmlns:xsd="http://www.w3.org/
2001/XMLSchema"><pssdk xmlns="http://www.
dictaphone.com/HSG/PSSDK/2004-11-30";
xmlns:html="http://www.w3.org/1999/
xhtml">>#xd;
<xmlTemplate>#xd;
<body>#xd;
<!-- The body contains <Section> or
<PlainSection>, They are same format as in
XML report -->#xd;

<section xmlns="http://www.w3.org/1999/xhtml"><p
xml:space="preserve"><html:span></p><p
xml:space="preserve"><html:span html:style="font-
family:Arial;font-size:10pt;color:#000000;source-type:0">
source="Tp">Patient was placed on teh angiography table. The
right upper quadrant was prepped and draped in standard sterile fashion,
and 2% lidocaine infiltrated into the soft tissues around the catheter for
local anesthesia. After performing a brief cholangiogram, demonstrating the
side-holes of the catheter in appropriate position with pigtail in the
duodenum, the hub of the catheter was cut and an Amplatz 0.035 guidewire
inserted into the duodenum. The catheter was removed and a new 12 French
percutaneous biliary drain inserted. Catheter was secured to the skin and
sterile dressings applied.</html:span></p><p
xml:space="preserve"><html:span></p></
section></body>#xd;
</xmlTemplate>#xd;
</pssdk>#xd;</longText>
<transcription xsi:type="xsd:string" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"></transcription>
<lastModifiedDate xsi:type="xsd:dateTime" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema">06/01/2007 16:40:57 PM
</lastModifiedDate>
<authorLongName xsi:type="xsd:string" xmlns:xsd="http://
www.w3.org/2001/XMLSchema"><None>, <None>;
</authorLongName>
<categoryName xsi:type="xsd:string" xmlns:xsd="http://
www.w3.org/2001/XMLSchema">Tube or Catheter Insertion
</categoryName>
</item>
</item>
</ArrayOfGroupBean>
```

# Managing Reports



*Note: Only Administrator level users can delete reports from the SDK application database.*

You delete a report from the database by calling the **DeleteReport()** method of the **PowerscribeSDK** object and pass it the report ID:

```
pscribeSDK.DeleteReport(strReportIdentifier);
```

## Code Summary

```
<HTML>
<HEAD>

<TITLE>Advanced SDK Administrator Tasks</TITLE>

<SCRIPT type="text/javascript">

// Global variables:
var strXML_List;
var strXMLCat_List;
var strXMLGrp_List;
var objAdmin;
var objUsersColl;
var objUser;
var strShortcuts_List;
var strWords_List;
var strReportIdentifier;

function InitializeAdminApp()
{
 try
 {
 // Instantiate Admin SDK Object
 pscribeSDK = new ActiveXObject("PowerscribeSDK.PowerscribeSDK");
 objAdmin = pscribeSDK.AdminSDK;
 objAdmin = new ActiveXObject("PSAdminSDK.PSAdminSDK");

 // Initialize Admin SDK Server
 var url = "http://server2/adminsdk/";
 objAdmin.Initialize(url, "UserConfigs");

 // Create EventMapper Object for Admin SDK Object
 UserSink = new ActiveXObject("PSAdminSDK.EventMapper");

 // Map Handlers for Each Event to SDK EventMapper Object
 UserSink.EndExport = HandleEndExport;
 UserSink.EndImport = HandleEndImport;
 }
}
```

```
 UserSink.Advise(objAdmin);
 }
 catch(error)
 {
 alert("InitializeAdmin: " + error.number + error.description);
 }
 finally
 {
 DoLogin()
 }
}

function DoLogin()
{
 ...
}

function ExportUsersAndVMs()
{
 strXML_List = objAdmin.ExportUsers();

 objUserColl = objAdmin.GetUsers();
 UserSink.Advise(objUser);

 objUsersColl = objAdmin.GetUsers();
 for (i = 0; i <= objUsersColl.Count; i++)
 {
 objUser = objUsersColl.Item(i);
 objUser.ExportVoiceModel();
 }
}

function ImportUsersAndVMs()
{
 objAdmin.ImportUsers(strXML_List);

 objUserColl = objAdmin.GetUsers();
 UserSink.Advise(objUser);

 for (i = 0; i <= objUsersColl.Count; i++)
 {
 strVoiceURL="http://server1/pscribesdk/Models/Voice/UserVoice_"+i+".zip";
 userObject.ImportVoiceModel(strVoiceURL);
 }
}

function ExportAllShortcuts()
{
 strShortcuts_List = objAdmin.ExportShortcuts();
 // save the list to an external file.
}
```

```
function ImportShortcutsFromFile()
{
 strShortcuts_List = txtShortcutsFile.value;
 objAdmin.ImportShortcuts(strShortcuts_List);
}

function ExportAllWords()
{
 strWords_List = objAdmin.ExportWords();
 // save the list to an external file.
}

function ImportWordsFromFile()
{
 strWords_List = txtWordsFile.value;
 objAdmin.ImportShortcuts(strWords_List);
}

function ExportAllCategories()
{
 strXMLCat_List = objAdmin.ExportCategories();
 // save the list to an external file.
}

function ImportCategoriesFromFile()
{
 strXMLCat_List = txtCategoriesFile.value;
 objAdmin.ImportUsers(strXMLCats_List);
}

function ExportAllGroups()
{
 strXMLGrp_List = objAdmin.ExportGroups();
 // save the list to an external file.
}

function ImportGroupsFromFile()
{
 strXMLGrp_List = txtGroupsFile.value;
 objAdmin.ImportGroups(strXMLGrp_List);
}

function PurgeReport()
{
 strReportIdentifier = txtReportID.value;
 pscribeSDK.DeleteReport(strReportIdentifier);
}

function HandleEndExport(strFileLoc, intErrNum, strErrMsg)
{
 if (strFileLoc != "")
 // Show location of file in status bar
 window.status = "Exported Voice Model to File Location: "+strFileLoc;
 else
}
```

```
// Show error message in status bar
window.status = "Export of Voice Models Failed: " + strErrMsg;
}

function HandleEndImport(intErrNum, strErrMsg)
{
 if (intErrNum == 0)
 // Show success message in status bar
 window.status = "Imported Voice Models Successfully";
 else
 // Show error message in status bar
 window.status = "Import of Voice Models Failed: " + strErrMsg;
}

function DoLogoff()
{
 ...
}

function cleanup()
{
 ...
}

</SCRIPT>
</HEAD>

<BODY bgcolor="#fffff" language="javascript" onload="InitializeAdminApp()"
onunload="DoLogoff()">

<form>

<!-- Set up window in HTML with list of users,
list of shortcuts, list of Words -->

...
<table>
<tr>
<td><input language="javascript" type="button" name="btnExportWords"
value="Export Words" onClick="ExportAllWords()"></td>

<td nowrap>Path to File of Custom Words:</td>
<input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
name="txtWordsFile">
<td><input language="javascript" type="button" name="btnImportWords"
value="Import Words" onClick="ImportWordsFromFile()"></td>

</tr>

<tr>
<td><input language="javascript" type="button" name="btnExportShortcuts"
value="Export Shortcuts" onClick="ExportAllShortcuts()"></td>

<td nowrap>Path to File of Shortcuts:</td>
<input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2"
name="txtShortcutsFile">
<td><input language="javascript" type="button" name="btnImportShortcuts"</pre>
```

```
 value="Import Shortcuts" onClick="ImportShortcutsFromFile()"">></td>

</tr>

<td><input language="javascript" type="button" name="btnExportCategories">
 value="Export Categories" onClick="ExportAllCategories()"></td>

<td nowrap>Path to File of Categories:</td>
<input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2" name="txtCategoriesFile">
<td><input language="javascript" type="button" name="btnImportCategories">
 value="Import Categories" onClick="ImportCategoriesFromFile()"></td>

</tr>

<td><input language="javascript" type="button" name="btnExportGroups">
 value="Export Groups" onClick="ExportAllGroups()"></td>

<td nowrap>Path to File of Groups:</td>
<input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2" name="txtGroupsFile">
<td><input language="javascript" type="button" name="btnImportGroups">
 value="Import Groups" onClick="ImportGroupsFromFile()"></td>

</tr>

<tr>
<td><input language="javascript" type="button" name="btnExportUsers" value="Export Users/Their Voice Models" onClick="ExportUsersAndVMs()">

<td nowrap>Path to File of Users:</td>
<input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2" name="txtUsersFile">
<td><input language="javascript" type="button" name="btnImportUsers" value="Import Users/Their Voice Models" onClick="ImportUsersAndVMs()">

</tr>

<tr>
<td nowrap>Report ID:</td>
<input style="WIDTH: 105, HEIGHT: 22" enabled maxlength="256" size="2" name="txtReportID">
<td><input language="javascript" type="button" name="btnDeleteReport">
 value="Delete Report" onClick="PurgeReport()">
</td>
</tr>

</table>

<input language="javascript" type="button" name="btnLogin">
 value="Login" onClick="DoLogin()">

<input language="javascript" type="button" name="btnLogout">
 value="Logout" onClick="DoLogoff()">

...
</form>
</BODY>
</HTML>
```

# *Taking Over Control of the Microphone*

## **Objectives**

This chapter presents how to take over the microphone in your *SDK* application. The same information applies to other dictation receiving devices, such as foot pedals. Before you proceed with this chapter, you should be familiar with the earlier chapter on the microphone, [Chapter 9, Configuring and Customizing the Microphone on page 219](#). You can then proceed with the following topics:

- [Taking Over Control of the Microphone](#)
- [Taking Custom Actions in ButtonDown Event Handler](#)
- [Code Summary](#)

# Taking Over Control of the Microphone



**Note:** Before you proceed with this topic, you should be familiar with the basics of working with the **Microphone** object and microphone buttons, as covered in [Chapter 9, Configuring and Customizing the Microphone](#) on page 219.

To take over control of all microphone buttons, after you create the **Microphone** object, you disable standard processing of button actions within your application. You disable that standard processing by calling the **ReceiveEvents()** method of the **Microphone** object and passing it **True**:

```
objMicrophone.ReceiveEvents(true);
```

Now, when you press the **DICTATE**, **TRANSCRIBE**, **LEFT**, and **RIGHT** buttons, the standard microphone actions do not occur. Instead, the application executes your custom code in response to the pressing of each button. You develop code for those actions in the **ButtonDown** event handler. In cases where it applies, you might also want to have custom code for responding when the button is released that would execute in the **ButtonUp** event handler.

## Taking Custom Actions in ButtonDown Event Handler

Your application receives the **ButtonDown** event whenever a button on the microphone or foot pedal has been pressed. If you have called **ReceiveEvents(true)**, you must have an action programmed in this handler for every button on the microphone that you want to use. Any button that you do not program will take no action at all.

You determine the particular button that has been pressed using the *MicButtonType* constants:

### MicButtonType Constants Sent by ButtonDown and ButtonUp Events

MicButtonType	Device	Button Label/Name or Graphic	Value
psMicButtonNone	PowerMic or PowerMic II	<i>None</i>	0
psMicButtonRecord	PowerMic or PowerMic II	DICTATE or	1
psMicButtonTranscribe	PowerMic or PowerMic II	Button A (upper left) or	2
psMicButtonForward	PowerMic or PowerMic II	FF or	3
psMicButtonTabForward	PowerMic or PowerMic II	Button B (upper right) or	4
psMicButtonPlayStop	PowerMic or PowerMic II	STOP/PLAY or	5
psMicButtonBottomLeft	PowerMic or PowerMic II	Button C (lower left) or left	6
psMicButtonRewind	PowerMic or PowerMic II	REW or	7

MicButtonType	Device	Button Label/Name or Graphic	Value
psMicButtonBottomRight	PowerMic or PowerMic II	Button D (lower right) or right 	8
psMicButtonScan	Scan	SCAN  or 	13
psMicButtonTabBackward	PowerMic II		14
psMicButtonEnter	PowerMic II		15
psFootButtonPlay	Foot Pedal	Play	9
psFootButtonRewind	Foot Pedal	Rewind	10
psFootButtonForward	Foot Pedal	Forward	11
psFootButtonStop	Foot Pedal	Stop	12

Suppose that you want to custom program the following buttons on your *PowerMic* microphone:

- **BOTTOM RIGHT—Create New Report.**

**Action:** Open new report using name from a text field.

Pop up a message prompting the user to begin dictating by pressing the **DICTATE** button.

- **DICTATE—Dictate report.**

**Action:** Begin recording dictated text. Pop up message indicating recording is in process.

- **TRANSCRIBE (upper left button)—Transcribe and Save (Leaving report open).**

**Action:** Transcribe dictated audio and save the report. Leave the report displaying in the editor. Pop up a message indicating action taken.

- **BOTTOM LEFT—Save and close.**

**Action:** Automatically save any current report in process, clear the text from the editor, and unlock (close) the report. Pop up a message indicating action taken.

- **STOP/PLAY—Copy Report.**

**Action:** Save the existing report, make a copy of it and hold it in the buffer. Pop up a message indicating action taken.

- **REW—Paste Copied Text into New Report.**

**Action:** Start a new report and paste the copied text from the buffer into the new report. Pop up a message indicating action taken.

None of the other buttons should take any action.

## Taking Custom Actions in Response to BOTTOM RIGHT Button

After you have disabled the standard processing of the microphone buttons, custom code that responds to a user pressing the **LOWER RIGHT** button could call several methods when the **btnPressed** passed to the handler is **psMicButtonBottomRight**.

To call these methods:

1. Create a new plain text report by calling the **NewReport()** method of the **PowerscribeSDK** object; then lock and activate the report, associate it with the **PlayerEditor** in the application, and be sure you initiate report events with **Advise()**:

```
objReport = pscribeSDK.NewReport(strOrdNumber);
objReport.SetPlayerEditor(objPlayerEditor);
objReport.ExclusiveLock(true);
pscribeSDK.ActivateReport(objReport);
objPlayerEditorCtl.EnablePlayer(true);
ReptSink.Advise(objReport);
```

2. Pop up a message prompting the user to begin dictating:

```
alert("Press DICTATE to begin dictating report " +strOrdNumber+ ".");
```

## Taking Custom Actions in Response to DICTATE Button

After you have disabled the standard processing of the microphone buttons, custom code that begins recording when the **DICTATE** button has been pressed could call the **Record()** method of the **Microphone** object when the **btnPressed** passed to the handler is **psMicButtonRecord**:

```
objMicrophone.Record();
alert("Recording report.");
```

## Taking Custom Actions in Response to TRANSCRIBE Button

After you have disabled the standard processing of the microphone buttons, custom code that responds to a user pressing the **UPPER LEFT** button could call the **Transcribe()** method of the **Microphone** object when the **btnPressed** passed to the handler is **psMicButtonTranscribe** and then save the report and pop up a message about the actions taken:

```
objMicrophone.Transcribe();
objReport.Save();
alert("Audio has been transcribed and saved.");
```

## Taking Custom Actions in Response to PLAY/STOP Button

After you have disabled the standard processing of the microphone buttons, custom code that responds to the user pressing the **PLAY/STOP** button could call several methods when the **btnPressed** passed to the handler is **psMicButtonPlayStop**.

### To call these methods:

1. After you save the report using the **Save()** method, retrieve all text from the report by calling the **GetText()** method of the **Report** object and store the text in a string:

```
objReport.Save();
strReptText = objReport.GetText();
```

2. Remove all text from the editor using the **ClearEditor()** method of the **Report** object:

```
objReport.ClearEditor();
```

3. Unlock the report by passing **False** to **ExclusiveLock()** and stop receiving events for the report by calling the **Unadvise()** method of associated **EventMapper** object:

```
objReport.ExclusiveLock(false);
ReptSink.Unadvise();
```

4. Pop up a message indicating the actions taken:

```
alert("Report text has been copied.");
```

## Taking Custom Actions in Response to REW Button

After you have disabled the standard processing of the microphone buttons, custom code that responds to the user pressing the **REW** button could call several methods when the **btnPressed** passed to the handler is **psMicButtonRewind**. To call these methods:

1. Create a new plain text report, just as you did for the **BOTTOM RIGHT** button:

```
if (strReptText != "")
{
 objReport = pscribeSDK.NewReport("CopyOf " + strOrdNumber);
 objReport.SetPlayerEditor(objPlayerEditor);
 objReport.ExclusiveLock(true);
 pscribeSDK.ActivateReport(objReport);
 objPlayerEditorCtl.EnablePlayer(true);
 ReptSink.Advise(objReport);
```

2. Merge the string you retrieved from the original report into the new report by calling the **MergeTemplate()** method of the **Report** object:

```
var statusMergeCode = objReport.MergeTemplate(strReptText);
}
```

3. Pop up a message indicating the actions taken:

```
alert("Copied text pasted from buffer into new report.");
```

## Taking Custom Actions in Response to BOTTOM LEFT Button

After you have disabled the standard processing of the microphone buttons, custom code that responds to the user pressing the **LOWER LEFT** button could call several methods when the **btnPressed** passed to the handler is **psMicButtonBottomLeft**. To call them:

1. Save the report by calling the **Save()** method of the **Report** object, then unlock it by passing **False** to the **ExclusiveLock()** method, and stop receiving events for that report by calling **Unadvise()** on the report **EventMapper**:

```
objReport.Save();
objReport.ClearEditor();
objReport.ExclusiveLock(false);
ReptSink.Unadvise();
```

2. Pop up a message indicating the actions that have transpired:

```
alert("Report has been saved and closed.");
```

## Restoring Default Microphone Functions

After your application no longer needs custom behavior of the microphone, or during exit from your application, you should restore the normal functioning of the microphone buttons. To reset the microphone buttons to their default functions, you call the **ReceiveEvents()** method of the **Microphone** object and passing it **False**:

```
objMicrophone.ReceiveEvents(false);
```

## Code Summary

```
<HTML> <HEAD>
<TITLE>Microphone Takeover</TITLE>
<!-- This code is not complete; it does not trap errors when buttons are
not used in the correct sequence --> <SCRIPT type="text/javascript"> ...

// Disable standard processing of microphone buttons:
function TakeOverMic()
{
 objMicrophone.ReceiveEvents(true);
}

// Re-enable standard processing of microphone buttons:
function ReleaseMic()
{
 objMicrophone.ReceiveEvents(false);
}
```

```
// Generate custom action to take in response to buttons pressed
function HandleButtonDown(btnPressed)
{
 // Define the constants for use in JavaScript
 var psMicButtonRecord = 1;
 var psMicButtonTranscribe = 2;
 var psMicButtonPlayStop = 5;
 var psMicButtonBottomLeft = 6;
 var psMicButtonRewind = 7;
 var psMicButtonBottomRight = 8;
 var strReptText; // String to hold report text

 switch btnPressed
 {

 case psMicButtonBottomRight:
 {
 objReport = pscribeSDK.NewReport(strOrdNumber);
 objReport.SetPlayerEditor(objPlayerEditor);
 objReport.ExclusiveLock(true);
 pscribeSDK.ActivateReport(objReport);
 objPlayerEditorCtl.EnablePlayer(true);
 ReptSink.Advise(objReport);

 alert("Press DICTATE to begin dictating " +strOrdNumber+ ".");
 break;
 }

 case psMicButtonRecord:
 {
 objMicrophone.Record();
 alert("Recording Report.");
 break;
 }

 case psMicButtonTranscribe:
 {
 objMicrophone.Transcribe();
 objReport.Save();
 alert("Audio has been transcribed and saved.");
 break;
 }

 case psMicButtonBottomLeft:
 {
 objReport.Save();
 objReport.ClearEditor();
 objReport.ExclusiveLock(false);
 ReptSink.Unadvise();
 alert("Report has been saved and closed.");
 break;
 }
 }
}
```

```
case psMicButtonPlayStop:
{
 objReport.Save();
 strReptText = objReport.GetText();
 objReport.ClearEditor();
 objReport.ExclusiveLock(false);
 ReptSink.Unadvise();
 alert("Report text has been copied into buffer.");
 break;
}

case psMicButtonRewind:
{
 if (strReptText != "")
 {
 objReport = pscribeSDK.NewReport("CopyOf " + strOrdNumber);
 objReport.SetPlayerEditor(objPlayerEditor);
 objReport.ExclusiveLock(true);
 pscribeSDK.ActivateReport(objReport);
 objPlayerEditorCtl.EnablePlayer(true);
 ReptSink.Advise(objReport);

 var statusMergeCode = objReport.MergeTemplate(strReptText);
 alert("Copied text pasted from buffer into new report.");
 }
 break;
}
}

...
</SCRIPT> </HEAD>
<BODY>
...
 <input language="javascript" type="button" name="btnTakeOver"
 value="Take Over Microphone" onClick="TakeOverMic()">

 <input language="javascript" type="button" name="btnReleaseMic"
 value="Release Mic from Takeover" onClick="ReleaseMic()">

...
</BODY>
</HTML>
```

# *Understanding Built-in Training and Adaptation*

## **Objectives**

This chapter presents how the training module built in to the *SDK* operates inside your custom application. In addition, the chapter presents how you can queue a user to be processed for training and processed for adaptation.

- [Operation of Built-in Training](#)
- [Working with Wav File Button](#)
- [Popping Up Modal Training Dialog](#)
- [Queueing User to Be Processed for Training](#)
- [Queueing User to Be Processed for Adaptation](#)

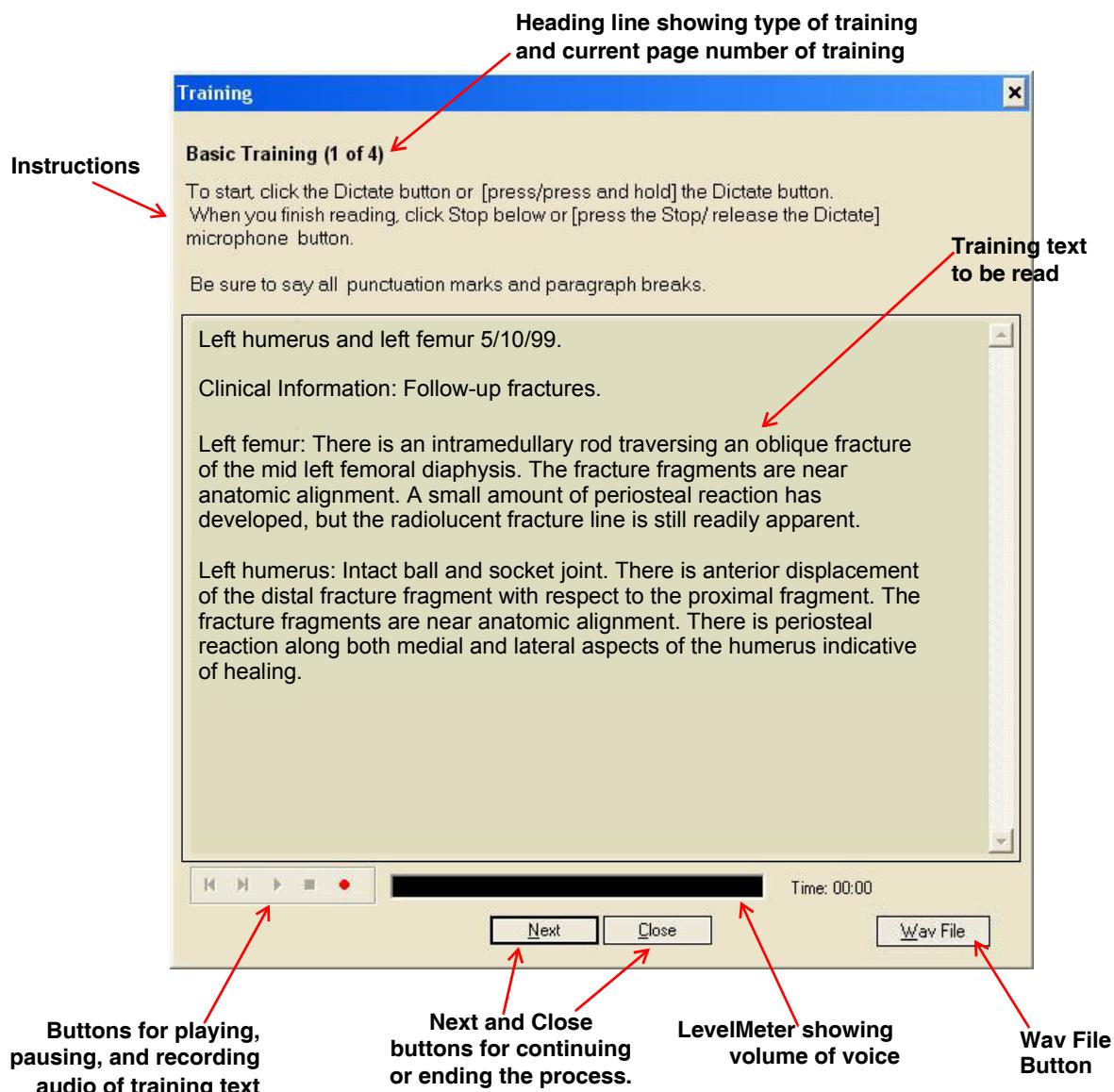
# Operation of Built-in Training

When any speech recognition user logs in for the first time, the built-in training **Enrollment** page pops up.

The user must then read the first page of training, called the **Enrollment** page. If the user does not complete reading of the **Enrollment** page, the attempt to log in is rejected.

Every time the user logs in, as long as he or she has not read the **Enrollment** page, the page pops up. After reading the **Enrollment** page (becoming *enrolled*), the user has the option to either continue training by clicking the **Next** button or, if satisfied with the quality of recognition being received, forego further training (by clicking the **Close** button) and proceeding to dictate.

Each page of the built-in training module includes several features shown in this illustration:



## Understanding Training Stages

The stages of training progress for each user include:

- **Enrollment**—First page of training. Required to log in.
- **Basic Training**—The first four pages of training .
- **Extended Training**—Any additional pages beyond **Basic Training**.

## Processing User for Training

If a user's training progresses to the beginning of the **Extended Training** pages, after the four pages of **Basic Training** or if the user exits from the last page of **Basic Training** and saves that last page, the *SDK* automatically adds that user to the queue of users to be *processed for training*. This operation builds a new acoustic model of the user's voice based on audio of his or her dictation of the training pages.

The *SDK* performs this processing operation on the *Recognition Server*. Depending on the processor speed of that server and how many other applications are running on that machine at the same time, processing a user for training can take from 5-10 minutes. The user can continue to work during this operation—the processing occurs in the background.

## Retrieving TrainingState of User

Whenever the user completes a page of training, the *SDK* evaluates the user's training progress. You can determine the phase of training the user has completed by retrieving the **TrainingState** property of the **User** object:

```
var intTrainLevel = objUser.TrainingState;
```

The normal levels of training progress for a user (starting at **psTrainingStateNeedsEnrollment** (0) and progressing to **psTrainingStateAdapted** (9)) are listed in the table below.

### TrainingState Property Settings

TrainingState Constant	Value	Explanation
psTrainingStateNeedsEnrollment	0	User has never read the first page of training, the <i>Enrollment</i> page.
psTrainingStateEnrolled	3	User has completed dictation of the first page of training.
psTrainingStateTrained	6	User has completed dictation of all pages of training.
psTrainingStateAdapted	9	The <i>Speech Recognition</i> engine has processed the required number of minutes of this user's dictation for adaptation.

# Working with Wav File Button

The **Wav File** button in the lower right corner of the **Training** page, below the time appears only if you set the **TrainingWaveFile** property of the **Training** object in your application.



*Note: The **Wav File** button appears only in standard training screens, not in your custom training screens.*

You need this button only if you have dictated or plan to dictate into a non-streaming audio device, such as a recorder, rather than a microphone, which is a streaming audio device.

## To record your training wave files on a recording device rather than using a microphone:

1. Print out the training pages supplied on the product DVD.
2. Read each page into a separate wave file.
3. After you read one page and save its wave file, be sure to name it to indicate it is page 1 of training (**JohnTrainingPage1.wav**). Name subsequent files to distinguish the particular page of training audio they contain.

## To ensure the button shows up in the Training pages, you take the following steps:

1. Create a **Training** object by calling the **Training** property of the **PowerscribeSDK** object:  

```
objTraining = pscribeSDK.Training;
```
2. Set the **TrainingWaveFile** property of the **Training** object to the full path to the first wave file that contains audio dictated through a non-streaming device:  

```
objTraining.TrainingWaveFile = "C:\WaveFiles\JohnTrainingPage1.wav";
```

Once you set this property to the name of a valid file containing recorded audio, the **Wav File** button appears in the lower right corner of each training page, starting with the **Enrollment** page, shown below. You click the button to open a browse dialog, where you browse until you find the wave file that contains the audio for the displaying screen's text. After you find that audio, you click **Next** and repeat the process of finding a wave file for each page's text, until you have selected a wave file for each page of text dictated.

Once you use the button to retrieve the wave file for the page, then the subsequent click of the **Next** or **Close** button saves that audio file as part of the training audio and associates it with the particular training page before proceeding.

# Popping Up Modal Training Dialog

In C#, C++, or Visual Basic .NET, to ensure that a **Training Wizard** dialog pops up after login has succeeded, you set the **ParentWindowHandle** property of the **PowerscribeSDK** object to the handle of the **Training Wizard** dialog's parent window:

```
pscribeSDK.ParentWindowHandle = m_Handle.ToInt32();
```

Be sure to cast the handle as a long integer. If you do not set this property, the parent remains NULL and the **Training Wizard** dialog does not appear.

# Queueing User to Be Processed for Training



*Note: If you have just created a user, you do not have to queue the user for training; that process is automatic.*

If a user is having trouble with speech recognition, you can have the user retrain the same data by queueing the user for training on the server. You queue the user for training by calling the **Train()** method of the **User** object:

```
objUser.Train();
```

This method does not invoke training immediately, but *schedules* (queues) the job. (The *SDK* does not allocate a specific date or time.) When the user has logged in and his or her training job reaches the front of the queue, the standard **Training Wizard** pops up. The server then processes the audio from the training, but you receive no special event when the training is complete, so to find out if it is complete, your application might proactively check the user's **TrainingState** property after the user logs in.

The levels of training progress for a user (from **psTrainingStateNeedsEnrollment** (0) and progressing to **psTrainingStateAdapted** (9)) are shown in the next table. (*Enrollment* is reading the first page of the training that pops up when the user logs in for the first time.)

## TrainingState Property Settings

TrainingState Constant	Value	Explanation
psTrainingStateNeedsEnrollment	0	User has never started training.
psTrainingStateEnrolled	3	User has completed dictation of the first page.
psTrainingStateTrained	6	User has completed dictation of all training pages.
psTrainingStateAdapted	9	The <i>Speech Recognition</i> engine has processed the required number of minutes of dictation by this user.
<i>SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.</i>		

# Queueing User to Be Processed for Adaptation

Similarly to the way the *PowerScribe SDK* automatically *processes a user for training*, it also *processes a user for adaptation*. When the user has dictated a sufficient number of reports, the *SDK* automatically adds that user to the queue of users to be processed for adaptation. That adaptation takes place on the *Recognition Server*.

The adaptation operation retrieves speech information from the reports that the user has dictated and uses that information to tweak his or her voice model.

*PowerScribe SDK* considers a user who has dictated the required number of reports *qualified for adaptation*.

You can also proactively queue a user for adaptation, by calling the **Adapt()** method of the **User** object:

```
objUser.Adapt();
```

The **Adapt()** method *schedules* (queues) the adaptation job for the particular user, but the user does not have to be qualified for adaptation for the job to be in the queue—when you place the job in the queue, you anticipate that by the time the user reaches the front of the queue, he or she becomes qualified for adaptation. If, at that time, the user is not qualified for adaptation, the process fails. If the process fails, you can requeue the job until the adaptation process occurs.

## Option to Modify Built-in Training Pages

If you are interested in modifying the built-in training pages, you can modify their look and feel, but not the text for training the user's voice. For more information on how to customize the training pages, proceed to the next chapter.

# *Creating Custom Training Module*

## **Objectives**

This chapter presents how to use an advanced feature of the *SDK*—creating your own custom training module and integrating it with your application. If you have not already read the previous chapter, [Understanding Built-in Training and Adaptation on page 435](#), you should at least peruse it before proceeding with this chapter:

- [Steps to Creating Custom Training Module](#)
- [Creating a Training Object](#)
- [Creating an EventMapper Object](#)
- [Securing Microphone and LevelMeter Exclusively for Training](#)
- [Starting Training](#)
- [Inserting Training Type into Your HTML](#)
- [Inserting Training Text into Your HTML](#)
- [Inserting Training Time into Your HTML](#)
- [Handling Training Events](#)
- [Releasing the Microphone and LevelMeter](#)
- [Integrating Custom Training into Login Process](#)
- [Adding Wave File Button to Custom Training](#)

# Steps to Creating Custom Training Module

Although you cannot customize the text that users read to teach the speech recognition engine their voices, you can customize the look and feel of the training pages.

To customize the training pages, take the following major steps in your application:

- Create HTML for your custom training page
- Create a **Training** object using the **PowerscribeSDK** object
- Create an **EventMapper** object to map event the **Training** object fires
- Create a **Microphone** object so that the application can later secure the microphone exclusively for the training process
- Secure use of the microphone and, if it is in the application, the **LevelMeter** for the **Training** object
- Start the training process
- Insert the type of training into the heading of your custom training HTML
- Insert the standard text into the main text block of your custom training HTML
- Modify the way the time displays in your custom training HTML
- Generate and respond to the buttons in your custom training for moving to next or previous pages and ending training or starting over
- When training is complete, release the microphone and **LevelMeter** from exclusive use by the **Training** object

Regardless of the type of training your application uses, built-in training or custom training, your application can always queue the user for training and adaptation. For details on how queue the user for training, refer to these topics in [Chapter 19, Understanding Built-in Training and Adaptation on page 435](#):

- [Queueing User to Be Processed for Training on page 439](#)
- [Queueing User to Be Processed for Adaptation on page 440](#)

Whenever your application calls the **Stop()** method of the **Training** object, the *SDK* evaluates the user's training progress. The value of the user's **TrainingState** property is based on the number of pages of training completed. For more information on the **TrainingState** property, refer to [Chapter 19, Understanding Built-in Training and Adaptation on page 435](#).

# Creating a Training Object

To customize training pages, you start by creating some HTML that you later integrate with standard training page text.

Your HTML needs to provide a window to display the training text and a heading to display the type and page number of the training. For these custom sections, you use `<span>` blocks within your window with the `id` attribute settings shown in the table below.

**Span Blocks for Custom Training HTML**

<b>id Attribute</b>	<b>Purpose</b>
trainingType	Display of the type of training and page number of training, to appear on each page.
trainingText	Display the training text the user needs to dictate.
trainingTime	Display the time that has elapsed during the current training session.

You can set the style to any you want for the heading or training text.

Then you create a **Training** object using the **Training** property of the **PowerscribeSDK** object:

```
objTraining = pscribeSDK.Training;
```

# Creating an EventMapper Object

To work effectively with the **Training** object, you create an **EventMapper** object to receive the **Training** object's events:

```
trainingSink = new ActiveXObject("PowerscribeSDK.EventMapper");
```

Using the **EventMapper** object, map the events to their handlers:

```
trainingSink.EnrollProgress = HandleEnrollProgress;
trainingSink.MoveToNextPage = HandleMoveToNextPage;
trainingSink.WaveFileProgress = HandleTrainWaveFileProgress;
```

To begin receiving the **Training** object events, call the **Advise()** method of the **EventMapper** object and pass it the **Training** object:

```
trainingSink.Advise(objTraining);
```

# Securing Microphone and LevelMeter Exclusively for Training

Training requires the use of the microphone, so you should instantiate the **Microphone** object and create an **EventMapper** object to map its events:

```
objMicrophone = pscribeSDK.Microphone;
micSink = new ActiveXObject("PowerscribeSDK.EventMapper");
micSink.ButtonDown = HandleButtonDown;
micSink.Advise(objMicrophone);
```

Before training can begin, your application should secure the microphone exclusively for the use of the **Training** object by calling the **SetMicrophone()** method of the **Training** object:

```
objTraining.SetMicrophone(objMicrophone);
```

You can also secure the **LevelMeter**, if you have one in your application, for exclusive use by the **Training** object:

```
objTraining.SetLevelMeter(objLevelMeter);
```

## Starting Training

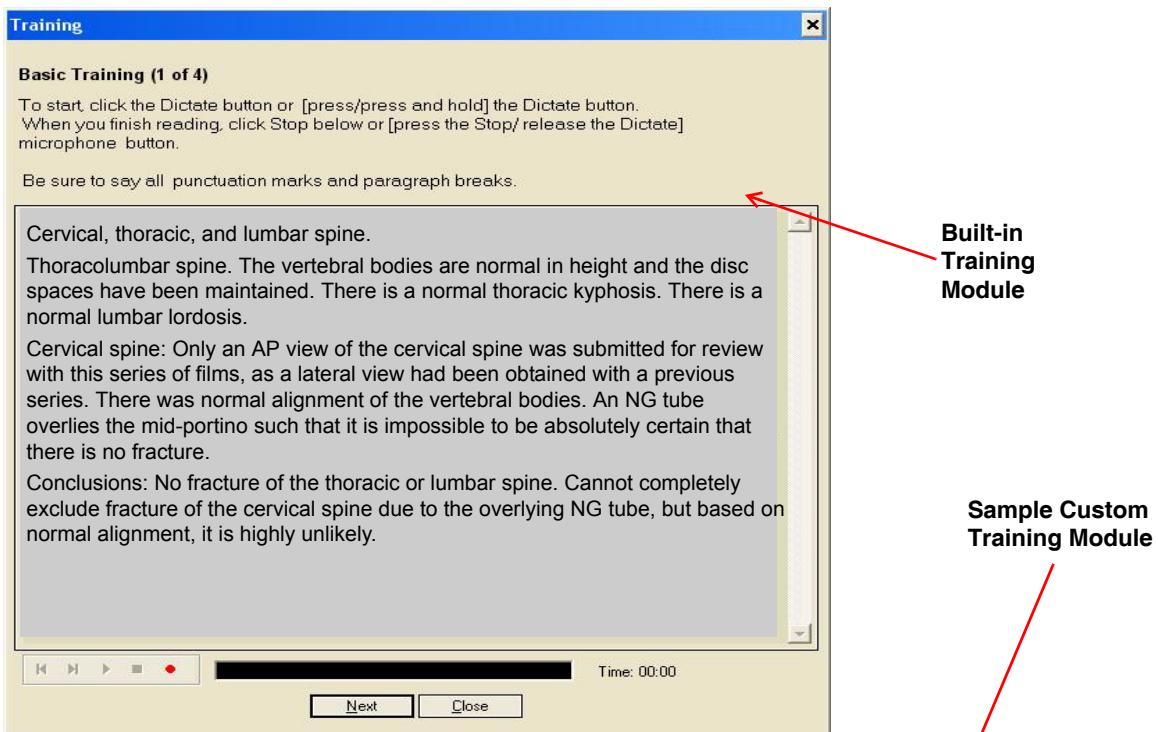
Once your application is receiving events and you have secured use of the microphone exclusively for training, you can start training by calling the **Start()** method of the **Training** object and passing it either **True** to start at enrollment, or **False** to continue with the training page that follows the last page the user completed. In this case, let's pass it **False** to pick up where training of the user last left off:

```
objTraining.Start(false);
```

The application might then construct your custom GUI by calling a custom function:

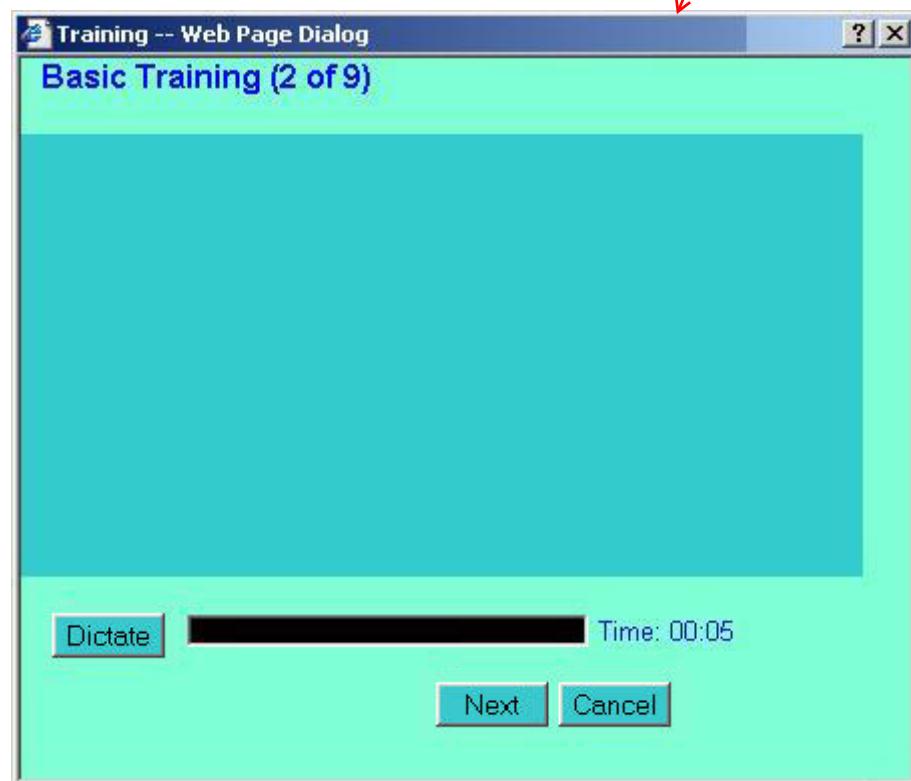
```
BuildCustomTrainingGUI();
```

The first illustration below shows the default training GUI. You might choose to create a



custom GUI that uses, for instance, different fonts and font colors, different background colors, and different buttons, like the GUI shown in the adjacent illustration, where the training text is not yet displaying.

You create this shell and the *SDK* supplies the text to insert into your HTML.



# Inserting Training Type into Your HTML

The first portion of the HTML code not only indicates the background color of the full page and font color of the heading text that says **Basic Training (# of #)**, but also contains a **<span>** block with an **id** attribute setting of **trainingType**.

```
<body bgcolor="#7FFFDD" onload="SetupTraining()" onunload="EndTraining()">
 ...


```

Your application can interact with this portion of the HTML by retrieving the **CurrentTrainingType** property of the **Training** object. This property indicates the current phase of the user's training and can have one of these values:

CurrentTrainingType	Value	Explanation
psTrainingTypeNone	0	User has not started training.
psTrainingTypeEnrollment	1	User has started training, but has not completed the <b>Enrollment</b> page.
psTrainingTypeBasic	2	User is in the process of training on the <b>Basic</b> pages.
psTrainingTypeExtended	3	User is in the process of training on the <b>Extended</b> pages.
<i>SDK constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.</i>		

Your code might start by creating a string to hold the text you want in the heading:

```
var strHeading = "";
```

You could then set the value of the heading string based on the value of the **CurrentTrainingType** property and two other properties of the **Training** object—the **CurrentPageNumber** and the **TotalPageNumber**, to indicate the number of the current page being trained and the total number of pages, respectively:

```
switch (objTraining.CurrentTrainingType)
{
 case psTrainingTypeEnrollment:
 strHeading = "Enrollment (" + objTraining.CurrentPageNumber
 + " of " + objTraining.TotalPageNumber + ") ";
 break;

 case psTrainingTypeBasic:
 strHeading = "Basic Training (" + objTraining.CurrentPageNumber
 + " of " + objTraining.TotalPageNumber + ") ";
 break;

 case psTrainingTypeExtended:
 strHeading = "Extended Training (" + objTraining.CurrentPageNumber
 + " of " + objTraining.TotalPageNumber + ") ";
 break;
}
```

```
 ...
}
```

Once you have set the heading string, you can pass it to the HTML <span> block by setting the **trainingType** attribute of the block (from **id=trainingType**) to the string:

```
trainingType.innerHTML = strHeading;
```

## Inserting Training Text into Your HTML

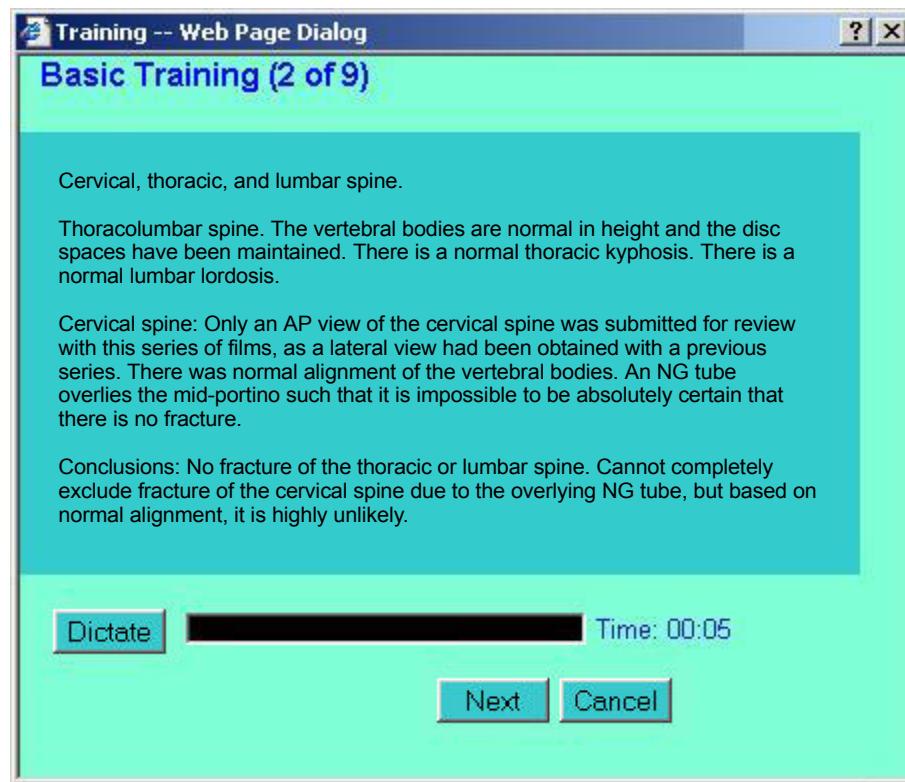
A second portion of the HTML to include for your training page contains another <span> block whose **id** attribute you set to **trainingText**:

```
<table border="0" cellpadding="10" cellspacing="0">...
 ...
</table>
```

You place the training text inside the inner block of your GUI by assigning the **trainingText** attribute (from **id=trainingText**) of the span block the value of the **CurrentPageText** property of the **Training** object:

```
var strTrainText = objTraining.CurrentPageText;
trainingText.innerHTML = strTrainText.replace(/$/mg, "
");
```

As a result of taking this action, the standard training text should now appear on the page.



# Inserting Training Time into Your HTML

You can include the time in your training page by having another `<span>` block with an `id` attribute set to **trainingTime**:

```
 ...
 ...

```

You would initialize the time to **00:00** by setting the **trainingTime** attribute value:

```
trainingTime.innerHTML = "Time: 00:00";
```

To update the time, you can handle the **Microphone** object's **WavePosition** event, fired as the number of milliseconds recorded increases during the recording process. If you choose to handle this event as part of your custom training, you can use it to update the display of the number of seconds of training that has elapsed.



*Note: The wave position value tends to differ for the same position when you switch from recording to playing back the audio, because the engine removes silences from the recording. As a result, the playback audio length is shorter than the recording length.*

Your **WavePosition** event handler might take the number of milliseconds it receives and update the time displaying in the GUI each time the event occurs:

```
function HandleWavePosition(milliseconds)
{
 // Update the number of milliseconds of training in display.
 var strTime = "Time: ";

 var seconds = Math.floor(milliseconds/1000);
 var minutes = Math.floor(seconds/60);
 seconds = seconds % 60;

 if (minutes < 60)
 {
 strTime = strTime + "00:";
 strTime = strTime + minutes + ":";

 }
 if (seconds < 60)
 strTime = strTime + seconds;

 trainingTime.innerHTML = strTime;
}
```

# Handling Training Events

At the bottom of the training GUI, you should create at least two buttons, one to proceed to the next screen (in this example **NEXT**) and one to stop (in this example **CANCEL**). In addition, you might include a button to start (in this example **DICTATE**) and a **PREVIOUS** or **BACK** button to return to the previous page.

In response to the **DICTATE** button, you might call the **Microphone.Record()** method. The buttons you would respond to by calling a **Training** object method are the **NEXT** button, the **PREVIOUS** or **BACK** button, and the **CANCEL** button. Your application might also include a **START OVER** button that you can respond to by calling a **Training** object method.

## Responding to the **NEXT** Button

In response to the **NEXT** button, your application could call the **NextPage()** method of the **Training** object to save the audio from the current training page and move to the next page:

 **Note:** *The audio must be a minimum of 15 seconds long or the **NextPage()** method cannot save it.*

```
function onNext()
{
 try
 {
 objTraining.NextPage();
 BuildCustomTrainingGUI();
 }
 catch(error)
 {
 alert(error);
 }
}
```

When your application calls **NextPage()**, after the method executes, the **Training** object fires an event. If the page just trained was the first (**Enrollment**) page, then the event that the object fires is an **EnrollProgress** event. The handler for this event receives two arguments, the *ProgressType* and *ProgressText* (refer to the next table for possible values).

### EnrollProgress Event, ProgressType Argument Values

Constant	Value	Corresponding ProgressText
psEnrollProgressTypeInProgress	1	Enrollment in progress.
psEnrollProgressTypeSaveInProgress	2	Enrollment being saved.
psEnrollProgressTypeSuccess	3	Enrollment successful.
psEnrollProgressTypeFail	4	Enrollment has failed.

Constants are available in C# and Visual Basic .NET; in JavaScript, you must use numeric values unless you define the constants.

The handler might check to see if the enrollment has failed and handle that situation:

```
function HandleEnrollProgress(progType, progText)
{
 if (progType == psEnrollProgressTypeFail)
 {
 alert(progText);
 }
}
```

If the page just trained was a Basic or Extended Training page, when your application calls the **NextPage()** method and the method has executed, the **Training** object fires the **MoveToNextPage** event. The handler for this event receives the result of the method, either success or an error code, and should handle the event:

```
function HandleMoveToNextPage(hresult)
{
 if (hresult == 0)
 BuildCustomTrainingGUI();
 else
 alert("Error " + hresult + " occurred during move to next page.");
}
```

## Responding to the PREVIOUS or BACK Button

When the user presses the **PREVIOUS** or **BACK** button, you can respond by calling the **PreviousPage()** method of the **Training** object, but you should give the user the option to cancel the change, because calling this method erases the wave file for the page and you can never recover that audio if you proceed.

```
answer = confirm("Are you sure you want to erase the audio for this page?");

if (answer == true)
 objTraining.PreviousPage();
 ...
else
 // Redisplay the current page
```

Another issue with returning to the previous page is that you cannot always go to the previous page. For instance, if the user is on the first page of Basic Training, you cannot use the **PreviousPage()** method to return to the enrollment page. Instead, you should first check the value of the **CurrentTrainingType** property, and if it indicates that the training type is **psTrainingTypeEnrollment**, you call the **Start()** method and pass it **True** to start training from enrollment:

```
if (answer == true)
{
 switch (objTraining.CurrentTrainingType)
 {
 case psTrainingTypeEnrollment:
 alert("Training starting again at Enrollment");
 objTraining.Start(true);
 BuildCustomTrainingGUI();
 break;
 }
}
```

You can make the same method call if the **CurrentTrainingType** property is equal to **psTrainingTypeBasic** and the **CurrentPageNumber** property is equal to **1**. As long as the **CurrentPageNumber** is greater than **1**, you can call **PreviousPage()** instead:

```
case psTrainingTypeBasic:
 if (objTraining.CurrentPageNumber <= 1)
 {
 alert("Training starting again at Enrollment");
 objTraining.RestartFromEnrollment();
 objTraining.Start(true);
 BuildCustomTrainingGUI();
 }
 if (objTraining.CurrentPageNumber > 1)
 {
 alert("Training returning to previous page.");
 objTraining.PreviousPage();
 BuildCustomTrainingGUI();
 }
 break;

...
default:
 break;
}
```

You can use the same logic if **CurrentTrainingType** is **psTrainingTypeExtended**, testing whether the **CurrentPageNumber** property is equal to or greater than **1**.

To erase all training completed so far, no matter what stage training is at, you can call the **Training.RestartFromEnrollment()** method, then call the **Start()** method to actually begin the training process.

When the **Start()** method executes and restarts at enrollment, the **Training** object fires the **EnrollProgress** event. By contrast, when the **PreviousPage()** method executes, the **Training** object does not fire an event.



*Note: The number of Basic Training pages varies depending on the language model.*

## Responding to the CANCEL Button

When the user presses the **CANCEL** button, you can respond by asking the user if he or she wants to save the audio, then calling the **Stop()** method of the **Training** object and passing it **True** to save the training from the current page or **False** to discard that audio:

```
function OnCancel()
{
 answer = confirm("Do you want to save the audio from this page?");

 if (answer == true)
 objTraining.Stop(true); // True saves the trained audio
 else
 objTraining.Stop(false); // False discards the audio
}
```



*Note: Although no **WaveFileProgress** event is fired after calling the **Stop()** method of the **Training** object with a **True** argument, the method returns only after the current wave file of the most recent page of training has been saved in its entirety.*

## Responding to the START OVER Button

If you have a button the user should press to indicate he or she wants to start training over from the beginning, you can respond to that button by calling the **Start()** method of the **Training** object and passing it **True** to start the training process from the **Enrollment** (first) page:

```
function OnStartOver()
{
 objTraining.Start(true);
}
```

# Releasing the Microphone and LevelMeter

After the user's training is complete, the application can release the **Microphone** from exclusive use of the training process by calling the **ReleaseMicrophone()** method of the **Training** object:

```
objPhoneTrnscriber.ReleaseMicrophone();
```

If the **LevelMeter** has also been secured for the **Training** object, it is automatically released.

# Integrating Custom Training into Login Process

To have your custom training module pop up when the user logs in, instead of the default training, you change the way you call the **Login()** method of the **PowerscribeSDK** object.

## To integrate your custom training into the login process:

1. Call the **Login()** method with the *ShowEnrollment* (fourth) argument set to **False**. This actions stops the automatic enrollment page from opening:

```
pscribeSDK.Login(user.text, passwd.text, true, false, psAudioDevicePowerMicII);
```

2. After the login succeeds, you need to determine whether or not the user is enrolled by testing the **User** object's **AcousticModel** object's **TrainingState** property. You start that process by retrieving the currently logged in user's **User** object using the  **GetUser()** method of the **PSAdminSDK** object:

```
objUser = objAdmin.GetUser();
```

3. Retrieve the currently logged in user's acoustic model by accessing the **UserProfile** object and retrieving the value of its **AcousticModelType** property:

```
var currAcousticMod = pscribeSDK.objUserProfile.AcousticModelType;
```

4. Find the user object's equivalent **AcousticModel** in the **AcousticModels** collection for the **User** object:

```
for (i = 0; i <= objUser.AcousticModels.Count; i++)
{
 if (objUser.AcousticModels.item(i).type == currAcousticMod)
 objAccousticModel = objUser.AcousticModels.item(i);
}
```

5. Test its **TrainingState** property and if it is at the pre-enrollment level (user is not enrolled), then start the custom training:

```
var psTrainingStateNeedsEnrollment = 0;

if (objAccousticModel.TrainingState == psTrainingStateNeedsEnrollment)
{
 // Execute the Custom Training:
 SetupTraining();
 // Have the user read enough pages for adequate training level
}
```

Your application can also allow the user to initiate training using a GUI button, in order to go beyond the basic (adequate) level your application requires.

# Adding Wave File Button to Custom Training

You can add a button to your custom training that is like the **Wav File** button that you can add to standard training. You need this button only if you have dictated or plan to dictate into a non-streaming audio device, such as a recorder, rather than a microphone, which is a streaming audio device.

## To record your training wave files on a recording device rather than using a microphone:

1. Print out the training pages supplied on the product DVD.
2. Read each page into a separate wave file.
3. After you read one page and save its wave file, be sure to name it to indicate it is page 1 of training (**JohnTrainingPage1.wav**). Name subsequent files to distinguish the particular page of training audio they contain.

To include a **Wave File** button that retrieves the wave files recorded with a non-streaming audio input device, such as a recorder, you create the button in HTML, then in response to a click of the button you can set the **TrainingWaveFile** property of the **Training** object to the path to the file selected in the select file dialog that pops up:

```
<html>
...
<input id="btnWaveFile" type="button" value="Wave File" onclick="SetTrainingWaveFile()">
<input id="btnNext" type="button" value="Next" onclick="NextPage()">
<input id="btnEndTraining" type="button" value="End Training" onclick="Stop()">
...
<script>
SetTrainingWaveFile()
{
 // Display browser dialog
 // and retrieve selected path in strTraining string
 objTraining.TrainingWaveFile = strTraining;
}
...
</script>
...
</html>
```

The user would click the **Wave File** button in each training page to select the file for that page.

# Code Summary

```

<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>Example of Custom Training</title>
</head>

<body bgcolor="#7FFFDD" onload="SetupTraining()" onunload="EndTraining()">

 <!-- Training Type Heading -->

<table border="0" cellpadding="10" cellspacing="0">
 <tr bgcolor="#33CCCC">
 <td width="99%" height="250" valign="top">
 <font color="#0000FF" face="arial,helvetica,sans-serif"
 point-size="12" style="text-align:justify">
 <!-- Training Text Block -->

 </td><td bgcolor="#7FFFDD" width="1%">&nbsp</td></tr>
 </table>

 <!-- Dictate Button -->
<input id="btnDictate" style="left:20px;position:absolute;width:56px;
 height:22px;background-color:#33CCCC;color=black"
 type="button" value="Dictate" onclick="OnDictate()">

<!-- Insert a LevelMeter into the Application in a Div block -->
<div id="LevelMeterobj">
 <object id="objLevelMeter" height=16 width=200 style="left:83px;
 position:absolute" data="data:application/x-oleobject;
 base64,DfwvSUoKrU68UpZq4VaPzQADAACBDwAAEQIAAA=="
 classid="clsid:492FFC0D-0A4A-4EAD-BC52-966AE1568FCD">
 </object>
</div>

 <!-- Elapsed Training Time Line -->
 Time:
 00:00


```

```
<!-- Next and Cancel Buttons -->
<input id="btnNext" style="left:190px;position:absolute;width:72px;
height:22px;background-color:#33CCCC;color:black" type="button"
value="Next" onclick="OnNext()">

<input id="btnCancel" style="left:274px;position:absolute;width:72px;
height:22px;background-color:#33CCCC;color:black" type="button"
value="Cancel" onclick="OnCancel()">

<!-- Start Over and Back Buttons -->

<input id="btnStartOver" style="left:190px;position:absolute;width:72px;
height:22px;background-color:#33CCCC;color:black" type="button"
value="Start Over" onclick="OnStartOver()">

<input id="btnBack" style="left:274px;position:absolute;width:72px;
height:22px;background-color:#33CCCC;color:black" type="button"
value="Back" onclick="OnBack()">

</body>

<script>

// Training Type Codes
var psTrainingTypeEnrollment = 1;
var psTrainingTypeBasic = 2;
var psTrainingTypeExtended = 3;

// Wave File Progress Types
var psWaveFileProgressLoadInProgress = 1;
var psWaveFileProgressSaveInProgress = 2;
var psWaveFileProgressSaveEnd = 3;
var psWaveFileProgressLoadEnd = 4;
var psWaveFileProgressFileSize = 5;

// Microphone Record Button
psMicButtonRecord = 1;

// Enrollment Progress Types
var psEnrollProgressTypeInProgress = 1;
var psEnrollProgressTypeSaveInProgress = 2;
var psEnrollProgressTypeSuccess = 3;
var psEnrollProgressTypeFail = 4;

// Variables
var objTraining;
var pscribeSDK;
var trainingSink;
var micSink;
var objMicrophone;
var win = window.dialogArguments;
var minutes = "00";
var seconds = "00";
var strTime = "Time: " + minutes + ":" + seconds;
```

```
function SetupTraining()
{
 try
 {
 // Instantiate Training Object
 pscribeSDK = win.top.script.pscribeSDK;
 alert("PowerScribe SDK object retrieved");
 objTraining = pscribeSDK.Training;
 alert("objTraining retrieved");

 // Create Training EventMapper and Map Events to Handlers
 trainingSink = new ActiveXObject("PowerscribeSDK.EventMapper");
 trainingSink.EnrollProgress = HandleEnrollProgress;
 trainingSink.MoveToNextPage = HandleMoveToNextPage;
 trainingSink.WaveFileProgress = HandleTrainWaveFileProgress;
 // Start firing and receipt of Training events in application
 trainingSink.Advise(objTraining);

 // Instantiate Microphone Object, Create EventMapper/Map Events
 objMicrophone = win.top.script.Microphone;
 alert("objMicrophone retrieved");
 micSink = new ActiveXObject("PowerscribeSDK.EventMapper");
 micSink.ButtonDown = HandleButtonDown;
 micSink.ButtonUp = HandleButtonUp;
 micSink.WavePosition = HandleWavePosition;

 // Start firing and receipt of Microphone events in application
 micSink.Advise(objMicrophone);

 // Call function to instantiate LevelMeter
 CreateObjectCtl("LevelMeterobj");
 alert("LevelMeter instantiated");

 // Secure Microphone and LevelMeter for Training Purposes
 objTraining.SetMicrophone(objMicrophone);
 alert("Microphone secured");
 objTraining.SetLevelMeter(objLevelMeter);
 alert("LevelMeter secured");

 // Start Training Process (False picks up where user last stopped)
 objTraining.Start(false);
 BuildCustomTrainingGUI();
 }
 catch(error)
 {
 alert(error.description);
 }
}

function BuildCustomTrainingGUI()
{
 // Set the strHeading to 'Training Type (Page X of X)':
 var strHeading = "";
 var strTrainText = "";
```

```
switch (objTraining.CurrentTrainingType)
{
 case psTrainingTypeEnrollment:
 strHeading = "Enrollment (" + objTraining.CurrentPageNumber +
 " of " + objTraining.TotalPageNumber + ") ";
 break;

 case psTrainingTypeBasic:
 strHeading = "Basic Training (" + objTraining.CurrentPageNumber +
 " of " + objTraining.TotalPageNumber + ") ";
 break;

 case psTrainingTypeExtended:
 strHeading = "Extended Training (" + objTraining.CurrentPageNumber +
 " of " + objTraining.TotalPageNumber + ") ";
 break;

 default:
 break;
}

// Insert heading into HTML:
trainingType.innerHTML = strHeading;

// Retrieve the training text and insert it into the HTML:
strTrainText = objTraining.CurrentPageText;
trainingText.innerHTML = strTrainText.replace(/\$/mg, "
");

// Insert elapsed training time into the HTML:
trainingTime.innerHTML = strTime;

// Activate LevelMeter:
objLevelMeter.Activate(true);
}

function EndTraining()
{
 alert("EndTraining function called.");

 objTraining.Stop(false);
 objTraining.ReleaseMicrophone();

 alert("Training stopped and microphone released.");

 micSink.Unadvise();
 micSink = null;

 trainingSink.Unadvise();
 trainingSink = null;

 alert("EventMappers released.");
}
```

```
function HandleMoveToNextPage(hresult)
{
 alert("MoveToNextPage event occurred.");
 if (hresult == 0)
 {
 BuildCustomTrainingGUI();
 }
 else
 {
 alert("Error " + hresult + " occurred during move to next page.");
 }
}

function HandleEnrollProgress(progType, progText)
{
 if (progType == psEnrollProgressTypeFail)
 {
 alert(progText);
 }
}

function HandleTrainWaveFileProgress(compress, type, txtSent, txtRemain)
{
 // Print message to status bar of Training window
 if (type == psWaveFileProgressSaveInProgress)
 window.status = "Bytes Sent: "+txtSent+"; Remaining: "+txtRemain;
}

function HandleButtonDown(nButton)
{
 switch (nButton)
 {
 case psMicButtonRecord:
 btnDictate.value = "Stop";
 break;

 default:
 break;
 }
}

function HandleButtonUp(nButton)
{
 switch (nButton)
 {
 case psMicButtonRecord:
 btnDictate.value = "Dictate";
 break;

 default:
 break;
 }
}
```

```
function HandleWavePosition(milliseconds)
{
 // Update the number of milliseconds of training in display.

 strTime = "Time: ";
 seconds = Math.floor(milliseconds/1000);
 minutes = Math.floor(seconds/60);
 seconds = seconds % 60;

 if (minutes < 60)
 {
 strTime = strTime + "00:";
 strTime = strTime + minutes + ":";

 }
 if (seconds < 60)
 strTime = strTime + seconds;

 trainingTime.innerHTML = strTime;
}

function OnDictate()
{
 try
 {
 // Switch the button text from Dictate to Stop or vice versa
 if (btnDictate.value == "Dictate")
 {
 btnDictate.value = "Stop";
 objMicrophone.Record();
 }
 else
 {
 btnDictate.value = "Dictate";
 objMicrophone.Stop();
 }
 }
 catch(error)
 {
 alert(error.description);
 }
}

function OnNext()
{
 alert("Next button pressed. OnNext function running.");
 try
 {
 objTraining.NextPage();
 BuildCustomTrainingGUI();
 }
 catch(error)
 {
```

```
 alert(error.description);
 }

}

function OnCancel()
{
 alert("Cancel button pressed. OnCancel function running.");
 answer = confirm("Do you want to save the audio from this page?");
 if (answer == true)
 {
 // True saves the training page if it has been dictated
 objTraining.Stop(true);
 alert("Saving training page");
 }
 else
 {
 objTraining.Stop();
 alert("Discarding training page");
 }

 window.close();
 EndTraining();
}

function OnBack()
{
 alert("Back button pressed. OnBack function running.");
 answer = confirm("Are you sure you want to erase the audio for this page?");
 if (answer == true)
 {
 switch (objTraining.CurrentTrainingType)
 {
 case psTrainingTypeEnrollment:
 alert("Training starting again at Enrollment");
 objTraining.Start(true);
 BuildCustomTrainingGUI();
 break;

 case psTrainingTypeBasic:
 if (objTraining.CurrentPageNumber <= 1)
 {
 alert("Training starting again at Enrollment");
 objTraining.Start(true);
 BuildCustomTrainingGUI();
 }

 if (objTraining.CurrentPageNumber > 1)
 {
 alert("Training returning to previous page.");
 objTraining.PreviousPage();
 }
 }
 }
}
```

```
 BuildCustomTrainingGUI();
 }
 break;

 case psTrainingTypeExtended:
 if (objTraining.CurrentPageNumber <= 1)
 {
 alert("Already on first page of Extended Training");
 break;
 }

 if (objTraining.CurrentPageNumber > 1)
 {
 alert("Training returning to previous page.");
 objTraining.PreviousPage();
 BuildCustomTrainingGUI();
 }
 break;

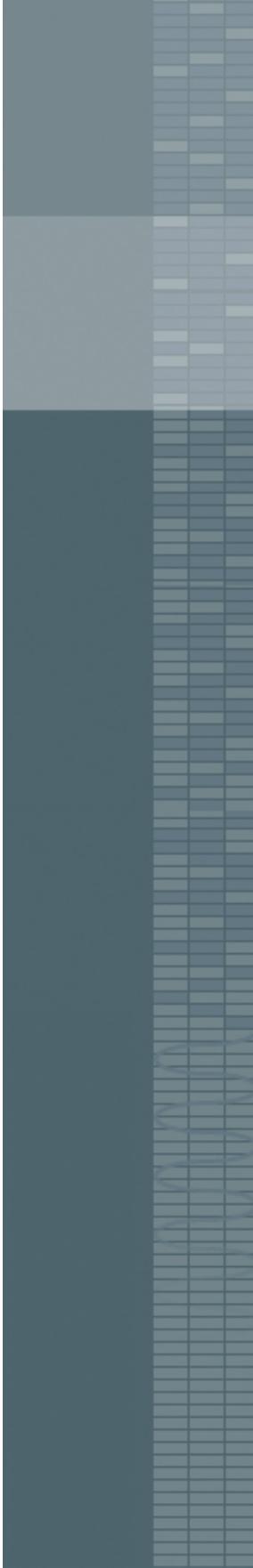
 default:
 break;
}
}

if (answer == false)
{
 alert("Audio for the last page you read retained.");
}
}

function OnStartOver()
{
 alert("StartOver button pressed. OnStartOver function running.");
 objTraining.Start(true);
 BuildCustomTrainingGUI();
}

function CreateObjectCtl(divID)
{
 var divRef = document.getElementById(divID);
 if (divRef)
 {
 divRef.innerHTML = divRef.innerHTML;
 }
}

</script>
</html>
```



## Appendix A

# *HeadingStyle and ParagraphStyle Options*

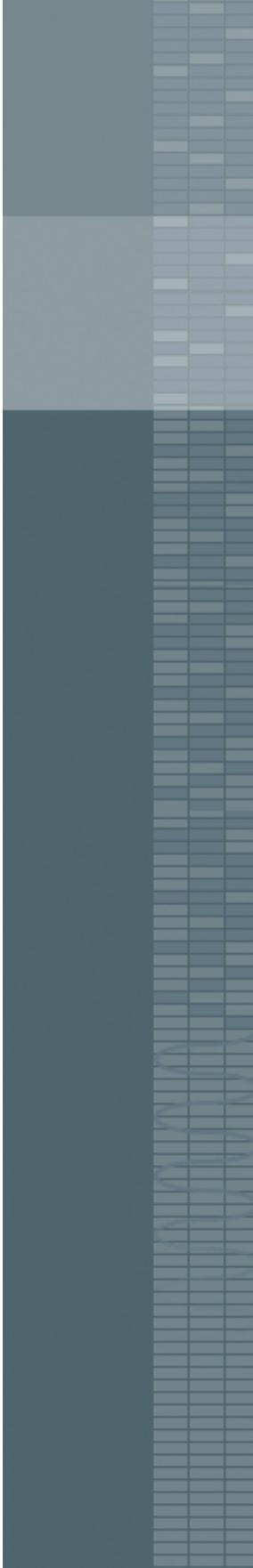
This appendix lists the **ParagraphStyle** and **HeadingStyle** formatting options available in structured reports.

# Heading and Paragraph Styles

The following table presents possible values for the attributes used to define paragraphs and heading styles in structured reports:

- **HeadingStyle** and **ParagraphStyle** properties of the **Section** object
- *HeadingStyle* and *ParagraphStyle* arguments of the **Add()** method of the **Sections** object

Attribute	Description	Values	Example(s)
font-size	Sets size of font.	points (pt) x-small small medium large x-large	{font-size: 12pt} {font-size: small}
font-family	Sets typeface.	typeface name font family name	{font-family: courier}
font-weight	Sets thickness of type.	normal bold	{font-weight: bold}
font-style	Italicizes text.	normal italic	{font-style: italic}
color	Sets color of text.	RGB triplet	{color: #00FF00}
text-decoration	Underlines the text.	none underline	{text-decoration: underline}
tab-count	Insert number of tabs before the text.		{tab-count: 2}
background	Sets background color of the text.	RGB triplet	{background-color: #33CC00}



## Appendix B

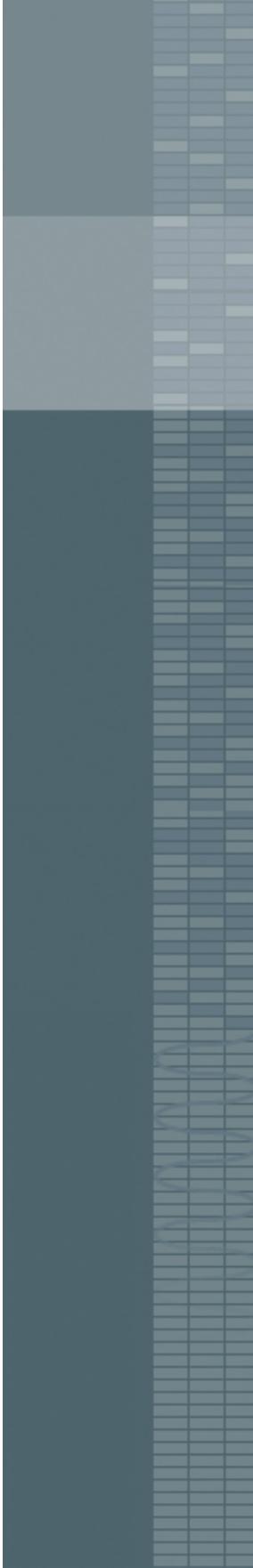
# *Empty Structured Report Template*

This appendix provides an empty structured report template.

# Empty Structured Report Template

Below is a sample of an empty structured report template:

```
<?xml version="1.0" encoding="UTF-8" ?>
<clu xmlns="http://www.dictaphone.com/HSG/CLU/Extraction/2002-12-02"
 xmlns:t="http://www.dictaphone.com/HSG/CLU/Template/2002-12-02"
 xmlns:html="http://www.w3.org/1999/xhtml" >
 <doc>
 <body>
 <!-- The body contains <Section> same format as in template.xsd -->
 </body>
 </doc>
</clu>
```



## Appendix C

# *XML Template Schema*

This appendix contains the schema for an empty XML template that you can use with the **XmlTemplateEdit** object.

# XML Template Schema

Below is a sample of an empty structured report template:

```
<xs:schema targetNamespace="http://tempuri.org/XMLSchema.xsd"
elementFormDefault="qualified"

 xmlns="http://tempuri.org/XMLSchema.xsd" xmlns:mstns=
 "http://tempuri.org/XMLSchema.xsd"

 xmlns:t="http://www.dictaphone.com/HSG/PSSDK/2004-11-30"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:complexType name="pssdk">
 <xs:sequence>
 <xs:element ref="shortcut" />
 </xs:sequence>
 </xs:complexType>

 <xs:element name="shortcut">
 <xs:complexType>
 <xs:sequence>
 <xs:element ref="body" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>

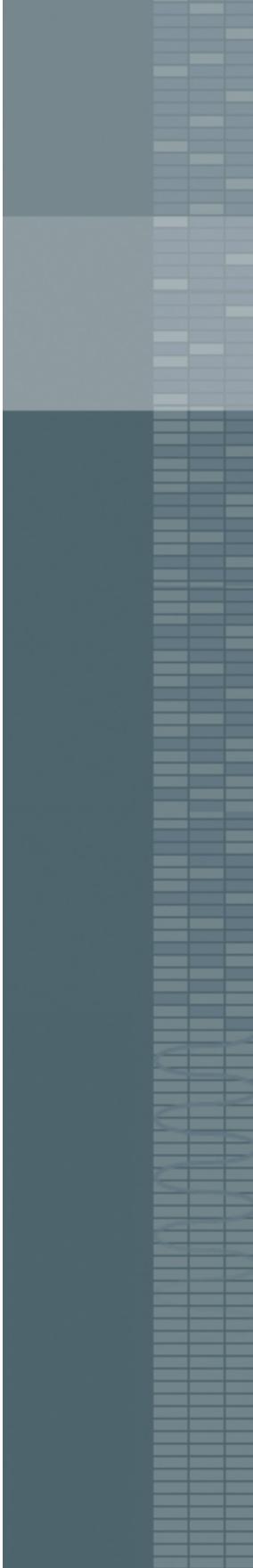
 <xs:element name="body">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="plainSection" type="plainSectionType" />
 <xs:element name="section" type="sectionType" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>

 <xs:complexType name="plainSectionType">
 <xs:sequence>
 <xs:element name="p" type="paragraphType" />
 </xs:sequence>
 </xs:complexType>

 <xs:complexType name="paragraphType">
```

```
<xs:sequence>
 <xs:element ref="t:span" />
</xs:sequence>
<xs:attribute name="html:style" type="xs:string" />
</xs:complexType>
<xs:element name="span">
 <xs:complexType>
 <xs:sequence />
 <xs:attribute name="html:style" type="xs:string" />
 </xs:complexType>
</xs:element>
<xs:complexType name="sectionType">
 <xs:sequence>
 <xs:element ref="heading" />
 <xs:element name="p" type="paragraphType" />
 </xs:sequence>
 <xs:attribute name="html:style" type="xs:string" />
 <xs:attribute name="Level" type="xs:string" />
</xs:complexType>
<xs:element name="heading">
 <xs:complexType>
 <xs:sequence />
 <xs:attribute name="html:style" type="xs:string" />
 </xs:complexType>
</xs:element>
</xs:schema>
```





## Appendix D

# ***XML Style Format Template Schema***

This appendix provides information about the schema for the XML *style formats* template used by the **XmlTemplateEdit** object to create a template for structured reports or structured shortcuts.

# Style Formats Schema

Here is the XML schema you should use to create the *style format* template:

```
<xs:schema targetNamespace="http://tempuri.org/XMLSchema.xsd"
 elementFormDefault="qualified"
 xmlns="http://tempuri.org/XMLSchema.xsd" xmlns:mstns="http://
 tempuri.org/XMLSchema.xsd"
 xmlns:t="http://www.dictaphone.com/HSG/PSSDK/2004-11-30"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="pssdk">
 <xs:sequence>
 <xs:element ref="template" />
 </xs:sequence>
</xs:complexType>
<xs:element name="template">
 <xs:complexType>
 <xs:sequence>
 <xs:element ref="styleFormats" maxOccurs="1" />
 </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="styleFormats">
 <xs:complexType>
 <xs:sequence>
 <xs:element ref="styleFormat"/>
 </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="styleFormat" minOccurs="1">
 <xs:complexType>
 <xs:attribute name="Name" type="xs:string" />
 <xs:attribute name="Type" type="xs:string" />
 <xs:attribute name="html:style" type="xs:string" />
 <xs:attribute name="Level" type="xs:string" />
 </xs:complexType>
</xs:element>
</xs:schema>
```

## ***PowerMic and PowerMic II Microphone Buttons***

This appendix shows the buttons on the *PowerMic* and *PowerMic II* microphones, indicating:

- Name used to refer to each button
- Default function(s) of each button
- Microphone object method that is equivalent to pressing the button
- *MicButtonType* value received by the **ButtonDown** event handler when the button is pressed

# PowerMic in PowerScribe® SDK



## Diction Mode Buttons, Their Equivalent Methods & MicButtonType

Microphone Button	Method & MicButtonType	Function
<b>LED</b>	<b>Solid Red</b> —Microphone is on and is recording. <b>Solid Green</b> —Microphone is on and the last barcode scan is validated. The LED remains solid green until another microphone button is pressed. <b>Flashing Green</b> —Microphone is on and the last barcode scan has activated a warning or dialog box. This LED remains flashing until the user acknowledges the dialog box. <b>Off</b> —Microphone is off and the LED indicator is not illuminated. This is the normal state when the microphone is not dictating or scanning.	
<b>DICTATE</b>	<b>Record()</b> method psMicButtonRecord	Begins audio recording.
<b>FF</b>	<b>Forward()</b> method psMicButtonForward	Fast forwards the recorded audio, highlighting corresponding text.
<b>STOP/PLAY</b>	<b>Play()</b> or <b>Stop()</b> method psMicButtonPlayStop	Stops or starts audio playback. Highlights corresponding text during playback.
<b>REW</b>	<b>Rewind()</b> method psMicButtonRewind	Rewinds the recorded audio, highlighting corresponding text.
<b>SCAN or Customizable</b>	<b>Scan()</b> method psMicButtonScan	Scans an accession number barcode. If the microphone has no scanner, this button can be programmed.

## Navigation and Selection Buttons, Their Equivalent Methods & MicButtonType

Microphone Button	MicButtonType	Function
<b>A (top left) Transcribe</b>	<b>Transcribe()</b> method psMicButtonTranscribe	Transcribes dictated audio and displays text.
<b>A (top left) Selection</b>	<b>Left()</b> method psMicButtonTranscribe	Accepts the default selection in a template.
<b>B (top right) Tab or Navigate Forward</b>	<b>Right()</b> method psMicButtonTabForward	Moves to the next field in square brackets.
<b>B (top right) Selection</b>	<b>Right()</b> method psMicButtonTabForward	Selects the next field in square brackets or, in a structured report, the next section.
<b>C (bottom left) Custom Button C</b>	psMicButtonBottomLeft	Takes the programmed action.
<b>D (bottom right) Custom Button D</b>	psMicButtonBottomRight	Takes the programmed action.

## PowerMic II in PowerScribe® SDK



Dictation/Navigation Mode Buttons, Equivalent Methods & MicButtonType		
Microphone Button	Method & MicButtonType	Function
MICROPHONE	Not applicable.	Talk into the PowerMic II microphone.
TRANSCRIBE	Transcribe() method psMicButtonTranscribe	Press to end current dictation/display transcribed text. Press to position insertion point or select a sentence.
DICTATE	Record() method psMicButtonRecord	Press to begin recording your report.
TAB BACK OR NAVIGATE BACKWARD	TabBackward() method psMicButtonTabBackward	Press to move to previous dialog box or field. After dictating, press to end dictation/display transcribed text.
TAB OR NAVIGATE FORWARD	TabForward() method psMicButtonTabForward	Press to move to next dialog box or field. Press to select text after transcribing.
REW	Rewind() method psMicButtonRewind	Press to rewind recorded audio. Press to scroll down through displayed list.
FF	Forward() method psMicButtonForward	Press to fast forward through recorded audio. Press to scroll up through displayed list.
STOP/PLAY	Play() and Stop() methods psMicButtonPlayStop	Press to start or stop audio playback.
ENTER/SELECT Customizable ENTER Button	psMicButtonEnter	Press to take the custom programmed action.
Customizable LEFT (Left of ENTER)	Left() method psMicButtonBottomLeft	Press to take the custom programmed action.
Customizable RIGHT (Right of ENTER)	Right() method psMicButtonBottomRight	Press to take the custom programmed action.
LEFT/RIGHT MOUSE BUTTONS	Not applicable.	Use the same way you use your mouse buttons.
POINTING DEVICE	Not applicable.	Use to position the insertion point and select objects, just as you would with the mouse.
SCAN (if no scanner, Customizable Scan Button)	Scan() method psMicButtonScan	On mic with scanner, scans MRN or accession number barcode. On mic without scanner, takes programmed action.
TRIGGER (on back of Microphone)	Not applicable.	Use the same way you use the left mouse button.

## PowerMic & PowerMic II in PowerScribe® SDK

When you use the scanner on your microphone to scan a barcode (MRN, accession number, order number, or any other barcode), it beeps to indicate that the scan was successful.

To turn the beep off if you do not want to hear it after a successful barcode scan, scan this barcode.

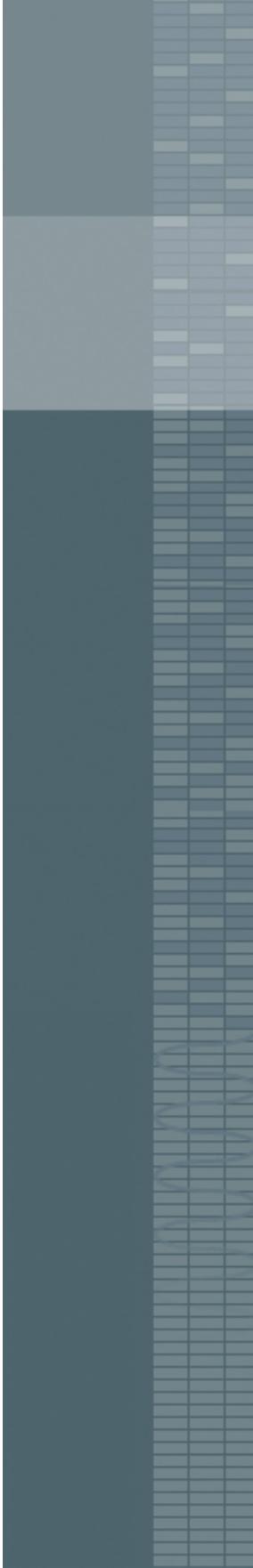


To turn the beep back on, scan this barcode.



### Dictation Tips & Techniques

- Dictation Preparation — Collect your thoughts and plan your words before speaking.
- Position the microphone between 2 and 5 inches from your mouth, but not touching. Keep the microphone at a constant distance from your mouth.
- At the start of your dictation, press the **DICTATE** button and pause briefly before you start speaking.
- Turn the microphone off when you are not speaking or if you move away from it.
- Speak naturally at your normal rate.
- Speak at a normal rate, not too quickly or too slowly. You should not:  
T a a a l k s l o o o o o w w w l y y..... Or. Say. Only. One. Word. At. A. Time.
- Speak at a normal, constant volume.
- Speak as you would to someone sitting across the desk from you. Do not speak too loudly or too softly.
- Enunciate properly — Fast dictation is acceptable as long as the words are spoken clearly and not slurred.
- Word beginning and ending sounds are important for recognition accuracy.
- Small words — Take more time when saying small words. Don't run the words together. For example, "and there was," "there is no," and similar small word phrases.
- Small words — Add vocal inflection to increase or emphasize where the words are important.
- Pause slightly before and after small words such as *a* and *the* if they are being lost or misrecognized.
- When pronouncing the word *a*, pronounce it as in *say*, rather than as in *uh*.
- Dictate punctuation.
- Speak in cadence of 6-8 word phrases followed by a brief pause.
- Stop dictating when engaging in side conversations or when there are excessive background noises. You must be the primary speaker.
- Dictate in a quiet area to minimize background noise as much as possible. Stay away from machines, radios, fans, and crowds.
- Avoid clearing your throat and yawning while you are dictating. Do not talk through a yawn or when you clear your throat. Please stop dictating.
- Eliminate utterances (*ums*, *ahs*, coughing) and similar sounds.
- Do not allow the tone or volume of your voice to trail off or fluctuate at the end of sentences or around small words.
- Do not over-enunciate or elongate spoken words.
- Do not speak too loudly or softly.
- Do not pause in the middle of a word.
- Do not chew gum or eat while dictating.
- Do not repeat large segments of text and verbal instructions to the editor.
- Do not use the microphone to point at films or other objects.



## Appendix F

# *Audio Converter Tool for Upgrading Compressed Audio*

This appendix shows you how to run the *Audio Converter* tool provided with *PowerScribe SDK*. You use this tool to convert compressed audio from earlier versions of *PowerScribe SDK* on your server to PCM format.

# Introducing Audio Converter

As of Version 3.2, *PowerScribe SDK* uses a new audio compression format. The *Audio Converter* tool converts previously existing compressed audio files to PCM format. You use the tool when you are upgrading from an earlier version of *PowerScribe SDK* to convert compressed audio of reports, adaptation, and training.

The tool looks for audio files compressed using the old format in the **Wave** directory at this path on your web server:

**[http://<servername>/pscribesdk\\_data/Wave/](http://<servername>/pscribesdk_data/Wave/)**

You must run the tool from a machine where you have the *PowerScribe SDK* client installed and where you have Vianix compression drivers installed. Before you run the tool, be sure that no users (or very few) are actually dictating on clients.

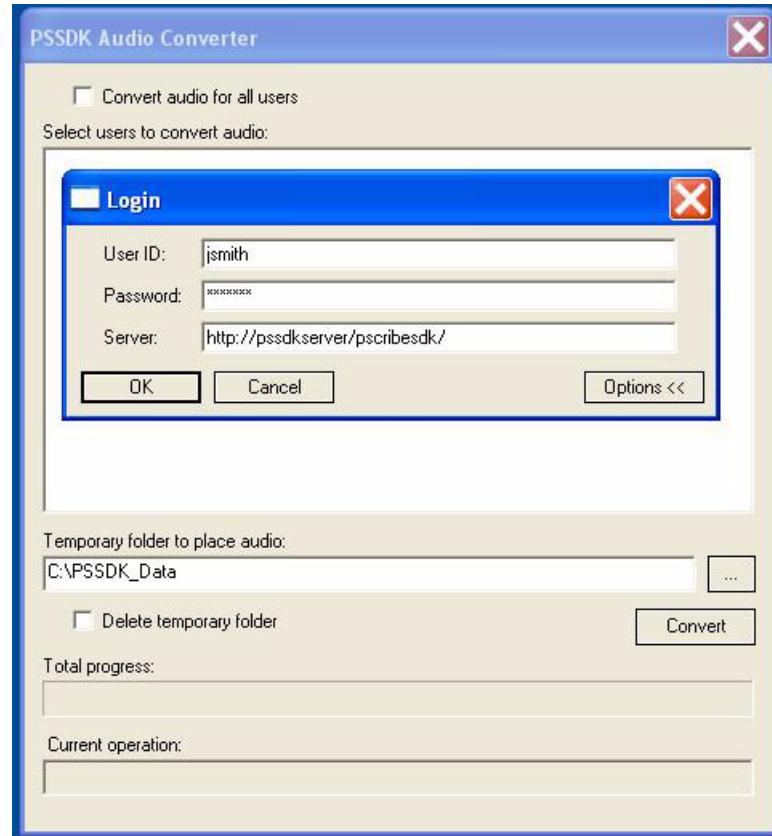
# Installing Audio Converter Tool



**Note:** You must have administrative privileges to use the Audio Converter tool.

To install the tool:

1. Find the **Tools** directory on the product DVD. In that directory, locate the **PSSDKAudioConverter.exe** file.
2. (optional) If you would like, you can copy the **.exe** file and the **PSAudi.dll** file that accompanies it onto your client machine, but if you do, be sure to keep the two files in the same directory.
3. Run the **.exe** file by double clicking on it. The tool pops up a **Login** dialog box where you provide a *PowerScribe SDK* user ID and password. Click the **Options>>** button to

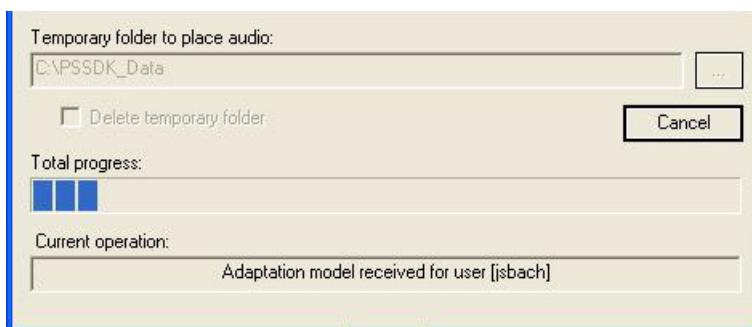
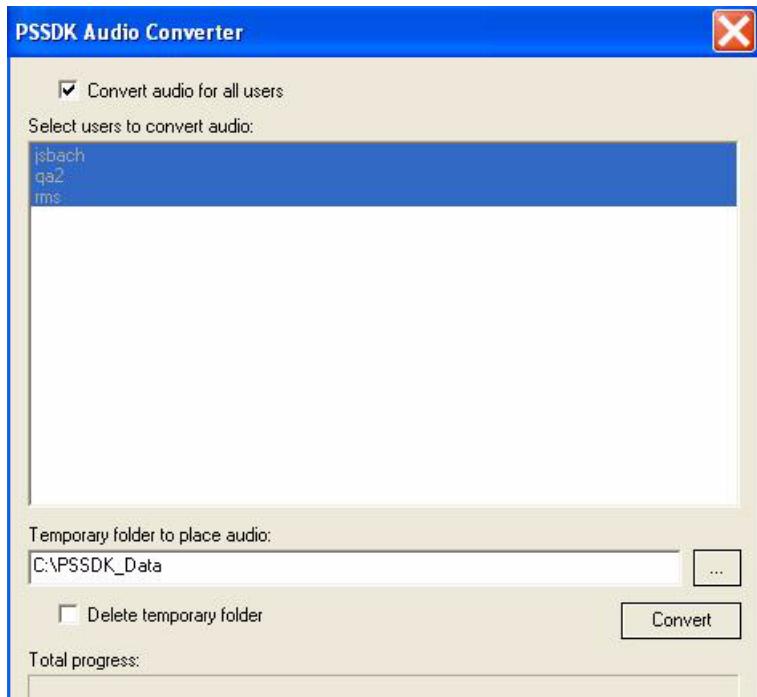


display the **Server** text box and enter the path (starting with **http://**) to your web server.

## Running Audio Converter Tool

Once you log in, you can convert audio either for all users or for a particular user/set of users:

1. To convert audio for all users, click the **Convert audio for all users** check box near the top of the dialog box; all users in the list box are then selected. To convert audio for particular users, ignore the check box and select those users in the list.
2. In the **Temporary folder to place audio** text box enter the full path to the temporary data folder on your client where the tool should store the converted files.
3. If you want to start with an empty temporary folder (which is a good idea), click the **Delete temporary folder** check box. Otherwise, the folder retains any previously converted files stored there.
4. To start the conversion process, click the **Convert** button (see above).



Watch the progress bar under **Total progress**. As each aspect of the conversion occurs, the **Current operation** box shows that operation and the name of the user whose audio it is converting.

The tool writes a log file that records action it has taken and places that log in the directory of the executable.

When the process is complete, the tool displays a message indicating the status of the results.

If the process has been successful, the tool then uploads the converted files to the server in the `<dataDirectory>\Wave\Report` directory and deletes them from the `<dataDirectory>\Wave\ReportCompressed` directory.



If the converted files are training files, the tool uploads the converted files to the `<dataDirectory>\Wave\Training` directory, where they replace the original audio files.

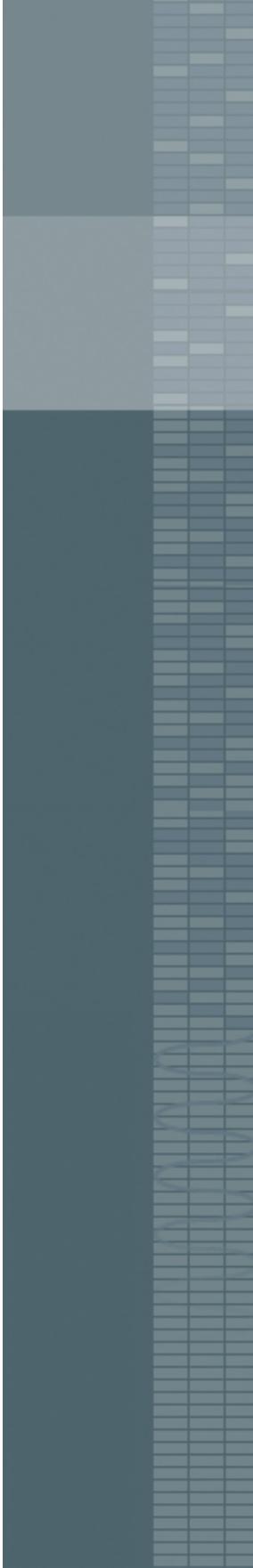
If the converted files are those used for adapting the user's acoustic model, the tool uploads them to the `<dataDirectory>\Adaptation` directory and deletes them from the `<dataDirectory>\Wave\AdaptReportCompressed` directory.

For a speaker with a user name of `jsbach`, the full path to, for instance, a training audio file that has been converted might be: `X:\PSSDK_Data\MyServer\jsbach\Wave\Training\125.wav`



*Note: If the user has more than one type of device that he or she has trained on and dictates with, the files are in separate subdirectories for each device type. Those directories have names such as `handheld`, `bluetooth`, `array`, or `telephony`, to name a few.*

*For a speaker with a user name of `jsbach`, the full path to a converted training audio file that he created using his handheld microphone rather than his bluetooth might be:  
`X:\PSSDK_Data\MyServer\jsbach\Wave\Training\handheld\125.wav`*



## Appendix G

# *Setting Up PowerMic II for Use on Virtual Machine*

This appendix shows you how to set up a virtual machine so that you can dictate on it using your *PowerMic II* microphone with *PowerScribe SDK*.

- [Requirements for Operating PowerMic II on VMWare Virtual Machine](#)
- [Configuring USB Ports for PowerMic II](#)
- [Notes on Configuring Dictaphone USB Device Ports on Virtual Machines](#)

# Requirements for Operating PowerMic II on VMWare Virtual Machine

Version 3.2 of *PowerScribe SDK* supports use of *PowerMic II* microphone on a VMWare Version 5 or 6 virtual machine. The configuration requirements for this functionality are:

- VMWare 5 or VMWare 6 Workstation software installed where you want to run virtual machines
- VMWare Tools supplied with VMWare Workstation installed on virtual operating systems
- *PowerMic II* drivers provided with *PowerScribe SDK* installed on both the host and virtual machines.
- *PowerScribe SDK* client installed on the virtual machines only (not required on host)

## Configuring USB Ports for PowerMic II

Even though you are going to be using the *PowerMic II* on the virtual machine, you must configure the USB port on both the host and the VMWare Workstation virtual machines.

### Host Machine

First you configure the USB port on the host machine:

1. Start a VMWare Workstation virtual machine on the computer where speakers will dictate using the *PowerMic II* microphone.
2. Plug the *PowerMic II* in to the host machine (**not** the machine running the VMWare Workstation).

The *PowerMic II* appears under the **Human Interface Devices** section of the Windows Explorer tree on the host system, as shown in the adjacent illustration.

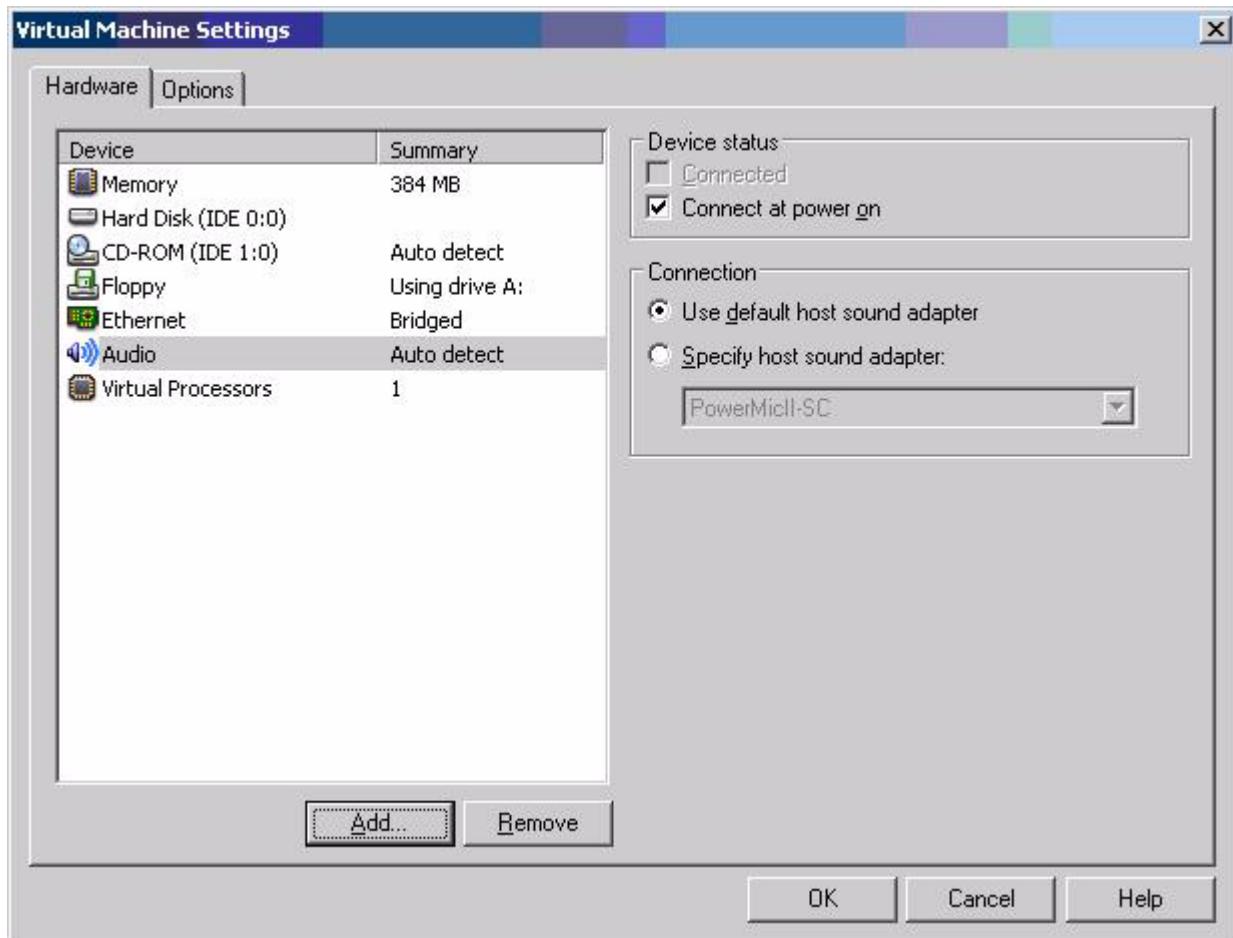


## Virtual Machine

Next, you configure the computer where you installed the VMWare Workstation virtual machine software.

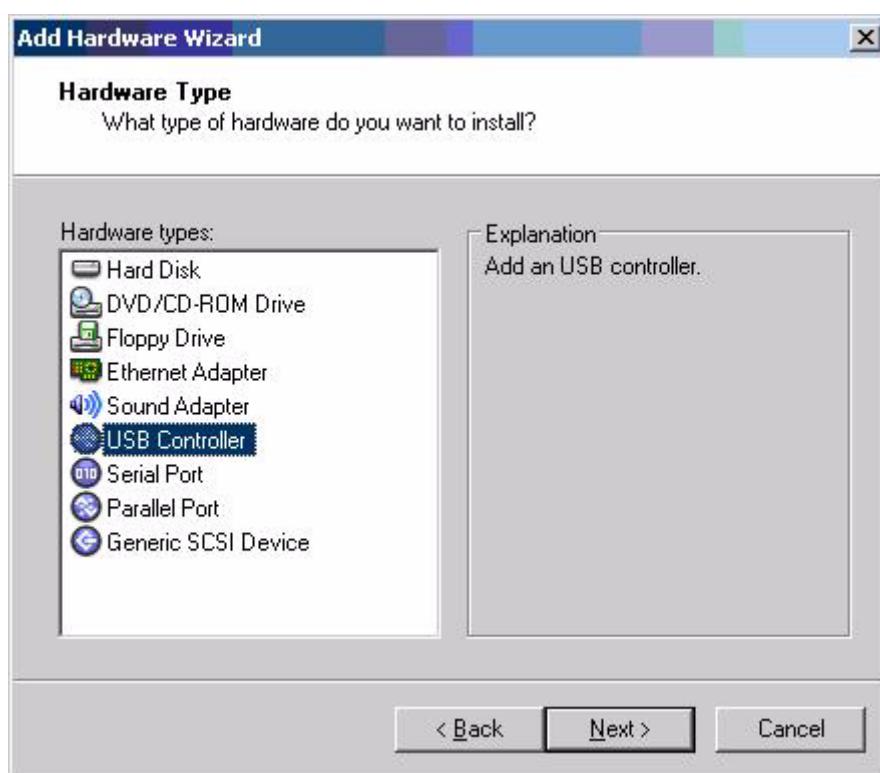
3. If the virtual machine does not already have one, add a USB controller to the virtual machine by going to the VMWare Workstation window and selecting **VM > Settings > Hardware**.

The following virtual machine dialog box appears.

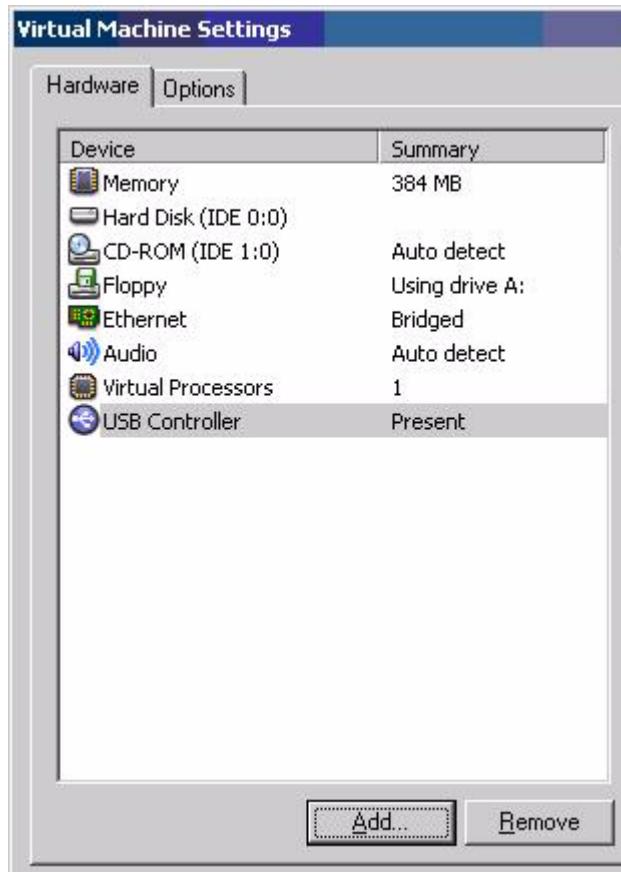


4. To add the USB controller, click the **Add...** button.

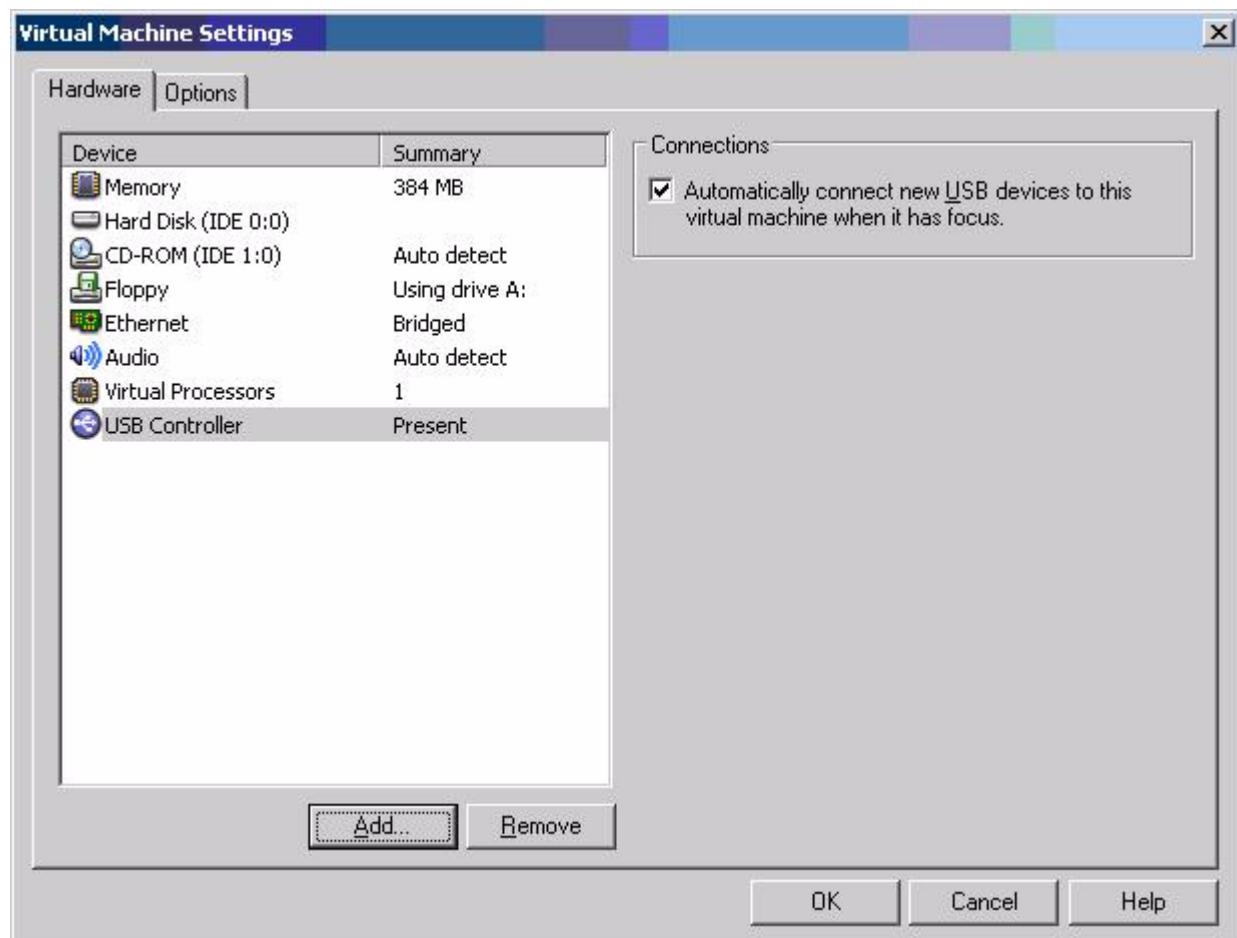
5. When the **Add Hardware Wizard** opens, select **USB Controller** in the **Hardware types** list and click **Next**.



6. Finish going through the wizard screens until you complete them. You then see the **USB Controller** under the **Hardware** tab, as shown in the next illustration.



7. Check the check box to the far right under **Connections** labeled **Automatically connect new USB device to this virtual machine when it has focus.**



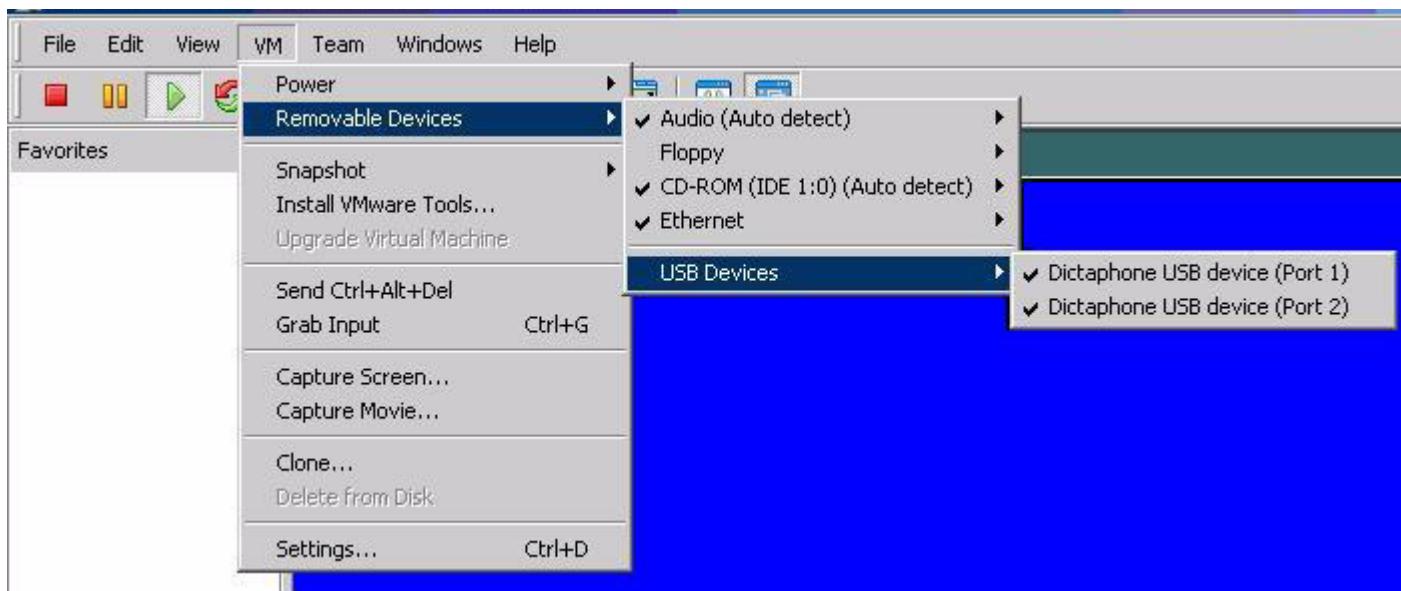
By checking this option you set up the *PowerMic II* to be engaged when you plug it in to the computer running the virtual machine. If you uncheck the option, the *PowerMic II* is recognized as being connected only to the host computer when you plug it in to the virtual machine.



**Note:** Once the virtual machine is running, when you later want to disconnect the *PowerMic II* to use it on a different virtual machine, you can go to the menu bar and select VM > **Removable Devices**.

8. After you have set up the **USB Controller**, you then set up **two** Dictaphone USB device ports on the virtual machine for a single *PowerMic II* microphone. (You do not have to set these ports up on the host machine, only on the virtual machine.) You configure the ports by going to the virtual machine menu bar and selecting VM >

**Removable Devices > USB Devices** and opening the menu to the right. You then see the **Dictaphone USB device** ports **Port 1** and **Port 2** in the menu (see below).



Both ports are already selected when a *PowerMic II* is plugged in. Both of these ports are required for a single microphone and must be used on a single virtual machine only.

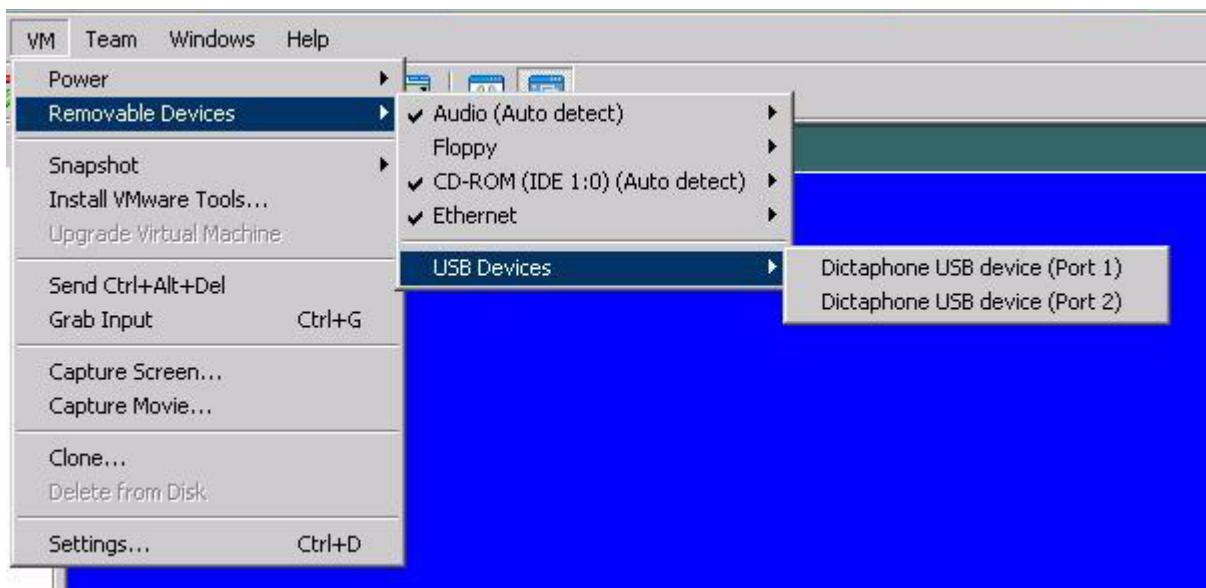


***Caution:*** If you try to use the same pair of Dictaphone USB device ports on more than one virtual machine at the same time, the *PowerMic II* will malfunction.

# Notes on Configuring Dictaphone USB Device Ports on Virtual Machines

**Notes:** Follow these guidelines when configuring **Dictaphone USB device ports** on virtual machines:

- When using a PowerScribe SDK client application with a PowerMic II device on several virtual machines, the PowerMic II can be connected to one of them and if a user dictates on that virtual machine, then switches focus to another virtual machine, pressing the Play/Stop button the the PowerMic II plays back the audio dictated on the original virtual machine.
- You do not have to restart your PowerScribe SDK client application if you disconnect/reconnect the microphone after you have configured it on that virtual machine.
- To switch the PowerMic II from one virtual machine to another, you must go to the menu bar and select **VM > Removable Devices > USB Devices** and uncheck both **Dictaphone USB device ports** on the first virtual machine; then run the other virtual machine and be sure to configure both the **USB Controller** and both **Dictaphone USB device ports** on that virtual machine before dictating.



- Before unplugging a PowerMic II or using the **VM > Removable Devices > USB Devices** menu to disconnect the microphone from a virtual machine, be sure it is in a safe state. Uncheck the PowerMic II devices from both **Dictaphone USB device ports**.
- You can use up to two **Dictaphone USB device ports** for a single PowerMic II device in a virtual machine if both the host operating system and the virtual

*machine operating system support USB. (Windows NT and versions of Linux earlier than 2.2.17 do not support USB connections.)*

- *Although the host operating system must support USB for you to use PowerMic II on a virtual machine of that host, you do not need to install USB device drivers on the host if you are using the PowerMic II only on the virtual machine.*
- *Only one operating system (host or virtual machine) can have control of the PowerMic II at any one time. If you connect the PowerMic II to the virtual machine, it disappears from the host, and vice versa.*
- *On a Windows 2000, Windows XP, or Windows Server 2003 host, when you connect a PowerMic II to a virtual machine, you may see a message on the host that says the device can be removed safely—you can ignore the message and do not have to remove the device from your physical computer. The virtual machine is given control of the device automatically.*

# **Index**

## **Symbols**

66, 152, 188, 222, 287  
.ast file  
    defined 90  
.doc files  
    exporting report files to 78  
    importing report files from 79  
.rtf files  
    exporting report files to 78  
    importing report files from 78  
.sig file 102  
.usr file 102  
.wav files  
    deleting cached on logout 44

## **A**

AccentID property of User object 353  
accents  
    assigning to users 353  
access privileges  
    types in SDK 344  
acoustic models  
    defined 356  
    marking reports for use in modifying 71  
AcousticModel object 356  
    Name property settings 357  
    TrainingState property 453  
    Type property 356  
AcousticModel property of User object 453  
AcousticModels collection object 356  
AcousticModelType property of UserProfile  
    object 453  
ActivateEnd event  
    handling 65  
    mapping 65  
ActivateEnd event of Report object 64  
ActivateReport() method 64  
activating reports 113  
ActiveX controls  
    making available for application 56  
    PlayerEditorCtl  
        mapping events 62  
Adapt() method 440  
adaptation  
    marking reports for 71  
queueing user to be processed for 440

Add() method of Categories object 291  
Add() method of Groups collection object 374  
Add() method of Shortcuts object 285  
Add() method of Users object 347  
Add() method of Words object 296  
Add() vs. LoadWord() method of Words object  
    299  
AddShortcut() method of Group object 375  
administrative privileges 345  
    assigning to users 349  
Administrator applications  
    initializing 272  
Administrator property of UserProfile object 41  
AdminPrivilege property 359  
AdminPrivileges object  
    setting properties of 350  
AdminSDK property 282, 373  
Advise() method 65, 67, 68, 69, 76, 305, 307  
applications  
    dictation modes 66  
    SDK types 3  
Attribute property 120  
attributes  
    text source markup 128  
audio  
    fast forwarding 234  
    going to start or end of 233  
    playing back 233  
    recording 232  
    rewinding 233  
    setting microphone volume 224  
    stopping playback 233  
    tracking position in report with microphone  
        237  
    transcribing 232  
audio devices  
    passed to Login() method 31  
    types 222  
audio files  
    adding reports to Recognition Server queue  
        99  
    determining length of Note  
    determining position of cursor in 191  
    exporting reports to local cache 89  
    exporting reports to server 89  
    importing report 92

importing reports  
    requirements for 92  
importing reports from alternative device 94  
Note object  
    using Microphone commands with 191  
retrieving from Note object 190  
saving reports to the sever 99  
audio playback speed  
    setting 224  
audio rewind speed  
    setting 224  
Author property of Group object 376  
Author property of Shortcut object 287  
Author property of UserProfile object 42  
Author property of Word object 297  
AutoAdvanceToNextShortcutField property 231  
AutoformatProfile  
    Name property 352  
AutoformatProfile Customization tool  
    assigning access to 350  
    URL to run 351  
AutoformatProfile object  
    ProfileID property 352  
AutoformatProfiles  
    assigning to user 352  
    creating 351  
    defined 351  
automatic punctuation  
    enabling or disabling 39  
AutoPunctuation property 39

**B**

BadgeID property 359  
bar code  
    scanning with microphone 239  
batch recognition  
    enabling 38  
    see also Speech recognition  
    vs. realtime 26  
BeepOnNavigate property 231  
bitwise values  
    using when adding to Words collection 295  
bulleted lists  
    options for default settings 159  
ButtonDown event  
    taking custom actions in response to 428  
ButtonDown event of Microphone object 235, 240  
buttons  
    MicButtonType constants 235  
    microphone hardware 473  
    programmable on microphone  
        customizing 239  
ButtonUp event of Microphone object 237, 240

**C**

cached wave files  
    deleting on logout 44  
CanRecover property 70  
categories  
    assigning to Shortcut objects 292  
    defined 280  
    defining for shortcuts 291  
    see also Categories object and Categories collection  
    vs. groups 280, 372  
Categories collection  
    creating 291  
Categories object  
    creating collection 291  
    defined 280  
    exporting collection 417  
    importing collection 418  
    importing collection from non-SDK system 418  
    vs. Groups object 280  
Category object  
    adding to collection 291  
    creating 291  
    Description property  
        modifying 292  
    Name property  
        modifying 292  
    removing from collection 292  
    Update() method 293  
    using to assign category to Shortcut object 286

Category property of Word object 297  
CategoryName property  
    modifying shortcut 286  
CheckSpelling() method 96  
cleanup  
    deleting cached wave files 44  
    releasing mapped events 43  
ClearAudioBuffer() method 71  
ClearEditor() method 71  
clipboard text  
    setting format of 161  
code samples  
    locating 22  
    running 22  
color of microphone light  
    setting 226  
Command mode 66  
    definition 66, 188  
    setting Note to 188  
    setting report to 66  
    switching to 66, 258  
    vs. Dictation mode 66, 258

- commands
    - creating set of custom 248
    - default grammar
      - setting Note object to receive 188
      - setting report to receive 66
    - dictating 66, 188
  - components of SDK
    - architecture 2, 16
    - installing from CD 19
    - installing from server 19
  - concordance file
    - defined 90, 92
    - importing for report 92
    - retrieving from Note object 190
    - retrieving reports for export 90
    - retrieving reports from local cache for export 90
  - control key sequences
    - events triggered by pressing handling 162
  - Correct That command
    - popping up Correction box 39
  - Correction box
    - popping up on Correct That command 39, 40
  - Correction dialog
    - enabling or disabling popup of 40
  - Correction property of UserProfile object 42
  - CorrectOptionPreference property 40
  - creating/modifying administrators
    - assigning access to capability 350
  - creating/modifying system administrators
    - assigning access to capability 350
  - creating/modifying users
    - assigning access to capability 350
  - current editor
    - setting for Note object 187
  - CurrentWavePosition property of Note object 191
  - cursor
    - events triggered by moving mapping 62
    - moving in report 124
    - positioning at end of line 167
    - positioning at particular character location 167
    - positioning at start of field 167
    - positioning at start of line 167
    - positioning at start of section 167
    - positioning in report programmatically 166
    - refocusing 166
  - custom training
    - creating HTML to integrate with training text 443
  - dictating wave files into a recording device 454
  - EnrollProgress event 449
  - GUI 445
  - handling events from Microphone 449
  - id attribute settings 443
  - inserting timer into HTML 448
  - inserting training text into HTML 447
  - inserting training type into HTML 446
  - integrating wave files dictated into a recording device 454
  - integrating with application 453
  - moving to next page 449
  - moving to previous page 450
  - popping up after login 453
  - releasing LevelMeter from 452
  - releasing microphone from 452
  - replacing standard training with custom module 453
  - securing LevelMeter for training 444
  - securing microphone for training 444
  - starting from enrollment 452
  - starting from first page 452
  - starting training process 444
  - steps to creating module 442
  - stopping training process 452
  - WavePosition event 448
  - custom words
    - loading after login 36
  - CustomEditController object
    - adding to application 59
    - associating with a Note object 186
    - instantiating 186
    - purposes 8
    - SetHWnd() method 186
    - use with Note object 184, 186
    - vs. PlayerEditor 59
- D**
- dangling logged in user
    - error code received 32, 272
    - logging out 32, 273
  - data retrieval
    - speech recognition applications 4
  - default grammar
    - commands 262
      - setting Note object to receive 188
      - setting report to receive 66
    - deploying commands in application 263
      - modifying commands 264
  - DefaultGrammar object 263
  - delegates
    - events in C# 28, 275
  - DeleteReport() method 77, 422

DeleteSelText() method 77  
Description property of Group object 376  
devices  
    audio  
        types in SDK 31  
        selecting microphone or foot pedal 23  
        testing microphone or foot pedal 23  
        types of audio 222  
    DICTATE button  
        setting action type 226  
Dictate mode  
    definition 66  
    setting Note object to 188  
    setting report to 66  
dictate only users 21  
DictateOnly mode  
    definition 66, 188  
    dictation 66  
    setting Note object to 188  
    setting report to 66  
dictation  
    activating reports for 64  
    commands 66, 188  
    efficiency analysis 191  
    modes of Note object 188  
    modes of report 66  
    numbers 66, 188  
    opening reports for 63  
    outside a report 184  
    setting microphone light color at phases of 226  
    setting microphone to play tone when ready for 225  
    speech and commands mixed 66, 188  
    speech only, no commands 66  
    spelling words into Note 188  
    spelling words into report 66  
    transient data 184  
    without commands 66, 188  
Dictation mode  
    vs. Command mode 66, 258  
dictation modes 66  
    Note object 188  
    reports 66  
digits  
    dictating into Note object 188  
    dictating into report 66  
distortion through microphone  
    reducing 224  
DLLs  
    installing SDK  
        from CD 19  
        from server 19  
    retrieving version 37  
DownloadProgressMessage event 34  
drag and drop capability  
    using with reports 171  
**E**  
editing  
    activating reports for 64  
    opening reports for 63  
editor 66, 188  
EditorCommand constants 156  
editors  
    adding COM components for 56  
    adding to application 58, 59  
    Note object 184, 186  
embedded speech recognition applications 4  
Enabled property 354  
EnableGainWizard property  
    setting 220  
EnableLogging property of User object 354  
EnableLogging() method 30  
EnablePlayer() method 72  
EnableRAF property 100  
EnableShortcuts() method 290  
EndList() method 157, 160  
enrollment  
    for training  
        built-in feature 436  
Enterprise Express Speech files  
    importing concordance to SDK system 93  
error codes  
    different for same type of error 74  
event delegates in C# 28, 275  
event handlers  
    see Events  
event maps  
    releasing 43, 275  
EventMapper object 27  
    Advise() method 65, 67, 68, 69, 76, 274, 305, 307  
    releasing mapped events 43, 275  
    Unadvise() method 43, 275  
events  
    Command event 259  
    cursor moves  
        mapping 62  
    Grammar object 258  
    keyboard key pressed  
        mapping 62  
    login of PowerScribe SDK application 33  
    Map Creation functionality 27  
    mapping in C# 28  
    mapping in JavaScript 27  
    Microphone object 234  
    mouse button pressed

- mapping 62
  - Note object 185, 189, 191
  - PhoneticTranscriber object
    - TranscriptionResult event 305, 307
  - PlayerEditorCtl 62
    - ButtonClick event 164
    - CombineKeysPress event 162
    - CtrlKeyPress event 162
    - EditorTextSelChanged
      - parsing style argument in handler 153
    - FunctionKeyPress event 163
    - NumLockKeyPress event 164
    - TextSelChanged 151
  - PSAdminSDK object 274
  - Report 95
    - QueuedForRecognition 98
    - WaveFileProgress 98
  - Report object 112
  - Sections object 112
  - Training object
    - EnrollProgress event 449
    - WavePosition event 448
  - triggered in structured reports 112
  - turning on Microphone object 235
  - User object
    - EndExport 410
    - EndImport 411
  - XmITemplateEdit object
    - EditorCursorMoveToNewLine 400
    - SectionNameChanged 397
    - SectionsChanged 395
  - exceptions
    - creating report 61
  - ExclusiveLock() method 63, 72
  - expanded text
    - modifying Shortcut 286
  - ExpandShortcuts() method 170
  - ExportCategories() method 417
  - ExportGroups() method 419
  - ExportShortcuts() method 415
  - ExportUsers() method 408
  - ExportVoiceModel() method 409
  - ExportWords() method 413
- F**
- fast forward monkey chatter
    - setting microphone 224
  - fields
    - dictating into 8
  - Find() method 149
  - foot pedals
    - handling connection change 238
    - setting up 222
- setting up hardware 23
  - testing hardware 23
  - Forward() method 233
  - full name
    - retrieving of user 42
  - function keys
    - events triggered by pressing 163
- G**
- GetConcordanceFile() method of Note object 190
  - GetConcordanceFile() method of Report object 90
  - GetCurrentSection() method 397, 398
  - GetDictatedWave() method of Report object 169
  - GetDocFile() method 78
  - GetEditorCurrentSection() method 124
  - GetEditorText() method 165
  - GetEditorText() method of Report object 89
  - GetGroups() method 373
  - GetInternalObject() method 157, 172
  - GetLanguageModels() method 346
  - GetLMWords() method of PowerscribeSDK object 300
  - GetReport() method 74
  - GetSections() method 114
  - GetSel() method 169
  - GetShortcuts() method 282, 285, 392
  - GetStatus() method 236
  - GetText() method of PlayerEditorCtl 165
  - GetText() method of Report object 88
  - GetText() method of the Note object 190
  - GetText() method of XmITemplateEdit object 391
  - GetTranscription() method 285, 296, 304, 306
  - GetUser() method 360
  - GetUserAccents() method 353
  - GetUsers() method 353, 358
  - GetWaveFile() method of Note object 190, 191
  - GetWaveFile() method of Report object 89
  - GetWords() method 294
  - Global property of Group object 376
  - Global property of Shortcut object 287
  - Global property of Word object 297
  - GoToEnd() method 233
  - GoToStart() method 233
  - grammar commands
    - setting Note object to receive 188
    - setting report to receive 66
  - Grammar object
    - Command event 259
    - creating 256
    - events 258
  - grammar templates

creating XML 249  
multi-level rules 252  
multiple verbs for same action  
    rules 255  
optional phrases 252  
rules 249  
single level of rules 250  
two-level rules 251  
grammars  
    activating 256  
    activating individual rules 257  
    creating 248  
    number you can have active 256  
SDK  
    defined 248  
graphical element  
    correlating with XML template structure of  
        `XmlTemplateEdit` object 398  
developing to reflect XML template structure  
    395  
group  
    see also Group object  
Group object  
    `addShortcut()` method 375  
    Author property 376  
    Description property 376  
    Global property 376  
    Name property 376  
    `RemoveShortcut()` method 377  
    removing Shortcut object from group 377  
    retrieving from a collection 375  
    steps to creating 372  
    `Update()` method 376  
groups  
    defined 280  
    see also Groups object or Groups collection  
    vs. categories 372  
    vs.categories 280  
Groups collection  
    exporting 419  
    importing 419  
    importing from non-SDK system 420  
Groups object  
    `Add()` method 374  
    creating collection 373  
    defined 280  
    exporting collection 419  
    filtering collection returned by group name  
        373  
    filtering collection returned by shortcut 374  
    global vs. user-specific 373  
    importing collection 419  
    importing collection from non-SDK system  
        420  
removing Group from collection 377  
steps to creating 372  
vs. Categories 280  
GUI component  
    associating with `CustomEditController` object  
        186  
    associating with Note object 186  
    setting to current editor for Note object 187  
GUI elements  
    mimicking microphone buttons with 232  
**H**  
HasAudioInMicBuffer() method 71  
HasData property 120  
HeadingName property 120  
headings  
    creating in XML for structured reports 111  
    style setting in XML for structured reports  
        112  
    XML style format string for `XmlTemplateEdit`  
        object 388  
highlight colors  
    text source indicators  
        default 127  
**I**  
ID property of `UserAccent` object 353  
identifier  
    dictating 192, 193  
ImportCategories() method 418  
ImportGroups() method 419  
ImportShortcuts() method 415  
ImportUsers() method 408  
ImportVoiceModel() method 410  
ImportWords() method 413, 415  
IncludeGlobals values 295  
Initialize() method of PowerscribeSDK object 26  
Initialize() method of PSAdminSDK object 272  
Initialize() method of `XmlTemplateEdit` object  
    390  
InRecognition property 100  
InsertText() method 76, 239  
installation  
    creating custom SDK installation module 22  
    SDK 15  
IsScanner property 238  
IsShortcutsEnabled() method 291  
**J**  
Java classes  
    installing SDK  
        from CD 19  
        from server 19  
Javascript

mapping events 27

**K**

keyboard keys  
events triggered by pressing  
mapping 62

**L**

language models  
definition in SDK 344  
in-memory 280  
modifying assigned to user 355  
retrieving assigned for user 355  
retrieving read-only collection of words in  
300  
retrieving to assign to User object 347  
storage and download of 344

LanguageModel object 347

LanguageModelID property 355

LanguageModels collection  
retrieving 346

LastName property 359

Letter mode  
definition 66, 188  
setting Note object to 188  
setting report to 66

letters  
dictating into Note object 188  
dictating into report 66  
dictating one by one into Note object 188  
dictating one by one into report 66

LevelMeterCtl  
instantiating 60  
making available for application 56  
see also LevelMeters

LevelMeters  
adding COM components for 56  
associating report with 62  
associating with Note object 187  
releasing from exclusive use of  
PhoneticTranscriber 308  
securing for use of the Note object 188  
securing to train punctuation marks 302  
securing to train shortcuts or words 302  
use with Note object 184

light on microphone  
setting color 226

LightColor property 226

lists  
changing start number of list 161  
see Bulleted Lists  
see Numbered Lists  
see Text

LoadAll() method of Report object 74

LoadEnd event  
handling 76  
mapping 75

loading Shortcuts collection after login 36

loading words after login 36

LoadShortcut() method 289

LoadShortcuts() method 36, 280, 289

LoadWord() method 299

LoadWords() method 36, 280, 299

log files  
URL to location 353

logging  
enabling or disabling for user 353  
locating log files 353  
turning on in application 30

logging in  
PowerScribe SDK application 30

login  
actions to take after 36  
logging out dangling logged in user 32  
session expiration 32  
user already logged in error 32

login events  
DownloadProgressMessage 34  
LoginEnd 35  
ProgressMessage 33  
ProgressMessageEx 34

login to PowerScribe SDK applications  
automatic SDK response to 31

Login() method  
login events 33

Login() method of PowerscribeSDK object 30

Login() method of PSAdminSDK object 272

LoginEnd event 35

LoginName property of User object 354, 358

Logoff() method 44, 276

Logoff() vs. LogoffUser() method 44

LogoffUser() method 33, 276

LongText  
modifying Shortcut 286

LongText property of Shortcut object 287

lookup key  
dictating 184

**M**

ManageReports property 359

managing reports  
assigning access to capability 350

managing Shortcut objects  
assigning access to capability 350

managing words  
assigning access to capability 350

Map Creation  
functionality 27

releasing mapped events 43  
MarkForAdaptation() method 71, 77  
MarkUnapprove() method 77  
markup  
    changing colors/styles for text source 128  
    default colors 127  
    displaying text source 127  
    setting attributes for text source 128  
    text source 126  
    text source types tracked 126  
    XML string for setting attributes 128  
MergeTemplate() method 138, 391  
MicButtonRecord property 227  
MicConnectionChange event 238  
Microphone events  
    custom actions in 428  
Microphone object  
    audio device types 222  
    AutoAdvanceToNextShortcutField property  
        231  
    BeepOnNavigate property 231  
    ButtonDown event 235, 240  
    ButtonUp event 237, 240  
    creating 27, 220  
    ensuring microphone settings take effect 227  
    events 234  
        mapping 234  
    Forward() method 233  
    GetStatus() method 236  
    GoToEnd() method 233  
    GoToString() method 233  
    IsScanner property 238  
    LightColor property 226  
    MicButtonRecord property 227  
    MicConnectionChange event 238  
    MonkeyChatter property 224  
    NavigatePreference property 229  
    NavigateSectionPreference property 230  
    NormalizePlaybackVolume property 225  
    Orientation property 225  
    Play() method 233  
    PlayAudioTone property 225  
    PlaybackSpeed property 224  
    PlaybackVolume property 224  
    ReceiveEvents() method 428  
    Record() method 232  
    Rewind() method 233  
    ScanText event 239  
    securing to train punctuation marks 302  
    securing to train Shortcuts or Words 302  
    see also Microphones  
    Setup() method 223  
    Stop() method 232, 233  
    Transcribe() method 232  
TranscribePreference property 228  
TranscribeSelectCount property 228  
Tuning() method 220  
turning off standard processing of buttons  
    428  
turning on events 235  
turning on standard processing of buttons  
    432  
Type property 222  
Update() method 227  
use with the Note object 184  
using command with Note object 191  
VoiceActivatedRecording property 227  
WavePosition event 237  
WindingSpeed property 224  
Microphone property 27  
Microphones  
    see also Microphone object  
microphones  
    audio device types 222  
    automatically set up 223  
    beep on navigating  
        setting 231  
    buttons 473  
        MicButtonType constants 235  
        mimicking with GUI elements 232  
        programmable 240  
        setting action type 226  
    configuring 223  
        customizing programmable buttons 239  
        default device type 223  
        determining whether has scanner 238  
        handling bar code scan 239  
        handling connection change 238  
        mimicking buttons with GUI elements 232  
        monkey chatter 224  
        navigating preferences 228  
            beep on 231  
            setting 229  
        navigation preferences 229  
        normalizing playback volume 225  
        playback speed 224  
        playing tone when ready to dictate 225  
        reducing voice distortion 224  
        regulating volume 220, 224  
        releasing from exclusive use by Note object  
            192  
        releasing from exclusive use of  
            PhoneticTranscriber 308  
    restoring default button actions 432  
    rewind speed 224  
    right-handed vs. left-handed 225  
    setting beep on navigation 231

- setting color of light at phases of dictation 226
  - setting playback speed 224
  - setting rewind speed 224
  - setting up 222
    - setting up hardware 23
    - setting volume 224
    - setting volume on login 31
    - taking over control of 428
    - testing hardware 23
    - tracking report audio position with 237
    - transcribing preferences 227, 228
      - setting 228
      - tuning 220
  - Micwiz.exe 222
  - MiddleName property 359
  - microphones
    - tuning wizard
      - embedding in application 220
  - modal dialog for Training Wizard
    - popping up 439
  - Mode property of Note object 188
  - Mode property of PowerscribeSDK object 66
  - modes
    - dictation 66
    - switching between dictation 66, 258
  - modifying system parameters
    - assigning access to capability 350
  - MonkeyChatter property 224
  - mouse buttons
    - events triggered by pressing
      - handling 164
      - mapping 62
- N**
- Name property of AcousticModel object 357
  - Name property of AutoformatProfile object 352
  - Name property of Category object
    - using to assign Shortcut category 286
  - Name property of Group object 376
  - Name property of Rule object 257
  - Name property of User object 359
  - Name property of UserAccent object 353
  - NavigatePreference property 229
  - NavigateSectionPreference property 230
  - NavigateTo() method of PlayerEditorCtl 167
    - NavigateDirection constants 168
    - NavigatePreference constants 168
  - navigating
    - microhpone preferences 228
      - setting 229
    - microphone beep on
      - setting 231
  - NewNote() method of Note object 185
- NewReport() method 391
    - plain reports 60
    - structured reports 113
  - NormalizePlaybackVolume property 225
  - Note object
    - associating LevelMeter with 187
    - concordance file 190
    - creating 185
    - CurrentWavePosition property 191
    - dictation modes 188
    - file for correlating text with audio 190
    - GetConcordanceFile() method 190
    - GetText() method 190
    - GetWaveFile() method 190, 191
    - Mode property 188
    - NewNote() method 185
    - purpose 8
    - purpose of 184
    - RecognizeEnd event
      - declaring 185
      - handling 189
    - ReleaseMicrophone() method 192
    - releasing microphone from exclusive use by 192
    - retrieving audio file from 190
    - retrieving concordance from 190
    - retrieving text from 190
    - saving content of 185
    - securing LevelMeter for exclusive use of 188
    - SetCustomEditControl() method 186
    - SetLevelMeter() method 188
    - SetMicrophone() method 189
    - SetPlayerEditor() method 187
    - setting current editor for 187
    - ShortcutsExpanded event
      - declaring 186
      - handling 191
    - SoundLength property 191
    - switching between Report and 192, 193
    - TextStreaming property 189
      - using in JavaScript 192
      - using Microphone commands with 191
    - Number mode
      - definition 66, 188
      - setting Note object to 188
      - setting report to 66
    - numbered lists
      - creating 157
      - modifying default settings 158
      - options for default settings 157
      - punctuation options in Word reports 158
      - starting 157
    - numberpad keys
      - events triggered by pressing

handling 164  
numbers  
dictating 66, 188

**O**

object model 2  
objects  
    EventMapper 27  
    in SDK architecture 2, 16  
    installing SDK 19  
        from CD 19  
        from server 19  
    instantiating PowerscribeSDK 26  
    types in SDK 2

Orientation property 225

**P**

paragraphs  
    creating in XML for structured reports 111  
    style setting in XML for structured reports  
        112

ParagraphStyle property 120

Parent property 121

ParentWindowHandle property of  
    PowerscribeSDK object 439

password  
    checking user's on login 30

Password property 358

PDA  
    importing report audio files from 94

personal digital assistant  
    importing report audio files from 94

phonetic transcription  
    retrieving while training Shortcuts or Words  
        303, 306  
    string for Shortcut object 285  
    string for Word 296  
    training for Shortcut or Word 303, 306  
    training with GUI buttons 304, 306  
    training with microphone buttons 305, 307  
    using string to add trained Shortcut to  
        collection 308  
    using string to add trained Word to collection  
        310  
    using TranscriptionResult event handler to  
        retrieve from event 305, 307

PhoneticTranscriber object  
    GetTranscription() method 285, 296, 304,  
        306  
    ReleaseMicrophone() method 308  
    releasing LevelMeter 308  
    releasing microphone 308  
    SetLevelMeter() method 303  
    SetMicrophone() method 302

Start() method 303, 306  
Stop() method 303, 306  
using to train punctuation marks 302  
using to train Shortcuts or Words 302  
using when adding Shortcut to collection 285  
using when adding Word to collection 296

Play() method 233

PlayAudioTone property 225

PLAYBACK button  
    setting action type 226

playback speed  
    setting audio 224  
    setting microphone 224

PlaybackSpeed property 224

PlaybackSpeed() method of Report object 170

PlaybackVolume property 224

PlayerEditor  
    Correct That command  
        turning on/off response to 39  
    dragging and dropping shortcuts into 314

Results box  
    turning on auto popup of 40

Select That command  
    turning on/off response to 40

zooming in on text 171

PlayerEditorCtl  
    EndList() method 157, 160  
    Find() method 149  
    GetSel() method 169  
    GetText() method 165  
    instantiating 59  
    making available for application 56  
    NavigateTo() method 167  
    ReplaceSel() method 150  
    see also PlayerEditors 56  
    SendCommand() method 155  
    SetEditorCurrentSection() method 156  
    SetEditorFocus() method 166  
    SetListDefaultOption() method 157, 158,  
        159  
    SetSel() method 167, 169  
    SetSelBold() method 153, 166  
    SetSelItalic() method 153  
    SetSelList() method 160  
    SetSelTextCase() method 155  
    SetSelUnderline() method 153  
    SetZoom() method 171  
    StartList() method 157, 160

PlayerEditorCtl object  
    see PlayerEditorCtl

PlayerEditors  
    activating reports for editing 148  
    adding COM components for 56  
    adding to application 58, 59

- associating report with 61
- associating with `XmlTemplateEdit` object 390
- choosing type for application 146
- copying and pasting text in 155
- cutting and pasting text in 155
- enabling or disabling 72
- events 62
- finding and replacing text 149
- inserting tabs at cursor 150
- inserting text at cursor 150
- locking reports for editing 148
- matching text case when searching 149
- pasting text in 155
- preparing to edit report in 148
- refocusing cursor 166
- searching and replacing text 149
- searching text up or down 149
- setting case of selected text 155
- setting font and style 149
- setting font and style of text 149
- setting style of selected text 152
- setting to current editor for `Note` object 187
- starting position for searching text 149
- types you can create 146
- undoing and redoing text 156
- use with `Note` object 184, 186
- zooming in on text 171
- PostProfiles**
  - assigning to user 352
  - creating 351
  - defined 351
  - see also `AutoformatProfiles`
- PowerMic buttons** 474
- PowerMic II buttons** 475
- PowerMic II microphone**
  - audio device type for 222
- PowerMic microphone**
  - audio device type for 222
- PowerScribe SDK**
  - installing 15
- PowerScribe SDK application**
  - logging into 30
  - login events 33
- PowerScribe SDK application login**
  - see also Login events
- PowerScribe SDK applications**
  - checking user's password on login 30
  - defined 3
  - disconnecting from SDK Web Server 44
  - displaying Training pages on login 30
  - logging into 30
  - logging out 44
  - login
    - automatic responses to 31
- tasks possible in 5
- technology types available 4
- vs. SDK Administrator applications 3, 16
- PowerScribe SDK product version**
  - retrieving 37
- PowerScribe SDK servers**
  - installing 17
- PowerscribeSDK DLL**
  - retrieving version 37
- PowerscribeSDK object**
  - `AdminSDK` property 282, 373
  - creating 26
  - `DefaultGrammar` property 263
  - `DeleteReport()` method 77, 422
  - `EnableGainWizard` property
    - setting 220
  - `EnableLogging()` method 30
  - `EnableShortcuts()` method 290
  - `GetLMWords()` method 300
  - `GetReport()` 74
  - initializing 26
  - instantiating 26
  - `IsShortcutsEnabled()` method 291
  - `LoadShortcuts()` method 36, 280
  - `LoadWord()` method 299
  - `LoadWords()` method 36, 280, 299
  - `Login()` method 30
    - login events 33
  - `Logoff()` method 44
  - `LogoffUser()` method 33
  - `Microphone` property 27
  - `Mode` property 66
  - `NewReport()` method 391
  - `ParentWindowHandle` property 439
  - `SetDataSourceMarkStyles()` method 128
  - `ShowDataSourceMark()` method 127
  - `Training` property 443
  - `Uninitialize()` method 44
- PowerscribeSDK objects in object model**
  - overview 5
- privileges**
  - objects for assigning user 10
- ProfileID** property 352
- ProgressMessage** event 33
- ProgressMessageEx** event 34
- pronunciation**
  - modifying shortcut 286
  - retrieving for Shortcuts, Words, or punctuation marks while training 303
- PSAdminSDK object** 346
  - creating with property of `PowerscribeSDK` object 282, 373
- events** 274
- ExportCCategories()** method 417

ExportGroups() method 419  
ExportShortcuts() method 415  
ExportUsers() method 408  
ExportWords() method 413  
GetGroups() method 373  
GetLanguageModels() method 346  
GetShortcuts() method 282, 392  
 GetUserAccents() method 353  
GetUsers() method 353, 358  
GetWords() method 294  
ImportCategories() method 418  
ImportGroups() method 419  
ImportShortcuts() method 415  
ImportUsers() method 408  
ImportWords() method 413, 415  
Initialize() method 272  
instantiating 272  
Login() method 272  
Logoff() method 276  
LogoffUser() method 276  
PSAdminSDK objects  
    overview 10  
punctuation  
    turning on/off automatic 39  
punctuation mark  
    adding trained to collection 310  
    using string to add trained Word to collection  
        310  
punctuation marks  
    overview of training 301  
    retrieving ways to dictate 300  
    securing LevelMeter to train 302  
    securing microphone to train 302  
    training audio of 306, 307  
    using PhoneticTranscriber object to train 302  
punctuation marks  
    steps to training 301

**Q**

QueuedForRecognition event 98, 100  
QueueForRecognition() method 99

**R**

RAF tool  
    logged data 102  
    purpose 100  
    using in application 100  
ReadOnly property of Word object 297  
realtime speech recognition  
    disabling 38  
    enabling or disabling 26  
    see also Speech Recognition  
technologies available 4  
vs. batch 26

working at sites where disabled 98  
ReceiveEvents() method 428  
recognition  
    see Speech recognition  
Recognition Accuracy Feedback tool  
    see RAF tool  
Recognition Monitor  
    assigning access to 350  
Recognition Server  
    installing 17  
recognized text  
    handling RecognizeEnd event 68  
RecognizeEnd event 68  
RecognizeEnd event of Note object  
    declaring 185  
    handling 189  
RecognizeEnd event of Report object  
    handling 68  
    mapping 67  
RecognizeWaveFile() method 95  
Record() method 232  
ReleaseMicrophone() method of Note object 192  
ReleaseMicrophone() method of  
    PhoneticTranscriber object 308  
ReleaseMicrophone() method of Training object  
    452  
Remove() method of Groups object 377  
Remove() method of Shortcuts collection object  
    287  
Remove() method of Users object 360  
Remove() method of Word object 298  
RemoveShortcut() method of Group object 377  
ReplaceSel() method 150  
Report object  
    ActivateEnd event 64  
    activating reports for editing 148  
    ActiveReport() method 64  
    CanRecover property 70  
    CheckSpelling() method 96  
    ClearAudioBuffer() method 71  
    ClearEditor() method 71  
    creating for plain report 60  
    creating for structured report 113  
    DeleteSelText() method 77  
    deleting a report 422  
    EnablePlayer() method 72  
    EnableRAF property 100  
    events in structured report 112  
    ExclusiveLock() method 63, 72  
    ExpandShortcuts() method 170  
    GetConcordanceFile() method 90  
    GetDictatedWave() method 169  
    GetDocFile() method 78  
    GetEditorCurrentSection() method 124

GetEditorText() method 89, 165  
 GetSections() method 114  
 GetText() method 88  
 GetWaveFile() method 89  
 HasAudioInMicBuffer() method 71  
 InRecognition property 100  
 InsertText() method 76, 239  
 LoadAll() method 74  
 LoadEnd event 75  
 locking reports for editing 148  
 MarkForAdaptation() method 71, 77  
 MarkUnapprove() method 77  
 MergeTemplate method 391  
 MergeTemplate() method 138  
 method for editing content 76  
 NewReport() method 60, 113  
 PlaybackSpeed() method 170  
 QueuedForRecognition event  
     handling 100  
     mapping 98  
 QueueForRecognition() method 99  
 RecognizeEnd event 67  
 RecognizeWaveFile() method 95  
 Save() method 68  
 SaveEndEx event 69  
 SaveRAFInfo() method 102  
 SectionsChanged event  
     handling 117, 123  
     mapping 112  
 see also Reports  
 SelectText() method 76  
 SetConcordanceFile() method 93  
 SetDocFile() method 78  
 SetEditorCurrentSection() 167  
 SetEditorCurrentSection() method 124  
 SetLevelMeter() method 62  
 SetPlayerEditor() method 61  
 SetReadOnly() method 75  
 SetTextEx() method 91  
 setting font and style of reports 149  
 SetWaveFile() method 92, 99  
 ShortcutsExpanded event 170  
 switching between Note and 192, 193  
 WaveFileProgress event 73, 95  
     mapping 98  
 WindingSpeed() method 170  
 report privileges 345  
     assigning to users 349  
 report production  
     statistical analysis of effort 126  
 ReportChanged event  
     handling 67  
     mapping 67  
 ReportRights property 359

reports  
     activating 64  
     activating before adding sections 114  
     activating for editing 148  
     approving 77  
     associating LevelMeter with 62  
     associating PlayerEditor with 61  
     audio files  
         exporting to local cache 89  
         exporting to server 89  
     concordance file 90, 92  
     copying and pasting text 155  
     creating 60  
     cutting and pasting text 155  
     deleting 422  
     deleting text 77  
     dictating without generating 184  
     dictation modes 65, 66  
     dragging and dropping text into 171  
     editing 74  
     editing content with Report object methods  
         76  
     editors  
         making controls available in application  
             56  
     exceptions to handle when creating 61  
     exporting SDK 88  
     exporting to .doc files 78  
     exporting to .rtf files 78  
     file for correlating text with audio 90  
     finding and replacing text 149  
     importing audio 92  
         compressed vs. uncompressed 92  
         concordance file 92  
         from alternative device 94  
     importing Enterprise Express Speech 93  
     importing file for correlating text with audio  
         92  
     importing from .rtf files 78  
     importing from other SDK 91  
     importing report concordance file 92  
     importing text 91  
         compressed vs. uncompressed 92  
         concordance file 92  
     importing to .doc files 79  
     inserting tabs at cursor 150  
     inserting text 76  
     inserting text at cursor 150  
     inserting unformatted text into plain 138  
     locking for dictation/editing 63  
     locking for editing 148  
     locking for import of text 91  
     managing 422  
     marking approved 77

marking unapproved 77  
matching text case when searching 149  
merging shortcuts into 139  
merging unformatted text into plain 138  
moving from one SDK system to another 88  
opening 74  
opening for dictation 63  
pasting text 155  
recognized text  
    working with 68  
RecognizeEnd event 68  
recovering from failed save 70  
refocusing cursor 166  
retrieving all text from PlayerEditor 165  
retrieving audio for export 89  
retrieving SDK concordance file from local cache 90  
retrieving SDK concordance file from server 90  
retrieving text from editor 89  
retrieving text from server 88  
reversing approval 77  
saving 68  
saving to local cache vs. server 68  
searching and replacing text 149  
searching text up or down 149  
see also Structured Reports  
selecting text 76  
setting case of selected text 152, 155  
setting font and style of entire report 149  
setting style of selected text 152  
spell checking 96  
starting position for searching 149  
structured  
    elements of 110  
types you can create 5, 56  
undoing and redoing text changes 156  
unlocking to end dictation/editing 72  
workstation requirements for Microsoft Word 146  
zooming in on text 171

**Result box**  
    enabling or disabling display of 40  
    popping up during dictation 40

**ResultBox**  
    enabling or disabling display of 40

retraining user 439

rewind monkey chatter  
    setting microphone 224

rewind speed  
    setting audio 224  
    setting microphone 224

Rewind() method 233

right-handed vs left-handed microphone setup 225

roles for users  
    assigning in Administrator SDK 21

RTF reports  
    retrieving text from the editor 89  
    retrieving text from the server 88

Rule object 257  
    Name property 257  
    State property 257

rules 249  
    activating 256  
    activating individual for grammar 257  
    deactivating 256  
    handling multi-level 260  
    multi-level 252  
    multiple verbs for same action 255  
    optional phrases 252  
    single level 250  
    two-level 251

Rules object 257

**S**

samples  
    locating 22  
    running 22

Save() method 68

SaveEndEx event of Report object 69  
    handling 69  
    mapping 69

SaveRAFInfo() method 102

saving reports 68  
    local cache vs. server 68  
    recovering from failed save 70

ScanText event 239  
    when to handle 238

**SDK**  
    architecture 2, 16  
    installation instructions, locating 17  
    installing 15  
    installing Recognition Server 17  
    installing Web Server 17  
    purpose 2

SDK Administrator applications  
    defined 3  
    initializing 272  
    objects required 272  
    see also SDK Administrator provided  
    tasks possible in 10  
    tasks you can code 271  
    vs. PowerScribe SDK applications 3, 16

SDK Administrator provided  
    creating users 20  
    features 270  
    running 270

- URL to run 270
- SDK client DLLs
  - installing
  - see also Objects
- SDK Web Server
  - running 19
- Section object
  - Attribute property 120
  - HasData property 120
  - HeadingName property 120
  - ParagraphStyle property 120
  - Parent property 121
  - properties 119
  - Subsections property 121
  - Text property 120
  - using to add/remove subsections to report 117
- SectionChanged event
  - handling 123
  - mapping 113
- sections
  - activating report before adding 114
  - adding/removing from report 114
  - traversing structured report 121
- Sections object
  - events in structured report 112
  - retrieving 114
  - SectionChanged event
    - handling 123
    - mapping 113
- SectionsChanged event
  - handling 117, 123
  - mapping 112
- Select That command
  - popping up Correction box 40
- SelectText() method 76
- SendCommand() method 155
- server
  - disconnecting from SDK 44
- servers
  - installing PowerScribe SDK 17
- SetConcordanceFile() method of Report object 93
- SetCustomEditControl() method of Note object 186
- SetDataSourceMarkStyles() method 128
- SetDocFile() method 78
- SetEditor() method of XmlTemplateEdit object 390
- SetEditorCurrentSection() method 124, 156, 167
- SetEditorFocus() method 166
- SetHWnd() method of CustomEditController object 186
- SetLevelMeter() method of Note object 188
- SetLevelMeter() method of PhoneticTranscriber object 303
- SetLevelMeter() method of Report object 62
- SetLevelMeter() method of Training object 444
- SetListDefaultOption() method 157, 158, 159
- SetMicrophone() method of PhoneticTranscriber object 302
- SetMicrophone() method of Training object 444
- SetMircophone() method of Note object 189
- SetPlayerEditor() method of Note object 187
- SetPlayerEditor() method of Report object 61
- SetReadOnly() method of Report object 75
- SetSel() method 167, 169
- SetSelBold() method 153, 166
- SetSelItalic() method 153
- SetSelList() method 160
- SetSelTextCase() method 155
- SetSelUnderline() method 153
- SetText() method 392
- SetTextEx() method 91
- Setup() method 223
- SetWaveFile() method of Report object 92, 99
- SetZoom() method 171
- Shortcut object
  - adding to a Group 375
  - adding to global collection vs. user-specific collection 285
  - adding trained to collection 308
  - assigning categories 292
  - assigning category 286
  - associating one with others 372
  - Author property 287
  - auto-expanded vs. custom 282
  - building plain text to expand to 285
  - building structured XML text to expand to 285
- CategoryName property
  - setting 286
- defining categories for 291
- determining type Group contains 376
- ensuring saved to database 285
- Global property 287
- global vs. user-specific 283
- grouping several 372
- how deployed 280
- loading single into in-memory language model 289
- LongText property 287
- LongText value
  - modifying by loading single shortcut 290
- LongText vs. ShortText 285
- modifying name 286
- modifying ShortText property 286
- phonetic transcription string 285

pronunciation 285  
properties 287  
putting multiple into groups 372  
removing from collection 287  
removing from Group 377  
removing Shortcut from collection 287  
returning particular one in Shortcuts collection 284  
revising existing after training 309  
securing LevelMeter to train 302  
securing microphone to train 302  
ShortText property 287  
ShortText vs. LongText 285  
steps to creating 281, 293  
steps to deploying collection of 288  
steps to training 301  
Transcription property 287  
type of objects necessary to work with 282  
type of SDK applications that work with 282  
Type property 287  
types 280, 282, 284  
Update() method 286  
using PhoneticTranscriber to train 302  
shortcut template  
  see Template builder  
shortcuts  
  associating with one another 372  
  categories 292  
  defined 280  
  dragging and dropping into PlayerEditor 314  
  expanding on drag/drop into PlayerEditor 314  
  expanding when dragged from ListView  
    C# 314  
    JavaScript 312  
  forcing expansion in report 170  
  modifying expanded text 286  
  see also Shortcut object, Shortcuts object, and Shortcuts collection  
Shortcuts collection  
  checking whether enabled 291  
  copying from one SDK system to another 415, 417, 419  
  creating 282  
  disabling or re-enabling 290  
  ensuring saved to database 285  
  exporting collection 415  
  filtering by category 283  
  filtering by group 283  
  filtering by particular Shortcut object 284  
  filtering by user 283  
  how deployed 280  
  importing collection 415  
  importing from non-SDK system 416  
  loading after login 36  
  loading into in-memory language model 289  
  retrieving global vs. user-specific 283  
  steps to deploying 288  
  type of objects necessary to work with 282  
  type of SDK applications that work with 282  
Shortcuts object  
  Add() method 285  
  creating collection 282  
  GetShortcuts() method 285  
ShortcutsExpanded event of Note object  
  declaring 186  
  handling 191  
ShortcutsExpanded event of Report object 170  
ShortText  
  modifying Shortcut 286  
ShortText property  
  modifying Shortcut object 286  
ShortText property of Shortcut object 287  
ShowDataSourceMark() method 127  
ShowDictationResultBox property 40  
SoundLength property of Note object 191  
Source property of Word object 297  
speech recognition  
  applications that do not create reports 4  
  batch vs. realtime 26  
  disabling realtime 98  
  embedded without report creation 4  
  handling batch 98  
  improving for specific words 293  
  technologies available 4  
  transient data retrieval with 4  
  users 21  
speech recognition technologies  
  types available 4  
spelling  
  checking report 96  
  dictating words into Note object 188  
  dictating words into report 66  
spelling words into Note object 188  
spelling words into report 66  
spoken sound  
  modifying shortcut 286  
standard SDK Administrator  
  creating users in 20  
Start() method 303, 306, 444  
StartList() method 157, 160  
State property 257  
State property of Word object 297  
statistical analyses  
  report creation effort 126  
Stop() method 232, 233, 303, 306, 452  
streaming text  
  enabling or disabling for user 41  
structured reports

- activating 113
  - adding new section to particular location 124
  - adding/removing sections 114
  - adding/removing subsections 117
  - appending new section 134
  - appending paragraph to particular section 133
  - creating 113
  - creating templates 111
  - determining content 119
  - determining current section 124
  - displaying structure in application 122
  - elements of 110
  - events 112
  - finding particular section 125
  - inserting unformatted text into 136
  - merging
    - appending new section 134
    - appending paragraph to particular section 133
    - inserting unformatted text into structured report 136
    - selected sections into report 131
    - shortcuts into report 139
    - two with same template 129
  - moving cursor 124
  - moving cursor to particular section 124
  - navigating 124
  - positioning cursor in particular section 167
  - removing selected section 125
  - traversing sections 121
  - types of merging possible 129
  - structured template builder
    - see `XmITemplateEdit` object
  - style information
    - setting in XML for structured reports 112
  - Subsections property 121
  - system parameters
    - assigning modifying capability 350
- T**
- tabs
    - inserting at cursor 150
  - template builder
    - adding to application 390
    - how application uses 386
  - template object
    - see Template builder
  - templates
    - creating XML grammar 249
    - rules 249
    - see XML templates
  - text
    - changing start number of list 161
    - converting selected to list 160
    - copying all and reusing in another report 165
    - copying and pasting 155
    - cutting and pasting 155
    - finding and replacing 149
    - indicating format of selected 151
    - inserting at cursor 150
    - matching case when searching 149
    - parsing style argument in `EditorTextSelChanged` handler 153
    - pasting 155
    - pasting from clipboard 161
    - positioning cursor at particular character location 167
    - positioning cursor at start/end of line 167
    - positioning cursor in programmatically 166
    - retrieving all from `PlayerEditor` 165
    - retrieving from `Note` object 190
    - searching and replacing 149
    - searching up or down 149
    - see Numbered Lists 157
    - setting case of selected 152, 155
    - setting default font and text for report 149
    - setting style of selected 152
    - starting position for searching 149
    - undoing and redoing changes 156
  - Text property 120
  - text source
    - changing markup colors/styles 128
    - default markup colors 127
    - displaying markup 127
    - markup 126
    - types tracked 126
  - text source markup
    - XML string for setting attributes 128
  - TextStreaming property of `Note` object 189
  - TextStreaming property of `UserProfile` object 41
  - tone
    - setting microphone to play when ready to dictate 225
  - Train() method 439
  - training
    - built-in feature 436
    - processing user for training 437
    - stages of 437
    - dictating wave files into a recording device 438
    - integrating wave files dictated into a recording device 438
    - queueing user to be processed for adaptation 440
    - queueing user to be processed for training 439
    - see also Custom training

training level  
    user voice model  
        retrieving 42

Training object  
    ReleaseMicrophone() method 452  
    SetLevelMeter() method 444  
    SetMicrophone() method 444  
    Start() method 444  
    Stop() method 452  
    TrainingWaveFile property 438

training on login 32  
    custom 453

Training pages  
    displaying on login 30

Training property 443

training state  
    retrieving user 453

training state of user  
    determining 357

Training Wizard  
    popping up dialog in C#, C++, or Visual Basic .NET 439

Training Wizard dialog  
    Wav File button 438  
    Wav File button in custom training 454

TrainingState property 437

TrainingState property of AcousticModel object 453

TrainingWaveFile property of Training object 438

Transcribe() method 232

TranscribePreference property 228

TranscribeSelectCount property 228

transcribing  
    microhpone preferences 227  
    setting 228

Transcription property  
    modifying shortcut 286

Transcription property of Shortcut object 287

Transcription property of UserProfile object 42

Transcription property of Word object 297

transcriptionist users 21

transfer of files  
    tracking progress 73

trigger words  
    importing and training 311

troubleshooting login  
    user already logged in 32

tuning wizard  
    embedding in application 220

Tuning() method 220

Type property  
    modifying shortcut 286

Type property of AcousticModel object 356

Type property of Microphone object 222

Type property of Shortcut object 287

**U**

Unadvise() method 43

Uninitialize() method 44

Update() method of AutoformatProfile object 352

Update() method of Category object 293

Update() method of Group object 376

Update() method of Microphone object 227

Update() method of Shortcut object 286

Update() method of User object 354

Update() method of UserProfile object 41

User object  
    AccentID property 353  
    AcousticModel property 453  
    Adapt() method 440  
    AdminPrivilege property 359  
    assigning language model 347  
    BadgeID property 359  
    copying from one SDK system to another 408  
    Enabled property 354  
    EnableLogging property 354  
    enabling or disabling logging 353  
    events 410, 411  
    exporting voice models 409  
    ExportVoiceModel() method 409  
     GetUser() method 360  
    importing from non-SDK system 411  
    importing voice models 410  
    ImportVoiceModel() method 410  
    LanguageModelID property 355  
    LastName property 359  
    LoginName property 354, 358  
    ManageReports property 359  
    MiddleName property 359  
    Name property 359  
    Password property 358  
    ReportRights property 359  
    retrieving 358  
    Train() method 439  
    TrainingState property of user  
        retrieving 437  
    Update() method 352, 354  
    UseRealTimeRecognition property 359

UserAccent object  
    ID property 353  
    Name property 353

UseRealTimeRecognition property 26, 38, 359

UserName property of UserProfile object 42

UserProfile  
    UserTrainingData property 42

UserProfile object  
    AcousticModelProperty 453

- 
- Administrator property 41
  - Author property 42
  - AutoPunctuation property 39
  - Correction property 42
  - CorrectOptionPreference property 40
  - properties
    - having take effect 41
  - ShowDictationResultBox property 40
  - TextStreaming property 41
  - Transcription property 42
  - UseRealTimeRecognition property 26, 38
  - UserName property 42
  - users
    - administrative privileges 345
    - assigning accent 353
    - assigning administrative privileges 349
    - assigning AutoformatProfiles 352
    - assigning PostProfiles 352
    - assigning report privileges 349
    - assigning roles in Administrator SDK
      - provided 21
    - copying from one SDK system to another 408
    - creating 345
    - creating in SDK Administrator application
      - provided 20
    - default privileges 344
    - definition in SDK 344
    - determining training state of 357
    - determining whether logged in is enrolled in training 453
    - dictate only, defined 21
    - enabling or disabling 354
    - exporting 408
    - importing 408
    - importing from non-SDK system 411
    - locating log files 353
    - modifying assigned language model 355
    - object for creating and assigning privileges 10
    - report privileges 345
    - retrieving assigned language model 355
    - retrieving training state of 453
    - speech recognition, defined 21
    - transcriptionist, defined 21
    - types for PowerScribe SDK applications 21
    - types of privileges 344
    - voice models
      - exporting 409
      - importing 410
  - Users collection object
    - Add() method 347
    - exporting collection 408
    - importing collection 408
  - Remove() method 360
  - UserTrainingData property of UserProfile 42
- V**
- version information
    - retrieving 37
  - Visual Studio applications
    - dictating into fields 8
  - voice commands
    - custom 248
  - voice model status
    - retrieving 42
  - voice models
    - copying from one SDK system to another 409
    - exporting 409
    - importing 410
  - voice modulation in dictation
    - reducing distortion 224
  - VoiceActivatedRecording property 227
  - volume
    - setting microphone 224
  - volume of microphone
    - setting on login 31
- W**
- Wav File button in custom Training Wizard dialog 454
  - Wav File button in Training Wizard dialog 438
  - wave files
    - deleting cached on logout 44
    - determining length of Note
    - determining position of cursor in Note 191
    - importing report 92
    - importing reports
      - requirements for 92
    - importing reports from alternative device 94
    - Note object
      - using Microphone commands with 191
    - retrieving from Note object 191
    - see also Audio files
  - WaveFileProgress event 73, 95
    - handling 73
  - WaveFileProgress event of Report object
    - mapping 73
  - WavePosition event of Microphone object 237
  - Web Server
    - disconnecting from 44
    - installing SDK 17
    - running SDK 19
  - WindingSpeed property 224
  - WindingSpeed() method of Report object 170
  - Windows forms components
    - dictating into 8

Word  
    Microsoft Word functionality  
        accessing in reports 172

Word collection  
    type of SDK applications that work with 282

Word object  
    adding trained to collection 308, 310  
    Author property 297  
    Category property 297  
    Global property 297  
    modifying 297  
    phonetic transcription string 296  
    pronunciation 296  
    properties 297  
    properties you can set 297  
    ReadOnly property 297  
    removing 298  
    revising existing after training 311  
    securing LevelMeter to train 302  
    securing microphone to train 302  
    Source property 297  
    State property 297  
    steps to training 301  
    Transcription property 297  
    type of objects necessary to work with 282  
    type of SDK applications that work with 282  
    using PhoneticTranscriber to train 302  
    Word property 297

Word property of Word object 297

WordCaseType constants 155

WordCategoryCode values 294

WordPlayerEditorCtl  
    GetInternalObject() 172  
    GetInternalObject() method 157  
    instantiating 147  
    making available for application 56  
    see also PlayerEditors

WordPlayerEditors  
    adding to application 147  
    associating report with 148  
    enabling or disabling 148  
    see also PlayerEditors

words  
    loading after login 36  
    see Word object, Words object, and Words collection

Words collection  
    copying from one SDK system to another 413  
    exporting collection 413  
    importing collection 413  
    importing from non-SDK system 413  
    retrieving read-only collection 300  
    see also Words object and Word object

type of objects necessary to work with 282

Words object  
    Add() method 296  
    Add() vs. LoadWord() method 299  
    adding words to collection 296  
    creating collection 294  
    IncludeGlobals values 295  
    loading collection into in-memory language model 299  
    modifying words in 297  
    phonetic transcription string 296  
    pronunciation of word 296  
    punctuation marks  
        retrieving ways can be dictated 301, 306, 307  
    retrieving read-only collection 300  
    steps to deploying 298  
    WordCategoryCode values 294  
    WordSourceCode values 295  
    WordStateCode values 295

WordSourceCode values 295

WordStateCode values 295

workstations  
    requirements for Microsoft Word reports 146

**X**

XML  
    creating grammar template 249  
    grammar rules 249

XML reports  
    retrieving text from the editor 89  
    retrieving text from the server 88  
    see Structured Reports  
    steps to developing templates for 387

XML Shortcut objects  
    LongText for 285

XML strings  
    creating to initialize XmlTemplateEdit object 388

XML style formats template  
    defining styles 389

XML tags  
    required to import Shortcuts collection 417  
    required to import Users 412  
    required to import Words 414

XML templates  
    attributes  
        getting/setting programmatically 120  
    creating for structured reports 111  
    empty 388  
    steps to developing 387

XmlTemplateEdit object  
    adding to application 390  
    associating PlayerEditor with 390

- changing text style in template 393
- creating 390
- creating structured shortcuts 392
- creating template using content of associated PlayerEditor 391
- creating with empty template 388
- determining where to position cursor in graphical element 398
- GetCurrentSection() method 397, 398
- GetText() method 391
- how application uses 386
- Initialize() method 390
- initializing 390
- merging shortcut into report 393
- merging template into report 391
- modifying structured shortcuts 392
- placing text in associated editor 392
- positioning cursor in template 399
- preparing to handle events 394
- responding to move of cursor 400
- retrieving text from template section 397
- SetEditor() method 390
- SetText() method 392
- steps to developing report templates with 387
- working with style of text 400

