



ActiveX[™] Developer's Guide

IBM ViaVoice[™] SDK for Windows[®]

Version 1.7

Printed in the USA

Note:

Before using this information and the product it supports, be sure to read the general information under Appendix A, "Notices."

First Edition (December 1999)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you. This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Requests for technical information about IBM products should be made to your IBM reseller or IBM marketing representative.

©Copyright International Business Machines Corporation 1999, 2000. All Rights Reserved.

Note to U.S. Government Users—Documentation related to restricted rights— Use, duplication or disclosure is subject to restrictions set forth in GS ADP Schedule Contract with IBM Corp.

Contents

About this Book	17
Who Should Read This Book	17
ViaVoice SDK Related Publications	17
How This Book Is Organized	18
Document Conventions	18
 Introduction to the Embedded Speech Reference	 19
Installing the ViaVoice ActiveX Components	19
ActiveX Controls	19
Part 1: ViaVoice TextBox Control	20
Part 2: ViaVoice RichEdit Control	20
Part 3: ViaVoice Phrases Control	20
Part 4: ViaVoice Grammar Control	21
Part 5: ViaVoice Lite Controls	21
Part 6: ViaVoice Engine Control	21
Part 7: ViaVoice Error Correction Window Control	22
Part 8: ViaVoice User Interface Control	22
Part 9: ViaVoice DictationManager Control	23
Part 10: ViaVoice Dictation Control	23
Part 11: Virtual Voices Control	23
Part 12: ViaVoice Detective Control	24
ViaVoice SDK Architecture	25
 Chapter 1	 Introduction to the TextBox Control 27
VVTextBox Object Hierarchy	27
 Chapter 2	 Getting Started with the TextBox Control 29
Creating an Instance of the Control	29
Capturing Speech	35

	Capturing Commands	36
	Text Correction	38
	Summary	39
Chapter 3	Properties, Methods, and Events	41
	TextBox Control Properties	41
	AutoDictationWindow (Read/Write at Run Time Only)	42
	AutoUI	46
	Commands	49
	CommandsEnabled	52
	DictationOn	54
	Engine	56
	LanguageUI	58
	ShowDictationIcon	61
	TextBox Control Methods	63
	ExecuteCommand	64
	Playback	66
	PlaybackEx	67
	PlaybackEx2	68
	TextBox Control Events	69
	Command	70
	DictationStateChange	73
	Error	76
	MaxText	80
Chapter 4	TextBox Control Frequently Asked Questions	83
Chapter 5	Introduction to the RichEdit Control	85
	VVRichEdit Object Hierarchy	85
Chapter 6	Getting Started with the RichEdit Control	87
	Creating an Instance of the Control	87
	Capturing Speech	93
	Capturing Commands	94
	Text Correction	97
	Summary	98
Chapter 7	Properties, Methods, and Events	99
	RichEdit Control Properties	99

AutoDictationWindow (Read/Write at Run Time Only)	100
AudioSourceType	103
AutoUI	106
BulletIndentation	109
Commands	111
CommandsEnabled	114
DictationOn	116
Engine	118
FileName	120
hWnd (Read Only).	122
LanguageUI	124
RightMargin.	127
SelAlignment (Read/Write at Run Time Only)	129
SelBold (Read/Write at Run Time Only).	131
SelBullet	133
SelCharOffset (Read/Write at Run Time Only).	135
SelColor (Read/Write at Run Time Only)	137
SelFontName(Read/Write at Run Time Only)	139
SelFontSize (Read/Write at Run Time Only)	141
SelHangingIndent (Read/Write at Run Time Only)	143
SelIndent (Read/Write at Run Time Only)	145
SelItalic (Read/Write at Run Time Only)	147
SelLength (Read/Write at Run Time Only).	149
SelProtected.	151
SelRightIndent (Read/Write at Run Time Only)	153
SelRTF (Read/Write at Run Time Only)	155
SelStart (Read/Write at Run Time Only).	157
SelStrikeThru (Read/Write at Run Time Only).	159
SelTabCount (Read/Write at Run Time Only).	161
SelTabs (Read/Write at Run Time Only).	163
SelText (Read/Write at Run Time Only)	165
SelUnderline (Read/Write at Run Time Only)	167
TextRTF.	169
RichEdit Control Methods	171
ExecuteCommand	172
LoadRTF	174
LoadTextFile	176
Playback.	178

	PlaybackEx	180
	PlaybackEx2	182
	SaveRTF	184
	SaveTextFile	186
	SelPrint	188
	RichEdit Control Events	190
	Command	191
	DictationStateChange	193
	Error	195
	MaxText	199
Chapter 8	RichEdit Control Frequently Asked Questions	201
Chapter 9	Introduction to the Phrases Control	203
	VVPhrases Object Hierarchy.	203
Chapter 10	Getting Started with the Phrases Control	205
	Creating an Instance	205
	Drag-Drop-n-Go Support	211
	Adding Phrases	212
	Enabling/Disabling Phrases.	214
	Working with the Custom Designer	216
	Object Hierarchy	218
Chapter 11	Properties, Methods, and Events	221
	VVPhrases Control	221
	VVPhrases Control Properties	221
	AutoConnect (VVPhrases)	222
	AutoUI (VVPhrases).	223
	Enabled (VVPhrases)	225
	Engine (VVPhrases)	227
	Layout (VVPhrases)	229
	Phrases (VVPhrases).	231
	VVPhrases Control Methods	233
	RefreshUIText (VVPhrases).	236
	VVPhrases Control Events	239
	BeginSpeechRecognition (VVPhrases)	240
	SpeechRecognized (VVPhrases)	241
	TrainingRequired (VVPhrases)	243

	VVPhraseColl Collection	245
	VVPhraseColl Collection Properties	245
	Count (VVPhraseColl)	246
	Enabled (VVPhraseColl)	248
	Item (Default Method - VVPhraseColl)	250
	VVPhraseColl Collection Methods	252
	Add (VVPhraseColl)	253
	Exists (VVPhraseColl)	255
	Remove (VVPhraseColl)	257
	RemoveAll (VVPhraseColl)	259
	VVPhrase Object	260
	VVPhrase Object Properties	260
	ActionDesc (VVPhrase)	261
	Description (VVPhrase)	263
	Enabled (VVPhrase)	265
	ID (VVPhrase)	267
	ItemData (VVPhrase)	269
	Name (VVPhrase)	271
	Text (VVPhrase)	273
	VVPhrase Object Methods	275
	VVPhrase Object Events	275
Chapter 12	Phrases Control Frequently Asked Questions	277
Chapter 13	Introduction to the Grammar Control	279
	VVCFGram Object Hierarchy	279
Chapter 14	Getting Started with the Grammar Control	281
	Creating an Instance of the Control	281
	Drag-Drop-n-Go Support	287
	Loading a Grammar	288
	Enabling/Disabling a Grammar	290
	Using External Lists	291
Chapter 15	Properties, Methods, and Events	295
	Grammar Control Properties	295
	Alternates (VVCFGram)	296
	Annotations (VVCFGram)	298
	AutoConnect (VVCFGram)	303

AutoLoad (VVCFGram)	305
AutoUI (VVCFGram)	307
Enabled (VVCFGram)	309
Engine (VVCFGram)	311
ExternLists (VVCFGram)	314
GrammarFormat (VVCFGram)	317
GrammarSource (VVCFGram)	320
Rules (VVCFGram)	322
SourceType (VVCFGram)	324
Grammar Control Methods	327
LoadFromSource (VVCFGram)	328
Refresh	330
RefreshUIText (VVCFGram)	331
ShowTrainDialog (VVCFGram)	334
Grammar Control Events	336
BeginSpeechRecognized (VVCFGram)	337
SpeechRecognized (VVCFGram)	339
TrainingRequired (VVCFGram)	342
VVPhraseCollGroup Object	345
VVPhraseCollGroup Object Properties	345
Count (VVPhraseCollGroup)	346
Enabled (VVPhraseCollGroup)	349
Item (VVPhraseCollGroup)	352
VVPhraseCollGroup Object Methods	354
Exists (VVPhraseCollGroup)	355

Chapter 16
Grammar Control Frequently Asked Questions 359
Chapter 17
Introduction to the Lite Controls 361
Chapter 18
Getting Started with the Lite Controls 363

VVDictLite Control	363
Using the Control	363
VVGrammarLite Control	368
Using the Control	368
VVPhrasesLite Control	374
Using the Control	374
Summary	380

Chapter 19	Properties, Methods, and Events 381
	VVDictLite Control Properties 381
	Enabled (VVDictLite) 382
	VVDictLite Control Methods 384
	VVDictLite Control Events 384
	PhraseRecognized (VVDictLite) 385
	VUMeter (VVDictLite) 387
	VVGrammarLite Control Properties 389
	Enabled (VVGrammarLite) 390
	GrammarSource (VVGrammarLite) 392
	VVGrammarLite Control Methods 394
	VVGrammarLite Control Events 394
	PhraseRecognized (VVGrammarLite) 395
	VUMeter (VVGrammarLite) 397
	VVPhrasesLite Control Properties 399
	Enabled (VVPhrasesLite) 400
	VVPhrasesLite Control Methods 402
	AddPhrase (VVPhrasesLite) 403
	RemoveAll (VVPhrasesLite) 405
	VVPhrasesLite Control Events 407
	PhraseRecognized (VVPhrasesLite) 408
	VUMeter (VVPhrasesLite) 410
Chapter 20	Lite Controls Frequently Asked Questions 413
Chapter 21	Introduction to the ECWin Control 415
Chapter 22	Getting Started with the ECWin Control 417
	Creating an Instance of the Control 417
	Initializing the Error Correction Window Control 423
	Handling Error Correction Window Control Events 426
	Error Correction Window Control Voice Command Support 429
Chapter 23	Properties, Methods, and Events 431
	Error Correction Window Control Properties 431
	AddPhraseChecked 432
	AddPhraseVisible 435
	Caption 437

Chapter 24	ECWin Control Frequently Asked Questions	495
Chapter 25	Introduction to the User Interface Control	497
Chapter 26	Getting Started with the User Interface Control	499
	Creating an Instance of the Control	499
	Initializing the UIClient	506
	Programming the ViaVoice User Interface	513
	Getting and Setting User Interface Characteristics	520
	Creating Custom Menus	523
	Summary	531

Chapter 27 **Classes, Structures, and Enumerations 533**

User Interface Control Classes	534
vvUIMenuInfo (Class - Visual Basic and MFC Only)	534
User Interface Control Structures	535
UIMenuItemInfo Structure (Custom Interface Only)	535
User Interface Control Constants	537
Component Index Constants	538
vvUIDockingAlgorithmConstants	540
vvUIDockingStyleConstants	542
vvUIEventCallbackFlags	543
vvUIExtendedMenuFlags	545
vvUIMaxConstants	546
vvUIMenuItemConstants	547
vvUIRemoveClientConstants	550
User Interface Control Enumerations	551
MICROPHONE_STATES (Enum)	552
TCID (Enum)	554
TVIEWTYPE (Enum)	555
UIEVENTRC (Enum)	556
UIMENUGROUP (Enum)	557
UIMENUTYPE (Enum)	558
UIRC (Enum)	559

Chapter 28 **Properties, Methods, and Events 561**

User Interface Control Properties	561
LanguageUI	562
User Interface Control Methods	565
AddApplicationByName	566
AddApplicationByWindow	568
AppendMenuItem	570
DeleteMenuItem	574
GetMenuItemInfo	577
GetNumberValue	580
GetStringValue	582
Initialize	585
InsertMenuItem	587
RemoveApplicationByName	591
RemoveApplicationByWindow	593

	SetClientCallback (Custom Interface)	595
	SetClientCallbackFlags.	597
	SetLanguageByID.	599
	SetLanguageByString.	601
	SetMenuItemInfo	603
	SetNumberValue	606
	SetStringValue.	609
	User Interface Control Events	611
	EventActiveApplication	612
	EventButtonPressed	615
	EventComponentUpdated.	617
	EventMenuItemSelected.	619
	EventQueryViewFlags	621
	EventQueryViewMenuInfo.	624
Chapter 29	User Interface Control Frequently Asked Questions	629
Chapter 30	Introduction to the DictationMgr Control	631
Chapter 31	Getting Started with the DictationMgr Control	633
	Creating an Instance of the Control.	633
	Capturing Speech.	639
	Summary	640
Chapter 32	Properties, Methods, and Events	641
	Dictation Manager Control Properties.	641
	AutoDictationWindow (Run Time Only).	642
	CursorIndex	645
	DictationOn.	647
	Engine (Run Time Only)	649
	ExpandMacros	651
	Locked	653
	ProcessingMacro (Run Time Only)	655
	UppercaseOn.	657
	DictationMgr Control Methods	659
	Command	660
	Correct	663
	DeleteText.	666
	GetAlternate	668
	GetText	671

	GetWordInfo	673
	Playback.	676
	PlaybackEx2	677
	PutText.	678
	SetSelection	680
	DictationMgr Control Events	682
	DeleteText	683
	DictationStateChange	685
	PutText.	687
Chapter 33	DictationMgr Control Frequently Asked Questions	691
Chapter 34	Getting Started with the Dictation Control	693
Chapter 35	Introduction to the Dictation Control	699
Chapter 36	Properties, Methods, and Events	701
	Dictation Control Properties.	701
	AutoDictationWindow.	702
	DictationOn	704
	Engine	706
	ExpandMacros	708
	ProcessingMacro	710
	Dictation Control Methods.	712
	Correct.	713
	GetAlternatePhrase	716
	GetFlags.	719
	GetWavData.	721
	GetWordInfo	723
	MergeRecoPhrases	725
	SetBookMark.	727
	SetContext	729
	SplitOutLeftWord	731
	Dictation Control Events	734
	DictationStateChange	735
	HitBookMark.	737
	PhraseReco	739
	VVDictation Phrase Formatting Flags	741

Chapter 37	Dictation Control Frequently Asked Questions	743
Chapter 38	Getting Started with the Virtual Voices Control	745
	Overview	745
	How the Virtual Voices Control Works	747
	Speak.	749
	Paste	750
	Properties	750
	Programming Interfaces	755
Chapter 39	Introduction to Virtual Voices Control	757
	Files and Directories that Support Virtual Voices	757
Chapter 40	Properties, Methods, and Events	759
	Virtual Voices Control Properties	759
	ActorName	760
	Age (Read Only)	761
	AllowProperties	762
	BackColor	763
	Clipping	764
	DefaultExpression	766
	Expression	767
	Gender (Read Only)	768
	ModeGuid	769
	Pitch	771
	ShowMenu	772
	SpeakText	773
	Speed	774
	UseFace	775
	UseWave	776
	Volume	777
	WaveFileName	778
	ViaVoice Outloud (Text-To-Speech) Engine Attributes	779
	Breathiness	780
	HeadSize	781
	PitchFluctuation	782
	Roughness	783
	Example - Setting a Property	784
	Other Useful Properties	786

Virtual Voices Control Methods	790
AboutBox.....	791
Cancel	792
DoProperties	793
Pause	794
Resume	795
Speak	796
Example - Using a Method	797
Virtual Voices Control Events	799
BookMark	800
InitDone.....	802
KeyPress	804
Pause	806
Reset	807
Resume	808
StartSpeaking.....	809
StopSpeaking.....	810
WordPosition.....	811

Chapter 41

Programming Notes 813

Visual Basic Notes	813
Visual C++ Notes	813
Face Customization Notes	814
Resources	815
Bitmaps	816
Face (.FAC) File	824
Parameter (.PAR) File	829
Registry Entry	830
Testing Your Face	831
Style Considerations.....	833

Chapter 42

Virtual Voices Control Frequently Asked Questions 835

Appendix A

Notices 837

Index

839

About This Document

This book provides information on incorporating speech technology into applications using the IBM ViaVoice ActiveX Controls. It describes the programming interfaces available for developers to take advantage of these features within their applications. This book is prepared in Portable Document Format (PDF) to provide the advantages of text search and cross-reference hyperlinking and is viewable with the Adobe Acrobat Reader v.3.x. We recommend that you print all or part of this guide for quick reference.

Who Should Read This Book

Read this book if you are a developer interested in writing Windows 95/98 or Windows NT 4.0 applications that use ViaVoice ActiveX technology controls.

ViaVoice SDK Related Publications

Programming, reference and design information needed to use this SDK is available in a variety of sources:

- *SAPI Reference*

Refer to the following sources for additional programming, reference, and design information:

- *Developer's Corner Web Page* for SDK downloads, updates, and other documentation at: http://www.ibm.com/ViaVoice/dev_home.html
- *IBM ViaVoice SDK Web Channel* at: <http://www.software.ibm.com/ViaVoice/subscribe.html>

How This Book Is Organized

The Introduction contains an overview of the ActiveX controls documented in this guide as well as installation information. Chapter 1 through Chapter 4 contain information about the ViaVoice TextBox Control. Chapter 5 through Chapter 8 describe the ViaVoice RichEdit ActiveX Control. Chapter 9 through Chapter 12 describe the ViaVoice Phrases ActiveX Control. Chapter 13 through Chapter 16 describe the ViaVoice Grammar ActiveX Control. Chapter 17 through Chapter 20 describe the ViaVoice Lite ActiveX Controls. Chapter 21 through Chapter 24 describe the ViaVoice Error Correction Window ActiveX Control. Chapter 25 through Chapter 29 discuss the ViaVoice User Interface Control. Chapter 30 through Chapter 33 describe the ViaVoice Dictation Manager ActiveX Control. Chapter 34 through Chapter 37 describe the ViaVoice Dictation ActiveX Control. Chapter 38 through 42 describe the Virtual Voices ActiveX Control. Finally, Appendix A contains notices and trademark information.

Document Conventions

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

<i>Italic</i>	Used for emphasis and for references to other documents.
Bold	Represents a menu option or other user interface control, such as command buttons. Also represents the names of properties, methods, and events.
Courier Regular	Represents sample code.
Courier Bold	Represents a new line of code in a code sample.

Introduction

The IBM ViaVoice ActiveX controls enable you to incorporate the power of IBM's speech engines to the applications you develop. This chapter contains information about installation and a general description of the controls.

Installing the ViaVoice ActiveX Components

The setup program of the IBM ViaVoice SDK automatically installs the components you select. There are sample programs that use different combinations of the controls. Refer to the ViaVoice SDK README file for complete installation instructions and other important information.

ActiveX Controls

This guide documents the attributes of the ActiveX controls included in the SDK:

- ViaVoice TextBox Control
- ViaVoice RichEdit Control
- ViaVoice Phrases Control
- ViaVoice Grammar Control
- ViaVoice Lite Controls
- ViaVoice Engine Control (available as a separate RTF file)
- ViaVoice Error Correction Window Control
- ViaVoice User Interface Control
- ViaVoice Dictation Manager Control
- ViaVoice Dictation Control
- Virtual Voices Control
- ViaVoice Detective Control (available as a separate PDF file)

Part 1: ViaVoice TextBox Control

The ViaVoice **TextBox** Control (**VVTextBox**) is an ActiveX control that can capture speech input and turn it into text. It is an edit control similar to the Visual Basic native TextBox control. What separates this control from other textboxes or edit controls in the market is that it enables users to not only type information into text fields, but to also dictate their text through an audio input device (such as a microphone). For more information about the **VVTextBox** control, refer to the following chapters:

- Chapter 1, “Introduction to the TextBox Control” on page 27
- Chapter 2, “Getting Started with the TextBox Control” on page 29
- Chapter 3, “Properties, Methods, and Events” on page 41
- Chapter 4, “TextBox Control Frequently Asked Questions” on page 83

Part 2: ViaVoice RichEdit Control

The ViaVoice **RichEdit** Control (**VVRichEdit**) can capture speech input and turn it into text. It is an edit control similar to the Visual Basic native RichTextBox control or the Windows RichEdit control. What separates this control from other textboxes or edit controls in the market is that it enables users to not only type information into text fields, but to also dictate their text through an audio input device (such as a microphone). For more information about the **VVRichEdit** control, refer to the following chapters:

- Chapter 5, “Introduction to the RichEdit Control” on page 85
- Chapter 6, “Getting Started with the RichEdit Control” on page 87
- Chapter 7, “Properties, Methods, and Events” on page 99
- Chapter 8, “RichEdit Control Frequently Asked Questions” on page 201

Part 3: ViaVoice Phrases Control

The ViaVoice **Phrases** Control (**VVPhrases**) enables developers to add simple phrase command recognition to their application. The main idea behind the control is to have the developer provide the control with a list of phrases or commands, and the control will notify the developer when the user speaks one of the phrases in the list. For more information about the **VVPhrases** control, refer to the following chapters:

- Chapter 9, “Introduction to the Phrases Control” on page 203

- Chapter 10, “Getting Started with the Phrases Control” on page 205
- Chapter 11, “Properties, Methods, and Events” on page 221
- Chapter 12, “Phrases Control Frequently Asked Questions” on page 277

Part 4: ViaVoice Grammar Control

The ViaVoice **Grammar** Control (**VVCFGram**) enables developers to use a context-free grammar file to add robust command recognition to the application. The main idea behind the control is that developers provide the control with a grammar file or source, and then the control will notify you when the user speaks a command constructed from the grammar. For more information about the **VVGrammar** control, refer to the following chapters:

- Chapter 13, “Introduction to the Grammar Control” on page 279
- Chapter 14, “Getting Started with the Grammar Control” on page 281
- Chapter 15, “Properties, Methods, and Events” on page 295
- Chapter 16, “Grammar Control Frequently Asked Questions” on page 359

Part 5: ViaVoice Lite Controls

The ViaVoice **Lite** Controls consist of three controls: **Dictation Lite** (**VVDictLite**), **Grammar Lite** (**VVGrammarLite**), and **Phrases Lite** (**VVPhrasesLite**). Each of these controls is a reduced-feature version of their counterparts optimized for the web. For more information about the **VVLite** controls, refer to the following chapters:

- Chapter 17, “Introduction to the Lite Controls” on page 361
- Chapter 18, “Getting Started with the Lite Controls” on page 363
- Chapter 19, “Properties, Methods, and Events” on page 381
- Chapter 20, “Lite Controls Frequently Asked Questions” on page 413

Part 6: ViaVoice Engine Control

The ViaVoice **Engine** Control (**VVEngine**) is a drag-and-drop control that communicates with SAPI compatible speech engines. This control enables developers to: specify a search criteria for finding what speech engine to connect to; read properties from the speech engine, such as vendor name,

language, dialect, etc.; and receive engine notifications, such as volume level, audio state changed, etc. For more information about the **VVEngine** control, refer to the following chapters:

- Refer to the Engine Control Guide on the SDK “Documentation” menu.

Part 7: ViaVoice Error Correction Window Control

The ViaVoice **Error Correction Window** Control (**VVECWin**) is an ActiveX control that enables developers to utilize a common error correction dialog allowing users to correct dictated text in an application. It provides a common, familiar interface that users will quickly and easily become accustomed to, enabling them to correct speech recognition errors and text formatting issues, by using their keyboard, mouse or voice commands. For more information about the **VVECWin** control, refer to the following chapters:

- Chapter 21, “Introduction to the ECWin Control” on page 415
- Chapter 22, “Getting Started with the ECWin Control” on page 417
- Chapter 23, “Properties, Methods, and Events” on page 431
- Chapter 24, “ECWin Control Frequently Asked Questions” on page 495

Part 8: ViaVoice User Interface Control

The ViaVoice **User Interface Client** Control (**UIClient**) provides a common interface for speech-enabled applications for managing all aspects of Speech UI. It is capable of presenting speech-related information in a number of ways, including a Taskbar View, a Docked View, a Minimal View, and an Agent View. For more information about the **User Interface** control, refer to the following chapters:

- Chapter 25, “Introduction to the User Interface Control” on page 497
- Chapter 26, “Getting Started with the User Interface Control” on page 499
- Chapter 27, “Classes, Structures, and Enumerations” on page 533
- Chapter 28, “Properties, Methods, and Events” on page 561
- Chapter 29, “User Interface Control Frequently Asked Questions” on page 629

Part 9: ViaVoice DictationManager Control

The ViaVoice **DictationManager** Control (**VVDictationMgr**) is a relatively high level control which provides much of the functionality a client needs to add dictation to their application. However, in order to use this control clients must be able to synchronize the **VVDictationMgr** with their application user interface through zero (0) based character indices. **VVDictationMgr** is a full ActiveX control, which means that it can be “dropped” onto a form and configured at “design-time” in most high-level language environments. Using **VVDictationMgr** allows the user to manage both typed and dictated text, get wave data for playback of dictated text, and perform correction of dictated text. For more information about the **VVDictationMgr** control, refer to the following chapters:

- Chapter 30, “Introduction to the DictationMgr Control” on page 631
- Chapter 31, “Getting Started with the DictationMgr Control” on page 633
- Chapter 32, “Properties, Methods, and Events” on page 641
- Chapter 33, “DictationMgr Control Frequently Asked Questions” on page 691

Part 10: ViaVoice Dictation Control

The ViaVoice **Dictation** Control (**VVDictation**) is a low-level dictation object providing only the basics necessary for dictation, correction and playback. This object is implemented as a simple COM object rather than a full ActiveX control and can not be “dropped” into a form and configured at design-time. For more information about the **VVDictation** control, refer to the following chapters:

- Chapter 35, “Introduction to the Dictation Control” on page 699
- Chapter 34, “Getting Started with the Dictation Control” on page 693
- Chapter 36, “Properties, Methods, and Events” on page 701
- Chapter 37, “Dictation Control Frequently Asked Questions” on page 743

Part 11: Virtual Voices Control

The ViaVoice **Virtual Voices** Control enables developers to incorporate personality into their applications. A personality is represented through a voice (using text-to-speech or prerecorded audio wave file) and an animated face. The voice and face become the spokesperson through which the user interacts with the application. For more information about the **Virtual Voices** control, refer to the following chapters:

- Chapter 39, “Introduction to Virtual Voices Control” on page 757
- Chapter 38, “Getting Started with the Virtual Voices Control” on page 745
- Chapter 40, “Properties, Methods, and Events” on page 759
- Chapter 41, “Programming Notes” on page 813
- Chapter 42, “Virtual Voices Control Frequently Asked Questions” on page 835

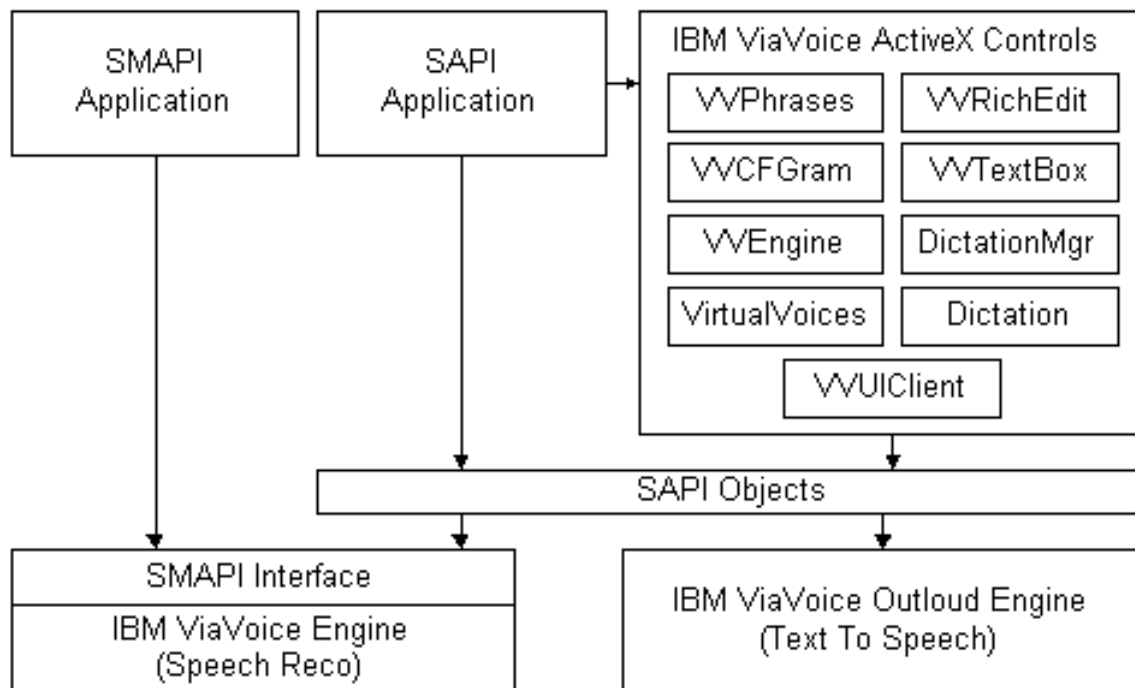
Part 12: ViaVoice Detective Control

The ViaVoice **Detective** Control (**VVDetective**) enables developers to perform speaker recognition. Speaker recognition is a biometric technology that uses voice patterns to recognize the speaker in various modes. **VVDetective** supports two modes of speaker recognition: verification and identification. For more information about the **VVDetective** control, refer to the following chapters:

- Refer to the Detective Control Guide on the SDK “Documentation” menu.

ViaVoice SDK Architecture

The following diagram shows the interaction of the components in the SDK.

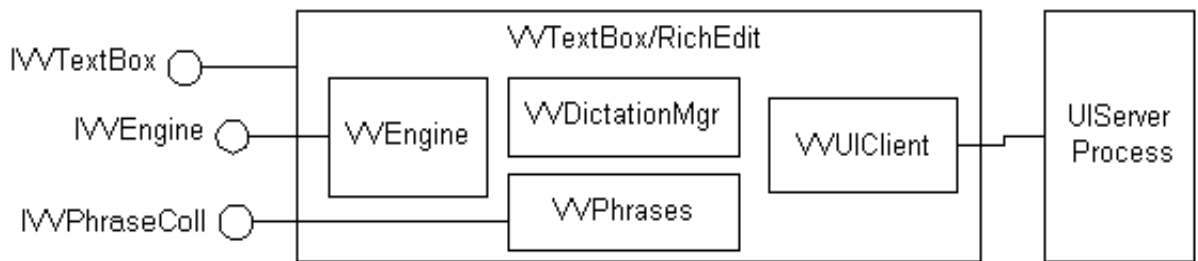


The ViaVoice **TextBox** Control (**VVTextBox**) is an ActiveX control that can capture speech input and turn it into text. It is an edit control similar to the Visual Basic native TextBox control. What separates this control from other textboxes or edit controls in the market is that it enables users to not only type information into text fields, but also to dictate their text through an audio input device (such as a microphone).

The **VVTextBox** control is also capable of understanding commands, which will enable the user to navigate and manipulate the contents of the textbox with ease.

VVTextBox Object Hierarchy

The following diagram shows the object hierarchy for the **VVTextBox**.



Getting Started with the TextBox Control

The following is a tutorial on how to incorporate the **VVTextBox** control into your Visual Basic or Visual C++ applications. This tutorial is designed to present you with the most commonly used properties and events in the **VVTextBox** control.

The following sections contain information to help you write code to create an instance of the **TextBox** control, then to capture speech, capture commands and create the environment to allow text correction.

Creating an Instance of the Control

This section contains step-by-step instructions for using Visual Basic or Visual C++ (MFC) to create an instance of the control.

In Visual Basic:

To add the **VVTextBox** control to your application, do the following:

1. From the **Project** menu, choose **Components**.
The Components dialog box, Figure 1, appears. The Components dialog lists all the ActiveX Controls that you can use in your application.

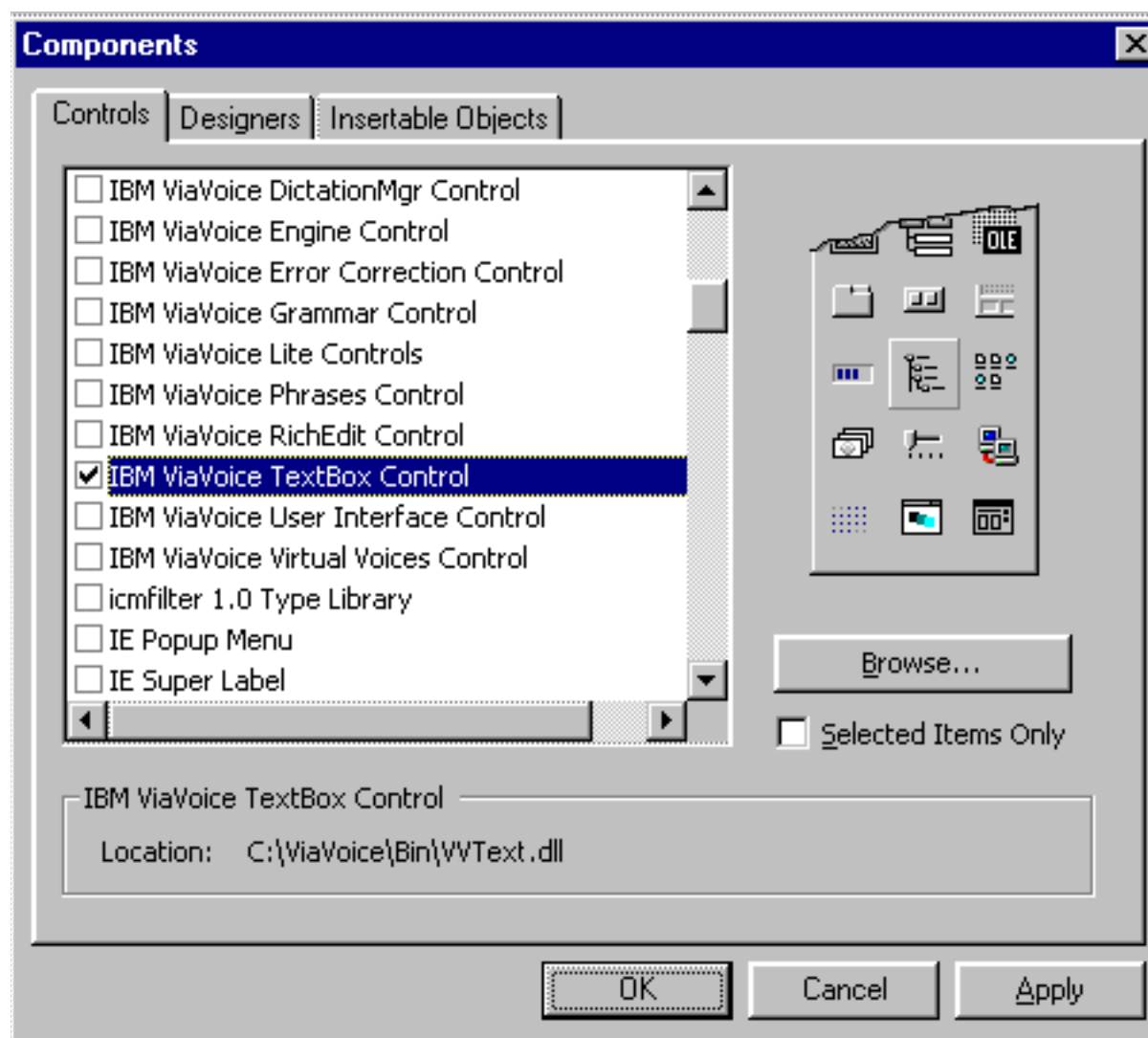


Figure 1. Component Selection Dialog - Visual Basic

2. Select **IBM ViaVoice TextBox Control** from the list and click **OK**.

Visual Basic adds the control to your project, and adds a new icon to the toolbar (Figure 2).



Figure 2. VVTextBox Control Toolbar Icon

3. Add an instance of the **VVTextBox** control to your form.

The **VVTextBox** control looks and acts much like the Visual Basic native TextBox control.

In Visual C++ (MFC):

To add the **VVTextBox** to your MFC project, do the following:

1. From the **Project** menu, select **Add To Project**, then select **Components and Controls**.

The 'Components and Controls Gallery' dialog box, Figure 3, appears.

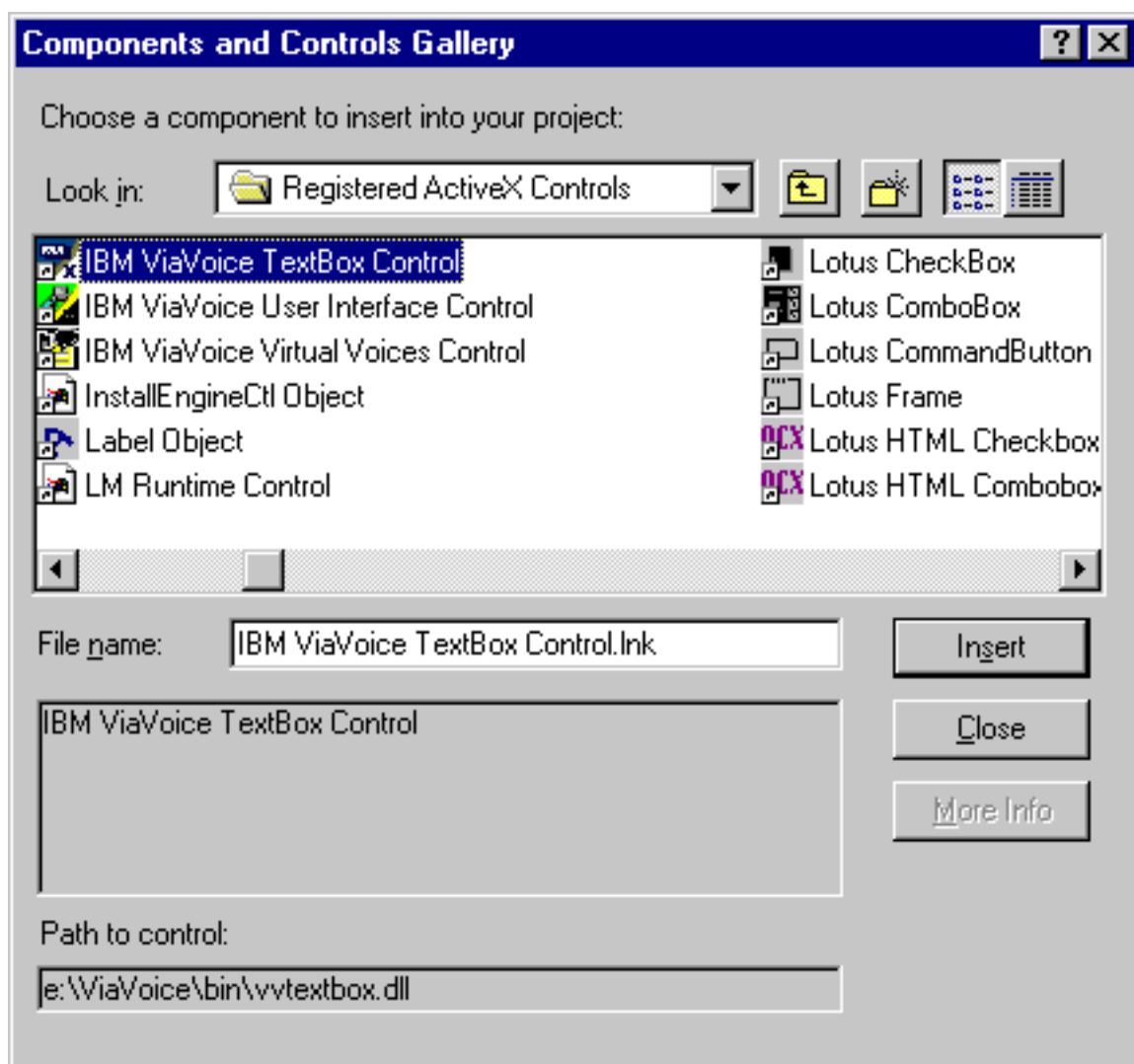


Figure 3. Insert ActiveX Control Dialog Box - Visual C++

2. Double-click the 'Registered ActiveX Controls' folder in the dialog box.
3. Select the **IBM ViaVoice TextBox Control** icon in the list of controls, then click **Insert**.

A confirmation message box appears, asking “Insert this component?”, just click **OK**.

4. Respond to the confirmation message box by clicking **OK**.

The ‘Confirm Classes’ dialog box, Figure 4, appears listing the Dual interface of the **TextBox** control (CVVTextBox), along with the accompanying Engine (CVVEngine), PhraseColl (CVVPhraseColl), Phrase (CVVPhrase), Font (COleFont), and Picture (CPicture) interfaces.

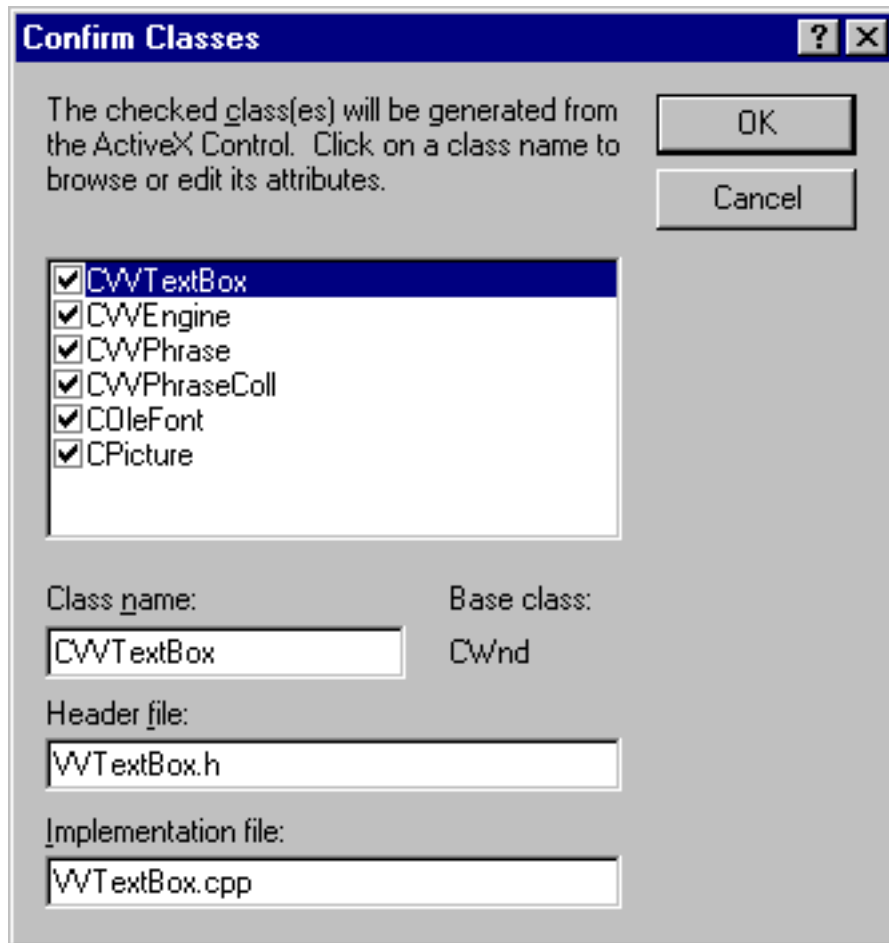


Figure 4. Confirm Classes Dialog Box

5. Click **OK** in the 'Confirm Classes' dialog box.
6. Close the 'Components and Controls Gallery' dialog box.
If you examine the Project Workspace window in the class view, you will notice six classes: CVVTextBox, CVVEngine, CVVPhraseColl, CVVPhrase, COleFont, and CPicture (assuming you accepted the default names for the class in the 'Confirm Classes' dialog box).
7. In the resource view of your Project Workspace window, double-click the dialog resource entry where you wish to insert the **VVTextBox** control.
The **VVTextBox** icon, Figure 5, appears in the Controls toolbar.

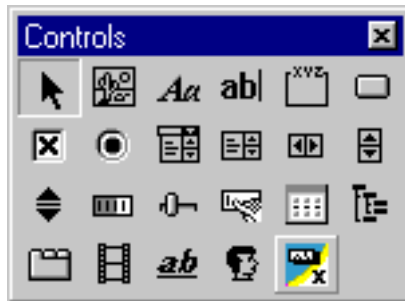


Figure 5. VVTextBox Icon in the Controls Toolbar

8. Add an instance of the **VVTextBox** control to the dialog box.
After you add the **VVTextBox** control to your dialog you can invoke Class Wizard to create a member variable for your class of type CVVTextBox. You might also decide to capture the events in the control by adding Event handlers to your dialog class. To add Event handlers, you can use the Class Wizard just like adding notification message handlers for a non-speech edit box.

Capturing Speech

The **VVTextBox** turns speech input into text; however, users of your application might not want every word they speak to be transformed. For example, users might need to perform other speaking tasks, such as answering the telephone or conversing with someone. By default, the **VVTextBox** operates in what is known as modeless operation. This means that whenever the control has window focus, the control will perform speech to text conversion, in the same way that it will accept keyboard input when the window has focus. But, you can also control dictation explicitly by specifying a window (see “AutoDictationWindow” on page 42) and setting the state of dictation (see “DictationOn” on page 54) as needed to implement almost any dictation activation logic you choose.

For example, you could use your application's top-most window for the **AutoDictationWindow**, so that dictation would automatically be disabled when your application is not the active application, and use **VVPhrases** control (or you could leverage the **VVTextBox Commands** property provided for extensibility) to add commands for “BEGIN-DICATION” and “STOP-DICATION” which would control the state of the **DictationOn** property. This example implements what is known as “modal dictation”.

When the control enters or exits dictation mode, it fires the **DictationStateChange** event.

One thing to keep in mind when working with the **VVTextBox** control is that the **TextBox** treats text from speech input the same as typed text. This means that properties like **MaxLength**, (which limits the number of characters the user can type into the **TextBox**) are still enforced the same way when the characters are generated from speech. Also, the control fires the **Change** event when the change occurs from spoken text just as it does when the change comes from typing.

Capturing Commands

In addition to dictated speech, the **VVTextBox** can recognize spoken commands. By default, the **VVTextBox** control listens for command speech when the control window has focus. If you specify an **AutoDictationWindow**, then that window will also be used for command activation tracking so that, by default, commands are always available when dictation is available. If you wish, you can achieve a finer granularity of control by explicitly setting the **CommandsEnabled** property in much the same way as using **DictationOn** to control dictation state.

The **VVTextBox** is capable of understanding eighteen command phrases. Refer to the following table for a complete listing.

Table 1. VVTextBox Command Phrases

ID (Value)	Command Phrase	Description
vvTBCapitalizeThis (13)	“CAPITALIZE-THIS”	Capitalizes selected text or word at the cursor.
vvTBCopyThis (8)	“COPY-THIS”	Copies selected text to the clipboard.
vvTBCorrectThis (24)	“CORRECT-THIS”	Shows the error correction window.
vvTBCutThis (7)	“CUT-THIS”	Cuts selected text to the clipboard.
vvTBDeleteThis (10)	“DELETE-THIS”	Clears selected text or word at the cursor.
vvTBHideEC (6)	“HIDE-CORRECTION-WINDOW”	Hides the error correction window.
vvTBLowercaseThis (15)	“LOWERCASE-THIS”	Changes selected text or word at the cursor to lowercase.
vvTBMoveBeginning (22)	“MOVE-TO-BEGINNING-OF-DOCUMENT”	Moves to the beginning of the text.
vvTBMoveEnd (23)	“MOVE-TO-END-OF-DOCUMENT”	Moves to the end of the text.
vvTBNextWord (11)	“NEXT-WORD”	Places the cursor at the beginning of the next word.

Table 1. VVTextBox Command Phrases

ID (Value)	Command Phrase	Description
vvTBPasteThis (9)	“PASTE-THIS”	Pastes text from the clipboard onto the VVTextBox control.
vvTBPreviousWord (12)	“PREVIOUS-WORD”	Moves cursor to the beginning of the previous word.
vvTBScratchThat (18)	“SCRATCH-THAT”	Deletes the last dictated phrase.
vvTBSelectThis (20)	“SELECT-THIS”	Selects the text at the cursor.
vvTBShowEC (5)	“SHOW-CORRECTION-WINDOW”	Shows the error correction window.
vvTBUppercaseOff (17)	“UPPERCASE-OFF”	Removes the “Dictation Caps Lock” feature so all subsequent text is lowercase.
vvTBUppercaseOn (16)	“UPPERCASE-ON”	Enables the “Dictation Caps Lock” feature so all subsequent dictated text is uppercase.
vvTBUppercaseThis (14)	“UPPERCASE-THIS”	Changes selected text or word at the cursor to uppercase.

Whenever the user speaks one of the command phrases in the above list, the **VVTextBox** control performs the corresponding action and fires the **Command** event indicating the ID of the command and the textual representation of the command. You can also use the **Commands** property to extend this list of commands and provide a more “natural language” type of control for the **VVTextBox**. To do this simply use the **Add** method of the **Commands** property to add additional commands using the same ID as the standard commands implementing the desired semantics. The **VVTextBox** will then respond to those commands in the same way as the original commands. You can also use the same **Commands** property to disable or remove any of the standard commands you choose.

Text Correction

The user can use the ViaVoice **TextBox** control's correction window to correct words that the engine has interpreted incorrectly. By default there are two ways for the user to invoke the correction window. If **CommandsEnabled** is true and the cursor is placed within the word to be corrected, the user can use the voice command "CORRECT-THIS" or "SHOW-CORRECTION-WINDOW". Or, the user can "right-click" on the textbox and choose correction from the context menu. Either approach will display the Error Correction window (see Figure 6) with the highlighted word/phrase or the word at the cursor location. You may extend this by using the **Commands** method to add additional voice commands or by using the **ExecuteCommand** method for programmatic invocation.

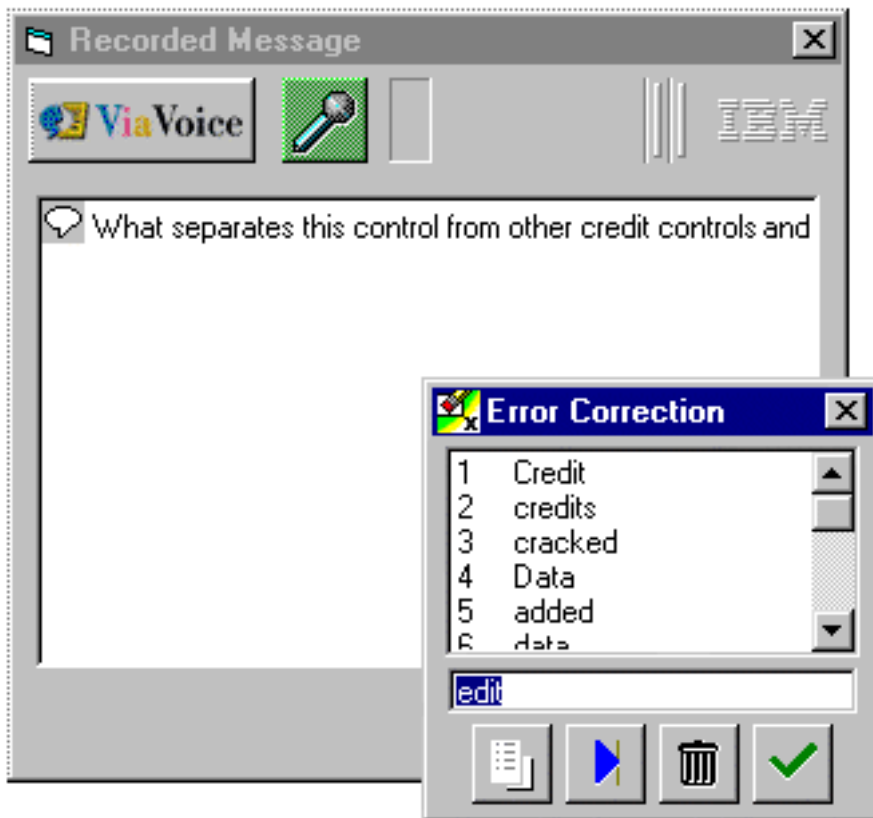


Figure 6. Error Correction Window

The error correction window shows a list of alternate words that the speech engine recognizes that are close in pronunciation to the word the user dictated. The user can choose a word from the list by clicking on it with the mouse and the word is replaced in the textbox control. Or the user can edit the mis-spoken word in the edit field and click the checkmark button to replace the selected text in the **VVTextBox** with whatever is in the edit field of the error correction window and apply the correction. After the user corrects the word, the **VVTextbox** control updates the speech recognition engine, so that the next time the user speaks the same word or phrase, the engine can interpret it correctly.

Summary

At this point, you should know how to do the following:

- How to incorporate the **VVTextBox** control into your project.
- How to control the activation of dictation.
- How to specify command words, and how to disable command capturing.
- How to correct dictated words using error correction.

The remainder of this documentation contains a reference for all the properties, methods, and events for the **VVTextBox** control.

TextBox Control Properties

The ViaVoice **TextBox** control supports the following properties:

Alignment^a	hWnd^a
Appearance^a	LanguageUI
AutoDictationWindow	Locked^a
AudioSourceType	MaxLength^a
AutoUI	MouseIcon^a
BackColor^a	MousePointer^a
BorderStyle^a	Multiline^a
Commands	PasswordChar^a
CommandsEnabled	ScrollBars^a
DictationOn	SelLength^a
Enabled^a	SelStart^a
Engine	SelText^a
Font^a	ShowDictationIcon
ForeColor^a	Text^a
HideSelection^a	

a. Represents standard properties in Visual Basic. For more information, refer to your Visual Basic documentation.

AutoDictationWindow (Read/Write at Run Time Only)

The **AutoDictationWindow** property provides a means of controlling the scope in which dictation is available. The default value of this property is the distinguished constant, **VV_HWND_AUTODICTATION** (-1), which will map dictation availability to ViaVoice TextBox window input focus^a. It can also be set to **VV_HWND_ALL** (0), which will enable dictation globally. However, please note that there can only be one global dictation object active (**DictationOn** is True) at any one time in the entire system (including other applications)! For this reason, it is strongly suggested that you avoid global dictation objects if at all possible. Alternatively, you can set this property to any valid "top-most"^b window handle, which maps dictation availability to that window's activation state (it or one of its children having focus).

Finer granularity of control can always be achieved by changing the state of **DictationOn** appropriately. When **DictationOn** is TRUE and the specified window is activated, the control will receive dictation. If **DictationOn** is FALSE, the control will not receive any dictation regardless of the value of this property.

Syntax

In Visual Basic:

<code>Property AutoDictationWindow As Long</code>

In Visual C++ (MFC):

<pre>long GetAutoDictationWindow(); void SetAutoDictationWindow(long nNewValue);</pre>

a. Dictation activation may potentially be modified by the state of the **DictationOn** property.

b. A "top-most" window is defined as any window without a parent. For more information on issues dealing with finding the "top-most" window, see the Microsoft Knowledge Base article Q84190.

In Visual C++:

```
HRESULT get_AutoDictationWindow(long * pVal);
HRESULT put_AutoDictationWindow(long newVal);
```

Parameters

??

Return Values

VV_HWND_AUTODICTATION

(Default) Dictation is available only when the **VVTextBox** control window has the focus.

Any valid “top-most^a” window handle

Dictation is available only when the indicated window is “active” as indicated by it, or one of its children, having the focus. **Note:** There can only be one dictation object active for the same window (**DictationOn** is True) at any one time

NULL

Dictation mode is always available and must be controlled manually by setting the **DictationOn** property to True or False. **Note:** There can only be one global dictation object active (**DictationOn** is True) at any one time!

Example

In Visual Basic:

```
'Assumes this form is the top-most form!
VVTextBox1.AutoDictationWindow = hWnd
```

a. A “top-most” window is defined as any window without a parent. For more information on issues dealing with finding the “top-most” window, see the Microsoft Knowledge Base article Q84190.

In Visual C++ (MFC):

```
// Makes no assumptions about m_hWnd
HWND Hwnd = m_hWnd;
// Due to the Win32 implementation of GetParent, this is necessary
// to find the "Foreground" window for SAPI grammar activation
// For more information see MS Knowledge Base article Q84190
while ( ::GetParent ( hwnd ) != NULL &&
        ! ( ::GetWindowLong( hwnd ,GWL_STYLE ) & WS_POPUP ) )
{
    hwnd = ::GetParent ( hwnd );
}
m_VVTextBox.SetAutoDictationWindow ( (long) hwnd );
```

In Visual C++:

```
// Makes no assumptions about m_hWnd
HWND Hwnd = m_hWnd;
// Due to the Win32 implementation of GetParent, this is necessary
// to find the "Foreground" window for SAPI grammar activation
// For more information see MS Knowledge Base article Q84190
while ( ::GetParent ( hwnd ) != NULL &&
        ! ( ::GetWindowLong( hwnd ,GWL_STYLE ) & WS_POPUP ) )
{
    hwnd = ::GetParent ( hwnd );
}
HRESULT hr = S_OK;
hr = m_pIVVDictationMgr->put_AutoDictationWindow ( (long)hwnd );
```

Remarks

If this property is set to VV_HWND_ALL and DictationOn = TRUE, the control will receive dictation. This enables an application to receive dictation when another application is active. Remember, if you use VV_HWND_ALL, be aware that there can only be one global dictation object active (**DictationOn** is True) at any one time. This includes your own *or any other application running on the system*. For this reason, global dictation objects should be used with extreme care and should probably be avoided unless absolutely necessary. Regardless of the value of this property, dictation will always be off if the **DictationOn** property is set to FALSE.

See Also

“DictationOn” on page 54

“DictationStateChange” on page 73

AutoUI

Controls whether ViaVoice **TextBox** displays and interacts with the ViaVoice **User Interface Server**.

Syntax

In Visual Basic:

```
Property AutoUI As Boolean
```

In Visual C++ (MFC):

```
BOOL GetAutoUI();  
void SetAutoUI(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_AutoUI(VARIANT_BOOL * pVal);  
HRESULT put_AutoUI(VARIANT_BOOL newVal);
```

Parameters

fNewValue
??

Return Values

TRUE

(Default) **VVTextBox** displays the **User Interface Server** and interacts with it automatically.

FALSE

VVTextBox does not display the **User Interface Server**. Also, it does not interact automatically with it if another control displays the **User Interface Server**.

Remarks

If multiple instances of the **VVTextBox** control have **AutoUI** set to True, the **User Interface Server** only gets created once, and all the instances of the control interact with the same **User Interface Server**. If you prefer not to display the **User Interface Server** or you do not want to have the **VVTextBox** control interact with it automatically, set this property to False.

By setting it to True, you will not be able to interact with the **UIServer** on behalf of your application. This means that even if you create an instance of the **UIClient**, you will never be able to control the state of the **UIServer**.

When **AutoUI** is True, the **VVTextBox** automatically updates the following components:

- Microphone
- Word History
- Volume Level

Example

In Visual Basic:

```
VVTextBox1.AutoUI = True
```

In Visual C++ (MFC):

```
m_VVTextBox.SetAutoUI(TRUE);
```

In Visual C++:

```
HRESULT hr = m_pIVVTextBox->put_AutoUI(VARIANT_TRUE);
```

See Also

Refer to the following chapters for more information about the ViaVoice **User Interface Control**:

Chapter 25, “Introduction to the User Interface Control” on page 497

Chapter 26, “Getting Started with the User Interface Control” on page 499

Chapter 27, “Classes, Structures, and Enumerations” on page 533

Chapter 28, “Properties, Methods, and Events” on page 561

Chapter 29, “User Interface Control Frequently Asked Questions” on page 629

Commands

Gets or sets the collection of commands used for voice control within the **VVTextBox**.

Syntax

In Visual Basic:

```
Property Commands As IVVPhraseColl
```

In Visual C++(MFC):

```
CVVPhraseColl GetCommands();  
void SetRefCommands(LPDISPATCH newValue);
```

In Visual C++:

```
HRESULT get_Commands(IVVPhraseColl ** pVal);  
HRESULT putref_Commands(IVVPhraseColl * pVal);
```

Settings

The **Commands** property settings for a ViaVoice **TextBox** control are:

Value	Description
Default Commands	(Default) The default commands used by the VVTextBox .
Any valid VVPhraseColl	Custom set of commands to be used by the VVTextBox .

Example

In Visual Basic:

```
Set VVPhrases1.Phrases = VVTextBox1.Commands
AddCustomCommands ( VVPhrases1 )
Set VVTextBox2.Commands = VVPhrases1.Phrases
```

In Visual C++ (MFC):

```
CVVPhraseColl Commands;
Commands = m_VVTextBox1.GetCommands ( );
AddCustomCommands ( Commands );
m_VVTextBox2.SetRefCommands ( Commands );
```

In Visual C++:

```
IVVPhraseColl* pIVVPhraseColl;
HRESULT hr = m_pIVVTextBox->get_Commands(&pIVVPhraseColl);
AddCustomCommands(pIVVPhraseColl);
hr = m_pIVVTextBox->putref_Commands(pIVVPhraseColl);
```

Remarks

The **TextBox** can recognize command words while dictation is off or on (see “DictationOn” on page 54) as long as speech input is available (see “AutoDictationWindow” on page 42). Finer granularity of control for command availability is always available through use of the **CommandsEnabled** property. Notice the distinction between the requested state and availability, which is dependent on the **AutoDictationWindow**.

Also, the **Commands** property is actually the **Phrases** property of an implicitly created **VVPhrases** control. For more information on **VVPhrases** see Chapter 9 “Introduction to the Phrases Control”.

If you wish to change the phrase used to invoke a given command, you may simply change the required command phrase as indicated in the **VVPhrases** documentation. If you wish to add

additional commands for existing functionality, you must use the correct ID for the desired functionality. For instance, if you wish to say either “NEXT-WORD” or “MOVE-NEXT” to move the cursor to the next word, then you would jsut add “MOVE-NEXT” as a new phrase with the ID of VVTBNextWord. Then, when the user speaks either command, the cursor will move to the next word.

See Also

“AutoDictationWindow (Read/Write at Run Time Only)” on page 42

“Capturing Commands” on page 36

“CommandsEnabled” on page 52

“DictationOn” on page 54

CommandsEnabled

Returns or sets whether the ViaVoice **TextBox** control will recognize commands or not, when speech input is available.

Syntax

In Visual Basic:

```
Property CommandsEnabled As Boolean
```

In Visual C++(MFC):

```
BOOL GetCommandsEnabled();  
void SetCommandsEnabled(BOOL bNewValue);
```

In Visual C++:

```
HRESULT get_CommandsEnabled(VARIANT_BOOL * pVal)  
HRESULT put_CommandsEnabled(VARIANT_BOOL newVal)
```

Settings

The **CommandsEnabled** property settings for a ViaVoice **TextBox** control are:

Value	Description
True	(Default) The ViaVoice Textbox recognizes commands, when available.
False	Command words are ignored.

Example

In Visual Basic:

```
VVTextBox1.CommandsEnabled = True
```

In Visual C++ (MFC):

```
m_VVTextBox.SetCommandsEnabled(TRUE);
```

In Visual C++:

```
HRESULT hr = m_pIVVTextBox->put_CommandsEnabled(VARIANT_TRUE);
```

Remarks

The **TextBox** can recognize command words while dictation is off or on (see “DictationOn” on page 54) as long as speech input is available (see “AutoDictationWindow” on page 42). Conversely, command recognition will not be available if speech input is not available. Finer granularity of control for command availability is always available through use of the **CommandsEnabled** property.

See Also

- “AutoDictationWindow (Read/Write at Run Time Only)” on page 42
- “Capturing Commands” on page 36
- “Commands” on page 49
- “DictationOn” on page 54

DictationOn

Returns or sets the desired state of the dictation mode. You can think of this property semantically as “Client want's dictation on”. What this means is that if dictation is available (i.e. nothing is preventing dictation), then the user will be able to dictate into the control. Some possible reasons why dictation would be unavailable when **DictationOn** is True are: **MaxLength** has been reached, **Locked** is True, **Enabled** is False, or the semantics of the **AutoDictationWindow** indicate that dictation is not available.

Syntax

In Visual Basic:

```
Property DictationOn As Boolean
```

In Visual C++(MFC):

```
BOOL GetDictationOn();  
void SetDictationOn(BOOL bNewValue);
```

In Visual C++:

```
HRESULT get_DictationOn(VARIANT_BOOL * pVal);  
HRESULT put_DictationOn(VARIANT_BOOL newVal)
```

Settings

The **DictationOn** property settings for a ViaVoice **TextBox** control are:

Value	Description
True	The control can receive dictation input when dictation is available.
False	The control ignores dictation input.

Example

In Visual Basic:

```
VVTextBox1.DictationOn = True
```

In Visual C++ (MFC):

```
m_VVTextBox.SetDictationOn(TRUE);
```

In Visual C++:

```
HRESULT hr = m_pIVVTextBox->put_DictationOn(VARIANT_TRUE);
```

Remarks

When dictation is off (`DictationOn = False`) the user may still be able to issue commands. For more information, refer to “`CommandsEnabled`” on page 52.

When the state of the dictation mode changes, the control fires the **DictationStateChange** event. You should not set the value of this property in the **DictationStateChange** event, as this will cause the event to trigger again.

See Also

“`AutoDictationWindow` (Read/Write at Run Time Only)” on page 42

“`CommandsEnabled`” on page 52

“`DictationStateChange`” on page 73

“`Enabled`” in Visual Basic Documentation

“`Locked`” in Visual Basic Documentation

“`MaxLength`” in Visual Basic Documentation

Engine

Contains a reference to the ViaVoice **Engine** control (**VVEngine**), which is used by the **VVTextBox** control. For more information, see the Engine Control Guide on the SDK “Documentation” menu.

Syntax

In Visual Basic:

```
Property Engine As IVVEngine
```

In Visual C++(MFC):

```
CVVEngine GetEngine();  
void SetRefEngine(LPDISPATCH newValue);
```

In Visual C++:

```
HRESULT get_Engine(IVVEngine * * pVal);  
HRESULT putref_Engine(IVVEngine * pVal);
```

Example

In Visual Basic:

```
VVTextBox1.Engine.AudioSourceType = vvstSAPICompliant
```

In Visual C++ (MFC):

```
m_VVTextBox.GetEngine( ).SetAudioSourceType(vvstSAPICompliant);
```


In Visual C++:

```
IVVEngine* pIVVEngine = NULL;
HRESULT hr = m_pIVVTextBox->get_Engine ( & pIVVEngine );
if ( SUCCEEDED ( hr ) )
    hr = pIVVEngine->put_AudioSourceType ( vvstSAPICompliant );
```

Remarks

The **Engine** property is actually holding an implicitly created ActiveX control (**VVEngine**), which can also be created separately. Inserting a **VVEngine** control in a project enables you to set the engine properties on this control and assign the resulting engine to multiple ViaVoice ActiveX controls.

See Also

Refer to the ViaVoice **Engine** Control Guide for more information.

LanguageUI

This property sets or gets the User Interface language used by the **VVTextBox** for this specific client. The language affects any dialogs, menus, strings or ToolTips displayed by the control.

Syntax

In Visual Basic:

```
Property LanguageUI As String
```

In Visual C++ (MFC):

```
CString GetLanguageUI();  
void SetLanguageUI(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_LanguageUI(BSTR * pVal);  
HRESULT put_LanguageUI(BSTR newVal);
```

Settings

The **LanguageUI** property settings for a **VVTextBox** control are:

Language	Property Value
U.S. English	"EN_US"
U.K. English	"EN_UK"
German	"GR_GR"
Italian	"IT_IT"
Spanish	"ES_ES"
French	"FR_FR"
Japanese	"JA_JP"

Example

In Visual Basic:

```
' Sets UI language to U.S. English
VVTextBox1.LanguageUI = "EN_US"
' Gets UI language and displays it in a message box
MsgBox VVTextBox1.LanguageUI
```

In Visual C++ (MFC):

```
// Sets UI language to U.S. English
m_VVTextBox.SetLanguageUI("EN_US");
CString sLangUI;
// Gets UI language and copies it into variable
sLangUI = m_VVTextBox.GetLanguageUI();
```

In Visual C++:

```
HRESULT hr;
BSTR bstrLangUI;

bstrLangUI = SysAllocString(OLESTR("EN_US"));
// Sets UI language to U.S. English
hr = pIVVTextBox->put_LanguageUI(bstrLangUI);
SysFreeString(bstrLangUI);

// Gets UI language into BSTR variable
hr = pIVVTextBox->get_LanguageUI(&bstrLangUI);
// Use language string now and when done free BSTR.
SysFreeString(bstrLangUI);
```

Remarks

None.

See Also

None.

ShowDictationIcon

Determines whether or not the **VVTextBox** displays a small “speech bubble” dictation icon indicating that it is capable of accepting dictation at run time. This can be used to easily differentiate between normal TextBox controls and **VVTextBox** controls, which can accept dictation.

Syntax

In Visual Basic:

```
Property ShowDictationIcon As Boolean
```

In Visual C++(MFC):

```
BOOL GetShowDictationIcon();  
void SetShowDictationIcon(BOOL bNewValue);
```

In Visual C++:

```
HRESULT get_ShowDictationIcon(VARIANT_BOOL * pVal);  
HRESULT put_ShowDictationIcon(VARIANT_BOOL newVal);
```

Settings

The **ShowDictationIcon** property settings for a ViaVoice **TextBox** control are:

Value	Description
True	VVTextBox displays a small dictation icon to indicate that it is capable of accepting dictation.
False	(Default) VVTextBox does not display the dictation icon.

Example

In Visual Basic:

```
VVTextBox1.ShowDictationIcon = True
```

In Visual C++ (MFC):

```
m_VVTextBox.SetShowDictationIcon(TRUE);
```

In Visual C++:

```
HRESULT hr = m_pIVVTextBox->put_ShowDictationIcon ( VARIANT_TRUE );
```

Remarks

The icon does not give any information about the current state or availability of dictation.

See Also

None.

TextBox Control Methods

The ViaVoice **TextBox** control supports the following methods:

- **About^a**
- **ExecuteCommand**
- **Playback**
- **PlaybackEx**
- **PlaybackEx2**
- **Refresh^a**

a. Represents a standard method in Visual Basic. For more information, refer to your Visual Basic documentation.

ExecuteCommand

This method allows the client to invoke any of the “voice commands” (see “Commands” property) programmatically simply by passing the ID of the command desired. For a complete list of voice commands and descriptions, see “Capturing Commands” on page 36. The primary use of this functionality is to expose voice command functionality for invocation via mouse or keyboard input. It can also be used to control actions based on voice commands in an external **VVPhrases** control. This might be useful, for instance, if you need to have voice commands activated/available based on your own logic using a different tracking window for the command phrases.

Syntax

In Visual Basic:

```
Sub ExecuteCommand(lCommandID As Long)
```

In Visual C++(MFC):

```
void ExecuteCommand(long lCommandID);
```

In Visual C++:

```
HRESULT ExecuteCommand(long lCommandID);
```

Example

In Visual Basic:

```
VVTextBox1.ExecuteCommand vvTBCorrectThis
```

In Visual C++ (MFC):

```
m_VVTextBox.ExecuteCommand ( vvTBCorrectThis );
```


In Visual C++:

```
HRESULT hr = m_pIVVTextBox->ExecuteCommand( vvTBCorrectThis );
```

Remarks

The **CommandsEnabled** property has no effect on commands invoked through the **ExecuteCommand** method.

See Also

“Capturing Commands” on page 36

“Commands” on page 49

“CommandsEnabled” on page 52

Playback

See Also

Chapter 7, “Playback” on page 178

PlaybackEx

See Also

Chapter 7, “PlaybackEx” on page 180

PlaybackEx2

See Also

Chapter 7, “PlaybackEx2” on page 182

TextBox Control Events

The ViaVoice **TextBox** control supports the following events:

Change^a	KeyUp^a
Click^a	MaxText
Command	MouseDown^a
DblClick^a	MouseMove^a
DictationStateChange	MouseUp^a
Error	
KeyDown^a	
KeyPress^a	

-
- a. Represents a standard event in Visual Basic. For more information, refer to your Visual Basic documentation.

Command

The control fires this event when the user speaks one of the command words the **VVTextBox** recognizes. (To see a complete list of commands that the **VVTextBox** control recognizes, refer to “Capturing Commands” on page 36.)

Syntax

In Visual Basic:

```
Event Command(CmdID As Long, strCommand As String)
```

In Visual C++(MFC):

```
void OnCommand (long CmdID, LPCTSTR strCommand);
```

Parameters

CmdID

Long

A number that uniquely identifies the command the **VVTextBox** recognized. For a complete list, refer to “Capturing Commands” on page 36.

strCommand

String

The actual command text the **VVTextBox** recognized. You should not write code that is dependent on this value as the phrases are subject to change and vary with the language of the engine. Use the **CmdID** parameter instead. It is recommended that you use the strCommand parameter for display only. This will be an empty string if invoked through any means other than the speech commands.

Example

In Visual Basic:

```
Private Sub VVTextBox1_Command(ByVal CmdID As Long, _
                                ByVal strCommand As String)

    Select Case CmdID
    Case vvTBShowEC
        ProcessTBShowEC
    . . .
    End Select

End Sub
```

In Visual C++ (MFC):

```
void CTestDlg::OnCommand(long CmdID, LPCTSTR strCommand)
{
    switch (CmdID)
    {
        case vvTBShowEC
            ProcessTBShowEC( );
            break;
        default:
            break
    }
}
```

Remarks

This control recognizes commands only when **CommandsEnabled** is set to True and there are no other limiting factors (see "Commands" and "CommandsEnabled" properties). Commands are never recognized when **CommandsEnabled** is False.

See Also

“Capturing Commands” on page 36

“Commands” on page 49

“CommandsEnabled” on page 52

DictationStateChange

The control fires this event when the control enters or exits dictation mode. There are several actions that affect the state of dictation – see the remarks below for more information.

Syntax

In Visual Basic:

```
Event DictationStateChange(DictationOn As Boolean)
```

In Visual C++(MFC):

```
void OnDictationStateChangeVvtextbox1(BOOL DictationOn);
```

Parameters

DictationOn

Boolean

The current state of dictation mode. True means the control is ready to receive dictation speech and turn it into text when speech input is available. False means that the control will ignore dictation input. This event implies nothing to do with the control being able to understand voice commands. For more information, refer to “CommandsEnabled” on page 52.

Example

In Visual Basic:

```
Private Sub VVTextBox1_DictationStateChange(ByVal DictationOn As Boolean)

    ProcessDictationOnEvent DictationOn

End Sub
```

In Visual C++ (MFC):

```
void CTestDlg::OnDictationStateChange(BOOL DictationOn)
{
    ProcessDictationOnEvent ( DictationOn )
}
```

Remarks

The following conditions can effect the state of dictation:

- The state of the **DictationOn** property is changed explicitly.
- Focus changes (see **AutoDictationWindow** on page 42).
- The state of the **Locked** property is changed explicitly.
- The state of the **Enabled** property is changed explicitly.
- The length of text reaches the max set in the **MaxLength** property.

A change indicated by the **DictationStateChange** event does not imply a change to the **DictationOn** property. **DictationOn** is semantically equivalent to "user wants dictation on". If dictation has been turned off in response to some action other than setting **DictationOn** to false (e.g., **Enabled** was set false) then **DictationOn** can be true after a **DictationStateChange** event indicates that dictation is "off".

Note:

If **AutoDictationWindow** is not set to `VV_HWND_AUTODICTIONATION`, then focus changes will not trigger this event, even if dictation availability has changed. If you need this information, then you must write code to track the window activation changes. This can be done by subclassing the window set for the **AutoDictationWindow**.

See Also

“CommandsEnabled” on page 52

“DictationOn” on page 54

“AutoDictationWindow (Read/Write at Run Time Only)” on page 42

“Enabled” in Visual Basic Documentation

“Locked” in Visual Basic Documentation

“MaxLength” in Visual Basic Documentation

Error

Event fired when the ViaVoice **TextBox** control reports an error.

Syntax

In Visual Basic:

```
Event Error(sErrorID As Integer, _  
    pstrDescription As String, _  
    hresult As Long, _  
    strSource As String, _  
    strHelp As String, _  
    lHelpID As Long, _  
    bShow As Boolean)
```

In Visual C++(MFC):

```
void OnError (short sErrorID, BSTR FAR* pstrDescription,  
    long FAR* hresult, BSTR FAR* strSource,  
    BSTR FAR* strHelp, long FAR* lHelpID, BOOL FAR* fShow);
```

Parameters

sErrorID

Integer. The error number. The error number can be one of the following values:

DICTERR_DICTATION_ACTIVATE	101 (Hex 65)
DICTERR_DICTATION_DEACTIVATE	102 (Hex 66)
DICTERR_COMMANDS_ACTIVATE	103 (Hex 67)
DICTERR_COMMANDS_DEACTIVATE	104 (Hex 68)
DICTERR_ENGINE_CONNECT	105 (Hex 69)

pstrDescription

String. The error description. The error message string is language-dependent and requires the use of the appropriate language resource DLL. The control will use the language of the container

application for error messages. If the control cannot find the appropriate language DLL, the error message will be in US English.

hresult

Long. The error code.

strSource

String. This parameter contains the name of the module where the error occurred.

strHelp

String. The name and path of the help file (HLP file) containing information about the error.

lHelpID

Long. The context ID of the page in the help file that explains the error.

fShow

Boolean. Set to True by default, the ViaVoice **TextBox** control will automatically display an error message dialog box when an error occurs. You can prevent the control from showing this dialog by setting this parameter to False.

Return Values

TRUE

(Default) ViaVoice **TextBox** control will automatically display an error message dialog box when an error occurs.

FALSE

The error message dialog will not display.

Remarks

The ViaVoice **TextBox** can report errors in one of two ways. If the error occurs from the setting of a property or the issuing of a method incorrectly, the control generates a trappable error (returns an error HRESULT). However, some errors can occur while the user is interacting with the control directly. Whenever the control needs to report this type of error, it fires the **Error** event.

Example

In Visual Basic:

```
Private Sub VVTextBox1_Error( _  
    sErrorID As Integer, _  
    pstrDescription As String, _  
    hresult As Long, _  
    strSource As String, _  
    strHelp As String, _  
    lHelpID As Long, _  
    bShow As Boolean)  
  
    Select Case sErrorID  
    Case DICTERR_ENGINE_CONNECT  
        MsgBox "Fatal Error! Unable to connect to speech engine."  
        bShow = False  
    End Select  
  
End Sub
```

In Visual C++ (MFC):

```
void CTestctrlDlg::OnError(
    short sErrorID,
    BSTR FAR* pstrDescription,
    long hresult,
    LPCTSTR strSource,
    LPCTSTR strHelp,
    long lHelpID,
    BOOL FAR* bShow)
{
    switch (sErrorID)
    {
    Case DICTERR_ENGINE_CONNECT
        MsgBox "Fatal Error! Unable to connect to speech engine.", "Speech
        Error", MB_OK);
        *bShow = TRUE;
        break;
    }
}
```

See Also

None.

MaxText

Event fired when the length of the text in the **TextBox** reaches the maximum number of characters allowed in the control.

Syntax

In Visual Basic:

```
Event MaxText()
```

In Visual C++ (MFC):

```
void OnMaxText();
```

Parameters

None.

Return Values

None.

Remarks

You can specify maximum number of characters through the **MaxLength** property. For more information about the **MaxLength** property, refer to the Visual Basic documentation

Setting the **MaxLength** property to zero means that the control accepts the maximum of a standard edit control, which is OS dependent. See Microsoft documentation for details.

Example

In Visual Basic:

```
Private Sub VVTextBox1_MaxText()  
    'AutoTab when all the information has been entered  
    SendKeys "{TAB}"  
End Sub
```

In Visual C++ (MFC):

```
void CTestDlg::MaxText()  
{  
    //AutoTab when all the information has been entered  
    GetNextDlgTabItem(GetFocus())->SetFocus();  
}
```

See Also

“MaxLength” in Visual Basic Documentation.

TextBox Control Frequently Asked Questions

This chapter contains answers to the most frequently asked questions about the ViaVoice **TextBox** Control.

How can I enter dictation mode automatically each time the VVTextBox gets focus?

Set the **AutoDictationWindow** property to `VV_HWND_AUTODICTIONATION (-1)` at run time (this is the default). By setting this value for **AutoDictationWindow** and setting **DictationOn** to true, the control will automatically enter dictation mode when it gets the focus, and exit dictation mode when it loses focus. Setting **DictationOn** to false will still disable dictation regardless of window focus.

How can I get more control in determining when dictation is available?

One way is to set the **AutoDictationWindow** property to `NULL` at design time to enable “global” dictation. By setting this value, the control will always accept dictation when **DictationOn** is true and will stop accepting dictation when **DictationOn** is false. There can only be one (1) global dictation object active (**DictationOn** set to true) in the entire system at any one time.

Another option is to use some other window for implicit dictation control. To do this, simply find the “top-most” window in the application of interest (your own or any other application) and assign it to **AutoDictationWindow** before setting **DictationOn** to true. This has the effect of enabling dictation any time that window, or any of its child windows, has focus. Using a window for dictation tracking provides the benefit of greater control without the problems associated with a “global” dictation object.

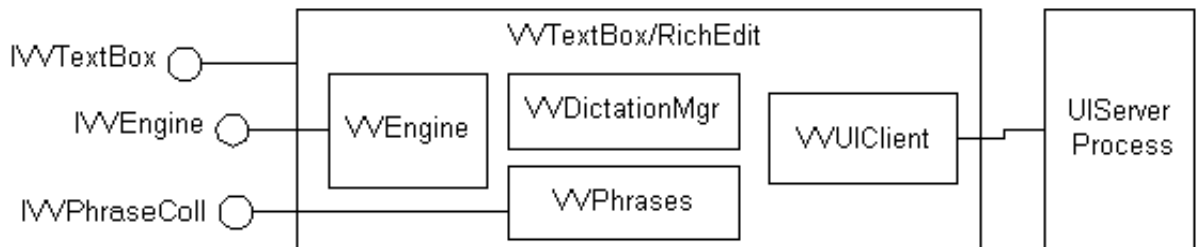
Introduction to the RichEdit Control

The ViaVoice **RichEdit** Control (**VVRichEdit**) is an ActiveX control which allows programmers to incorporate dictation-enabled word processing into their applications. The control contains many methods and properties which allow the user to create rich text format (RTF) files like a Windows rich edit control. What separates **VVRichEdit** from other rich edit controls is its ability to receive dictation from an audio input device (such as a microphone). The dictation will be adorned with any rich text formatting that would be required if the user was actually typing the text. The **VVRichEdit** control also provides voice commands to allow the user to navigate and manipulate its contents.

VVRichEdit Object Hierarchy

VVRichEdit actually utilizes controls of the ViaVoice SDK for many aspects of its functionality. To receive, synchronize, and correct dictation, it interacts with **VVDictationMgr**. For an error correction user interface, the control uses **VVECWin**. Internally a **VVPhrases** control is used to notify **VVRichEdit** if the user has uttered any voice commands.

The following diagram shows the object hierarchy for the **VVRichEdit**.



Getting Started with the RichEdit Control

The following is a tutorial on how to incorporate the **VVRichEdit** control into your Visual Basic or Visual C++ applications. This tutorial is designed to present you with the most commonly used properties and events in the **VVRichEdit** control.

The following sections contain information to help you write code to create an instance of the **RichEdit** control, then to capture speech, capture commands and create the environment to allow text correction.

Creating an Instance of the Control

This section contains step-by-step instructions for using Visual Basic or Visual C++ (MFC) to create an instance of the control.

In Visual Basic:

To add the **VVRichEdit** control to your application, do the following:

1. From the **Project** menu, choose **Components**.
The 'Components' dialog box, Figure 7, appears. The 'Components' dialog lists all the ActiveX Controls that you can use in your application.

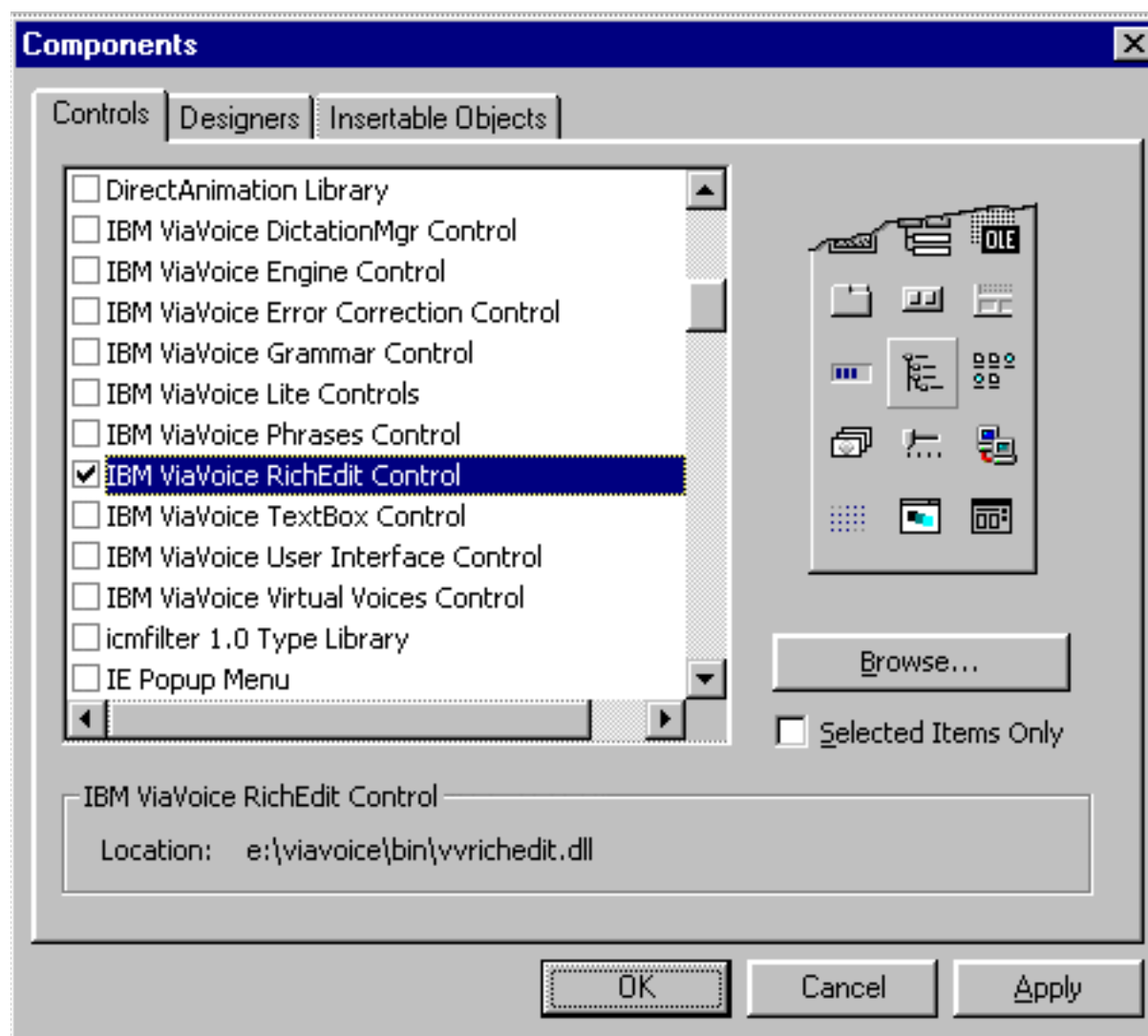


Figure 7. Component Selection Dialog - Visual Basic

2. Select **IBM ViaVoice RichEdit Control** from the list and click **OK**.

Visual Basic adds the control to your project, and adds a new icon to the toolbar (Figure 8).



Figure 8. VVRichEdit Control Toolbar Icon

3. Add an instance of the **VVRichEdit** control to your form.
The **VVRichEdit** control looks and acts much like the Visual Basic native RichTextBox control.

In Visual C++ (MFC):

To add the **VVRichEdit** to your MFC project, do the following:

1. From the **Project** menu, select **Add To Project**, then select **Components and Controls**.
The 'Components and Controls Gallery' dialog box, Figure 9, appears.

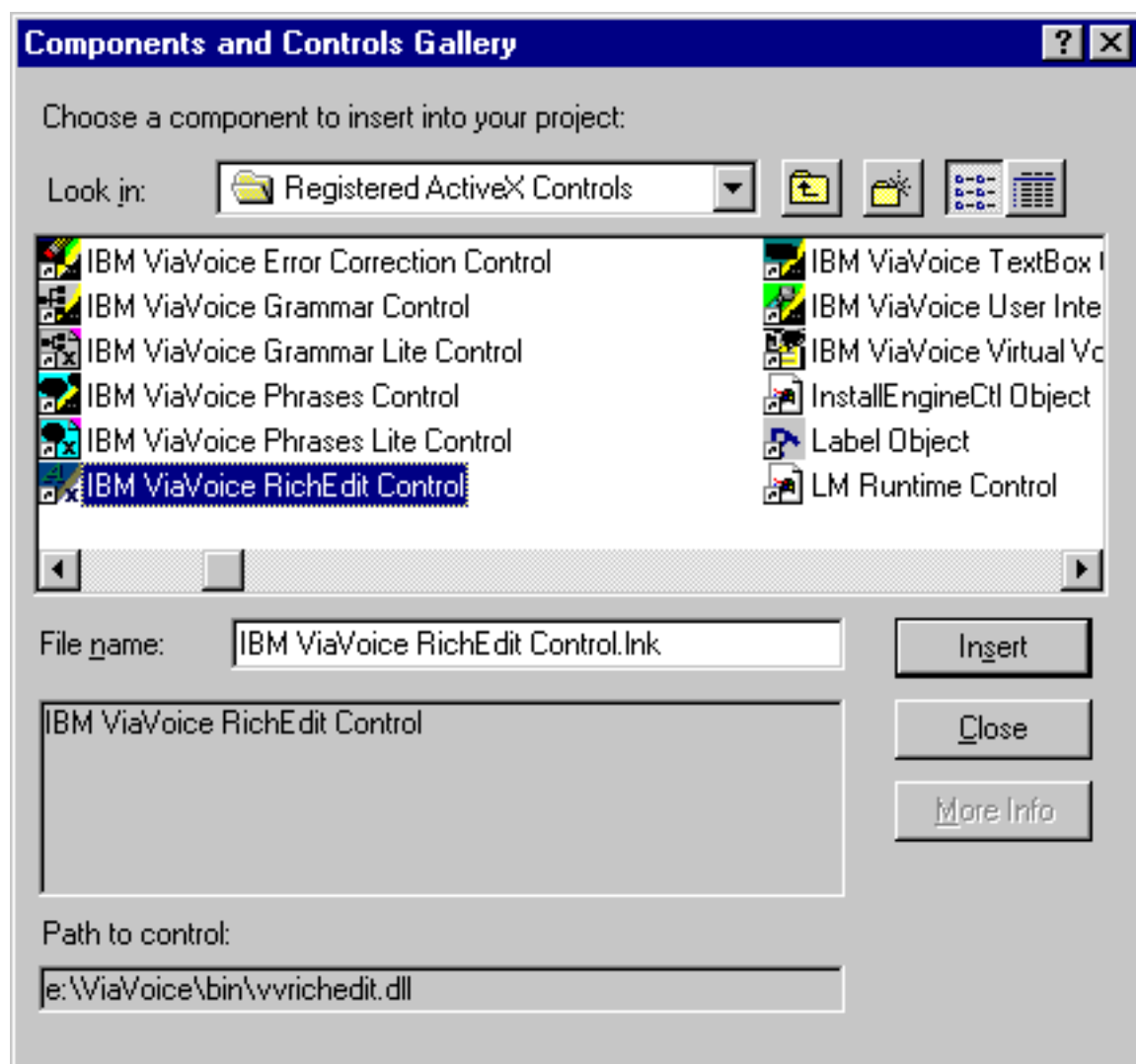


Figure 9. Insert ActiveX Control Dialog Box - Visual C++

2. Double-click the 'Registered ActiveX Controls' folder in the dialog box.
3. Select the **IBM ViaVoice RichEdit Control** icon in the list of controls, then click **Insert**.

A confirmation message box appears, asking “Insert this component?”, just click **OK**.

4. Respond to the confirmation message box by clicking **OK**.

The ‘Confirm Classes’ dialog box, Figure 10, appears listing the Dual interface of the **RichEdit** control (CVVRichEdit), along with the accompanying Engine (CVVEngine), Phrase (CVVPhrase), PhraseColl (CVVPhraseColl), Font (COleFont), and Picture (CPicture) interfaces.

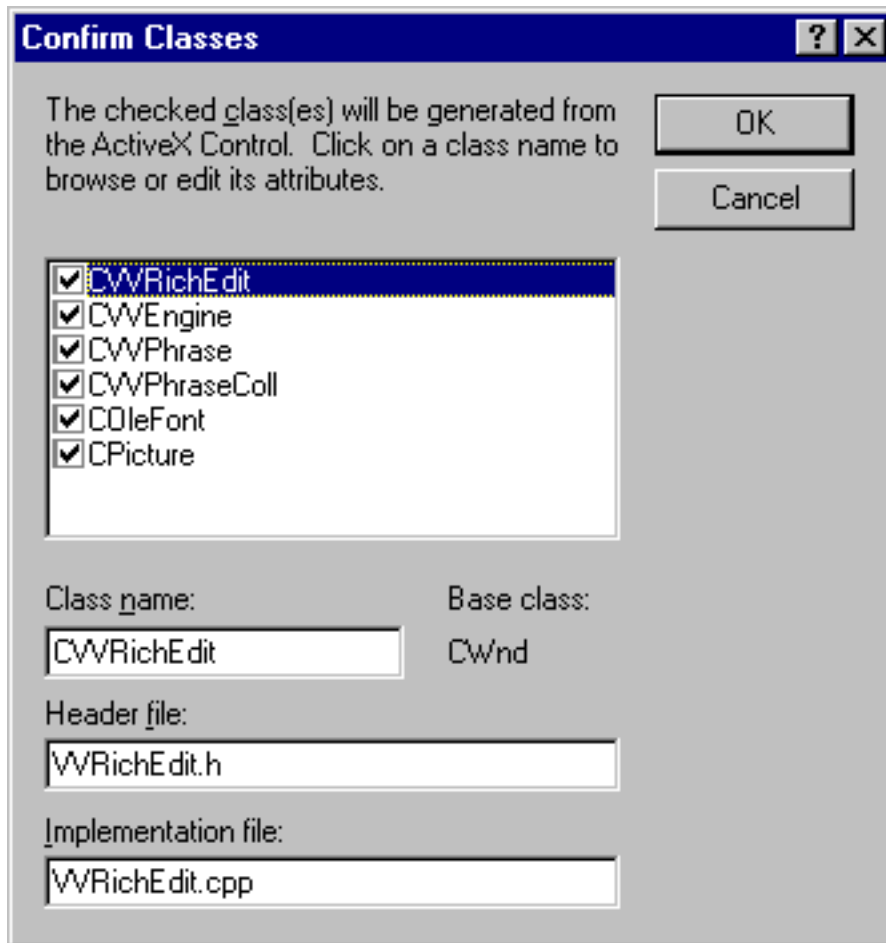


Figure 10. Confirm Classes Dialog Box

5. Click **OK** in the 'Confirm Classes' dialog box.
6. Close the 'Components and Controls Gallery' dialog box.
If you examine the Project Workspace window in the class view, you will notice six classes: CVVRichEdit, CVVEngine, CVVPhrase, CVVPhraseColl, COleFont, and CPicture (assuming you accepted the default names for the class in the 'Confirm Classes' dialog box).
7. In the resource view of your Project Workspace window, double-click the dialog resource entry where you wish to insert the **VVRichEdit** control.
The **VVRichEdit** icon, Figure 11, appears in the Controls toolbar.

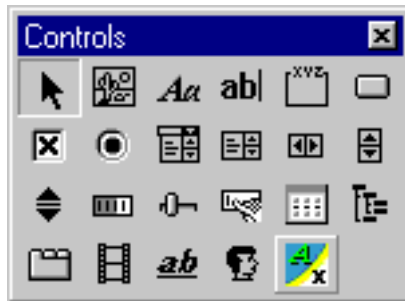


Figure 11. VVRichEdit Icon in the Controls Toolbar

8. Add an instance of the **VVRichEdit** control to the dialog box.
After you add the **VVRichEdit** control to your dialog you can invoke Class Wizard to create a member variable for your class of type CVVRichEdit. You might also decide to capture the events in the control by adding Event handlers to your dialog class. To add Event handlers, you can use the Class Wizard just like adding notification message handlers for a non-speech edit box.

Capturing Speech

The **VVRichEdit** turns speech input into text; however, users of your application might not want every word they speak to be transformed into text. For example, users might need to perform other speaking tasks, such as answering the telephone or conversing with someone. By default, the **VVRichEdit** operates in what is known as modeless operation. This means that whenever the control has window focus, the control will perform speech to text conversion, in the same way that it will accept keyboard input when the window has focus. But, you can also control dictation explicitly by specifying a window (see “AutoDictationWindow (Read/Write at Run Time Only)” on page 100) and setting the state of dictation (see “DictationOn” on page 116) as needed to implement almost any dictation activation logic you choose.

For example, you could use your application's top-most window for the **AutoDictationWindow**, so that dictation would automatically be disabled when your application is not the active application, and use **VVPhrases** control (or you could leverage the **VVRichEdit** Commands property provided for extensibility) to add commands for "BEGIN-DICATION" and "STOP-DICTATION" which would control the state of the **DictationOn** property. This example implements what is known as "modal dictation".

When the control enters or exits dictation mode, it fires the **DictationStateChange** event.

One thing to keep in mind when working with the **VVRichEdit** control is that the **VVRichEdit** treats text from speech input the same as typed text. This means that properties like **MaxLength**, (which limits the number of characters the user can type into the TextBox) are still enforced the same way when the characters are generated from speech. Also, the control fires the **Change** event when the change occurs from spoken text just as it does when the change comes from typing.

Capturing Commands

In addition to dictated speech, the **VVRichEdit** can recognize spoken commands. By default, the **VVRichEdit** control listens for command speech when the control window has focus. If you specify an **AutoDictationWindow**, then that window will also be used for command activation tracking so that, by default, commands are always available when dictation is available. If you wish, you can achieve a finer granularity of control by explicitly setting the **CommandsEnabled** property in much the same way as using **DictationOn** to control dictation state.

The **VVRichEdit** is capable of understanding twenty command phrases. Refer to the following table for a complete listing.

Table 2. VVRichEdit Command Phrases

ID (Value)	Command Phrase	Description
vvTBCapitalizeThis (13)	“CAPITALIZE-THIS”	Capitalizes selected text or word at the cursor.
vvTBCopyThis (8)	“COPY-THIS”	Copies selected text to the clipboard.
vvTBCorrectThis (24)	“CORRECT-THIS”	Shows the error correction window.
vvTBCutThis (7)	“CUT-THIS”	Cuts selected text to the clipboard.
vvTBDeleteThis (10)	“DELETE-THIS”	Clears selected text or word at the cursor.
vvTBHideEC (6)	“HIDE-CORRECTION-WINDOW”	Hides the error correction window.
vvTBLowercaseThis (15)	“LOWERCASE-THIS”	Changes selected text or word at the cursor to lowercase.
vvTBMoveBeginning (22)	“MOVE-TO-BEGINNING-OF-DOCUMENT”	Moves the cursor to the beginning of the text.
vvTBMoveEnd (23)	“MOVE-TO-END-OF-DOCUMENT”	Moves the cursor to the end of the text.

Table 2. VVRichEdit Command Phrases

ID (Value)	Command Phrase	Description
vvTBNextWord (11)	“NEXT-WORD”	Places the cursor at the beginning of the next word.
vvTBPasteThis (9)	“PASTE-THIS”	Pastes text from the clipboard onto the VVRichEdit control.
vvTBPreviousWord (12)	“PREVIOUS-WORD”	Moves cursor to the beginning of the previous word.
vvTBScratchThat (18)	“SCRATCH-THAT”	Deletes the last dictated phrase.
vvTBSelectPhrase (27)	“SELECT-%S”	(%S represents any visible word or phrase) Selects the text specified.
vvTBSelectThis (20)	“SELECT-THIS”	Selects the text at the cursor.
vvTBShowEC (5)	“SHOW-CORRECTION-WINDOW”	Shows the error correction window.
vvTBUppercaseOff (17)	“UPPERCASE-OFF”	Removes the “Dictation Caps Lock” feature so all subsequent text is lowercase.
vvTBUppercaseOn (16)	“UPPERCASE-ON”	Enables the “Dictation Caps Lock” feature so all subsequent dictated text is uppercase.
vvTBUppercaseThis (14)	“UPPERCASE-THIS”	Changes selected text or word at the cursor to uppercase.

Whenever the user speaks one of the command phrases in the above list, the **VVRichEdit** control performs the corresponding action and fires the **Command** event indicating the ID of the command and the textual representation of the command. The **Commands** property can be used to extend or reduce this list of commands or provide a more “natural language” type of control for the **VVRichEdit**. The **Commands** property is an IVVPhraseColl interface. Internally the **VVRichEdit** uses the **VVPhrases** control and gets or sets the **Phrases** property on it. Any of the IVVPhraseColl interface methods can be used to modify or enable the collection of phrases, which the **VVRichEdit** uses to recognize commands.

The **VVRichEdit** will perform operations for events with the command IDs specified in VVRICHEDIT.H file. To add a new command, use the **Add** method on the IVVPhrasesColl and

specify a new command ID. **VVRichEdit** will not perform any specific functionality for the new command, but it will fire the **Command** event when it is spoken. In the **Command** event handler of the client application, add custom code for the command, which will be executed if the spoken command ID is the same as the ID of the new command. Since you can execute any methods on the **IVVPhrasesColl** and the **VVPhrases** contained within it, it is possible to disable or remove any of the standard commands you choose.

Text Correction

The user can use the ViaVoice **RichEdit** control's correction window to correct individual words that the engine has interpreted incorrectly. By default there are two ways for the user to invoke the correction window. If **CommandsEnabled** is true and the cursor is placed within the word to be corrected, the user can use the voice command "CORRECT-THIS" or "SHOW-CORRECTION-WINDOW". Or, the user can "right-click" the textbox and choose correction from the context menu. Either approach will display the Error Correction window (see Figure 12) with the highlighted word/phrase or the word at the cursor location. You may extend this by using the **Commands** method to add additional voice commands or by using the **ExecuteCommand** method for programmatic invocation.

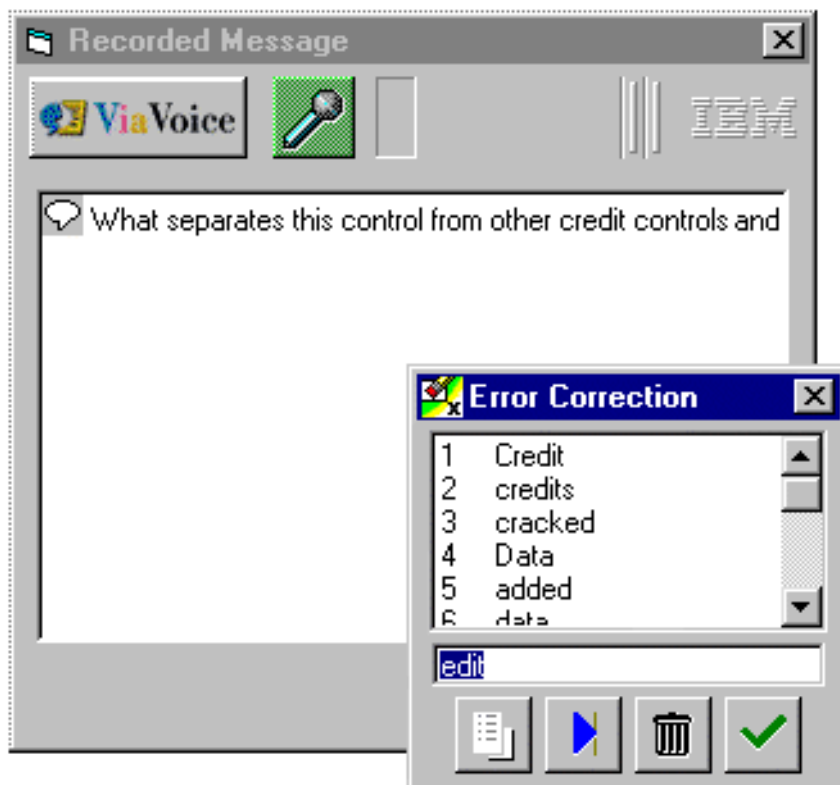


Figure 12. Error Correction Window

The error correction window shows a list of alternate words that the speech engine recognizes that are close in pronunciation to the word the user dictated. The user can choose a word from the list by clicking on it with the mouse and the word is replaced in the textbox control. Or, the user can edit the mis-spoken word in the edit field and click the checkmark button to replace the selected text in the **VVRichEdit** with whatever is in the edit field of the error correction window and apply the correction. After the user corrects the word, the **VVRichEdit** control updates the speech recognition engine, so that the next time the user speaks the same word or phrase, the engine can interpret it correctly.

Summary

At this point, you should know how to do the following:

- How to incorporate the **VVRichEdit** control into your project.
- How to control the activation of dictation.
- How to specify command words, and how to disable command capturing.
- How to correct dictated words using error correction.

The remainder of this documentation contains a reference for all the properties, methods, and events for the **VVRichEdit** control.

RichEdit Control Properties

The ViaVoice **RichEdit** control supports the following properties:

Appearance^a	LanguageUI	SelItalic
AutoDictationWindow	Locked^a	SelLength
AudioSourceType	MaxLength^a	SelProtected
AutoUI	MouseIcon^a	SelRightIndent^b
BackColor^a	MousePointer^a	SelRTF
BorderStyle^a	Multiline^a	SelStart
BulletIndentation^b	RightMargin^b	SelStrikeThru
Commands	ScrollBars^a	SelTabCount
CommandsEnabled	SelAlignment	SelTabs
DictationOn	SelBold	SelText^a
Enabled^a	SelBullet	SelUnderline
Engine	SelCharOffset	Text^a
FileName	SelColor	TextRTF
Font^a	SelFontName	
ForeColor^a	SelFontSize	
HideSelection^a	SelHangingIndent^b	
hWnd	SelIndent^b	

a. Represents standard properties in Visual Basic. For more information, refer to your Visual Basic documentation.

b. This property represents an amount of distance. It is entered as a string with the amount followed by one of the abbreviations for supported units. Inches can be represented as quotes ("), in, inch, or inches. Centimeters are specified by cm, point by pt, and pica by pi. If no units are specified, the property defaults to twips.

AutoDictationWindow (Read/Write at Run Time Only)

Controls the scope in which dictation is available.

Syntax

In Visual Basic:

```
[VVRichEditTextBox].AutoDictationWindow = [Valid "Top-Most" Window handle]
```

In Visual C++ (MFC):

```
[VVRichEdit].SetAutoDictationWindow (long);  
  
HWND = (HWND) [VVRichEdit].GetAutoDictationWindow();
```

In Visual C++:

```
HRESULT [pIVVRichEdit]->get_AutoDictationWindow( long* );  
  
HRESULT [pIVVRichEdit]->put_AutoDictationWindow( long );
```

Note:

[Valid “Top-Most” Window handle] can be replaced by `VV_HWND_AUTODICTATION` or `VV_HWND_ALL`, as indicated above.

Parameters

VV_HWND_AUTODICATION

(Default) Dictation is available only when the **VVRichEdit** control window has the focus.

Return Values

??

Remarks

The default value of this property is the distinguished constant, `VV_HWND_AUTODICTIONATION` (-1), which will map dictation availability to ViaVoice **RichEdit** window input focus^a. It can also be set to `VV_HWND_ALL` (0), which will enable dictation globally. However, please note that there can only be one global dictation object active (**DictationOn** is True) at any one time in the entire system (including other applications)! For this reason, it is strongly suggested that you avoid global dictation objects if at all possible. Alternatively, you can set this property to any valid "top-most"^b window handle, which maps dictation availability to that window's activation state (it or one of its children having focus).

Finer granularity of control can always be achieved by changing the state of **DictationOn** appropriately. When **DictationOn** is TRUE and the specified window is activated, the control will receive dictation. If **DictationOn** is FALSE, the control will not receive any dictation regardless of the value of this property.

If this property is set to `VV_HWND_ALL` and `DictationOn = TRUE`, the control will receive dictation. This enables an application to receive dictation when another application is active. Remember, if you use `VV_HWND_ALL`, be aware that there can only be one global dictation object active (**DictationOn** is True) at any one time. This includes your own *or any other application running on the system*. For this reason, global dictation objects should be used with extreme care and should probably be avoided unless absolutely necessary. Regardless of the value of this property, dictation will always be off if the **DictationOn** property is set to FALSE.

a. Dictation activation may potentially be modified by the state of the **DictationOn** property.

b. A "top-most" window is defined as any window without a parent. For more information on issues dealing with finding the "top-most" window, see the Microsoft Knowledge Base article Q84190.

Example

In Visual Basic:

```
Private Sub Form_Load()  
    'Assumes this form is the top-most form!  
    VVRichEdit1.AutoDictationWindow = hWnd  
End Sub
```

In Visual C++ (MFC):

```
void CTestDlg::InitializeRichEdit()  
{  
    // Makes no assumptions about m_hWnd  
    HWND hWnd = m_hWnd;  
    while ( GetParent ( hWnd ) != NULL )  
        hWnd = GetParent ( hWnd );  
    m_VVRichEdit.SetAutoDictationWindow((long) hWnd);  
}
```

In Visual C++:

```
void CTestDlg::OnInitDialog()  
{  
    // Makes no assumptions about m_hWnd  
    HWND hWnd = m_hWnd;  
    while ( GetParent ( hWnd ) != NULL )  
        hWnd = GetParent ( hWnd );  
  
    m_pIVVRichEdit->put_AutoDictationWindow((long) hWnd);  
}
```

See Also

“DictationOn” on page 116

“DictationStateChange” on page 193

AudioSourceType

AudioSourceType property to specify an alternative source containing spoken words and phrases.

This property only takes effect at design time.

This property takes one of the following enum values:

vvFixedAudio

- default values.

vvMultimediaDevice

- the default multimedia device on the system is used.

vvCustomCLSID

- Custom CLSID of a creatable object in a registered COM server which implements the SAPI custom audio source specification.

vvWAVFile

- a wave file.

NOTE: This is a new property, which could break some legacy client applications.

If this happens, recompile the code after installation of SDK.

Syntax

In Visual Basic:

```
[enum] = [VVRichEdit].AudioSourceType  
[VVRichEdit].AudioSourceType = [enum]
```

In Visual C++ (MFC):

```
SetAudioSourceType(VVAudioSourceConstants *pVal)  
SetAudioSourceType(VVAudioSourceConstants newVal)
```

In Visual C++:

```
get_AudioSourceType(VVAudioSourceConstants *pVal)  
put_AudioSourceType(VVAudioSourceConstants newVal)
```

Parameters

None.

Return Values

None.

Remarks

None.

Example

In Visual Basic:

```
VVRichEdit1.Engine.AudioSource="c:\\temp\\my.wav"  
VVRichEdit1.Engine.AudioState=vvasStarted
```

In Visual C++ (MFC):

```
void CTestRichEditDlg::WAVinput()  
{  
    pVVRichEdit->GetEngine()->SetAudioSource(L"c:\\temp\\my.wav");  
    pVVRichEdit->GetEngine()->SetAudioState(vvasStarted);  
}
```

In Visual C++:

```
void CTestRichEditDlg::WAVinput()  
{  
    IVVEngine * pVVEngine=NULL;  
    m_pvvRichEdit->get_Engine(&pVVEngine);  
    pVVEngine->put_AudioSource(L"c:\\temp\\my.wav");  
    pVVEngine->put_AudioState(vvasStarted);  
}
```

See Also

None.

AutoUI

Controls whether **VVRichEdit** displays the **User Interface Server** and interacts with it automatically.

Syntax

In Visual Basic:

```
VVRichEdit.AutoUI = [Boolean]
```

In Visual C++ (MFC):

```
BOOL GetAutoUI();  
void SetAutoUI(BOOL fNewValue);
```

In Visual C++:

```
get_AutoUI(VARIANT_BOOL *pVal)  
put_AutoUI(VARIANT_BOOL newVal)
```

Parameters

fnewValue
??

Return Values

TRUE

(Default) **VVRichEdit** displays the **User Interface Server** and interacts with it automatically.

FALSE

VVRichEdit does not display the **User Interface Server**. Also, it does not interact automatically with it if another control displays the **User Interface Server**

Remarks

When **AutoUI** is True, the **VVRichEdit** automatically updates the following components: Microphone, Word History, and Volume Level.

When this property is set to True, the ViaVoice **RichEdit** control automatically displays and interacts with the ViaVoice **User Interface Server**. If multiple instances of the **VVRichEdit** control have **AutoUI** set to True, the **User Interface Server** only gets created once, and all the instances of the control interact with the same **User Interface Server**. If you prefer not to display the **User Interface Server** or you do not want to have the **VVRichEdit** control interact with it automatically, set this property to False. By setting it to True, you will not be able to interact with the **UIServer** on behalf of your application. This means that even if you create an instance of the **UIClient**, you will never be able to control the state of the **UIServer**.

Example

In Visual Basic:

```
Private Sub Form_Load()  
    VVRichEdit1.AutoUI = True  
End Sub
```

In Visual C++ (MFC):

```
m_RichEdit.SetAutoUI(TRUE);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_AutoUI(VARIANT_TRUE);
```

See Also

Refer to the following chapters for more information about the ViaVoice **User Interface Control**:

Chapter 25, “Introduction to the User Interface Control” on page 497

Chapter 26, “Getting Started with the User Interface Control” on page 499

Chapter 27, “Classes, Structures, and Enumerations” on page 533

Chapter 28, “Properties, Methods, and Events” on page 561

Chapter 29, “User Interface Control Frequently Asked Questions” on page 629

BulletIndentation

Sets or gets the amount of space to indent.

Syntax

In Visual Basic:

```
VVRichEdit.BulletIndentation = [String]
```

In Visual C++ (MFC):

```
CString GetBulletIndentation();  
void SetBulletIndentation(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_BulletIndentation(BSTR * pVal);  
HRESULT put_BulletIndentation(BSTR newVal);
```

Parameters

??

Return Values

TRUE

Sets or gets the amount of space to indent.

FALSE

??

Remarks

This property represents an amount of distance. It is entered as a string with the amount followed by one of the abbreviations for supported units. Inches can be represented as quotes ("), in, inch, or

inches. Centimeters are specified by cm, point by pt, and pica by pi. If no units are specified, the property defaults to twips.

Example

In Visual Basic:

```
VVRichEdit1.BulletIndentation = "0.5in"
```

In Visual C++ (MFC):

```
m_RichEdit.SetBulletIndentation( _T("0.5in") );
```

In Visual C++:

```
BSTR Indent = SysAllocString ( OLESTR("0.5in"));  
HRESULT hr = m_pIVVRichEdit->put_BulletIndentation(Indent);  
SysFreeString(Indent);
```

See Also

None.

Commands

Gets or sets the IVVPhraseColl interface used by the internal VVPhrases object, used by **VVRichEdit** to recognize voice commands.

Syntax

In Visual Basic:

```
IVVPhraseColl = [VVRichEdit].Commands
```

In Visual C++ (MFC):

```
CVVPhraseColl GetCommands();  
void SetRefCommands(LPDISPATCH newValue);
```

In Visual C++:

```
HRESULT get_Commands(IVVPhraseColl ** pVal);  
HRESULT put_Commands(IVVPhraseColl * pVal);
```

Parameters

??

Return Values

??

Remarks

The RichEdit can recognize command words while dictation is off or on as long as speech input is available. Finer granularity of control for command availability is always available through use of the CommandsEnabled property. Notice the distinction between the requested state and availability, which is dependent on the **AutoDictationWindow**.

Also, the **Commands** property is actually the **Phrases** property of an implicitly created VVPhrases control.

If you wish to change the phrase used to invoke a given command, you may simply change the required command phrase as indicated in the **VVPhrases** documentation. If you wish to add additional commands for existing functionality, you must use the correct ID for the desired functionality. For instance, if you wish to say either “NEXT-WORD” or “MOVE-NEXT” to move the cursor to the next word, then you would jsut add “MOVE-NEXT” as a new phrase with the ID of VVTBNextWord. Then, when the user speaks either command, the cursor will move to the next word.

Example

In Visual Basic:

```
Set VVPhrases1.Phrases = VVRichEdit1.Commands
AddCustomCommands ( VVPhrases1 )
Set VVRichEdit2.Commands = VVPhrases1.Phrases
```

In Visual C++ (MFC):

```
CVVPhraseColl Commands;
Commands = m_VVRichEdit1.GetCommands ( );
AddCustomCommands ( Commands );
m_VVRichEdit2.SetRefCommands ( Commands );
```

In Visual C++:

```
IVVPhraseColl* pIVVPhraseColl;
HRESULT hr = m_pIVVRichEdit->get_Commands(&pIVVPhraseColl);
AddCustomCommands(pIVVPhraseColl);
hr = m_pIVVRichEdit->putref_Commands(pIVVPhraseColl);
```

See Also

For more information on **VVPhrases** see Chapter 9, “Introduction to the Phrases Control” on page 203.

“AutoDictationWindow (Read/Write at Run Time Only)” on page 100

“DictationOn” on page 116

“CommandsEnabled” on page 114

“VVPhraseColl Collection” on page 245

CommandsEnabled

Returns or sets whether the ViaVoice **RichEdit** control will recognize commands or not, when available based on the **AutoDictationWindow**.

Syntax

In Visual Basic:

```
VVRichEdit.CommandsEnabled = [Boolean]
```

In Visual C++ (MFC):

```
BOOL GetCommandsEnabled();  
void SetCommandsEnabled(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_CommandsEnabled(VARIANT_BOOL * pVal)  
HRESULT put_CommandsEnabled(VARIANT_BOOL newVal)
```

Parameters

fNewValue
??

Return Values

TRUE

(Default) The ViaVoice **RichEdit** recognizes commands, when available.

FALSE

Command words are ignored.

Remarks

The property can be set at design time or runtime, although commands will never be enabled until runtime. The **RichEdit** can recognize command words while dictation is off or on as long as speech input is available. Command recognition will not be available if dictation is not available. Enabling or disabling an individual command can be done through the **IVVPhraseColl** returned by the **Commands** property. An individual phrase can be accessed through the collection and enabled or disabled.

Example

In Visual Basic:

```
Private Sub Form_Load()  
    VVRichEdit1.CommandsEnabled = True  
End Sub
```

In Visual C++ (MFC):

```
m_RichEdit.SetCommandsEnabled(TRUE);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_CommandsEnabled(VARIANT_TRUE);
```

See Also

“AutoDictationWindow (Read/Write at Run Time Only)” on page 100

“DictationOn” on page 116

“VVPhraseColl Collection” on page 245

DictationOn

Returns or sets the desired state of the dictation mode.

Syntax

In Visual Basic:

```
VVRichEdit.DictationOn = [Boolean]
```

In Visual C++ (MFC):

```
BOOL GetDictationOn();  
void SetDictationOn(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_DictationOn(VARIANT_BOOL * pVal);  
HRESULT put_DictationOn(VARIANT_BOOL newVal)
```

Parameters

fNewValue
??

Return Values

TRUE

The control can receive dictation input when dictation is available.

FALSE

The control ignores dictation input

Remarks

If this property is set to TRUE, it indicates that the programmer wants dictation to be on. What this means is that if dictation is available (i.e. nothing is preventing dictation), then the user will be able to

dictate into the control. Some possible reasons why dictation would be unavailable when **DictationOn** is True are: **MaxLength** has been reached, **Locked** is True, **Enabled** is False, or the semantics of the **AutoDictationWindow** indicate that dictation is not available. If DictationOn = TRUE and these conditions are alleviated, the control will automatically begin receiving dictation. If the property is set to FALSE, the user will never be able to dictate into the control. When dictation is off (DictationOn = False) the user may still be able to issue commands.

When the state of the dictation mode changes, the control fires the **DictationStateChange** event. You should not set the value of this property in the **DictationStateChange** event, as this will cause the event to trigger again.

Example

In Visual Basic:

```
Private Sub Form_Load()  
    VVRichEdit1.DictationOn = True  
End Sub
```

In Visual C++ (MFC):

```
m_VVRichEdit.SetDictationOn(TRUE);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_DictationOn(VARIANT_TRUE);
```

See Also

“AutoDictationWindow (Read/Write at Run Time Only)” on page 100

“CommandsEnabled” on page 114

“DictationStateChange” on page 193

“Enabled” in Visual Basic Documentation

“Locked” in Visual Basic Documentation

“MaxLength” in Visual Basic Documentation

Engine

Contains a reference to the ViaVoice **Engine** control (**VVEngine**), which is used by the **VVRichEdit** control.

Syntax

In Visual Basic:

```
VVRichEdit.Engine
```

In Visual C++ (MFC):

```
CVVEngine GetEngine();  
void SetRefEngine(LPDISPATCH newValue);
```

In Visual C++:

```
HRESULT get_Engine(IVVEngine * * pVal);  
HRESULT putref_Engine(IVVEngine * pVal);
```

Parameters

??

Return Values

??

Remarks

The **Engine** property is actually holding an implicitly created ActiveX control (**VVEngine**), which can also be created separately. Inserting a **VVEngine** control in a project enables you to set the engine properties on this control, and then assign the resulting engine to multiple ViaVoice ActiveX controls.

Example

In Visual Basic:

```
VVRichEdit1.Engine.AudioSourceType = vvstSAPICompliant
```

In Visual C++ (MFC):

```
m_VVRichEdit.GetEngine( ).SetAudioSourceType(vvstSAPICompliant);
```

In Visual C++:

```
IVVEngine* pIVVEngine = NULL;  
HRESULT hr = m_pIVVRichEdit->get_Engine ( & pIVVEngine );  
if ( SUCCEEDED ( hr ) )  
    hr = pIVVEngine->put_AudioSourceType ( vvstSAPICompliant );
```

See Also

Refer to the Engine Control Guide on the SDK “Documentation” menu for more information. For more information, see the Engine Control Guide.

FileName

Loads a file into the **VVRichEdit** when the control is created.

Syntax

In Visual Basic:

```
VVRichEdit.FileName = [String]
```

In Visual C++ (MFC):

```
CString GetFileName();  
void SetFileName(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_FileName(BSTR * pVal)  
HRESULT put_FileName(BSTR newVal);
```

Parameters

??

Return Values

??

Remarks

If the file extension is ".rtf", then it is loaded as a RTF file. Otherwise it is loaded as a text file. If the filename is not valid at run time, the control will not contain any text.

Example

In Visual Basic:

```
VVRichEdit1.filename = "test.rtf"
```

In Visual C++ (MFC):

```
m_RichEdit.SetFileName(_T("test.rtf"));
```

In Visual C++:

```
BSTR File = SysAllocString ( OLESTR ( "test.rtf" ) );  
HRESULT hr = m_pIVVRichEdit->put_FileName( File );  
SysFreeString ( File );
```

See Also

None.

hWnd (Read Only)

Sets or gets window handle to the Windows rich edit common control used by the **VVRichEdit** control.

Syntax

In Visual Basic:

```
VVRichEdit(hWnd)
```

In Visual C++ (MFC):

```
long GetHWnd();
```

In Visual C++:

```
HRESULT get_hWnd(long * pVal);
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
REWindowHandle = m_RichEdit.hWnd
```

In Visual C++ (MFC):

```
HWND hWnd = m_RichEdit.GetHhWnd();
```

In Visual C++:

```
HWND hWnd = 0;  
HRESULT hr = m_pIVVRichEdit->get_HWnd( & (long)hWnd );
```

See Also

None.

LanguageUI

Sets or gets the language used by the **VVRichEdit** for this specific client.

Syntax

In Visual Basic:

```
[VVRichEdit].LanguageUI = [String]
```

In Visual C++ (MFC):

```
CString GetLanguageUI();  
void SetLanguageUI(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_LanguageUI(BSTR * pVal);  
HRESULT put_LanguageUI(BSTR newVal);
```

Parameters

?? The **LanguageUI** property settings for a **VVRichEdit** control are:

Language	Property Value
U.S. English	“EN_US”
U.K. English	“EN_UK”
German	“GR_GR”
Italian	“IT_IT”
Spanish	“ES_ES”
French	“FR_FR”
Japanese	“JA_JP”

Return Values

??

Remarks

The language affects any dialogs, menus, strings or ToolTips displayed by the control.

Example

In Visual Basic:

```
' Sets UI language to U.S. English
VVRichEdit1.LanguageUI = "EN_US"
' Gets UI language and displays it in a message box
MsgBox VVRichEdit1.LanguageUI
```

In Visual C++ (MFC):

```
// Sets UI language to U.S. English
m_VVRichEdit.SetLanguageUI("EN_US");
CString sLangUI;
// Gets UI language and copies it into variable
sLangUI = m_VVRichEdit.GetLanguageUI();
```

In Visual C++:

```
HRESULT hr;
BSTR bstrLangUI;

bstrLangUI = SysAllocString(OLESTR("EN_US"));
// Sets UI language to U.S. English
hr = pIVVRichEdit->put_LanguageUI(bstrLangUI);
SysFreeString(bstrLangUI);

// Gets UI language into BSTR variable
hr = pIVVRichEdit->get_LanguageUI(&bstrLangUI);
// Use language string now and when done free BSTR.
SysFreeString(bstrLangUI);
```

See Also

None.

RightMargin

(Not Yet Implemented) Sets or gets the amount the text is indented from the right margin.

Syntax

In Visual Basic:

```
VVRichEdit.RightMargin = [String]
```

In Visual C++ (MFC):

```
CString GetRightMargin();  
void SetRightMargin(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_RightMargin(BSTR * pVal);  
HRESULT put_RightMargin(BSTR newVal);
```

Parameters

??

Return Values

??

Remarks

This property represents an amount of distance. It is entered as a string with the amount followed by one of the abbreviations for supported units. Inches can be represented as quotes (“), in, inch, or inches. Centimeters are specified by cm, point by pt, and pica by pi. If no units are specified, the property defaults to twips.

Example

In Visual Basic:

```
VVRichEdit1.RightMargin = "0.5in"
```

In Visual C++ (MFC):

```
m_RichEdit.SetRightMargin(_T("0.5in");
```

See Also

None.

SelAlignment (Read/Write at Run Time Only)

Sets or gets the alignment of the selected text.

Syntax

In Visual Basic:

```
VVRichEdit.SelAlignment = [Integer]
```

In Visual C++ (MFC):

```
long GetSelAlignment();  
void SetSelAlignment(long nNewValue);
```

In Visual C++:

```
HRESULT get_SelAlignment(enumParaAlignmentType * pVal);  
HRESULT put_SelAlignment(enumParaAlignmentType newVal);
```

Parameters

?? AlignLeft = 0

Aligns the selected text flush with the left margin.

AlignRight = 1

Aligns the selected text flush with the right margin.

AlignCenter = 2

Centers the selected text.

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
Private Sub CenterText_Click()  
    VVRichEdit1.SelAlignment(AlignCenter)  
End Sub
```

In Visual C++ (MFC):

```
m_RichEdit.SetSelAlignment(2);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelAlignment( AlignCenter );
```

See Also

None.

SelBold (Read/Write at Run Time Only)

Sets or gets the boldness of the selected text.

Syntax

In Visual Basic:

```
VVRichEdit.SelBold = [Boolean]
```

In Visual C++ (MFC):

```
BOOL GetSelBold();  
void SetSelBold(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_SelBold(VARIANT_BOOL * pVal);  
HRESULT put_SelBold(VARIANT_BOOL newVal);
```

Parameters

fNewValue
??

Return Values

TRUE

The selected text is displayed in bold face.

FALSE

The selected text is displayed without bold face.

Remarks

None.

Example

In Visual Basic:

```
'cause the selected text to be displayed in bold face  
VVRichEdit1.SelBold = True
```

In Visual C++ (MFC):

```
m_RichEdit.SetSelBold(TRUE);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelBold ( VARIANT_TRUE );
```

See Also

None.

SelBullet

Sets or gets whether the selected text has a bullet.

Syntax

In Visual Basic:

```
VVRichEdit.SelBullet = [Bool]
```

In Visual C++ (MFC):

```
BOOL GetSelBullet();  
void SetSelBullet(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_SelBullet(VARIANT_BOOL * pVal);  
HRESULT put_SelBullet(VARIANT_BOOL newVal);
```

Parameters

fNewValue
??

Return Values

TRUE

The selected text is displayed with a bullet and indented by the amount of the **BulletIndentation** property.

FALSE

The selected text is not displayed with a bullet.

Remarks

None.

Example

In Visual Basic:

```
'Toggle whether the control has a bullet or not  
VVRichEdit1.SelBullet = Not VVRichEdit1.SelBullet
```

In Visual C++ (MFC):

```
m_RichEdit.SetSelBullet(TRUE);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelBullet( VARIANT_TRUE );
```

See Also

None.

SelCharOffset (Read/Write at Run Time Only)

Sets or gets the offset of the selected character from the left margin in twips.

Syntax

In Visual Basic:

```
VVRichEdit.SelCharOffset = [Integer]
```

In Visual C++ (MFC):

```
long GetSelCharOffset();  
void SetSelCharOffset(long nNewValue);
```

In Visual C++:

```
HRESULT get_SelCharOffset(long * pVal);  
HRESULT put_SelCharOffset(long newVal);
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
'offset the selected characters one inch from the left margin  
VVRichEdit1.SelCharOffset = 720
```

In Visual C++ (MFC):

```
// offset the selected characters one inch from the left margin  
m_RichEdit.SetSelCharOffset( = 720);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelCharOffset(720);
```

See Also

None.

SelColor (Read/Write at Run Time Only)

Sets or gets the color of the selected text.

Syntax

In Visual Basic:

```
VVRichEdit.SelColor = [Integer]
```

In Visual C++ (MFC):

```
unsigned long GetSelColor();  
void SetSelColor(unsigned long newValue);
```

In Visual C++:

```
HRESULT get_SelColor(OLE_COLOR * pVal);  
HRESULT put_SelColor(OLE_COLOR newVal);
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
Private Sub ChangeSelColor_Click()  
    CommonDialog1.ShowColor  
    VVRichEdit.SelColor = CommonDialog1.Color  
End Sub
```

In Visual C++ (MFC):

```
OLE_COLOR color = GetColor();  
m_RichEdit.SetSelColor((long)color);
```

In Visual C++:

```
OLE_COLOR color = GetColor();  
HRESULT hr = m_pIVVRichEdit->put_SelColor ( color );
```

See Also

None.

SelfFontName(Read/Write at Run Time Only)

Sets or gets the font name of the selected text.

Syntax

In Visual Basic:

```
VVRichEdit.SelfFontName = [String]
```

In Visual C++ (MFC):

```
CString GetSelfFontName();  
void SetSelfFontName(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_SelfFontName(BSTR* pVal);  
HRESULT put_SelfFontName(BSTR newVal);
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
'change the selected font to arial  
m_RichEdit.SelFontName = "Arial"
```

In Visual C++ (MFC):

```
// change the selected font to arial  
m_RichEdit.SetSelFontName("Arial" strFontName);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelFontName("Arial" );
```

See Also

None.

SelfFontSize (Read/Write at Run Time Only)

Sets or gets the size of the font of the selected text.

Syntax

In Visual Basic:

```
VVRichEdit.SelFontSize = [Integer]
```

In Visual C++ (MFC):

```
long GetSelFontSize();  
void SetSelFontSize(long nNewValue);
```

In Visual C++:

```
HRESULT get_SelFontSize(long * pVal);  
HRESULT put_SelFontSize(long newVal);
```

Parameters

Return Values

Remarks

None.

Example

In Visual Basic:

```
VVRichEdit1.SelFontSize = 12;
```

In Visual C++ (MFC):

```
m_RichEdit.SetSelFontSize(12);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelFontSize(12);
```

See Also

None.

SelHangingIndent (Read/Write at Run Time Only)

Sets or gets the amount to indent subsequent lines in a paragraph from the left edge of the first line in the paragraph.

Syntax

In Visual Basic:

```
VVRichEdit.SelHangingIndent = [Integer]
```

In Visual C++ (MFC):

```
CString GetSelHangingIndent();  
void SetSelHangingIndent(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_SelHangingIndent(BSTR * pVal);  
HRESULT put_SelHangingIndent(BSTR newVal);
```

Parameters

??

Return Values

??

Remarks

This property represents an amount of distance. It is entered as a string with the amount followed by one of the abbreviations for supported units. Inches can be represented as quotes ("), in, inch, or inches. Centimeters are specified by cm, point by pt, and pica by pi. If no units are specified, the property defaults to twips.

Example

In Visual Basic:

```
Private Sub SetSelHangingIndent()  
    'Indent the subsequent lines in a paragraph a half inch from  
    'from the first line  
    VVRichEdit1.SelHangingIndent = "0.5in"  
End Sub
```

In Visual C++ (MFC):

```
VVRichEdit1.SetSelHangingIndent(_T("0.5in"strIndent));
```

In Visual C++:

```
BSTR Indent = SysAllocSting ( OLESTR ("0.5in") );  
HRESULT hr = m_pIVVRichEdit->put_SelHangingIndent(Indent);  
SysFreeString(Indent);
```

See Also

None.

SelIndent (Read/Write at Run Time Only)

Sets or gets the paragraph indentation of the selected text.

Syntax

In Visual Basic:

```
VVRichEdit.SelIndent = [String]
```

In Visual C++ (MFC):

```
CString GetSelIndent();  
void SetSelIndent(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_SelIndent(BSTR * pVal);  
HRESULT put_SelIndent(BSTR newVal);
```

Parameters

??

Return Values

??

Remarks

This property represents an amount of distance. It is entered as a string with the amount followed by one of the abbreviations for supported units. Inches can be represented as quotes (“), in, inch, or inches. Centimeters are specified by cm, point by pt, and pica by pi. If no units are specified, the property defaults to twips.

Example

In Visual Basic:

```
'set the indentation of the current text to one inch  
VVRichEdit1.SelIndent = "1in"1
```

In Visual C++ (MFC):

```
SetSelIndent(_T("1in"));
```

In Visual C++:

```
BSTR Indent = SysAllocString ( OLESTR("1in"));  
HRESULT hr = put_SelIndent(Indent);  
SysFreeString(Indent);
```

See Also

None.

SelItalic (Read/Write at Run Time Only)

Sets or gets the italicized nature of the selected text.

Syntax

In Visual Basic:

```
VVRichEdit.SelItalic = [Bool]
```

In Visual C++ (MFC):

```
BOOL GetSelItalic();  
void SetSelItalic(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_SelItalic(VARIANT_BOOL * pVal);  
HRESULT put_SelItalic(VARIANT_BOOL newVal);
```

Parameters

fNewValue
??

Return Values

TRUE

The selected text is italicized.

FALSE

The selected text is not italicized.

Remarks

None.

Example

In Visual Basic:

```
'Turn on the Italics for the selected text  
VVRichEdit1.SelItalic=True
```

In Visual C++ (MFC):

```
// Turn off the Italics for the selected text  
m_RichEdit.SetSelItalic(TRUE);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelItalic(VARIANT_TRUE);
```

See Also

None.

SelLength (Read/Write at Run Time Only)

Sets or gets the number of characters that are selected.

Syntax

In Visual Basic:

```
VVRichEdit.SelLength = [Integer]
```

In Visual C++ (MFC):

```
long GetSelLength();  
void SetSelLength(long nNewValue);
```

In Visual C++:

```
HRESULT get_SelLength(long * pVal);  
HRESULT put_SelLength(long newVal);
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
'Display a message box with the number of characters currently selected  
MsgBox "Number of Characters Selected = " + CStr(VVRichEdit1.SelLength)
```

In Visual C++ (MFC):

```
// Clear the selection  
m_RichEdit.SetSelLength(0);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelLength(0);
```

See Also

None.

SelProtected

(Not Yet Implemented) Sets or gets the value indicating whether the selected text is protected from editing.

Syntax

In Visual Basic:

```
VVRichEdit.SelProtected = [Bool]
```

In Visual C++ (MFC):

```
BOOL GetSelProtected();  
void SetSelProtected(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_SelProtected(VARIANT_BOOL * pVal);  
HRESULT put_SelProtected(VARIANT_BOOL newVal);
```

Parameters

fNewValue
??

Return Values

TRUE

The selected text is protected.

FALSE

The selected text is not protected.

Remarks

None.

Example

In Visual Basic:

```
'Protect the selected text  
VVRichEdit1.SelProtected = True
```

In Visual C++ (MFC):

```
CtestDlg::OnToggleProtectText()  
m_RichEdit.SetSelProtected(True);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelProtected(VARIANT_TRUE);
```

See Also

None.

SelRightIndent (Read/Write at Run Time Only)

Sets or gets the value indicating whether the selected text is protected from editing.

Syntax

In Visual Basic:

```
VVRichEdit.SelRightIndent = [String]
```

In Visual C++ (MFC):

```
CString GetSelRightIndent();  
void SetSelRightIndent(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_SelRightIndent( BSTR * pVal );  
HRESULT put_SelRightIndent( BSTR newVal );
```

Parameters

??

Return Values

??

Remarks

This property represents an amount of distance. It is entered as a string with the amount followed by one of the abbreviations for supported units. Inches can be represented as quotes ("), in, inch, or inches. Centimeters are specified by cm, point by pt, and pica by pi. If no units are specified, the property defaults to twips.

Example

In Visual Basic:

```
'Indent the selected text one inch from the right margin  
VVRichEdit1.SelRightIndent = "1in"
```

In Visual C++ (MFC):

```
m_RichEdit.SetSelRightIndent("1in"strRightIndent);
```

In Visual C++:

```
BSTR Indent = SysAllocString ( OLESTR ("1in") );  
HRESULT hr = m_pIVVRichEdit->put_SelRightIndent(Indent);  
SysFreeString(Indent);
```

See Also

None.

SelRTF (Read/Write at Run Time Only)

Sets or gets the selected text in RTF format.

Syntax

In Visual Basic:

```
VVRichEdit.SelRTF = [String]
```

In Visual C++ (MFC):

```
CString GetSelRTF();  
void SetSelRTF(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_SelRTF( BSTR * pVal );  
HRESULT put_SelRTF( BSTR newVal );
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
m_RichEdit.SetSelRTF(_Ts(""));
```

In Visual C++ (MFC):

```
m_RichEdit.SetSelRTF(_Ts(""));
```

In Visual C++:

```
BSTR Empty = SysAllocString(OLESTR(""));
HRESULT hr = m_pIVVRichEdit->put_SelRTF(Empty);
SysFreeString(Empty);
```

See Also

None.

SelStart (Read/Write at Run Time Only)

Sets or gets the index in the control of the start of the selection.

Syntax

In Visual Basic:

```
VVRichEdit.SelStart = [Integer]
```

In Visual C++ (MFC):

```
long GetSelStart();  
void SetSelStart(long nNewValue);
```

In Visual C++:

```
HRESULT get_SelStart ( long * pVal );  
HRESULT put_SelStart ( long newVal );
```

Parameters

??

Return Values

??

Remarks

For more information about the **MaxLength** property, refer to the Visual Basic documentation.

Example

In Visual Basic:

```
Private Sub Form_Load()  
    'set the start of the selected text to the first character in the  
    'control  
    VVRichEdit1.SelStart = 0  
End Sub
```

In Visual C++ (MFC):

```
m_RichEdit.SetSelStart(0);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelStart(0);
```

See Also

None.

SelStrikeThru (Read/Write at Run Time Only)

Sets or gets a value indicating whether the selected text is displayed with a line crossing through it.

Syntax

In Visual Basic:

```
VVRichEdit.SelStrikeThru = [Bool]
```

In Visual C++ (MFC):

```
BOOL GetSelStrikeThru();  
void SetSelStrikeThru(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_SelStrikeThru(VARIANT_BOOL * pVal);  
HRESULT put_SelStrikeThru(VARIANT_BOOL newVal);
```

Parameters

fNewValue
??

Return Values

TRUE

The selected text is displayed with a line through it.

FALSE

The selected text is displayed without a line through it.

Remarks

None.

Example

In Visual Basic:

```
Private Sub Form_Load
    VVRichEdit1.SelStrikeThru = False
End Sub
```

In Visual C++ (MFC):

```
m_RichEdit.SetSelStrikeThru(FALSE);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelStrikeThru( VARIANT_FALSE );
```

See Also

None.

SelTabCount (Read/Write at Run Time Only)

Sets or gets the number of tabs in the control.

Syntax

In Visual Basic:

```
VVRichEdit.SelTabCount = [Integer]
```

In Visual C++ (MFC):

```
short GetSelTabCount();  
void SetSelTabCount(short nNewValue);
```

In Visual C++:

```
HRESULT get_SelTabCount(short * pVal);  
HRESULT put_SelTabCount(short newVal);
```

Parameters

??

Return Values

??

Remarks

This function is used in conjunction with the **SelTabs** property. Specifying an index higher than the SelTabCount property will return an error from the control.

Example

In Visual Basic:

```
Private Sub Form_Load
    ' Set 3 TabStops at 0.5"intervals
    VVRichEdit1.SelTabCount = 3
    For I = 0 To VVRichEdit1.SelTabCount - 1
        VVRichEdit1.SelTabs(I) = 720 * I
    Next I
End Sub
```

In Visual C++ (MFC):

```
m_RichEdit.SetSelTabCount(3);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelTabCount(3);
```

See Also

None.

SelTabs (Read/Write at Run Time Only)

Sets or gets the position of a specified tabstop in twips.

Syntax

In Visual Basic:

```
VVRichEdit.SelTabs([Integer]) = [Integer]
```

In Visual C++ (MFC):

```
long GetSelTabs(short sIndex);  
void SetSelTabs(short sIndex, long nNewValue);
```

In Visual C++:

```
HRESULT get_SelTabs(short sIndex, long * pVal);  
HRESULT put_SelTabs(short sIndex, long newVal);
```

Parameters

??

Return Values

??

Remarks

When specifying 0 for the index, the first tabstop is retrieved. This is a zero-based index.

Example

In Visual Basic:

```
Private Sub Form_Load()  
    'Set the third tab to one half inch past second tab  
    VVRichEdit1.SelTabs(2) = 720  
End Sub
```

In Visual C++ (MFC):

```
m_RichEdit.SetSelTabs(2, 720);
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelTabs(2, 720);
```

See Also

None.

SelText (Read/Write at Run Time Only)

Sets or gets the selected text without RTF formatting codes.

Syntax

In Visual Basic:

```
VVRichEdit.SelText = [String]
```

In Visual C++ (MFC):

```
CString GetSelText();  
void SetSelText(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_SelText( BSTR * pVal );  
HRESULT put_SetText( BSTR newVal );
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
'replace the selected text with a string  
VVRichEdit1.SelText = "Test String"
```

In Visual C++ (MFC):

```
m_RichEdit.SetSelText( _T("") strText);
```

In Visual C++:

```
BSTR Empty = SysAllocString(OLESTR(""));  
HRESULT hr = m_pIVVRichEdit->put_SelText(Empty);  
SysFreeString(Empty);
```

See Also

None.

SelUnderline (Read/Write at Run Time Only)

Sets or gets the value indicating whether the selected text is underlined or not.

Syntax

In Visual Basic:

```
VVRichEdit.SelUnderline = [Bool]
```

In Visual C++ (MFC):

```
BOOL GetSelUnderline();  
void SetSelUnderline(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_SelUnderline(VARIANT_BOOL * pVal);  
HRESULT put_SelUnderline(VARIANT_BOOL newVal);
```

Parameters

fNewValue
??

Return Values

TRUE

The selected text is underlined

FALSE

The selected text is not underlined

Remarks

None.

Example

In Visual Basic:

```
'Turn on the Underline for the selected text  
VVRichEdit1.SelUnderline=True
```

In Visual C++ (MFC):

```
// Turn off the Underline for the selected text  
CtestDlg::OnUnderlineOff()  
{  
    m_RichEdit.SetSelUnderline(TRUE);  
}
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->put_SelUnderline(VARIANT_TRUE);
```

See Also

None.

TextRTF

Sets or gets text for the entire control in RTF format.

Syntax

In Visual Basic:

```
VVRichEdit.TextRTF = [String]
```

In Visual C++ (MFC):

```
CString GetTextRTF();  
void SetTextRTF(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT get_TextRTF( BSTR * pVal );  
HRESULT put_TextRTF( BSTR newVal );
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
VVRichEdit1.TextRTF = ""
```

In Visual C++ (MFC):

```
m_RichEdit.SetTextRTF(_T(""));
```

In Visual C++:

```
BSTR Empty = SysAllocString ( OLESTR("") );  
HRESULT hr = m_pIVVRichEdit->put_TextRTF(Empty);  
SysFreeString ( Empty );
```

See Also

None.

RichEdit Control Methods

The ViaVoice **RichEdit** control supports the following methods:

- **About^a**
- **ExecuteCommand**
- **LoadRTF**
- **LoadTextFile**
- **Playback**
- **PlaybackEx**
- **PlaybackEx2**
- **Refresh^a**
- **SaveFile**
- **SaveRTF**
- **SelPrint**

a. Represents a standard method in Visual Basic. For more information, refer to your Visual Basic documentation.

ExecuteCommand

Allows the client to invoke any of the “voice commands” (see ”Commands” property) programmatically simply by passing the ID of the command desired.

Syntax

In Visual Basic:

```
Call VVRichEdit.ExecuteCommand ( vvTBCCommand )
```

In Visual C++ (MFC):

```
void ExecuteCommand(long lCommandID);
```

In Visual C++:

```
HRESULT ExecuteCommand(long lCommandID);
```

Parameters

??

Return Values

??

Remarks

The primary use of this functionality is to expose voice command functionality for invocation via mouse or keyboard input, although it can also be used to control actions based on voice commands in an external **VVPhrases** control. This might be useful, for instance, if you need to have voice commands activated/available based on your own logic using a different tracking window for the command phrases.

The **CommandsEnabled** property has no effect on commands invoked through the **ExecuteCommand** method. In this version of the SDK, the **ExecuteCommand** method cannot be used to invoke playback.

Example

In Visual Basic:

```
Private Sub Correct_Click()  
    VVRichEdit1.ExecuteCommand vvTBCorrectThis  
End Sub
```

In Visual C++ (MFC):

```
m_RichEdit.ExecuteCommand( vvTBCorrectThis );
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->ExecuteCommand( vvTBCorrectThis );
```

See Also

“Commands” on page 111

“CommandsEnabled” on page 114

LoadRTF

Loads an RTF file with the specified path name into the **VVRichEdit** control.

Syntax

In Visual Basic:

```
VVRichEdit.LoadRTF ( [String] )
```

In Visual C++ (MFC):

```
void LoadRTF(LPCTSTR bstrFileName);
```

In Visual C++:

```
HRESULT LoadRTF(BSTR bstrFileName);
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
VVRichEdit1.LoadRTF "TestFile.rtf"
```

In Visual C++ (MFC):

```
m_RichEdit.LoadRTF( _T("TestFile.rtf") );
```

In Visual C++:

```
BSTR File = SysAllocString ( OLESTR ("TestFile.rtf" ) );  
HRESULT hr = m_pIVVRichEdit->LoadRTF(File);  
SysFreeString ( File );
```

See Also

None.

LoadTextFile

Loads a text file with the specified path name into the **VVRichEdit** control.

Syntax

In Visual Basic:

```
VVRichEdit.LoadTextFile ( [String] )
```

In Visual C++ (MFC):

```
void LoadTextFile(LPCTSTR bstrFileName);
```

In Visual C++:

```
HRESULT LoadTextFile(BSTR bstrFileName);
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
Private Sub Form_Load()  
    VVRichEdit1.LoadTextFile ( "c:\My Documents\TestFile.txt" )  
End Sub
```

In Visual C++ (MFC):

```
VVRichEdit.LoadTextFile ( _T("TestFile.txt" strTextFile ) );
```

In Visual C++:

```
BSTR File = SysAllocString ( OLESTR ("TestFile.txt") );  
HRESULT hr = m_pIVVRichEdit->LoadTextFile ( File );  
SysFreeString ( File );
```

See Also

None.

Playback

Play back voice data for the selected range of text.

Syntax

In Visual Basic:

```
Playback
```

In Visual C++ (MFC):

```
void Playback();
```

In Visual C++:

```
HRESULT Playback();
```

Parameters

None.

Return Values

None.

Remarks

None.

Example

In Visual Basic:

```
VVRichEdit1.Playback
```

In Visual C++ (MFC):

```
void CTestRichEditDlg::OnPlayback()  
{  
    m_pvvRichEdit->Playback();  
}
```

In Visual C++:

```
void CTestRichEditDlg::OnPlayback()  
{  
    m_pvvRichEdit->Playback();  
}
```

See Also

None.

PlaybackEx

Play back and/or retrieve the wav data for a range of text.

Syntax

In Visual Basic:

```
PlaybackEx(StartIndex as long, TextLength as long, PlayAudio as BOOL,  
WavData as string)
```

In Visual C++ (MFC):

```
void PlaybackEx(long StartIndex, long TextLength, VARIANT_BOOL  
PlayAudio, BSTR *ppWavData)
```

In Visual C++:

```
HRESULT PlaybackEx(long StartIndex, long TextLength, VARIANT_BOOL  
PlayAudio, BSTR *ppWavData)
```

Parameters

StartIndex

Index of into text buffer of where to start playing the wav audio.

TextLength

The length of selected text. End offset is StartIndex + TextLength.

PlayAudio

TRUE - Playback Audio, FALSE - Play Audio to wave out.

WavData

Buffer to hold returned wave data if PlayAudio is FALSE.

Return Values

None.

Remarks

None.

Example

In Visual Basic:

```
VVRichEdit1.PlaybackEx(StartIndex, TextLength, FALSE, strWavData)
VVRichEdit1.PlaybackEx(StartIndex, TextLength, TRUE, NULL)
```

In Visual C++ (MFC):

```
void CTestRichEditDlg::OnPlaybackEx()
{
    CComBSTR bsWavData;
    pVVRichEdit->PlaybackEx(StartIndex, TextLength, VARIANT_FALSE,
    &bsWavData);

    pVVRichEdit->PlaybackEx(StartIndex, TextLength, VARIANT_TRUE, NULL);
}
```

In Visual C++:

```
void CTestRichEditDlg::OnPlaybackEx()
{
    CComBSTR bsWavData;
    pVVRichEdit->PlaybackEx(StartIndex, TextLength, VARIANT_FALSE,
    &bsWavData);
    pVVRichEdit->PlaybackEx(StartIndex, TextLength, VARIANT_TRUE, NULL);
}
```

See Also

None.

PlaybackEx2

Retrieve WAV data for a range of text.

Syntax

In Visual Basic:

```
PlaybackEx2(StartIndex as long, TextLength as long, pszFile as string)
```

In Visual C++ (MFC):

```
void PlaybackEx2(long StartIndex, long TextLength, LPCTSTR pszFile);
```

In Visual C++:

```
HRESULT PlaybackEx2(long StartIndex, long TextLength, BSTR pszFile);
```

Parameters

StartIndex

Index of into text buffer of where to start playing the wav audio.

TextLength

The length of selected text. End offset is $\text{StartIndex} + \text{TextLength}$.

pszFile

Wave file name for output.

Return Values

None.

Remarks

None.

Example

In Visual Basic:

```
VVRichEdit1.PlaybackEx2(0, 0, "c:\temp\my.wav")
```

In Visual C++ (MFC):

```
void CTestRichEditDlg::OnPlaybackEx2()  
{  
    m_pvvRichEdit->PlaybackEx2(0,0, L"c:\\temp\\my.wav");  
}  
  
void CTestRichEditDlg::OnPlayback()  
{  
    m_pvvRichEdit->Playback();  
}
```

In Visual C++:

```
void CTestRichEditDlg::OnPlaybackEx2()  
{  
    m_pvvRichEdit->PlaybackEx2(0,0, L"c:\\temp\\my.wav");  
}  
  
void CTestRichEditDlg::OnPlayback()  
{  
    m_pvvRichEdit->Playback();  
}
```

See Also

None.

SaveRTF

Saves the contents of the **VVRichEdit** control as an RTF file to the specified path name.

Syntax

In Visual Basic:

```
VVRichEdit.SaveRTF ( [String] )
```

In Visual C++ (MFC):

```
void SaveRTF(LPCTSTR FileName);
```

In Visual C++:

```
HRESULT SaveRTF(BSTR FileName);
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
VVRichEdit1.SaveRTF "TestFile.rtf"
```

In Visual C++ (MFC):

```
m_RichEdit.SaveRTF(_T("TestFile.rtf"));
```

In Visual C++:

```
BSTR File = SysAllocString ( OLESTR("TestFile.rtf"));  
HRESULT hr = m_pIVVRichEdit->SaveRTF(File);  
SysFreeString(File);
```

See Also

None.

SaveTextFile

Saves the contents of the **VVRichEdit** control as a text file.

Syntax

In Visual Basic:

```
VVRichEdit.SaveTextFile ( [PathName] )
```

In Visual C++ (MFC):

```
void SaveFile(LPCTSTR FileName);
```

In Visual C++:

```
HRESULT SaveFile(BSTR FileName);
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
Private Sub OnSave()  
    VVRichEdit1.SaveTextFile ( "c:\My Documents\TestFile.txt" )  
End Sub
```

In Visual C++ (MFC):

```
m_RichEdit.SaveTextFile ( _T(Cstring("TestFile.txt")) );
```

In Visual C++:

```
BSTR File = SysAllocString ( OLESTR("TestFile.rtf") );  
HRESULT hr = m_pIVVRichEdit->SaveFile( File );  
SysFreeString( File );
```

See Also

None.

SelPrint

Allows the client to invoke any of the “voice commands” programmatically simply by passing the ID of the command desired.

Syntax

In Visual Basic:

```
VVRichEdit.SelPrint
```

In Visual C++ (MFC):

```
void SelPrint(long lHDC);
```

In Visual C++:

```
HRESULT SelPrint(long lHDC);
```

Parameters

??

Return Values

??

Remarks

The primary use of this functionality is to expose voice command functionality for invocation via mouse or keyboard input, although it can also be used to control actions based on voice commands in an external **VVPhrases** control. This might be useful, for instance, if you need to have voice commands activated/available based on your own logic using a different tracking window for the command phrases.

Example

In Visual Basic:

```
Private Sub Form_Load()  
    VVRichEdit1.SelPrint  
End Sub
```

In Visual C++ (MFC):

```
m_RichEdit.SelPrint( (long)hDC );
```

In Visual C++:

```
HRESULT hr = m_pIVVRichEdit->SelPrint( (long)hDC );
```

See Also

None.

RichEdit Control Events

The ViaVoice **RichEdit** control supports the following events:

Change^a	KeyPress^a
Click^a	KeyUp^a
Command	MaxText
DblClick^a	MouseDown^a
DictationStateChange	MouseMove^a
Error	MouseUp^a
KeyDown^a	

-
- a. Represents a standard event in Visual Basic. For more information, refer to your Visual Basic documentation.

Command

Event fired when the user speaks one of the command words the **VVRichEdit** recognizes.

Syntax

In Visual Basic:

```
Command (ByVal CmdID As Long, ByVal strCommand As String)
```

In Visual C++ (MFC):

```
void OnCommand(long CmdID, LPCTSTR strCommand);
```

Parameters

CmdID

Long. A number that uniquely identifies the command the **VVRichEdit** recognized.

strCommand

String. The actual text selected that the **VVRichEdit** recognized. You should not write code that is dependent on this value as the phrases are subject to change and vary with the language of the engine. Use the **CmdID** parameter instead. It is recommended that you use the *strCommand* parameter for the UI Server's Word History display only. This will be an empty string if invoked through any means other than the default speech commands.

Return Values

??

Remarks

This control recognizes commands from the default voice commands only when **CommandsEnabled** is set to True and there are no other limiting factors (see "Commands" and "CommandsEnabled" properties). Commands are never recognized when **CommandsEnabled** is False. Also, be aware that

this event is called only for voice commands and not for commands invoked programmatically through the **ExecuteCommand** method.

Example

In Visual Basic:

```
Private Sub VVRichEdit1_Command(ByVal CmdID As Long, ByVal strCommand As String)
    Select Case CmdID
        Case vvTBShowEC
            ProcessTBShowEC
        . . .
    End Select
End Sub
```

In Visual C++ (MFC):

```
void CTestDlg::OnCommand(long CmdID, LPCTSTR strCommand)
{
    switch (CmdID)
    {
        case vvTBShowEC
            ProcessTBShowEC( );
            break;
        default:
            break
    }
}
```

See Also

“Capturing Commands” on page 36

“Commands” on page 111

“CommandsEnabled” on page 114

DictationStateChange

Event fired when the control enters or exits dictation mode. There are several actions that affect the state of dictation.

Syntax

In Visual Basic:

```
DictationStateChange (ByVal DictationOn As Boolean)
```

In Visual C++ (MFC):

```
void OnDictationStateChangeVvrichedit1(BOOL DictationOn);
```

Parameters

DictationOn

Boolean. The current state of dictation mode.

Return Values

TRUE

The control is ready to receive dictation speech and turn it into text.

FALSE

The control will ignore dictation input.

Remarks

This event implies nothing to do with the control being able to understand voice commands. The following conditions can effect the state of dictation:

- The state of the **DictationOn** property is changed explicitly.
- The state of the **Locked** property is changed explicitly
- The state of the **Enabled** property is changed explicitly
- The length of text exceeds the max set in the **MaxLength** property.

Note:

If **AutoDictationWindow** is not set to **VV_HWND_AUTODICTATION**, then focus changes will not trigger this event, even if dictation availability has changed. If you need this information, then you must write code to track the focus changes.

Example

In Visual Basic:

```
Private Sub VVRichEdit1_DictationStateChange(ByVal DictationOn As
Boolean)

    'Handler Code Here

End Sub
```

In Visual C++ (MFC):

```
void CTestDlg::OnDictationStateChange(BOOL DictationOn)
{

    // Handler Code Here

}
```

See Also

“CommandsEnabled” on page 114

“DictationOn” on page 116

“AutoDictationWindow (Read/Write at Run Time Only)” on page 100

“Enabled” in Visual Basic Documentation

“Locked” in Visual Basic Documentation

“MaxLength” in Visual Basic Documentation

Error

Reports an error.

Syntax

In Visual Basic:

```
Error (sErrorID As Integer, pstrDescription As String, hresult As Long,
strSource As String, strHelp As String, lHelpID As Long, fShow As
Boolean)
```

In Visual C++ (MFC):

```
void OnError(short sErrorID,
    BSTR FAR* pstrDescription,
    long FAR* hresult,
    BSTR FAR* strSource,
    BSTR FAR* strHelp,
    long FAR* lHelpID,
    BOOL FAR* fShow)
```

Parameters

sErrorID

Integer. The error number. The error number can be one of the following values:

DICTERR_DICTATION_ACTIVATE	101 (Hex 65)
DICTERR_DICTATION_DEACTIVATE	102 (Hex 66)
DICTERR_COMMANDS_ACTIVATE	103 (Hex 67)
DICTERR_COMMANDS_DEACTIVATE	104 (Hex 68)
DICTERR_ENGINE_CONNECT	105 (Hex 69)

pstrDescription

String. The error description. The error message string is language-dependent and requires the use of the appropriate language resource DLL. The control will use the language of the container

application for error messages. If the control cannot find the appropriate language DLL, the error message will be in US English.

hresult

Long. The COM-generated error code.

strSource

String. This parameter contains the name of the module where the error occurred.

strHelp

String. The name and path of the help file (HLP file) that the control will invoke when the user clicks the help button in an error message dialog.

lHelpID

Long. The context ID of the page in the help file that explains the error.

fShow

Boolean.

Return Values

TRUE

Displays an error message dialog box when an error occurs.

FALSE

Prevents the control from showing this dialog

Remarks

The ViaVoice **RichEdit** control can report errors in one of two ways. If the error occurs from the setting of a property or the issuing of a method incorrectly, the control generates a trappable error (returns an error HRESULT). However, some errors can occur while the user is interacting with the control directly. Whenever the control needs to report this type of error, it fires the **Error** event. After the event fires and execution returns to the control, the control shows an error message dialog box.

Example

In Visual Basic:

```
Private Sub VVRichEdit1_Error( _  
    sErrorID As Integer, _  
    pstrDescription As String, _  
    hresult As Long, _  
    strSource As String, _  
    strHelp As String, _  
    lHelpID As Long, _  
    bShow As Boolean)  
  
    Select Case sErrorID  
        Case DICTERR_ENGINE_CONNECT  
            MsgBox "Unable to connect to a speech engine."  
            bShow = False  
        End Select  
  
End Sub
```

In Visual C++ (MFC):

```
void CTestctrlDlg::OnError(
    short sErrorID,
    BSTR FAR* pstrDescription,
    long hresult,
    LPCTSTR strSource,
    LPCTSTR strHelp,
    long lHelpID,
    BOOL FAR* bShow)
{
    switch (sErrorID)
    {
        case DICTERR_ENGINE_CONNECT:
            MessageBox ("Unable to connect to a speech engine.",
                "Speech Error", MB_OK);
            *bShow = FALSE;
            break;
    }
}
```

MaxText

Event fires when the length of the text in the **RichEdit** reaches the maximum number of characters allowed in the control.

Syntax

In Visual Basic:

```
MaxText()
```

In Visual C++ (MFC):

```
void OnMaxText();
```

Parameters

None.

Return Values

None.

Remarks

You can specify maximum number of characters through the **MaxLength** property. Setting the **MaxLength** property to zero means that the control accepts the maximum of a standard edit control, which is OS dependent. See Microsoft documentation for details.

Example

In Visual Basic:

```
Private Sub VVRichEdit1_MaxText()  
    'AutoTab to the next control on the form when all the  
        information has been entered  
    SendKeys "{TAB}"  
End Sub
```

In Visual C++ (MFC):

```
void CTestDlg::MaxText()  
{  
    //AutoTab to the next control on the form when all the  
        information has been entered  
    GetNextDlgTabItem(GetFocus())->SetFocus();  
}
```

See Also

“MaxLength” in Visual Basic Documentation

RichEdit Control Frequently Asked Questions

This chapter contains answers to the most frequently asked questions about the ViaVoice **RichEdit** control.

How can I enter dictation mode automatically each time the VVRichEdit gets focus?

This is the default operation of the control. Set the **AutoDictationWindow** property to **VV_HWND_AUTODICTATION** (-1) at run time (this is the default). By setting this value for **AutoDictationWindow** and setting **DictationOn** to true, the control will automatically enter dictation mode when it gets the focus, and exit dictation mode when it loses focus. Setting **DictationOn** to false will still disable dictation regardless of window focus.

How can I get more control in determining when dictation is available?

One way is to set the **AutoDictationWindow** property to **NULL** at design time to enable “global” dictation. By setting this value, the control will always accept dictation when **DictationOn** is true and will stop accepting dictation when **DictationOn** is false. There can be only one (1) global dictation object active (**DictationOn** set to true) in the entire system at any one time.

Another option is to use some other window for implicit dictation control. To do this, simply find the “top-most” window in the application of interest (your own or any other application) and assign it to **AutoDictationWindow** before setting **DictationOn** to true. This has the effect of enabling dictation any time that window, or any of its child widows, has focus. Using a window for dictation tracking provides the benefit of greater control without the problems associated with a “global” dictation object.

If I have a project using the standard Visual Basic RichTextBox, can I use VVRichEdit without changing any code?

The **VVRichEdit** control can be substituted in your project for the standard Visual Basic **RichTextBox** with minimal code changes. In Visual Basic, the name of the **VVRichEdit** control can be kept the same as the name of the original control. If this is done, most code will work without changes. There are a few differences between the controls.

- **VVRichEdit** has two methods for loading files, **LoadRTF** and **LoadTextFile**. The **RichTextBox** control has one method for both operations by passing a flag indicating what kind of file to load.

- **VVRichEdit** allows the programmer to specify actual units in properties specifying a distance rather than specifying TWIPS like the RichTextBox. If no units are specified for a distance, the VVRichEdit control does not default to the units of the container. It defaults to TWIPS.
- **VVRichEdit** does not make extensive use of the VARIANT data type for parameters like the RichTextBox.

Why aren't the methods prefixed with "SEL" shown in the Object Browser in Visual Basic?

Using these methods only makes sense at run time because they operate on the current selection of text. At design time, the control doesn't accept user input, so there is no valid selection.

When I invoke Correction, nothing shows up in the list box?

If the cursor is in a dictated word, invoking the **VVRichEdit** will get the alternate words from **VVDictationMgr** and display them in the error correction window, **VVECWin**. If the cursor is not in a word, or it is in typed text, the error correction window will be shown without any alternates. This is because it is not currently possible to determine alternates for typed text.

I press the Correction button in the Error Correction window and it doesn't correct the text?

It is not currently possible to correct typed text.

Why can I only correct one word at a time?

In most cases, correcting one word at a time allows for better accuracy.

I can't dictate into the edit field in the Error Correction Window?

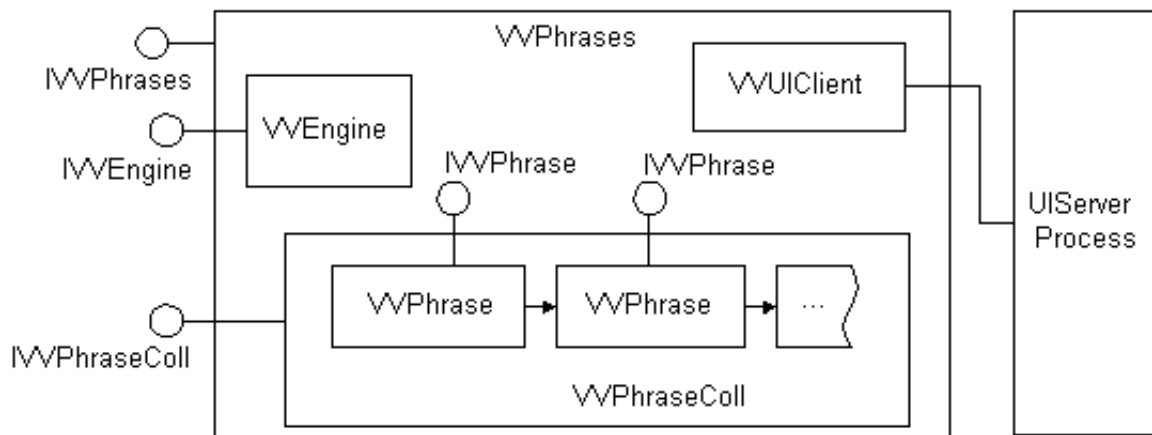
It is more accurate to type the text you would like to replace the currently select word with, rather than dictate it.

The IBM ViaVoice **Phrases** Control (**VVPhrases**) is an ActiveX control that enables developers to add simple phrase command recognition to their applications. The main idea behind the control is that the developer provides the control with a list of phrases or commands, and the control will notify the developer when the user speaks one of the phrases in the list.

VVPhrases is one of two controls in the ViaVoice SDK capable of recognizing phrases and commands. The second control is **VVCFGram**. **VVCFGram** uses compiled grammar files to interpret user speech. A grammar is a script that defines the way that the user can construct sentences. It, therefore, offers a more robust solution for command recognition than **VVPhrases**. However, **VVCFGram** requires that you learn the grammar language, and that you compile the grammar script beforehand. **VVPhrases** on the other hand offers a simple command and phrase recognition solution that does not require knowledge of grammars.

VVPhrases Object Hierarchy

The following diagram shows the object hierarchy for **VVPhrases**.



Getting Started with the Phrases Control

The following is a tutorial on how to incorporate the **VVPhrases** control into your Visual Basic or Visual C++ applications. This tutorial is designed to present you with the most commonly used properties and events in the **VVPhrases** control.

Creating an Instance

In Visual Basic:

To add the **VVPhrases** control to your application, do the following:

1. From the **Project** menu, choose **Components**.
The 'Components' dialog box, Figure 13, appears. The 'Components' dialog lists all the ActiveX Controls that you can use in your application.

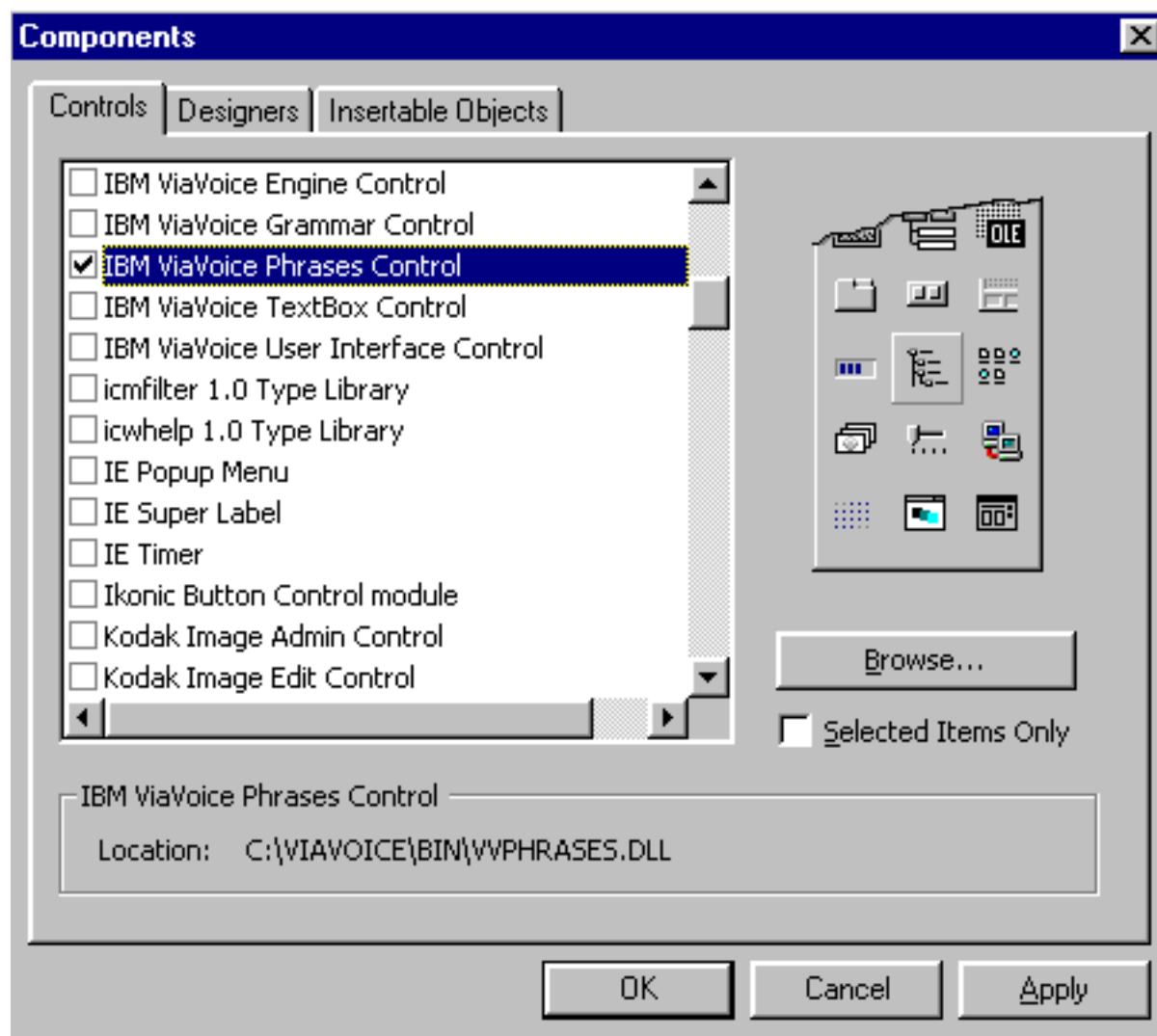


Figure 13. Component Selection Dialog - Visual Basic

2. Select **IBM ViaVoice Phrases Control** from the list and choose **OK**.

Visual Basic adds the control to your project, and adds a new icon to the toolbar (Figure 14).



Figure 14. VVPhrases Control Toolbar Icon

3. Add an instance of the **VVPhrases** control to your form.
The **VVPhrases** control is an invisible control at run time.

In Visual C++ (MFC):

To add the **VVPhrases** control to your MFC project do the following:

1. From the **Project** menu, select **Add To Project**, then select **Components and Controls**.

The Components and Controls Gallery dialog box, Figure 15, appears.

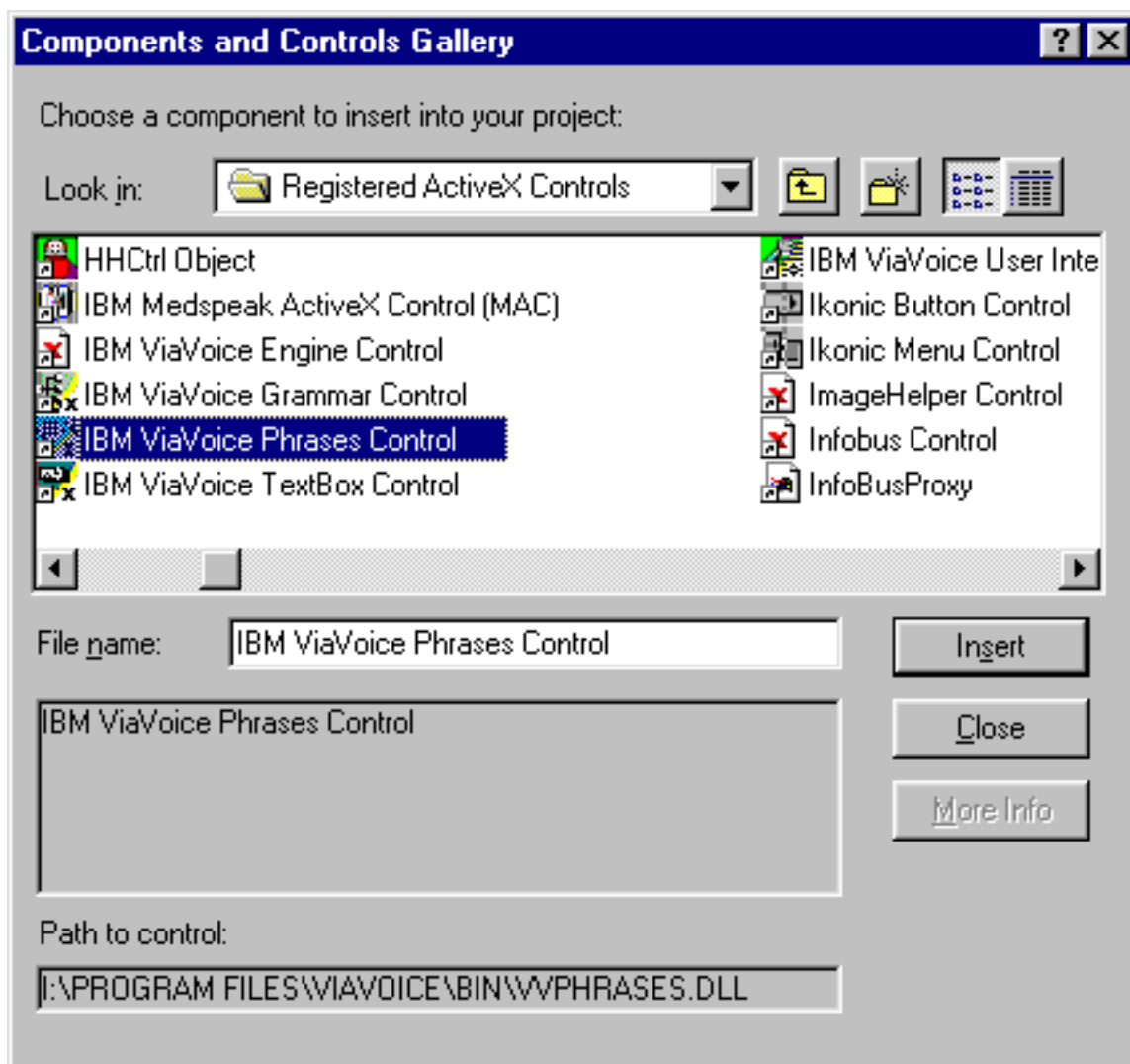


Figure 15. Insert ActiveX Control Dialog Box - Visual C++ (MFC)

2. Double-click the 'Registered ActiveX Controls' folder in the dialog box.
3. Select the **IBM ViaVoice Phrases Control** icon in the list of controls, then click **Insert**.
A confirmation message box appears, asking "Insert this component?"

4. Respond to the confirmation message box by choosing **OK**.

The 'Confirm Classes' dialog box, Figure 16, appears listing the components in the **VVPhrases** control: CVVPhrases, CVVPhrase, CVVPhraseColl, CVVBookmarkColl, CVVBookmark, and CVVEngine. (The CVVBookmarkColl and CVVBookmark will be implemented in the future.)

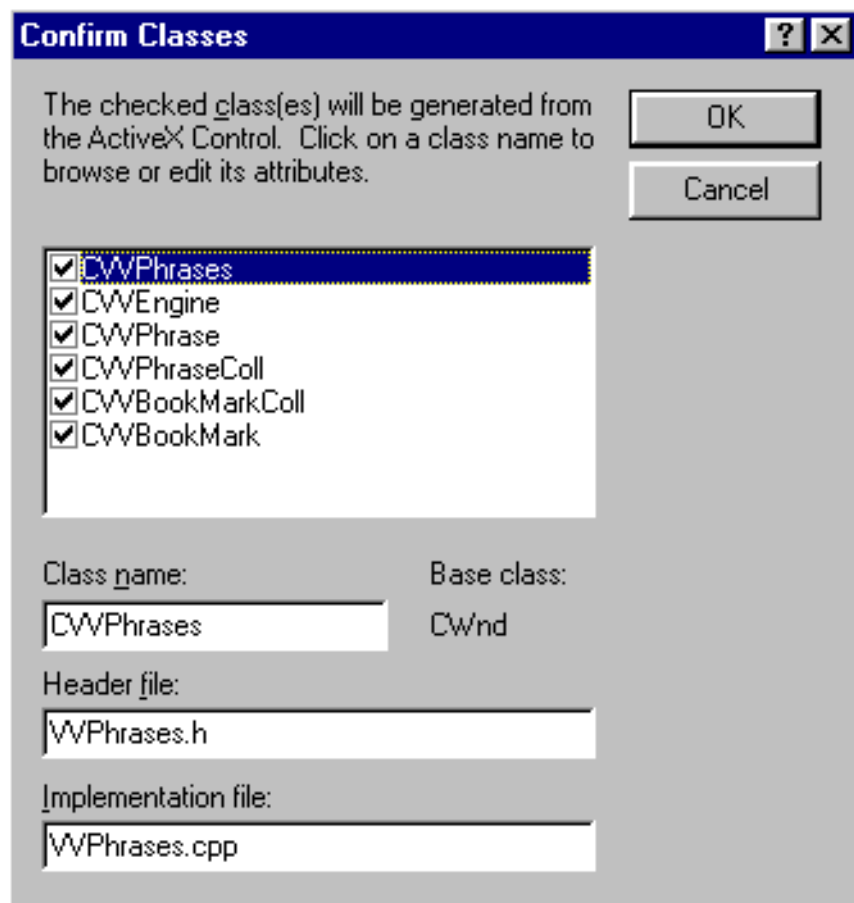


Figure 16. Confirm Classes Dialog Box

5. Click **OK** in the 'Confirm Classes' dialog box.
6. Close the 'Components and Controls Gallery' dialog box.

If you examine the Project Workspace window in the class view you will notice six new classes: CVVPhrases, CVVPhrase, CVVPhraseColl, CVVBookmarkColl, CVVBookmark, and CVVEngine (assuming you accepted the default names for the class in the Confirm Classes dialog box).

7. In the resource view of your Project Workspace window double-click the dialog resource entry where you wish to insert the **VVPhrases** control.

The **VVPhrases** icon, Figure 17, appears in the Controls toolbar.

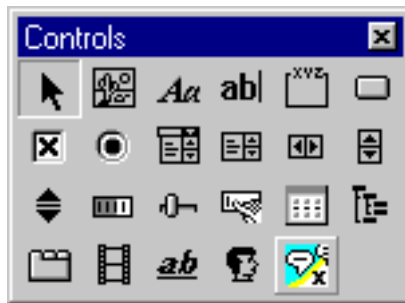


Figure 17. VVPhrases Icon in the Controls Toolbar

8. Add an instance of the **VVPhrases** control to the dialog box.
After you add the **VVPhrases** control to your dialog you can invoke Class Wizard to create a member variable for your CVVPhrases class. You might also decide to capture the events in the control by adding Event handlers to your dialog class. To add Event handlers, click the **VVPhrases** control with the right mouse button, then select **Events** from the pop-up menu.

Drag-Drop-n-Go Support

The **VVPhrases** control fully incorporates IBM's Drag-Drop-n-Go technology. With this technology you can add the control to your application and use it without writing many lines of code.

The **VVPhrases** control has the following Drag-Drop-n-Go properties:

Property	Type	Default Value
AutoConnect	Boolean	True
AutoUI	Boolean	True

AutoConnect when set to True causes the control to automatically search for a speech recognition engine at run time and connect to it. This is necessary before the control can recognize commands. If you set this property to False you must connect manually by calling the **Engine.Connect** method of the control.

AutoUI when set to True causes the **VVPhrases** control to display and manipulate the ViaVoice **User Interface Server**. Refer to "Introduction to the User Interface Control" on page 497 for more information.

By leaving the properties set to their default values you guarantee that the control will connect to the speech recognition engine as well as display and control the IBM ViaVoice **User Interface Server** automatically.

Adding Phrases

The first step in using the **VVPhrases** control is to specify the list of commands that you wish the control to recognize. As soon as the control hears the user speak one of the command words, it triggers the **SpeechRecognized** event. You can then write code in this event to handle the command in your application.

The **VVPhrases** control stores the list of phrases in the phrases collection object – **VVPhraseColl**. The **VVPhraseColl** object stores instances of the phrase object – **VVPhrase**. The phrase object stores the actual command phrase text, as well as a programmer assigned ID for the command. (You will get more information about the **VVPhrase** object later in this chapter).

You can access the **VVPhraseColl** object through the **Phrases** property in the control. To add a phrase (command) to the collection simply use the **VVPhraseColl**'s **Add** method as follows:

In Visual Basic:

```
VVPhrases1.Phrases.Add "Hello", 100, "Say Hello World"
```

In Visual C++ (MFC):

```
m_VVPhrases.GetPhrases().Add("Hello", 100, "Say Hello World", TRUE)
```

The first parameter in the **Add** method is a programmer assigned unique name for the phrase. You can use this name later in your code to access the phrase object within the collection. The second parameter is a programmer assigned ID for the command. This ID enables you to write code to handle the command without relying on the text. It also enables you to group commands by assigning to them the same ID. The third parameter is the actual phrase or command the control will recognize. (The string is not case-sensitive.) The fourth parameter is the **Enabled** property. If the phrase is enabled then the control will recognize the phrase. Otherwise, the control will ignore the phrase. If you are using Visual Basic, this parameter is optional and it will be defaulted to True.

The second step in using the control is to handle the **SpeechRecognized** event. The **VVPhrases** control listens to the user. When the user speaks one of the phrases in the phrase collection, the control fires the **SpeechRecognized** event to notify you. The following code segment shows how to handle this event.

In Visual Basic:

```
Private Sub VVPhrases1_SpeechRecognized(ByVal Name As String, ByVal ID As Long, ByVal Phrase As String, UpdateUIText As Boolean, ByVal BegTime As String, ByVal EndTime As String)

Select Case ID
    Case 100
        MsgBox "Hello Sue"
    Case 101
        MsgBox "Hello James"
    End Select

End Sub
```

In Visual C++ (MFC):

```
void CVVPhrtest::OnSpeechRecognized(LPCTSTR Name, long ID, BSTR FAR *Phrase, BOOL FAR *UpdateUIText, LPCTSTR BegTime, LPCTSTR EndTime)

{

    switch (ID)
    {
    case 100:
        MessageBox ("Hello World" ,"VVPhrases",MB_OK);
        break;
    default:
        break;
    }
}
```

Enabling/Disabling Phrases

During the execution of your application, it may be necessary at times to turn on or off speech recognition for certain commands, or even for all the commands in the phrase collection.

The **VVPhrase** object, which encapsulates each individual phrase in the control's phrase collection, has an **Enabled** property. This property lets you turn on or off recognition for a single command. By default, the control sets this property to True when the object is created. To turn off recognition for the command, you first specify which item in the collection you wish to modify, then change the value of the **Enabled** property to False as follows:

In Visual Basic:

```
VPPhrases1.Phrases("Hello").Enabled = False
```

In Visual C++ (MFC):

```
VARIANT va;  
VariantInit(&va);  
va.vt = VT_BSTR;  
va.bstrVal = SysAllocString(L"Hello");  
m_VPPhrases.GetPhrases().GetItem(va).SetEnabled(FALSE);
```

You may recall from our earlier discussion that the **Phrases** property in the control returns a **VVPhraseColl** collection object. To find a phrase in the collection, issue the **Item** method passing the phrase text in the first parameter. The **Item** method returns an instance of the **Phrase** object that encapsulates the phrase. Once you have an instance of the **Phrase** object you can change its properties. In the previous example, you saw how to enable and disable a phrase. You can also change the ID number of the phrase, by changing the object's ID property.

The **VVPhraseColl** object also has an **Enabled** property, which allows you to disable or enable all the phrases without removing them from the collection. To disable all the phrases simply set the **Enabled** property of this object to False, as follows:

In Visual Basic:

```
VVPhrases1.Phrases.Enabled = False
```

In Visual C++ (MFC):

```
m_VVPhrases.GetPhrases().SetEnabled(FALSE);
```

Working with the Custom Designer

The **VVPhrases** control ships with a Custom Designer window that enables you to add Phrases at design time. To invoke it simply click the ellipsis (...) next to the **Custom** property (see Figure 18).

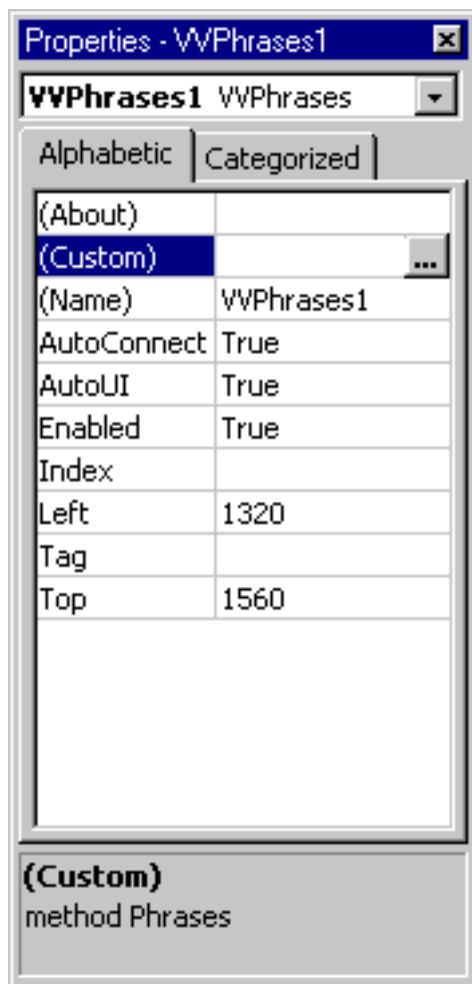


Figure 18. The Custom Property

This will bring up the Designer window, depicted in Figure 19.

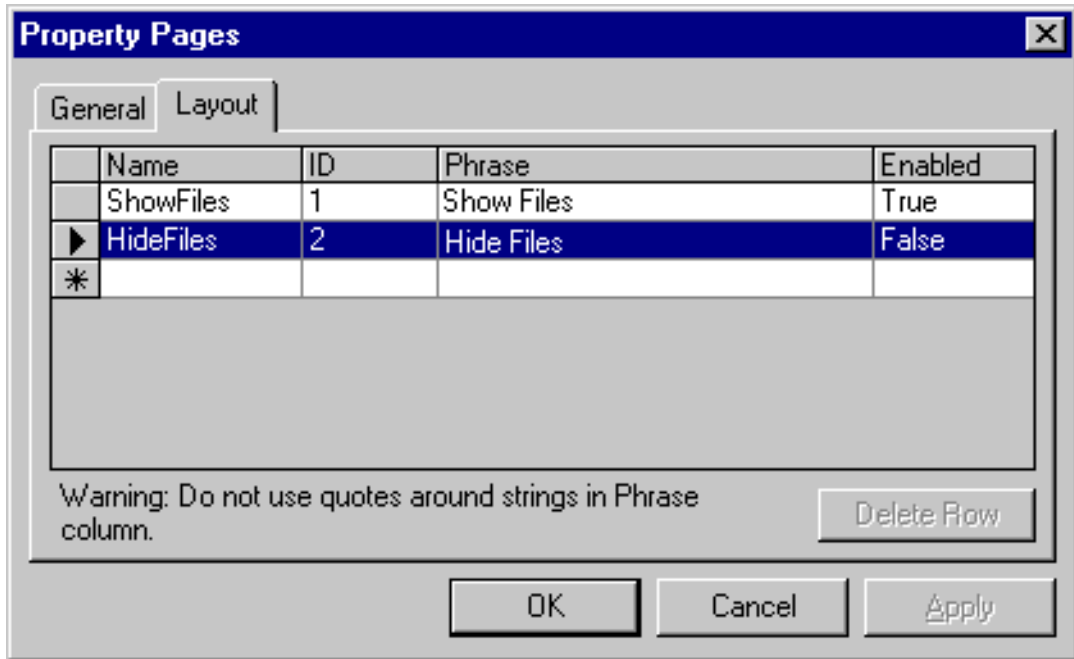


Figure 19. The Designer Window

The Custom Designer allows you to add, remove, and modify phrases at design time. To add a phrase, simply enter the necessary information to fill out a row in the layout grid.

1. Enter the name of the phrase item in the name field.
2. Enter an ID for the phrase. [Recommended but optional.]
3. Enter the phrase that you wish the control to recognize in the Phrase field. Do not use quotes around strings in Phrase column.
4. Indicate whether the phrase should be initially enabled or not by typing "t" or "f" in the enabled field which will then display "True" or "False" as appropriate.

Repeat these steps for each phrase you wish the control to recognize.

You can modify a previously entered phrase simply by selecting the desired field of an existing phrase and changing it as necessary. You may also remove an entry completely by clicking on the "button" to the left of the phrase to be deleted, which will highlight the entire row (as indicated in Figure 19). Then click the **Delete Row** button to remove the phrase.

Object Hierarchy

The **VVPhrases** control defines the following object hierarchy:

Concept: Control → Phrases → Phrase

Object names: VVPhrases → VVPhraseColl → VVPhrase

The control (**VVPhrases**) contains a property called **Phrases**, (or **GetPhrases**, if using MFC) which exposes a **Phrases** collection object (**VVPhraseColl**). Each member of the phrase collection is an instance of a **VVPhrase** object.

Example

To change the properties in a phrase object:

In Visual Basic:

```
VVPhrases1.Phrases.Item("Hello").Enabled = True
```

In Visual C++ (MFC):

```
m_VVPhrases.GetPhrases().GetItem(va).SetEnabled(TRUE);
```

In Visual Basic, using default properties that line can be shortened to the following:

```
VVPhrases1.Phrases("Hello").Enabled = True
```

Note:

The above line of code is possible because **Item** is the default property of **VVPhraseColl**. Because the **Phrases** property is the default property of the **VVPhrases** control you can shrink the line of code even further, as follows:

```
VVPhrases1("Hello").Enabled = True
```

There is a macro definition for Visual C++ developers using MFC which simulates the default property behavior in Visual Basic. The macro definition is the following:

```
#define GetPhrases(a) GetPhrases().GetItem(a)
```

Using this macro you can then convert the first example to the following:

```
m_VVPhrase.GetPhrases(va).SetEnabled(TRUE);
```


The following sections describe the properties, methods, and events for the **VVPhrases** Control, **VVPhraseColl** Collection, and **VVPhrase** Object.

VVPhrases Control

VVPhrases Control Properties

The **VVPhrases Control** supports the following properties:

- **AutoConnect**
- **AutoUI**
- **Enabled**
- **Engine**
- **Index^a**
- **Layout**
- **Phrases**

a. Represents a standard method in Visual Basic. For more information, refer to your Visual Basic documentation.

AutoConnect (VVPhrases)

Automatically connects to the speech recognition engine when created. Before the **VVPhrases** control can recognize phrases it needs to connect to a speech engine. When this property is set to True the control will try to connect to an engine with the following properties: IBM Manufacturer, continuous speech, with dictation, limited domain grammar, and context free grammar support. If you wish to override the default behavior, set this property to False, then at run time modify the properties in the **Engine** object and call the **Connect** method.

Syntax

```
VVPhrases.AutoConnect = [Boolean]
```

Parameters

AutoConnect
??

Return Values

TRUE

(Default) The control automatically finds and connects to a speech engine.

FALSE

The control does not connect to a speech engine. You must issue the **Connect** method to cause the control to connect to the engine

Remarks

Changing the value of this property at run time has no effect. This property is meant to be used only at design time.

If **AutoConnect** is True, the control will connect to the speech engine before the **Form_Load** event takes place in Visual Basic and before the **InitDialog** method gets executed in an MFC application.

AutoUI (VVPhrases)

sets the **VVPhrases** control to automatically displays and interacts with the IBM ViaVoice **User Interface Server**.

Syntax

```
VVPhrases.AutoUI = [Boolean]
```

Parameters

??

Return Values

TRUE

(Default) **VVPhrases** displays the **User Interface Server** and interacts with it automatically

FALSE

VVPhrases does not display the **User Interface Server** or interact automatically with it (if another control has already displayed the **User Interface Server**, for example).

Remarks

When **AutoUI** is True, the **VVPhrases** automatically updates the following components in the **User Interface Control**: Microphone, Word History, and Volume Level.

If multiple instances of the **VVPhrases** control have **AutoUI** set to True, the **User Interface Server** only gets created once, and all the instances of the control interact with the same **User Interface Server**. If you prefer not to display the **User Interface Server**, or not to have the **VVPhrases** control interact with it automatically, set this property to False.

See Also

Chapter 25, “Introduction to the User Interface Control” on page 497

Chapter 26, “Getting Started with the User Interface Control” on page 499

Chapter 27, “Classes, Structures, and Enumerations” on page 533

Chapter 28, “Properties, Methods, and Events” on page 561

Chapter 29, “User Interface Control Frequently Asked Questions” on page 629

Enabled (VVPPhrases)

Enables/disables command recognition.

Syntax

```
[Control].Enabled = [Boolean]
```

Parameters

??

Return Values

TRUE

(Default) **VVPPhrases** listens for commands.

FALSE

VVPPhrases does not listen for commands.

Remarks

Changing the value of this property does not change the value of the **Enabled** property in the collection object, or the value of the **Enabled** property in each of the **Phrase** objects in the collection. It does, however, disable all commands in the control.

Example

In Visual Basic:

```
Private Sub chkSpeechRecognition_Click()  
    VVPhrases1.Enabled = Not VVPhrases1.Enabled  
End Sub
```

In Visual C++ (MFC):

```
void SpeechEnabledToggle()  
{  
    m_VVPhrases.SetEnabled(!m_VVPhrases.GetEnabled());  
}
```

See Also

VVPhraseColl collection's "Enabled (VVPhraseColl)" on page 248

VVPhrase object's "Enabled (VVPhrase)" on page 265

Engine (VVPhrases)

Contains a reference to the ViaVoice **Engine** control (**VVEngine**). If **AutoConnect** is True, the engine property will refer to a connected engine control at run-time; otherwise, the internal engine control is disconnected. When **AutoConnect** is False, the desired properties of the engine can be set – for instance the speaking style as discrete or continuous – and then Engine.Connect can be called to start up a speech engine with the desired attributes.

Syntax

```
[Control].Engine
```

Parameters

None.

Return Values

None.

Remarks

The **Engine** property is actually holding an implicitly created ActiveX control (**VVEngine**), which can also be created separately. Inserting a **VVEngine** control in a project enables you to set the engine properties on this control, call connect, and then assign the resulting connected engine to multiple **VVPhrases**, **VVCFGram**, and **VVTextBox** controls. The **AutoConnect** or **AutoInit** property in **VVTextBox** control must be False for all controls besides the **VVEngine** control, however. Then the **VVEngine** control can be assigned to the writable **Engine** property of **VVPhrases**, **VVTextBox**, and **VVCFGram**.

Example

In Visual Basic:

```
Private Sub Form_Load()  
    VVPhrases1.Engine.SpeakingStyle = vvssContinuous  
    VVPhrases1.Engine.AudioSourceType = vvstSAPICompliant  
    VVPhrases1.Engine.Connect  
End Sub
```

In Visual C++ (MFC):

```
void StartEngine()  
{  
    m_VVPhrases.GetEngine().SetSpeakingStyle(vvasContinuous);  
    m_VVPhrases.GetEngine().SetAudioSourceType(vvstSAPICompliant);  
    m_VVPhrases.GetEngine().Connect();  
}
```

See Also

Refer to the [ViaVoice Engine Control Guide](#) for more information.

Layout (VVPhrases)

Binary representation of the current state of the control. The **Layout** property allows the developer to obtain a string of bytes encoding all currently loaded phrases in the **VVPhraseColl** collection, together with the values of all other properties. The developer can then save this binary snapshot of the control to a file or registry key and reload it to restore the exact state of the control.

Syntax

```
[Control].Layout = [String]
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
Private Sub Form_Load()  
Dim A As String  
  
    VVPhrases1.AddPhrase "yes", 100, "yes", True  
  
    Open "MyFile.dat" For Random As #1  
    Put #1, , (VVPhrases1.Layout)  
    Close #1  
  
    VVPhrases1.Phrases.RemoveAll  
  
    Open "MyFile.dat" For Random As #1  
    Get #1, , A  
    Close #1  
  
    VVPhrases1.Layout = A  
End Sub
```

See Also

None.

Phrases (VVPhrases)

Returns a reference to the **VVPhraseColl** collection object. The **VVPhrases** control stores phrases in **VVPhrase** objects and then adds them to a collection. Through this collection you can add phrases, remove phrases, and select phrases. You can also enable/disable all the phrases in the collection by modifying the collection's **Enabled** property.

Syntax

```
[Control].Phrases
```

Parameters

None.

Return Values

None.

Remarks

The Phrases property is the control's default property. Because the Item property is the VVPhraseColl collection's default property, you can write the following:

```
VVPhrases1("Save").Enabled = True
```

Example

In Visual Basic:

```
VVPhrases1.Phrases("Hello").Enabled = False
```

In Visual C++ (MFC):

```
VARIANT va;  
VariantInit(&va);  
va.vt = VT_BSTR;  
va.bstrVal = SysAllocString(L"Hello");  
m_VVPhrases.GetPhrases().GetItem(va).SetEnabled(FALSE)
```

See Also

“VVPhraseColl Collection” on page 245

VVPhrases Control Methods

The **VVPhrases Control** supports the following methods:

- **About**^a
- **AddPhrase**
- **Refresh**^a
- **RefreshUIText**

a. Represents a standard method in Visual Basic. For more information, refer to your Visual Basic documentation.

AddPhrase (VVPhrases)

Creates a new **Phrase** object and adds it to the collection.

Syntax

```
VVPhrase = [VVPhrases].AddPhrase(ByVal Name As String, ByVal ID As Long,  
ByVal Phrase As String, Optional ByVal Enabled As Boolean)
```

Parameters

Name

String. A programmer assigned string name to identify the phrase object. You can use this name to refer back to the object within the phrase collection.

ID

Long. A programmer assigned number. This parameter enables you to associate a command or group of commands with a number.

Text

String. The actual phrase that the control will recognize (Example: "Save Record"). This parameter is not case-sensitive.

Enabled

Boolean. Enables/Disables the phrase. In Visual Basic this parameter is optional and defaulted as True.

Return Values

VVPhrase object

An instance of the newly created **Phrase** object that encapsulates the new phrase.

Remarks

By default the newly created phrase will become active immediately (if the **Enabled** parameter, the VVPhrasesColl's **Enabled** property, and the control's **Enabled** property are set to True).

The **AddPhrase** method serves as a shortcut for `VVPhrase.Phrases.Add`. Both methods are identical in functionality.

You should not add two phrases with the same name. The Name serves a unique identifier for the phrase object within the collection. However, you can add many phrases with the same ID number and with the same Text.

Example

In Visual Basic:

```
'Add Commands
VVPhrases1.AddPhrase("Begin", 100, "Begin New Record")
VVPhrases1.AddPhrase("Add", 100, "Add New Record")
VVPhrases1.AddPhrase("Edit", 200, "Edit Record")
VVPhrases1.AddPhrase("Delete", 300, "Delete Record")
VVPhrases1.AddPhrase("Trash", 300, "Trash Record")
```

In Visual C++ (MFC):

```
'Add Commands
m_VVPhrases1.AddPhrase("Begin",100,"Begin New Record",TRUE);
m_VVPhrases1.AddPhrase("Add", 100, "Add New Record",TRUE);
m_VVPhrases1.AddPhrase("Edit", 200, "Edit Record",TRUE);
m_VVPhrases1.AddPhrase("Delete", 300, "Delete Record", TRUE);
m_VVPhrases1.AddPhrase("Trash", 300, "Trash Record",TRUE);
```

See Also

“Remove (VVPhraseColl)” on page 257

RefreshUIText (VVPhrases)

Forces an update of the ViaVoice **User Interface Server** when **AutoUI** is True.

Syntax

```
[Control].RefreshUIText(Text As String)
```

Parameters

Text

Text to display in the **UIServer**.

Return Values

??

Remarks

Used primarily in the SpeechRecognized event together with the **UpdateUIText** parameter.

Example

In Visual Basic:

```
Private Sub VVPhrases1_SpeechRecognized(ByVal Name As String, ByVal ID As
Long, Phrase As String, UpdateUIText As Boolean, ByVal BegTime As String,
ByVal EndTime As String)
    String WordHistoryText = Name

    Select Case ID
        Case 100 ` If user said "Open File"...
            WordHistoryText = "Opening document..."
        Case 110 ` If user said "Close File"...
            WordHistoryText = "Closing document..."
        Case 120 ` If user said "Close Window"...
            WordHistoryText = "Goodbye!"
    End Select

    VVPhrases1.RefreshUIText WordHistoryText
End Sub
```

In Visual C++ (MFC):

```
void CMYWnd::OnSpeechRecognized (LPCTSTR Name, long ID, BSTR FAR* Phrase,
BOOL FAR* UpdateUIText, LPCTSTR BegTime, LPCTSTR EndTime)
{
    LPCTSTR lpszWordHistoryText = Name;

    switch (ID) {
        // If user said "Open File"...
        case 100: lpszWordHistoryText = "Opening document..." break;
        // If user said "Close File"...
        case 110: lpszWordHistoryText = "Closing document..." break;
        // If user said "Close Window"...
        case 120: lpszWordHistoryText = "Goodbye!" break;
    }

    m_VVPhrases1.RefreshUIText(lpszWordHistory);
}
```

See Also

“SpeechRecognized (VVPhrases)” on page 241

VVPhrases Control Events

The **VVPhrases** Control supports the following events:

- **BeginSpeechRecognition**
- **BookMarkReached** (Not Supported)
- **Paused** (Not Supported)
- **SpeechHypothesis** (Not Supported)
- **SpeechRecognized**
- **SpeechRejected** (Not Supported)
- **TrainingRequired**

BeginSpeechRecognition (VVPhrases)

Event fired when the speech engine receives audio input it identifies as coming from user speech, rather than background noise.

Syntax

```
BeginSpeechRecognition(ByVal BegTime As String)
```

Parameters

BegTime

Bookmark indicating the time when the user began to speak.

Return Values

??

Remarks

The event does not necessarily mean a specific phrase in the **VVPhraseColl** Collection has been recognized; it simply indicates that the user has started speaking. If you are using the IBM Speech Engine, then this event will fire at the same time as the **SpeechRecognized** event.

See Also

“SpeechRecognized (VVPhrases)” on page 241

SpeechRecognized (VVPhrases)

Event fired by **VVPhrases** control when it recognizes one of the phrases in the **Phrase** collection (**VVPhraseColl**).

Syntax

```
SpeechRecognized (ByVal Name As String, ByVal ID As Long, Phrase As String, UpdateUIText As Boolean, ByVal BegTime As String, ByVal EndTime As String)
```

Parameters

Name

The programmer assigned unique identifier for the phrase object.

ID

A programmer assigned numeric identifier for the item.

Phrase

The actual phrase text the user spoke.

UpdateUIText

When **AutoUI** is True, this parameter tells the **VVPhrases** control to use the Phrase text to update the Word History component in the ViaVoice **UIServer**.

BegTime

A bookmark indicating the time when the user began to speak the phrase.

EndTime

A bookmark indicating the time the user finished speaking the phrase.

Return Values

Remarks

Handling this event is necessary when using the control.

Example

In Visual Basic:

```
Private Sub VVPhrases1_SpeechRecognized(ByVal Name As String, ByVal ID As
Long, ByVal Phrase As String, UpdateUIText As Boolean, ByVal BegTime As
String, ByVal EndTime As String)

    Select Case ID
    Case 100
        MsgBox "Hello Sue"
    Case 101
        MsgBox "Hello James"
    End Select
End Sub
```

In Visual C++ (MFC):

```
void CVVPhrtest::OnSpeechRecognized(LPCTSTR Name, long ID, BSTR FAR
*Phrase, BOOL FAR *UpdateUIText, LPCTSTR BegTime, LPCTSTR EndTime)
{
    switch (ID)
    {
        case 100:
            MessageBox ("Hello World" ,"VVPhrases",MB_OK);
            break;
        default:
            break;
    }
}
```

See Also

“AddPhrase (VVPhrases)” on page 234.

Chapter 25, “Introduction to the User Interface Control” on page 497.

TrainingRequired (VVPhrases)

Notification from the engine that the currently active speech user needs to train the speech engine in order to improve recognition.

Syntax

TrainingRequired (ByVal TrainingType As Long)

Parameters

TrainingType

One of the following SAPI training types (refer to the Microsoft SAPI documentation for further information).

General (SRGNSTRAIN_GENERAL)	1
Grammar (SRGNSTRAIN_GRAMMAR)	2
Grammar (SRGNSTRAIN_GRAMMAR)	4

Return Values

??

Remarks

In the case of **VVPhrases** this means that the engine is using tentative pronunciations for some of the phrases in the **VVPhraseColl** collection because it cannot find the words in its base vocabulary.

Example

In Visual Basic:

```
Private Sub VVPhrases1_TrainingRequired(ByVal TrainingType As Long)
    Select Case TrainingType
        Case 1:
            Debug.Print "General training required"
        Case 2:
            Debug.Print "Vocabulary needs to be added"
        Case 3:
            Debug.Print "Microphone levels need to be changed"
    End Select
End Sub
```

In Visual C++ (MFC):

```
void CMyWnd::OnTrainingRequired(long TrainingType)
{
    switch (TrainingType) {
        default:
            MessageBox("Training recommended.", "ViaVoice SDK", MB_OK);
    }
}
```

See Also

“Engine (VVPhrases)” on page 227

VVPhraseColl Collection

VVPhraseColl Collection Properties

The **VVPhraseColl** (**IVVPhraseColl**) Collection supports the following properties:

- **Count**
- **Enabled**
- **Item**

Count (VVPhraseColl)

Returns the number of phrases in the collection.

Syntax

```
lValue = [VVPhrases.Phrases].Count
```

Parameters

None.

Return Values

Long

The number of items in the collection

Remarks

None.

Example

In Visual Basic:

```
'Get all the phrases
Dim sPhrase As String

For i = 1 To VVPhrases1.Phrases.Count
    sPhrase = VVPhrases1.Phrases(i).Text
Next
```

In Visual C++ (MFC):

```
//Get all phrases
CString sPhrase;

for (int i = 1; i <= VVPPhrases1.GetPhrases().GetCount(); i++)
{
    VARIANT va;
    VariantInit(&va);
    va.vt = VT_I2;
    va.iVal = i;
    sPhrase = VVPPhrases1.GetPhrases().GetItem(va).GetText();
}
```

See Also

None.

Enabled (VVPhraseColl)

Turns on/off command recognition for all the commands in the collection. When this property is False the control ignores all the commands in the collection.

Syntax

```
[VVPhrases1.Phrases].Enabled = Boolean
```

Parameters

??

Return Values

TRUE

(Default) Activates command recognition for the collection.

FALSE

Turns off recognition for all the phrases in the collection.

Remarks

Changing the value of this property has no effect on the value of the **Enabled** property in each object in the collection.

Example

In Visual Basic:

```
VVPhrases1.Phrases.Enabled = False  
Call VVPhrases1.Phrases.Add("Minimize", 100, "Minimize Windows", TRUE)  
Call VVPhrases1.Phrases.Add("Maximize", 200, "Maximize Windows", TRUE)  
VVPhrases1.Phrases.Enabled = True
```

In Visual C++ (MFC):

```
m_VVPhrases1.GetPhrases().SetEnabled(FALSE);  
m_VVPhrases1.GetPhrases().Add("Minimize",100,"Minimize Windows", TRUE);  
m_VVPhrases1.GetPhrases().Add("Maximize", 200, "Maximize Windows, TRUE);  
m_VVPhrases1.GetPhrases().SetEnabled(TRUE);
```

See Also

VVPhrases Control's "Enabled (VVPhrases)" on page 225

VVPhrase Object's "Enabled (VVPhrase)" on page 265

Item (Default Method - VVPhraseColl)

Returns a **VVPhrase** object from the collection.

Syntax

```
VVPhrase = [VVPhrases1.Phrases].Item(ByVal Key As VARIANT)
```

Parameters

Key

VARIANT. The item identifier. This parameter can be numeric – indicating the ordinal position of the item within the collection, or a string – indicating the text of the item.

Return Values

VVPhrase

The **Phrase** object that contains the command text.

Remarks

Whenever you add a phrase to the control, the control creates a **VVPhrase** object to encapsulate the command text and adds it to the **VVPhraseColl** Collection. There are two ways to access a member of the collection. One way is through the item's command text. The other way is through the item's position within the collection.

Because the Phrases property is the default property of the control, and the Item function is the default method of the **VVPhraseColl** object, it is possible in Visual Basic to write:

```
VVPhrases1("Add ").ID = 100
```

Example

In Visual Basic:

```
VVPhrases1.Phrases.Item("Add").ID = 100  
Or  
VVPhrases.Phrases("Add").ID = 100
```

In Visual C++ (MFC):

```
VARIANT va;  
VariantInit(&va);  
va.bstrVal = SysAllocString(L"Add");  
va.vt = VT_BSTR  
  
m_VVPhrases.GetPhrases().GetItem(va).SetID(100);
```

See Also

None.

VVPhraseColl Collection Methods

The **VVPhraseColl** Collection supports the following methods:

- **Add**
- **Exists**
- **Remove**
- **RemoveAll**

Add (VVPhraseColl)

Creates a new phrase. By default the newly created phrase will become active immediately (if the VVPhrasesColl's **Enabled** property and the control's **Enabled** property are set to True).

Syntax

```
VVPhrase = [VVPhrases.Phrases]Add(ByVal Name As String, ByVal ID As Long,  
ByVal Phrase As String,Optional ByVal Enabled As Boolean)
```

Parameters

Name

String. A programmer assigned string name to identify the **Phrase** object. You can use this name to refer back to the object within the **Phrase** collection.

ID

Long. A programmer assigned number. This parameter enables you to associate a command or group of commands with a number.

Text

String. The actual phrase that the control will recognize (Example: "Save Record"). This parameter is not case-sensitive.

Enabled

Boolean. Enables/Disables the phrase. In Visual Basic this parameter is optional and defaulted as True.

Return Values

VVPhrase

An instance of the newly created Phrase object that encapsulates the command.

Remarks

You cannot add two phrases with the same Name. The Name serves a unique identifier for the phrase object within the collection. However, you can add many phrases with the same ID number and with the same Text.

Example

In Visual Basic:

```
'Add Commands
VVPhrases1.Phrases.Add("Begin", 100, "Begin New Record")
VVPhrases1.Phrases.Add("Add", 100, "Add New Record")
VVPhrases1.Phrases.Add("Edit", 200, "Edit Record")
VVPhrases1.Phrases.Add("Delete", 300, "Delete Record")
VVPhrases1.Phrases.Add("Trash", 300, "Trash Record")
```

In Visual C++ (MFC):

```
'Add Commands
m_VVPhrases1.GetPhrases().Add("Begin",100,"Begin New Record",TRUE)
m_VVPhrases1.GetPhrases().Add("Add", 100, "Add New Record",TRUE)
m_VVPhrases1.GetPhrases().Add("Edit", 200, "Edit Record",TRUE)
m_VVPhrases1.GetPhrases().Add("Delete", 300, "Delete Record", TRUE)
m_VVPhrases1.GetPhrases().Add("Trash", 300, "Trash Record",TRUE)
```

See Also

“Remove (VVPhraseColl)” on page 257

Exists (VVPhraseColl)

Use this method to find out if a certain phrase is part of the collection.

Syntax

```
[Boolean] = [VVPhrases1.Phrases].Exists(ByVal Key As VARIANT)
```

Parameters

Key

VARIANT. The item identifier. This parameter can be numeric – indicating the ordinal position of the item within the collection, or a string – indicating the text of the item.

Return Values

TRUE

The phrase exists in the collection;

FALSE

The phrase does not exist in the collection.

Remarks

None.

Example

In Visual Basic:

```
If VVPhrases1.Phrases.Exists("Delete") = True Then
    VVPhrases1.Phrases("Delete").Enabled = False
End If
```

In Visual C++ (MFC):

```
VARIANT va;
VariantInit(&va);
va.bstrVal = SysAllocString(L"Delete");
va.vt = VT_BSTR;

if (VVPhrases1.GetPhrases().Exists(va) )
    VVPhrases1.GetPhrases().GetItem(va).SetEnabled(FALSE);
```

See Also

None.

Remove (VVPhraseColl)

Removes a **Phrase** object from the collection.

Syntax

```
[VVPhrases.Phrases].Remove(ByVal Key As Variant)
```

Parameters

Key

VARIANT. The item identifier. This parameter can be numeric – indicating the ordinal position of the item within the collection, or a string – indicating the text of the item.

Return Values

None.

Remarks

Whenever you add a phrase to the control, the control creates a **Phrase** object to contain the command text and adds it to the **VVPhraseColl** Collection. There are two ways to remove a member of the collection. One way is through the items command text. The other way is through the item's position within the collection.

Example

In Visual Basic:

```
If VVPhrases1.Phrases.Exists("Delete") = True Then  
    VVPhrases1.Phrases.Remove("Delete")  
End If
```

In Visual C++ (MFC):

```
VARIANT va;  
VariantInit(&va);  
va.bstrVal = SysAllocString(L"Add");  
va.vt = VT_BSTR;  
  
If (VVPPhrases1.GetPhrases().Exists(va) == TRUE)  
    VVPPhrases1.GetPhrases().Remove(va);
```

See Also

None.

RemoveAll (VVPhraseColl)

Removes all the **VVPhrase** objects from the collection.

Syntax

```
[VVPhrases.Phrases].RemoveAll
```

Parameters

None.

Returns

None.

Remarks

Removes and destroys all phrase objects in the collection. If you want to only temporarily disable them, it is much faster to use the **VVPhraseColl** Collection's **Enabled** property.

Example

In Visual Basic:

```
VVPhrases1.Phrases.RemoveAll
```

In Visual C++ (MFC):

```
m_VVPhrases1.GetPhrases().RemoveAll();
```

See Also

“Remove (VVPhraseColl)” on page 257

VVPhrase Object

VVPhrase Object Properties

The **VVPhrase** object supports the following properties:

- **ActionDesc**
- **Description**
- **Enabled**
- **ID**
- **ItemData**
- **Name**
- **Text**

ActionDesc (VVPhrase)

A programmer assigned description of the action that the program will take after the user speaks the phrase. This property will be used later by other ViaVoice programs to give the user a list of commands that your program will recognize.

Syntax

```
[Phrase].ActionDesc = [String]
```

Parameters

??

Return Values

String

Any string value.

Remarks

The **ActionDesc** property does not have any effect on the functionality of the phrase object at this time. However, it is a good practice to set the value of this property in you programs since the future releases of the SDK will contain programs that will display a list of all the commands in your application along with the description, and the action description.

Example

In Visual Basic:

```
VVPhrases1.Phrases("Save").ActionDesc = "Save Current Record"
```

In Visual C++ (MFC):

```
VARIANT va;  
VariantInit(&va);  
va.vt = VT_BSTR;  
va.bstrVal = SysAllocString(L"Save");  
  
VVPhrases1.GetPhrases().GetItem(va).SetActionDesc("Writes the current  
record to the database.");
```

See Also

“Description (VVPhrase)” on page 263

Description (VVPhrase)

A programmer assigned description of the command. This property will be used later by other ViaVoice programs to give the user a list of commands that your program will recognize.

Syntax

```
[Phrase].Description = [String]
```

Parameters

??

Return Values

String

Any string value.

Remarks

The **Description** property does not have any effect on the functionality of the **Phrase** object at this time. However, it is a good practice to set the value of this property in you programs since the future releases of the SDK will contain programs that will display a list of all the commands in your application along with the description, and the action description.

Example

In Visual Basic:

```
VVPhrases1.Phrases("Save").Description = "Save Current Record"
```

In Visual C++ (MFC):

```
VARIANT va;  
VariantInit(&va);  
va.vt = VT_BSTR;  
va.bstrVal = SysAllocString(L"Save");  
  
VVPhrases1.GetPhrases().GetItem(va).SetDescription("Save Current  
Record");
```

See Also

“ActionDesc (VVPhrase)” on page 261

Enabled (VVPhrase)

Enables/disables speech recognition for a particular phrase.

Syntax

```
[Phrase].Enabled = [Boolean]
```

Parameters

??

Return Values

TRUE

(Default) Activates command recognition for a single phrase.

FALSE

Turns off recognition for a single phrase.

Remarks

If the user speaks the phrase and it is enabled, the **VVPhrases** control will fire the **SpeechRecognized** event. If the phrase is disabled the control simply ignores the phrase.

Setting the collection's **Enabled** property to True/False will not affect the value of the **Enabled** property of each individual phrase. However, it will cause the control to ignore the command. In other words, if the Phrase's **Enabled** property is True, and then you set the Collection's **Enabled** property to False, the Phrase's **Enabled** property will remain True but the control will not recognize it.

Example

In Visual Basic:

```
VVPhrases1.Phrases("Speak").Enabled = False
```

In Visual C++ (MFC):

```
VARIANT va;  
VariantInit(&va);  
va.bstrVal = SysAllocString(L"Speak");  
va.vt = VT_BSTR;  
  
m_VVPhrases1.GetPhrases().GetItem(va).Enabled = False;
```

See Also

VVPhraseColl's "Enabled (VVPhraseColl)" on page 248

ID (VVPhrase)

A programmer assigned identifier for the phrase.

Syntax

```
Phrase.ID = [long]
```

Parameters

??

Return Values

Long

Any long numeric value.

Remarks

The **ID** number does not have to be unique. It offers a more reliable way to refer to the command than using the command text. For example, your program may have the same command in different languages, but has the same **ID** for all versions of the item. The **ID** number also enables you to group similar commands by functionality, such as “Say Hello World,” and “Hello World.”

Example

In Visual Basic:

```
Private Sub VVPhrases1_SpeechRecognized(ByVal Name As String, ByVal ID As
Long, ByVal Phrase As String, UpdateUIText As Boolean, ByVal BegTime As
String, ByVal EndTime As String)

    Select Case ID
    Case 100
        MsgBox "Hello Sue"
    Case 101
        MsgBox "Hello James"
    End Select
End Sub
```

In Visual C++ (MFC):

```
void CVVPhrtest::OnSpeechRecognized(LPCTSTR Name, long ID, BSTR FAR
*Phrase, BOOL FAR *UpdateUIText, LPCTSTR BegTime, LPCTSTR EndTime)
{
    switch (ID)
    {
        case 100:
            MessageBox ("Hello World" ,"VVPhrases",MB_OK);
            break;
        default:
            break;
    }
}
```

See Also

None.

ItemData (VVPhrase)

Stores additional programmer-defined data with an individual phrase.

Syntax

```
[Phrase].ItemData = [Variant]
```

Parameters

??

Return Values

Variant

Any values or object reference.

Remarks

This data value is not used by the **Phrases** control at anytime. Instead, its meaning is only known by the client application, which stores it.

Example

In Visual Basic:

```
Set VVPhrases1.Phrases("Save").ItemData=VVTextBox1
```

In Visual C++ (MFC):

```
CVVTextBox VVTextBox1;  
...  
VARIANT va1, va2;  
VariantInit(&va1);  
VariantInit(&va2);  
va1.vt = VT_BSTR;  
va2.vt = VT_BYREF | VT_DISPATCH;  
va1.bstrVal = SysAllocString(L"Save");  
VVTextBox1.AddRef();  
va2.ppvUnknown = &VVTextBox1;  
  
VVPhrases1.GetPhrases().GetItem(va1).SetItemData(va2);
```

See Also

None.

Name (VVPhrase)

A programmer-assigned string name for the **Phrase** object.

Syntax

```
[Phrase].Name = [String]
```

Parameters

??

Return Values

String

Any string value. The name property is not case sensitive.

Remarks

You can use this name later in your code to search for a particular **Phrase** object.

Although you can change the name of the **Phrase** object at any time, to find the object in the **Phrase** Collection, you must use the name assigned at the time when the object was first created.

Example

In Visual Basic:

```
Dim Phrase As VVPhrase

VVPhrases1.AddPhrase "Hello",100, "Say Hello World", TRUE
VVPhrases1.AddPhrase "Goodbye", 150, "Say Goodbye World", TRUE
For Each Phrase In VVPhrases1.Phrases
    List1.AddItem Phrase.Name
Next
```

In Visual C++ (MFC):

```
VARIANT va;
VariantInit (&va);
va.vt = VT_I4;
for (va.lVal= 1; va.lVal<=
m_VVPhrases.GetPhrases().GetCount();va.lVal++)
{
    TRACE("%s\n", m_VVPhrases.GetPhrases().GetItem(va).GetText());
}
```

See Also

“Add (VVPhraseColl)” on page 253

“AddPhrase (VVPhrases)” on page 234

Text (VVPhrase)

Exact text of the user speech to be recognized.

Syntax

```
[Phrase].Text = [String]
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
VVPhrase1.Phrases("Save").Text="Save me now"
```

In Visual C++ (MFC):

```
VARIANT va;  
VariantInit(&va);  
va.vt = VT_BSTR;  
va.bstrVal = SysAllocString(L"Save");  
  
VVPhrases1.GetPhrases().GetItem(va).SetText("Save me now");
```

See Also

None.

VVPhrase Object Methods

There are no methods for this object.

VVPhrase Object Events

There are no events for this object.

Phrases Control Frequently Asked Questions

This chapter contains answers to the most frequently asked questions about the ViaVoice **Phrases Control**.

Can I put more than one phrase control on a form?

Yes. In fact it can be more efficient to group related phrases in separate controls.

What characters are permitted in a phrase?

The characters permitted in a phrase can vary depending on the SAPI speech engine you are using. As a general rule, however, non-printing ASCII characters and punctuation marks could render a phrase unrecognizable. In the case of the IBM speech engine, conventional punctuation marks such as the comma, apostrophe, and exclamation point usually do not interfere with recognition unless they are placed at or near the beginning of a word.

How can I improve load time when adding many phrases to a VVPhrases control?

When the **Phrases** collection of the control is enabled, each call to `AddPhrase` or `Phrases.Add` updates the speech engine with the new phrase immediately. If you are adding many phrases to the control at the same time it is more efficient to set `Phrases.Enabled = False`, perform all the `AddPhrase` calls you desire, and set `Phrases.Enabled = True` when finished. The speech engine will then be updated only once – after all the phrases have been added.

How can I improve phrase recognition?

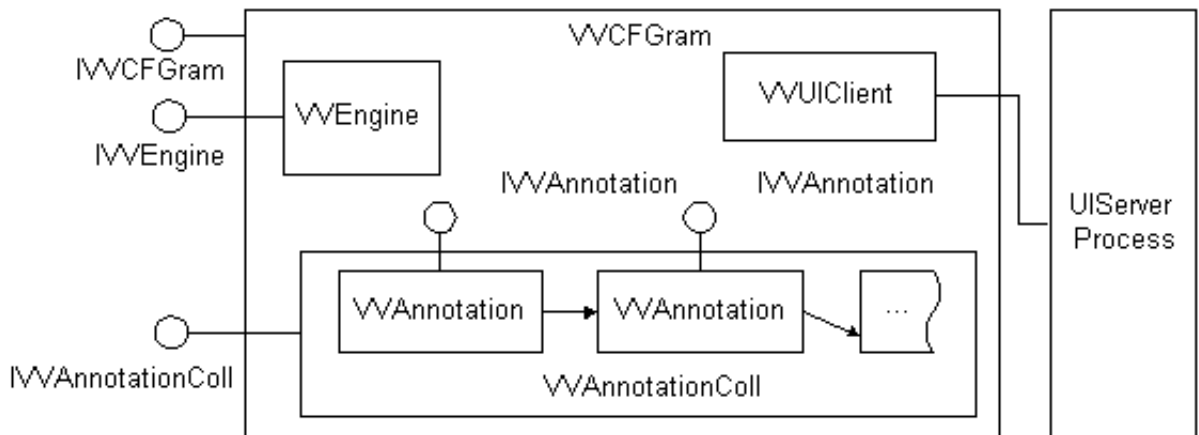
The fewer phrases added to an enabled phrase control, the fewer possibilities the speech engine has to sort through to find a match with the user's speech. Also, the more distinct the phrases are in sound and spelling the better the likelihood that the speech engine will not confuse them.

Introduction to the Grammar Control

The ViaVoice **Context-Free Grammar Control (VVCFGram)** is an ActiveX control that enables developers to use a compiled context-free grammar file to add robust command recognition to their application. The main idea behind the control is that the developer provides the control with a compiled grammar file, and the control will notify the developer when the user speaks a command constructed from the grammar.

VVCFGram Object Hierarchy

The following diagram shows the components of the **VVCFGram** Control.



Getting Started with the Grammar Control

The following is a tutorial on how to incorporate the **VVCFGram** control into your Visual Basic or Visual C++ applications. This tutorial is designed to present you with the most commonly used properties and events in the **VVCFGram** control.

Creating an Instance of the Control

In Visual Basic:

To add the **VVCFGram** control to your application, do the following:

1. From the **Project** menu, choose **Components**.
The Components dialog box, Figure 20, appears. The Components dialog lists all the ActiveX Controls that you can use in your application.

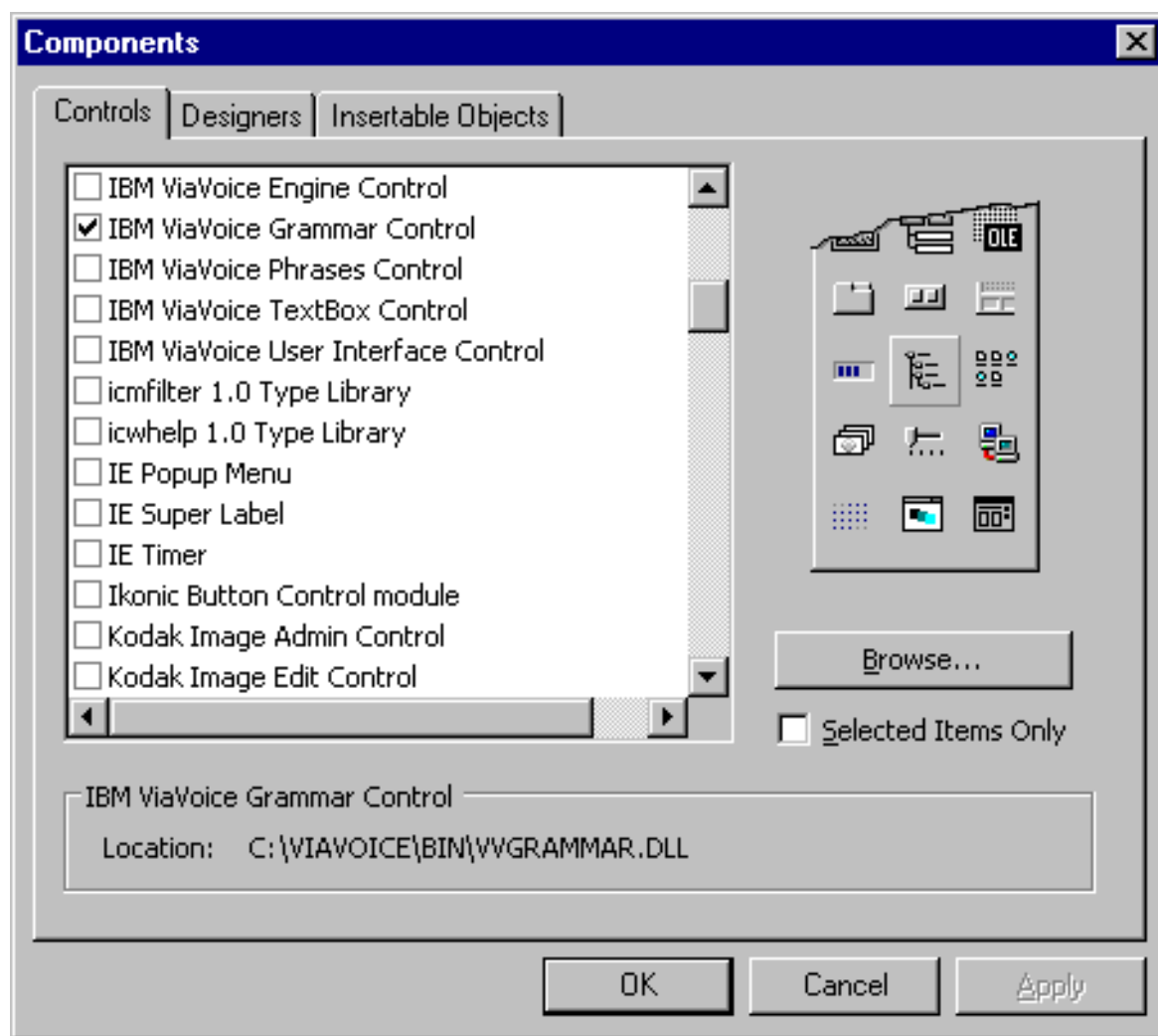


Figure 20. Component Selection Dialog - Visual Basic

2. Select **IBM ViaVoice Grammar Control** from the list and choose **OK**.
Visual Basic adds the control to your project, and adds a new icon to the toolbar (Figure 21).



Figure 21. VVCFGram Control Toolbar Icon

3. Add an instance of the **VVCFGram** control to your form.
The **VVCFGram** control is an invisible control at run time.

In Visual C++ (MFC):

To add the **VVCFGram** control to your MFC project, do the following:

1. From the **Project** menu, select **Add To Project**, then select **Components and Controls**.
The Components and Controls Gallery dialog box, Figure 22, appears.

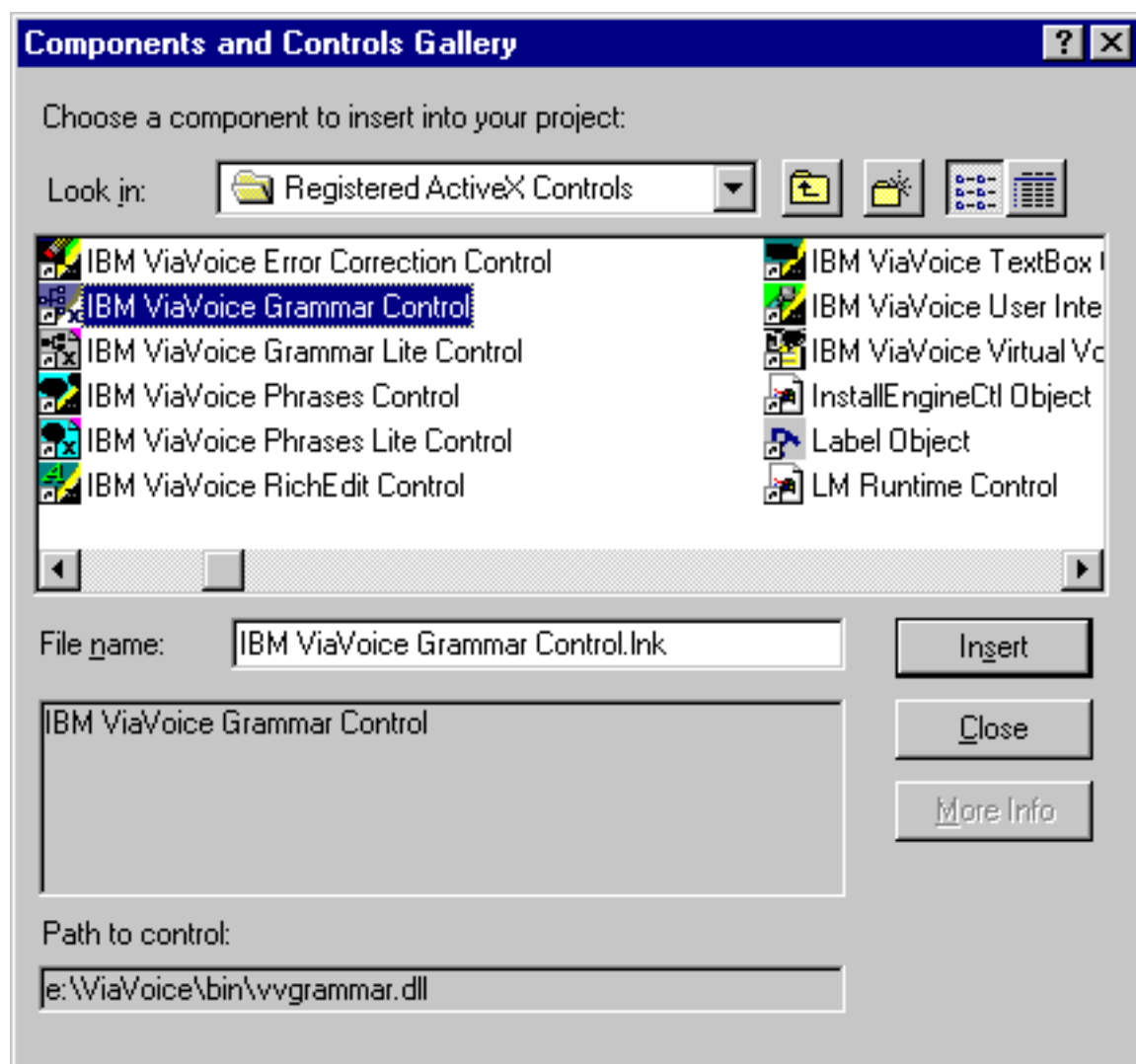


Figure 22. Insert ActiveX Control Dialog Box - Visual C++

2. Double-click the Registered ActiveX Controls folder in the dialog box.
3. Select the **IBM ViaVoice Grammar Control** icon in the list of controls, then click **Insert**.

A confirmation message box appears, asking “Insert this component?” A confirmation message box appears, asking “Insert this component?”

4. Respond to the confirmation message box by choosing **OK**.

The Confirm Classes dialog box, Figure 23, appears listing the components in the **VVCFGram** control: CVVCFGram, CVVAnnotation, CVVRule, CVVBookmarkColl, CVVBookmark, CVVAnnotationColl, CVVEngine, and CVVRuleColl. (The CVVRule, CVVRuleColl, CVVBookmarkColl, and CVVBookmark will be implemented in the future.)

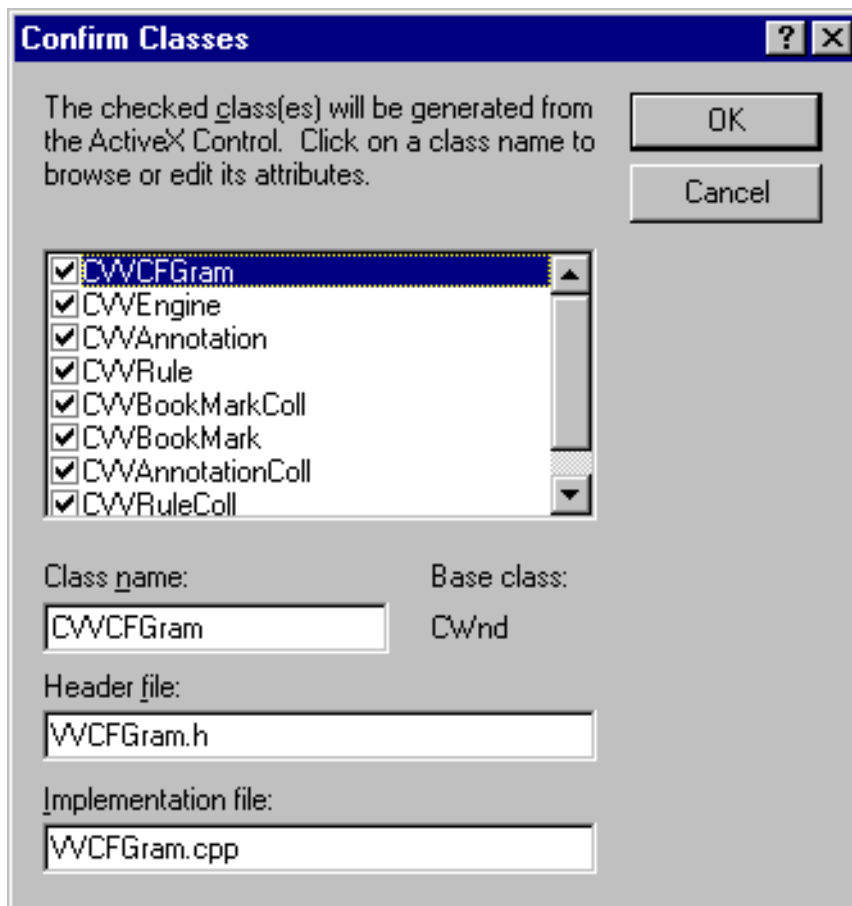


Figure 23. Confirm Classes Dialog Box

5. Click **OK** in the Confirm Classes dialog box.
6. Close the “Components and Controls Gallery” dialog box.
If you examine the Project Workspace window in the class view you will notice eight new classes: **VVCFGram** control: CVVCFGram, CVVAnnotation, CVVRule, CVVBookmarkColl, CVVBookmark, CVVAnnotationColl, CVVEngine, and CVVRuleColl (assuming you accepted the default names for the class in the Confirm Classes dialog box).
7. In the resource view of your Project Workspace window double-click the dialog resource entry where you wish to insert the **VVCFGram** control.
The **VVCFGram** icon, Figure 24, appears in the Controls toolbar.



Figure 24. VVCFGram Icon in the Controls Toolbar

8. Add an instance of the **VVCFGram** control to the dialog box.
After you add the **VVCFGram** control to your dialog you can invoke Class Wizard to create a member variable for you class of type CVVCFGram. You might also decide to capture the events in the control by adding Event handlers to your dialog class. To add Event handlers, click the **VVCFGram** control with the right mouse button, then select **Events** from the pop-up menu.

Drag-Drop-n-Go Support

The **VVCFGram** control fully incorporates IBM's Drag-Drop-n-Go technology. Through this technology you can add the control to your application and use it without having to write many lines of code.

The **VVCFGram** control has the following Drag-Drop-n-Go properties:

Property	Type	Default Value
AutoConnect	Boolean	True
AutoUI	Boolean	True
AutoLoad	Boolean	True

AutoConnect when set to True causes the control to automatically search for a speech recognition engine at run time and connect to it. This is necessary before the control can recognize commands and phrases. If you set this property to False, you must connect manually by calling the `Event.Connect` method of the control.

AutoUI, when set to True, causes the **VVCFGram** control to display and manipulate the **ViaVoice User Interface Server**. For more information on the **ViaVoice User Interface Server**, refer to the **ActiveX User Interface** control documentation.

AutoLoad, when set to True, causes the control to automatically load a grammar file. Use the **GrammarSource** property to specify the path to the grammar file. Setting this property to True is equivalent to issuing the **LoadFromSource** method at run time.

By leaving the property values set to their default values you guarantee that the control will connect to the speech recognition engine and display and interact with the **IBM ViaVoice User Interface Server** automatically. Furthermore, if you specify a grammar file (by setting the **GrammarSource** property), the control will automatically load the compiled grammar file and activate it.

Loading a Grammar

The first step in using the **VVCFGram** control is to specify the grammar source. In this release the grammar source is a compiled grammar file, whether it uses IBM native grammar format, or SAPI 4.0 format. (For information on compiling grammars, see the SMAPI Reference or the SAPI Reference included with this SDK). Guide Simply set the **GrammarSource** property to the path of the grammar file.

If you set the **AutoLoad** property to True, the control will automatically load the grammar file and activate it. Otherwise, you must issue the **LoadFromSource** method at run time, as seen below.

In Visual Basic:

```
VVCFGram1.GrammarSource = "C:\Grammars\sample.grm"  
VVCFGram1.LoadFromSource
```

In Visual C++ (MFC):

```
VVCFGram1.SetGrammarSource("C:\\Grammars\\sample.grm");  
VVCFGram1.LoadFromSource();
```

The control then is ready to interpret expressions from the grammar file. As soon as the control hears the user speak one of the phrases defined in the grammar, it triggers the **SpeechRecognized** event. You can then write code in this event to handle the command in your application.

In Visual Basic:

```
Private Sub VVCFGram1_SpeechRecognized(ByVal Name As String, _  
    ByVal ID As Long, ByVal Phrase As String, _  
    UpdateUIText As Boolean, ByVal BegTime As String, _  
    ByVal EndTime As String)  
  
    If Phrase="Exit Program" Then  
        Call CloseDatabase  
        Call ExitProgram  
    End If  
  
End Sub
```

In Visual C++ (MFC):

```
void CVVGramtest::OnSpeechRecognized(LPCTSTR Name,  
    long ID, BSTR FAR *Phrase, BOOL FAR *UpdateUIText,  
    LPCTSTR BegTime, LPCTSTR EndTime)  
{  
  
    If (wcscmp (Phrase, L"Exit Program") ==0)  
    {  
        CloseDatabase();  
        ExitProgram();  
    }  
}
```

Enabling/Disabling a Grammar

During the execution of your application, it may be necessary at times to turn on or off speech recognition for a particular grammar.

The **VVCFGram** control has an **Enabled** property. This property lets you turn on or off speech recognition for the current grammar.

Using External Lists

It is possible to create a grammar that contains a placeholder for words that can only be resolved at run time. For example, suppose that in your program you wish the user to issue the following verbal commands: “Delete Files from Temp”, or “Copy Files From Temp to Download.” In these two examples the user is able to copy files or delete files from any directory on his or her computer. However, it is not possible to account for all the directory names the user might have at the time the grammar is compiled (at design time). Instead, it is possible to create a grammar with a placeholder for the directory name, then at run time after loading the grammar, your program can provide the engine with a list of words to use in the directory placeholder. The source code for such a grammar might look like the following:

In **TXT** format:

```
[Grammar]
langid=1033
type=cfg

[Lists]
=DirectoryA
=DirectoryB

[<Start>]
<Start> = Delete [opt] <object> from <DirectoryA>
<Start> = Copy [opt] <object> from <DirectoryB>

<object> = Files
<object> = All
```

In **BNF** format:

```
EXTERN <DirectoryA>
EXTERN <DirectoryB>

<start> = <delete> | <copy>.

<delete>=Delete <object>? from <DirectoryA>.
<copy>=Copy <object>? from <DirectoryA> to <DirectoryB>.

<object> = FILES | ALL.
```

The placeholders in a grammar file are called **External Lists**. Each external list has a unique name. In the example above, “DirectoryA” and “DirectoryB” are external lists. The grammar control has an **ExternLists** property that enables you to manage the words in an external list.

The **ExternLists** property in the grammar control provides an instance of the **Phrase Collection Group** Object (**IVVPhraseCollGroup**). Through this object you are able to enable, disable, and enumerate external lists in your control. For example, to provide words for the “DirectoryA” and “DirectoryB” external lists in the grammar example above, you would first load the grammar using the **LoadFromSource** method as normally done. Then, add words to the two external lists as follows:

In Visual Basic:

```
With VVCFGram1.ExternLists("DirectoryA")
    .Add "C:\Junk", 100, "Junk", True
    .Add "C:\Temp", 200, "Temp", True
End With

With VVCFGram1.ExternLists("DirectoryB")
    .Add "C:\Download",100, "Download", True
    .Add "C:\Temp",200,"Temp", True
    .Add "C:\Windows",300, "Windows", True
    .Add "C:\Windows\System",300,"System", True
End With
```

In Visual C++ (MFC):

```

CVVPhraseColl DirectoryA;
CVVPhraseColl DirectoryB;

VARIANT vaItemA
VariantInit(&vaItemA);
vaItemA.vt=VT_BSTR;
vaItemA.bstr.Val=SysAllocString(L "DirectoryA");

DirectoryA = m_VVCFGram1.GetExternLists().GetItem(vaItemA);
DirectoryA.Add(L"C:\\Junk", 100, L"Junk", TRUE);
DirecotryA.Add(L"C:\\Temp", 200, L"Temp", TRUE);

VARIANT vaItemB;
VariantInit(&vaItemB);
vaItemB.vt=VT_BSTR;
vaItemB.bstrVal=SysAllocString(L"DirectoryB");

DirectoryB = m_VVCFGram1.GetExternLists().GetItem(vaItemB);
DirectoryB.Add(L"C:\\Download",100, L"Download", TRUE);
DirectoryB.Add(L"C:\\Temp",200,L"Temp", TRUE);
DirectoryB.Add(L"C:\\Windows",300, L"Windows", TRUE);
DirectoryB.Add(L"C:\\Windows\\System",300,L"System", TRUE);

```

Adding words or phrases to each external list is identical to adding phrases to the phrases control (see “Adding Phrases” on page 212 for more information). The first parameter in the **Add** command is the unique name of the phrase item within the phrase collection. The second parameter is a numeric ID for the phrase. (This number does not have to be unique.) The third parameter is the actual word or phrase the user will speak to complete the command. In the example above, the statement. Add “C:\Windows\System”, 300, “System”, True tells the grammar control that the user can speak the phrase “Copy Files from Temp to System.” The fourth parameter tells the grammar control if the phrase is enabled (True) or disabled (False).

You can enable or disable phrases within the external list at any time by changing the **Enabled** property of the phrase item as follows:

In Visual Basic:

```
VVCFGram1.ExternLists("DirectoryB")("C:\Windows").Enabled = False
```

In Visual C++ (MFC):

```
VARIANT vaExtName;  
VariantInit(&vaExtName);  
vaExtName.vt = VT_BSTR;  
vaExtName.bstrVal = SysAllocString(L"DirectoryB");  
  
VARIANT vaPhrase;  
VariantInit(&vaPhrase);  
vaPhrase.vt = VT_BSTR;  
vaPhrase.bstrVal = SysAllocString(L"C:\Windows");  
m_VVCFGram1.GetExternLists().GetItem(vaExtName).GetItem(vaPhrase).SetEnabled (FALSE);
```

Changing the **Enabled** property for a phrase causes the grammar control to unload the external list that contains that phrase and reload it without the disabled phrase. You can also disable the entire list by issuing the following command:

In Visual Basic:

```
VVCFGram1.ExternLists("DirectoryB").Enabled = False
```

In Visual C++ (MFC):

```
VARIANT vaExtName;  
VariantInit(&vaExtName);  
vaExtName.vt = VT_BSTR;  
vaExtName.bstrVal = SysAllocString(L"DirectoryB");  
m_VVCFGram1.GetExternLists(vaExtName).SetEnabled(FALSE);
```

The following is a list of properties, methods, and events pertaining to the **VVCFGrammar** Control.

Grammar Control Properties

The ViaVoice **Grammar** control supports the following properties:

- **Alternates**
- **Annotations**
- **AutoConnect**
- **AutoLoad**
- **AutoUI**
- **Enabled**
- **Engine**
- **ExternLists**
- **GrammarFormat**
- **GrammarSource**
- **Refresh**
- **Rules**
- **SourceType**

Alternates (VVCFGram)

??

Syntax

In Visual Basic:

In Visual C++ (MFC):

In Visual C++:

Parameters

??

Return Values

??

Remarks

Example

In Visual Basic:

In Visual C++ (MFC):

In Visual C++:

See Also

Annotations (VVCFGram)

??

Syntax

In Visual Basic:

```
VVAnnotationColl = [VVCFGram].Annotations
```

In Visual C++ (MFC):

```
CVVAnnotationColl AnnColl = [CVVCFGram].Annotations();
```

In Visual C++:

```
IVVAnnotationColl *pIAnnColl;  
HRESULT hr = pVVCFGram -> get_Annotations(&pIAnnColl);
```

Parameters

??

Return Values

??

Remarks

Grammars enable you to associate a word with a related piece of information. When the **VVCFGram** control recognizes a phrase and fires the **SpeechRecognized** event, it also refreshes the **Annotation** collection. This collection contains the annotations for each of the words in the phrase recognized. The **Annotations** property returns an **Annotation Collection** object.

The **Annotation Collection** contains instances of the **Annotation** object, and each of which contains a word. The **Annotation Collection** object (**VVAnnotationColl**) has two methods for navigating through its members: **Count** and **Item**. The following code shows you how to retrieve the annotations in the **SpeechRecognized** event.

Although this property is available throughout the lifetime of the control, its value only makes sense when **SpeechRecognized** event takes place.

Example

In Visual Basic:

```
Private Sub VVCFGram1_SpeechRecognized(ByVal Name As String, _
    ByVal ID As Long, Phrase As String, _
    UpdateUIText As Boolean, _
    ByVal BegTime As String, ByVal EndTime As String)

    Dim oAnnotation As VVAnnotation
    Dim sAnnotation As String

    If VVCFGram1.Annotations.Count > 0 Then

        For Each oAnnotation In VVCFGram1.Annotations
            sAnnotation = oAnnotation.Word
        Next

    End If

End Sub
```

In Visual C++ (MFC):

```
void CVVPhrtest::OnSpeechRecognized(LPCTSTR Name, long ID,
    BSTR FAR *Phrase, BOOL FAR *UpdateUIText,
    LPCTSTR BegTime, LPCTSTR EndTime)
{
    CString sAnnotation;

    If (m_VVCFGram.GetAnnotations().GetCount > 0)
    {
        for (int i=1; i <= m_VVCFGram1.GetAnnotations().GetCount();
            i++)
        {
            VARIANT va;
            VariantInit(&va);
            va.iVal = i;
            va.vt = VT_I2;

            m_VVCFGram1.GetAnnotations().GetItem(va).GetWord
                (&sAnnotation);
        }
    }
}
```

In Visual C++:

```

HRESULT SpeechRecognized (BSTR Name, long ID, BSTR *Phrase, VARIANT_BOOL
*UpdateUIText, BSTR BegTime, BSTR EndTime)
{
    IVVAnnotationColl *pIAnnColl;
    long                lCount = 0;
    HRESULT             hr = S_OK;

    hr = pVVCFGram->get_Annotations(&pIAnnColl);
    if (FAILED(hr)) return hr;

    hr = pIAnnColl->get_Count(&lCount);
    if (FAILED(hr)) return hr;

    if (lCount > 0)
    {
        for (long l=1; l <= lCount; l++)
        {
            VARIANT vaItem;
            VariantInit(&vaItem);
            va.vt = VT_I4;
            va.lVal = l;

            IVVAnnotation    *pIAnn;
            BSTR              sAnnotation;

            hr = pIAnnColl->get_Item(vaItem,&pIAnn);
            if (FAILED(hr)) return hr;

            pIAnn->get_Word(&sAnnotation);

            //Place code to use annotation here
        }
    }
}

```

See Also

“SpeechRecognized (VVCFGram)” on page 339

AutoConnect (VVCFGram)

Automatically connects to the speech recognition engine when created.

Syntax

In Visual Basic:

```
[VVCFGram].AutoConnect = [Boolean]
```

In Visual C++ (MFC):

```
VARIANT_BOOL = [CVVCFGram].GetAutoConnect()  
[CVVCFGram].SetAutoConnect(VARIANT_BOOL)
```

In Visual C++:

```
HRESULT[pIVVCFGram] -> get_AutoConnct(VARIANT_BOOL *)  
HRESULT[pIVVCFGram] -> put_AutoConnect(VARIANT_BOOL)
```

Parameters

??

Return Values

TRUE

(Default) The control automatically finds and connects to a speech engine.

FALSE

The control does not connect to a speech engine. You must issue the **Connect** method to cause the control to connect to the engine.

Remarks

Before the **VVCFGram** control can recognize commands, it needs to connect to a speech engine. When this property is set to True the control will try to connect to an engine with the following

properties: IBM Manufacturer, continuous speech engine, with dictation, limited domain grammar, and context free grammar support. If you wish to override the default behavior, set this property to False; then at run time modify the properties in the **Engine** object and call the **Connect** method. Changing the value of this property at run time has no effect. This property is meant to be used only at design time.

If **AutoConnect** is True, the control will connect to the speech engine before the Form_Load event takes place in Visual Basic and before the **InitDialog** method gets executed in an MFC application.

Examples

None.

See Also

“Engine (VVCFGram)” on page 311

AutoLoad (VVCFGram)

Loads a binary grammar file at run time.

Syntax

In Visual Basic:

```
[VVCFGram].AutoLoad = [Boolean]
```

In Visual C++ (MFC):

```
VARIANT_BOOL = [CVVCFGram].GetAutoLoad();  
[CVVCFGram].SetAutoLoad(VARIANT_BOOL)
```

In Visual C++:

```
HRESULT [pIVVCFGram] ->get_AutoLoad(VARIANT_BOOL *)  
HRESULT [pIVVCFGram] ->put_AutoLoad(VARIANT_BOOL)
```

Parameters

??

Return Values

TRUE

(Default) Automatically load a binary grammar file at run time. You must set the **GrammarSource** property to the full path of the binary file.

FALSE

VVCFGram does not load the binary grammar file automatically at run time. To load the grammar file you must issue the **LoadFromSource** method.

Remarks

You must specify the path to the grammar by setting the value of the **GrammarSource** property. **AutoLoad** only happens once when the control is first created. Changing the **GrammarSource** after **AutoLoad** has occurred does not automatically load the new file. To activate the new file, simply issue the **LoadFromSource** method.

Examples

None.

See Also

“GrammarSource (VVCFGram)” on page 320

AutoUI (VVCFGram)

Displays and interacts with the IBM ViaVoice **User Interface Server**.

Syntax

In Visual Basic:

```
[VVCFGram].AutoUI = [Boolean]
```

In Visual C++ (MFC):

```
VARIANT_BOOL = [CVVCFGram].GetAutoUI()  
[CVVCFGram].SetAutoUI(VARIANT_BOOL)
```

In Visual C++:

```
HRESULT[pIVVCFGram] ->Get_AutoUI(VARIANT_BOOL *)  
HRESULT[pIVVCFGram] ->Put_AutoUI(VARIANT_BOOL)
```

Parameters

??

Return Values

TRUE

(Default) **VVCFGram** displays the **User Interface Server** and interacts with it automatically.

FALSE

VVCFGram does not display the **User Interface Server**. It does not interact automatically with it either (if another control displays the **User Interface Server**, for example).

Remarks

If multiple instances of the **VVCFGram** control have **AutoUI** set to True, the **User Interface Server** only gets created once, and all the instances of the control interact with the same **User Interface**

Server. If you prefer not to display the **User Interface Server** or not to have the **VVCFGram** control interact with it automatically, set this property to False. When **AutoUI** is True, the **VVCFGram** automatically updates the following components: Microphone, Word History, and Volume Level.

Examples

None.

See Also

Chapter 18, “Introduction to the User Interface Control” on page 319.

Refer to the following chapters for more information about ViaVoice **User Interface Control**:

Chapter 25, “Introduction to the User Interface Control” on page 497

Chapter 26, “Getting Started with the User Interface Control” on page 499

Chapter 27, “Classes, Structures, and Enumerations” on page 533

Chapter 28, “Properties, Methods, and Events” on page 561

Chapter 29, “User Interface Control Frequently Asked Questions” on page 629

Enabled (VVCFGram)

Enables/Disables command recognition.

Syntax

In Visual Basic:

```
[VVCFGram].Enabled = [Boolean]
```

In Visual C++ (MFC):

```
VARIANT_BOOL = [CVVCFGram].GetEnabled()  
[CVVCFGram].SetEnabled (VARIANT_BOOL)
```

In Visual C++:

```
HRESULT [pIVVCFGram] ->get_Enabled(VARIANT_BOOL *)  
HRESULT [pIVVCFGram] ->put_Enabled(VARIANT_BOOL)
```

Parameters

??

Return Values

TRUE

(Default) **VVCFGram** listens for commands.

FALSE

VVCFGram does not listen for commands.

Remarks

Changing the value of this property does not unload the grammar from memory, but only deactivates it.

Example

In Visual Basic:

```
Private Sub ToggleSpeech_Click ( )  
    VVCFGram1.Enabled = Not VVCFGram1.Enabled  
End Sub
```

In Visual C++ (MFC):

```
void SpeechEnabledToggle()  
{  
    m_VVCFGram.SetEnabled(!m_VVCFGram.GetEnabled());  
}
```

In Visual C++:

```
void SpeechEnabledToggle()  
{VARIANT_BOOL bVal;  
    pIVVCFGram ->get_Enabled(&bVal);  
    pIVVCFGram ->put_Enabled(bVal);  
}
```

See Also

None.

Engine (VVCFGram)

Contains a reference to the ViaVoice **Engine** control (**VVEngine**).

Syntax

In Visual Basic:

```
[VVCFGram].Engine
```

In Visual C++ (MFC):

```
CVVEngine = [CVVCFGram].GetEngine()  
[CVVCFGram].SetEngine(CVVEngine)
```

In Visual C++:

```
HRESULT [pIVVCFGram] ->get_Engine(IVVEngine **)  
HRESULT [pIVVCFGram] ->put_Engine(IVVEngine *)
```

Parameters

??

Return Values

None.

Remarks

If **AutoConnect** is True, the engine property will refer to a connected **Engine** object at run-time; otherwise, the **Engine** object is disconnected. When **AutoConnect** is False, the desired properties of the engine can be set – for instance the speaking style as discrete or continuous – and then Engine.Connect can be called to start up a speech engine with the desired attributes.

The **Engine** property is actually holding an implicitly created ActiveX control (**VVEngine**), which can also be created separately. Inserting a **VVEngine** control in a project enables you to set the engine properties on this control, call connect, and then assign the resulting connected engine to multiple **VVPhrases**, **VVCFGram**, and **VVTextBox** controls. The **AutoConnect** or **AutoInit** property must be False for all controls besides the **VVEngine** control, however. Then the **VVEngine** control can be assigned to the writable engine property of **VVPhrases**, **VVTextBox**, and **VVCFGram**.

Example

In Visual Basic:

```
Private Sub Form_Load()  
VVCFGram1.Engine.NeedsDictation = True  
VVCFGram1.Engine.AudioSourceType = vvFixedAudio  
VVCFGram1.Engine.Connect  
End Sub
```

In Visual C++ (MFC):

```
void StartEngine()  
{  
m_VVCFGram.GetEngine().SetNeedsDictation (TRUE)  
m_VVCFGram.GetEngine().SetAudioSourceType(vvFixedAudio);  
m_VVCFGram.GetEngine().Connect();  
}
```


In Visual C++:

```
void StartEngine()
{
    IVVEngine *pIVVEngine;
    HRESULT hr;

    hr = pIVVCFGram->get_Engine(&pIVVEngine);
    if (FAILED(hr)) return hr;

    pIVVEngine->put_NeedsDictation(VARIANT_TRUE);
    pIVVEngine->put_AudioSourceType(vvFixedAudio);
    pIVVEngine->Connect();
}
```

See Also

Refer to the ViaVoice Engine Control Guide for more information.

ExternLists (VVCFGram)

Accesses the **Phrase Collection Group** object (**IVVPhraseCollGroup**) in the **Grammar** control.

Syntax

In Visual Basic:

```
[VVCFGram].ExternLists
```

In Visual C++ (MFC):

```
CVVPhraseCollGroup = [CVVCFGram].GetExternLists
```

In Visual C++:

```
HRESULT [pIVVCFGram] ->get_ExternLists(IVVPhraseCollGroup **)
```

Parameters

??

Return Values

None.

Remarks

You can use this object to provide external lists for the control at run time. For a complete description of this property and how to use external lists see the section titles “Using External Lists” in this chapter. The **IVVPhraseCollGroup** exposed through this property is a group of **Phrase Collections**. The **Phrase Collection** object is discussed in detail on “Adding Phrases” on page 212.

Example

In Visual Basic

```
With VVCFGram1.ExternLists("DirectoryA")
    .Add "C:\Junk", 100, "Junk", True
    .Add "C:\Temp", 200, "Temp", True
End With

With VVCFGram1.ExternLists("DirectoryB")
    .Add "C:\Download",100, "Download", True
    .Add "C:\Temp",200,"Temp", True
    .Add "C:\Windows",300, "Windows", True
    .Add "C:\Windows\System",300,"System", True
End With
```

In Visual C++ (MFC):

```
CVVPhraseColl DirectoryA;
CVVPhraseColl DirectoryB;

VARIANT vaItemA
VariantInit(&vaitemA);
vaItemA.vt=VT_BSTR;
vaItemA.bstr.Val=SysAllocString(L "DirectoryA");

DirectoryA = m_VVCFGram1.GetExternLists().GetItem(vaItemA);
DirectoryA.Add(L"C:\Junk", 100, L"Junk", TRUE);
DirecotryA.Add(L"C:\Temp", 200, L"Temp", TRUE);

VARIANT vaItemB;
VariantInit(&vaItemB);
vaItemB.vt=VT_BSTR;
vaItemB.bstrVal=SysAllocString(L "DirectoryB");

DirectoryB = m_VVCFGram1.GetExternLists().GetItem(vaItemB);
DirectoryB.Add(L"C:\Download",100, L"Download", TRUE);
DirectoryB.Add(L"C:\Temp", 200, L"Temp", TRUE);
DirectoryB.Add(L"C:\Windows", 300, L"Windows", TRUE);
DirectoryB.Add(L"C:\Windows\System", 300, L"System", TRUE);
```

In Visual C++:

```
HRESULT hr;
IVVPhraseCollGroup *pExtLists;
hr = pIVVCFGram->get_ExternLists(&pExtLists);

if (FAILED(hr)) return hr;

VARIANT va;
VariantInit(&va);
va.vt = VT_BSTR;
va.bastrBal = SysAllocString(L"Directory1");

VARIANT_BOOL bExists;
hr = pExtLists->Exists(va,&bExists);

if (bExists == VARIANT_TRUE)
{
    IVVPhraseColl *pList;
    pExtLists->get_Item(va,&pList);
    pList->put_Enabled(VARIANT_FALSE);
}
```

See Also

Refer to the following chapters for more information about the ViaVoice **Engine** Control:
Chapter 14, “Using External Lists” on page 291

GrammarFormat (VVCFGram)

Specifies the format of the **GrammarSource**.

Syntax

In Visual Basic:

```
[VVCFGram].GrammarFormat = [Enum VVFormatConstants]
```

In Visual C++ (MFC):

```
Enum VVFormatConstants = [CVVCFGram].GetGrammarFormat()  
[CVVCFGram].SetGrammarFormat(Enum VVFormatConstants)
```

In Visual C++:

```
HRESULT [pIVVCFGram] ->get_GrammarFormat(Enum VVFormatConstants *)  
HRESULT [pIVVCFGram] ->put_GrammarFormat(Enum VVFormatConstants)
```

Parameters

??

Return Values

None.

Remarks

This value tells the **VVCFGram** control how to load the **GrammarSource**. It tells the grammar control if the **GrammarSource** represents a compiled grammar, or a grammar source, and whether it uses IBM native grammar format, or SAPI 4.0 format. (For information on compiling grammars, see the SMAPI Reference or the SAPI Reference included with this SDK) The **VVCFGram** control supports the following formats:

0 - vvgfAuto: Automatically determines the format of the grammar source. This option can be more time consuming than telling the control the format of the grammar prior to loading it.

1 - vvgfNativeCompiled: The grammar is in IBM (.SAR) binary format.

2 - vvgfNativeSource: The grammar is in IBM (.BNF) source format.

3 - vvgfSAPICompiled: The grammar is in SAPI 4.0 (.GRM) binary format.

4 - vvgfSAPISource: The grammar is in SAPI 3.0 (.TXT) source format.

When using the vvgfAuto setting, in combination with vvstFile for the **SourceType** property, the **VVCFGram** control looks at the file extension in order to determine the **GrammarSource** format.

Example

In Visual Basic:

```
VVCFGram1.GrammarFormat = vvgfSAPICompiled
VVCFGram1.GrammarSource = "F:\Grammar\sample.grm"
VVCFGram1.LoadFromSource
```

In Visual C++ (MFC):

```
m_VVCFGram1.SetGrammarFormat(vvgfSAPICompiled);

VARIANT vaSource
VariantInit(&vaSource)
vaSource.vt = VT_BSTR;
vaSource.bstrVal = SysAllocString(L "F:\\Grammar\\sample.grm")

m_VVCFGram1.SetGrammarFormat(vaSource)
m_VVCFGram1.LoadFromSource();
```

In Visual C++:

```
pIVVCFGram->put_GrammarFormat(vvgfSAPICompiled);  
  
VARIANT vaSource  
VariantInit(&vaSource);  
VaSource.vt = VT_BSTR;  
VaSource.bstrVal = SysAllocString(L "F:\\Grammar\\sample.grm");  
pIVVCFGram->put_GrammarFormat(vaSource);  
pIVVCFGram->LoadFromSource();
```

See Also

“SourceType (VVCFGram)” on page 324

GrammarSource (VVCFGram)

Sets/Gets the path to a binary grammar file.

Syntax

In Visual Basic:

```
[VVCFGram].GrammarSource = String
```

In Visual C++ (MFC):

```
BSTR = [CVVCFGram].SetGrammarSource()  
[CVVCFGram].SetGrammarSource(BSTR)
```

In Visual C++:

```
HRESULT [pIVVCFGram] ->get_GrammarSource(BSTR *)  
HRESULT [pIVVCFGram] ->put_GrammarSource(BSTR)
```

Parameters

??

Return Values

None.

Remarks

Use this property in conjunction with the **LoadFromSource** method to load a grammar file into memory.

Example

In Visual Basic:

```
VVCFGram1.GrammarSource = "F:\Grammar\sample.grm"  
VVCFGram1.LoadFromSource
```

In Visual C++ (MFC):

```
VARIANT vaSource  
VariantInit(&vaSource)  
vaSource.vt = VT_BSTR;  
vaSource.bstrVal = SysAllocString (L "F:\\Grammar\\sample.grm")  
  
m_VVCFGram1.SetGrammarSource(vaSource)  
m_VVCFGram1.LoadFromSource();
```

In Visual C++:

```
VARIANT vaSource  
VariantInit(&vaSource);  
VaSource.vt = VT_BSTR;  
vaSource.bstrVal = SysAllocString(L "F:\\Grammar\\sample.grm");  
pIVVCFGram ->put_GrammarSource(vaSource);
```

See Also

“LoadFromSource (VVCFGram)” on page 328

Rules (VVCFGram)

??

Syntax

In Visual Basic:

In Visual C++ (MFC):

In Visual C++:

Parameters

??

Return Values

??

Remarks

Example

In Visual Basic:

In Visual C++ (MFC):

In Visual C++:

See Also

SourceType (VVCFGram)

Tells the **VVCFGram** control whether the **GrammarSource** property specifies a path to a file or is a grammar source string.

Syntax

In Visual Basic:

```
[VVCFGram].SourceType = [Enum SourceTypeConstants]
```

In Visual C++ (MFC):

```
Enum SourceTypeConstants = [CVVCFGram].SetSourceType()  
[CVVCFGram].SetSourceType(Enum SourceTypeConstants)
```

In Visual C++:

```
HRESULT [pIVVCFGram] ->get_SourceType(Enum SourceTypeConstants *)  
HRESULT [pIVVCFGram] ->put_SourceType(Enum SourceTypeConstants)
```

Parameters

??

Return Values

None.

Remarks

The **VVCFGram** control supports the following settings:

- 10 - vvstFile: The **GrammarSource** property contains the path to a file.
- 50 - vvstString: The **GrammarSource** property contains a string that represents a grammar source.

The contents of GrammarSource are determined by the value of this property and the value of the GrammarFormat property. The possible combinations are:

Setting	vvstFile	vvstString
vvgfAuto	X	X
vvgfNativeCompiled	X	N/A
vvgfNativeSource	X	X
vvgfSAPICompiled	X	N/A
vvgfSAPISource	X	X

Example

In Visual Basic:

```
VVCFGram1.SourceType = vvstFile
VVCFGram1.GrammarSource = "F:\Grammar\sample.grm"
VVCFGram1.LoadFromSource
```

In Visual C++ (MFC):

```
m_VVCFGram1.SourceType(vvstFile);

VARIANT vaSource
VariantInit(&vaSource)
vaSource.vt = VT_BSTR;
vaSource.bstrVal = SysAllocString (L "F:\\Grammar\\sample.grm")

m_VVCFGram1.SetGrammarFormat(vaSource)
m_VVCFGram1.LoadFromSource();
```

In Visual C++:

```
pIVVCFGram->put_SourceType(vvstFile);

VARIANT vaSource
VariantInit(&vaSource);
VaSource.vt = VT_BSTR;
VaSource.bstrVal = SysAllocString(L "F:\\Grammar\\sample.grm");

pIVVCFGram->put_GrammarFormat(vaSource);
pIVVCFGram->LoadFromSource();
```

See Also

“GrammarFormat (VVCFGram)” on page 317

Grammar Control Methods

The ViaVoice **Grammar** control supports the following methods:

- **About**^a
- **LoadFromSource**
- **RefreshUIText**
- **ShowTrainDialog**

a. Represents a standard method in Visual Basic. For more information, refer to your Visual Basic documentation.

LoadFromSource (VVCFGram)

Manually loads a binary grammar file.

Syntax

In Visual Basic:

```
[VVCFGram].LoadFromSource
```

In Visual C++ (MFC):

```
[CVVCFGram].LoadFromSource()
```

In Visual C++:

```
HRESULT [pIVVCFGram] ->LoadFromSource()
```

Parameters

None.

Return Values

None.

Remarks

There are two ways to load a grammar file, either manually using this method, or automatically, by setting the **AutoLoad** property to True at design time. Before the control can load a grammar file you must specify the path of the file to load by setting the value of the **GrammarSource** property.

If you wish to load a second grammar file into the same control make sure to set the **Enabled** property in the control to False. This will ensure that the speech engine unloads the first grammar from memory.

Example

In Visual Basic:

```
VVCFGram1.GrammarSource = "F:\Grammar\sample.grm"  
VVCFGram1.LoadFromSource
```

In Visual C++ (MFC):

```
VARIANT vaSource  
VariantInit(&vaSource)  
vaSouve.vt = VT_BSTR;  
vaSource.bstrVal = SysAllocString (L "F:\\Grammar\\sample.grm")  
  
m_VVCFGram1.SetGrammarSource(VaSource)  
m_VVCFGram1.LoadFromSource();
```

In Visual C++:

```
VARIANT vaSource  
VariantInit(&vaSource);  
VaSource.vt = VT_BSTR;  
vaSource.bstrVal = SysAllocString(L "F:\\Grammar\\sample.grm");  
pIVVCFGram ->put_GrammarSource(vaSource);
```

See Also

“AutoLoad (VVCFGram)” on page 305

Refresh

This method is exactly the same as `LoadFromSource()`.

RefreshUIText (VVCFGram)

Forces an update of the ViaVoice **User Interface Server** when **AutoUI** is True.

Syntax

In Visual Basic:

```
[VVCFGram].RefreshUIText (Text As String)
```

In Visual C++ (MFC):

```
[CVVCFGram].RefreshUIText (BSTR)
```

In Visual C++:

```
HRESULT [pIVVCFGram] ->RefreshUIText (BSTR)
```

Parameters

Text

Text to display in the **UIServer**.

Return Values

??

Remarks

Used primarily in the **SpeechRecognized** event together with the **UpdateUIText** property.

Example

In Visual Basic:

```
Private Sub VVCFGram1_SpeechRecognized(ByVal Name As String, ByVal ID As
Long, Phrase As String, UpdateUIText As Boolean, ByVal BegTime As String,
ByVal EndTime As String)
    ' This will cause the control to update the word history component in
    ' the UI Server immediately. Otherwise the text will be updated when
    ' the function ends (if UpdateUIText is True).
    VVCFGram1.RefreshUIText Phrase
    Update UIText = False
End Sub
```

In Visual C++ (MFC):

```
void CVVCFGramtest::OnSpeechRecognizedVvphrases1(LPCTSTR Name, long ID,
BSTR FAR* Phrase, BOOL FAR* UpdateUIText, LPCTSTR BegTime, LPCTSTR
EndTime)
{
    //This will cause the control to update the word history component in
    //the UI Server immediately. Otherwise the text will be updated when
    //the function ends (if UpdateUIText is True).
    m_VVCFGram1.RefreshUIText(*Phrase);
    *UpdateUIText = VARIANT_FALSE;
}
```

In Visual C++:

```
HRESULT SpeechRecognized (BSTR Name, long ID, BSTR *Phrase, VARIANT_BOOL
*UpdateUIText, BSTR BegTime, BSTR EndTime)
{
    //This will cause the control to update the word history component in
    //the UI Server immediately. Otherwise the text will be updated when
    //the function ends (if UpdateUIText is True).
    pIVVCFGram->RefreshUIText(*Phrase);
    *UpdateUIText = VARIANT_FALSE;
}
```

See Also

“SpeechRecognized (VVCFGram)” on page 339

ShowTrainDialog (VVCFGram)

Event fired if the **VVCFGram** control loads a grammar that contains a word that the speech engine cannot recognize.

Syntax

In Visual Basic:

```
[VVCFGram].ShowTrainDialog(ByVal Title As String, ByVal hWndParent As Long)
```

In Visual C++ (MFC):

```
[VVCFGram].ShowTrainDialog(CString Title, long hWndParent)
```

In Visual C++:

```
HRESULT ShowTrainDialog([in] BSTR Title, [in] long hWndParent);
```

Parameters

??

Return Values

??

Remarks

In this event you can call the **ShowTrainDialog** method to invoke the speech engine's training dialog. This dialog enables the user to train words that are unrecognizable. You do not have to provide the engine with a list of words - it automatically populates the dialog with the words that require training. Example

In Visual Basic:**In Visual C++ (MFC):****In Visual C++:****See Also**

None.

Grammar Control Events

The ViaVoice **Grammar** control supports the following events:

- **BeginSpeechRecognition**
- **Paused** (Not Supported)
- **SpeechRecognized** (Not Supported)
- **SpeechRejected**
- **TrainingRequired**

BeginSpeechRecognized (VVCFGram)

Fired when the speech engine receives audio input; it identifies as coming from user speech, rather than background noise. The event does not necessarily mean a specific expression in the **GrammarSource** that has been recognized; it simply indicates that the user has starting speaking.

Syntax

In Visual Basic:

```
BeginSpeechRecognition(ByVal BegTime As String)
```

In Visual C++ (MFC):

```
void OnBeginSpeechRecognition(LPCSTR BegTime)
```

In Visual C++:

```
HRESULT BeginSpeechRecognition(BSTR BegTime)
```

Parameters

BegTime

Bookmark indicating the time when the user began to speak.

Return Values

??

Remarks

None.

Examples

See Also

“SpeechRecognized (VVCFGram)” on page 339

SpeechRecognized (VVCFGram)

Event fired when the **VVCFGram** control recognizes one of the phrases contained in the grammar.

Syntax

In Visual Basic:

```
SpeechRecognized(ByVal Name As String, ByVal ID As Long, Phrase As
String, UpdateUIText As Boolean, ByVal BegTime As String, ByVal EndTime
As string)
```

In Visual C++ (MFC):

```
OnSpeechRecognized(LPCTSTR Name, long ID, BSTR FAR *Phrase, BOOL FAR
*UpdateUIText, LPCTSTR BegTime, LPCTSTR EndTime)
```

In Visual C++:

```
HRESULT SpeechRecognized(BSTR Name, long ID, BSTR FAR *Phrase,
VARIANT_BOOL *UpdateUIText, BSTR BegTime, BSTR EndTime)
```

Parameters

Name

The programmer assigned unique identifier for the phrase object.

ID

A programmer assigned numeric identifier for the item.

Phrase

The actual phrase text the user spoke.

UpdateUIText

When **AutoUI** is True, this parameter tells the **VVCFGram** control to use the Phrase text to update the Word History component in the ViaVoice **UIServer**.

BegTime

A bookmark indicating the time when the user began to speak the phrase.

EndTime

A bookmark indicating the time the user finished speaking the phrase.

Return Values

??

Remarks

Handling this event is necessary when using the **VVCFGram** control.

Example

In Visual Basic:

```
Private Sub VVCFGram1_SpeechRecognized(ByVal Name As String, _  
    ByVal ID As Long, ByVal Phrase As String, _  
        UpdateUIText As Boolean, ByVal BegTime As String,  
        ByVal EndTime As String)  
  
    Select Case ID  
    Case 100  
        MsgBox "Hello Sue"  
    Case 101  
        MsgBox "Hello James"  
    End Select  
  
End Sub
```

In Visual C++ (MFC):

```

void CVVCFGGramtest::OnSpeechRecognized(LPCTSTR Name, long ID,
    BSTR FAR *Phrase, BOOL FAR *UpdateUIText,
    LPCTSTR BegTime, LPCTSTR EndTime)
{
    switch (ID)
    {
    case 100:
        MessageBox ("Hello World" ,"VVCFGram",MB_OK);
        break;
    default:
        break;
    }
}

```

In Visual C++:

```

HRESULT SpeechRecognized(BSTR Name, long ID, BSTR *Phrase, VARIANT_BOOL
UpdateUIText, BSTR BegTime, BSTR EndTime)
{
    switch (ID)
    {
    case 100:
        MessageBox ("Hello World", "VVCFGram", MB_OK);
        break;
    default:
        break;
    }
}

```

See Also

“GrammarSource (VVCFGram)” on page 320

“LoadFromSource (VVCFGram)” on page 328

TrainingRequired (VVCFGram)

Notification from the engine that the currently active speech user needs to train the speech engine in order to improve recognition.

Syntax

In Visual Basic:

```
TrainingRequired(ByVal TrainingType As Long)
```

In Visual C++ (MFC):

```
void OnTrainingRequired(long TrainingType)
```

In Visual C++:

```
HRESULT TrainingRequired(long TrainingType)
```

Parameters

TrainingType

One of the following SAPI training types (refer to the Microsoft SAPI documentation for further description):

General (SRGNSTRAIN_GENERAL)	1
Grammar (SRGNSTRAIN_GRAMMAR)	2
Microphone (SRGNSTRAIN_MICROPHONE)	4

Return Values

??

Remarks

In the case of **VVCFGram** this means that the engine is using tentative pronunciations for some of the phrases in the **GrammarSource** because it cannot find the words in its base vocabulary.

Example

In Visual Basic:

```
Private Sub VVCFGram1_TrainingRequired(ByVal TrainingType As Long)
    Call VVCFGram1.ShowTrainDialog("Train Unrecognized Words. . .",
        m_hwnd)
End Sub
```

In Visual C++ (MFC):

```
void CVVCFGramTest::OnTrainingRequired(long TrainingType)
{
    m_VVCFGram.ShowTrainDialog("Train Unrecognized Words. . .",
        m_hwnd)
}
```

In Visual C++:

```
HRESULT Training Required(long TrainingType)
{
    pVVCFGram->ShowTrainDialog(L"Train Unrecognized Words...", hWnd);
}

Count
=====

//Get all external list names
IVVPhraseCollGroup *pExtLists;
pVVCFGram->get_ExtLists(&pExtLists);

long lCount;
pExtLists->get_Count(&lCount);

for (long l=1; l <= lCount; l++)
{
    IVVPhraseColl *pList;
    BSTR bstrListName;

    VARIANT va;
    VariantInit (&va);
    va.vt = VT_I4;
    va.lVal = l;

    pExtList->get_Item(va,&pList);
    pList->get_Name(&bstrListName);
}
```

See Also

Microsoft SAPI documentation

VVPhraseCollGroup Object

VVPhraseCollGroup Object Properties

The **VVPhraseCollGroup** (**IVVPhraseCollGroup**) Object has the following properties:

- **Count**
- **Enabled**
- **Item**

Count (VVPhraseCollGroup)

Returns the number of external lists in the group.

Syntax

In Visual Basic:

```
lValue = [VVCFGram.ExternList].Count
```

In Visual C++ (MFC):

```
long = [CVVPhraseCollGroup].GetCount()
```

In Visual C++:

```
HRESULT [pIVVPhraseCollGroup] ->get_Count(long *)
```

Parameters

None.

Return Values

Long

The number of items in the collection.

Remarks

None.

Example

In Visual Basic:

```
'Get all the external list names
Dim i As Integer
Dim sListName As String

For i = 1 To VVCFGram1.ExternList.Count
    sListName = VVCFGram1.ExternList(i).Name
Next
```

In Visual C++ (MFC):

```
//Get all the external list names
CString sListName;

for (i = 1; i <= VVCFGram1.GetExternList().GetCount(); i++)
{
    VARIANT va;
    VariantInit(&va);
    va.vt = VT_I2;
    va.iVal = i;
    sListName = VVCFGram1.GetExternList(va).GetName();
}
```

In Visual C++:

```
//Get all external list names
IVVPhraseCollGroup *pExtLists;
pVVCFGram->get_ExtLists(&pExtLists);

long lCount;
pExtLists ->get_Count(&lCount);

for (long l=1; l <= lCount; l++)
{
    IVVPhraseColl *pList;
    BSTR bstrListName;

    VARIANT va;
    VariantInit (&va);
    va.vt = VT_I4;
    va.lVal = l;

    pExtList ->get_Item(va, &pList);
    pList ->get_Name(&bstrListName);
}
```

See Also

None.

Enabled (VVPhraseCollGroup)

Enables/disables all the external lists within the group.

Syntax

In Visual Basic:

```
[VVCFGram.ExternList].Enabled = Boolean
```

In Visual C++ (MFC):

```
VARIANT_BOOL = [CVVPhraseCollGroup].GetEnabled()  
[CVVPhraseCollGroup].SetEnabled(VARIANT_BOOL)
```

In Visual C++:

```
HRESULT [pIVVPhraseCollGroup] ->get_Enabled(VARIANT_BOOL *)  
HRESULT [pIVVPhraseCollGroup] ->put_Enabled(VARIANT_BOOL)
```

Parameters

??

Return Values

TRUE

(Default) Enables all external lists in the control.

FALSE

Disables all external lists in the control.

Remarks

When an external list is disabled, the engine will not recognize commands in the grammar that use external lists.

Each time you add a phrase to an external list, the control unloads the existing external list and reloads the new external list. If you disable the external list object, you can build external lists without having the **Grammar** control unload and load the list for each item. You can then load the entire list at once by setting this property to True.

Example

In Visual Basic:

```
VVCFGram1.ExternList.Enabled = False

With VVCFGram1.ExternList("DirectoryA")
    .Add "C:\Junk", 100, "Junk", True
    .Add "C:\Temp", 200, "Temp", True
End With

VVCFGram1.ExternList.Enabled = True
```

In Visual C++ (MFC):

```
m_VVCFGram1.GetExternList().SetEnabled(FALSE);
CVVPhraseColl DirectoryA;

DirectoryA = m_VVCFGram1.GetExternList().Add(L"DirectoryA")
DirectoryA.Add(L"C:\Junk", 100, L"Junk", TRUE)
DirectoryA.Add(L"C:\Temp", 200, L"Temp", TRUE)

m_VVCFGrams1.GetExternLists().SetEnabled(TRUE);
```

In Visual C++:

```
IVVPhraseCollGroup *pExtLists;  
pVVCFGram ->get_ExtLists(&pExtLists);  
  
IVVPhraseColl *pList;  
  
pExtList ->put_Enabled(VARIANT_FALSE);  
  
VARIANT va;  
VariantInit (&va);  
va.vt = VT_BSTR;  
va.bstrVal = SysAllocString(L"DirectoryA");  
pExtList->get_Item(va,&pList);  
pList ->Add(L"C:\\Windows",100,L"Windows", VARIANT_TRUE);  
pList ->Add(L"C:\\Temp",200,L"Temp", VARIANT_TRUE);  
  
pExtList->put_Enabled(VARIANT_TRUE);
```

See Also

None.

Item (VVPhraseCollGroup)

Returns an external list from the **Phrase Collection** Group (**IVVPhraseCollGroup**) Object.

Syntax

In Visual Basic:

```
[VVPhraseColl] = [VVCFGram.ExternList].Item(ByVal Key As VARIANT)
```

In Visual C++ (MFC):

```
CVVPhraseColl = [CVVPhraseCollGroup].GetItem(VARIANT Key)
```

In Visual C++:

```
HRESULT IVVPhraseCollGroup::get_Item(VARIANT Key, IVVPhraseColl  
**pRetVal)
```

Parameters

Key

VARIANT. The item identifier. This parameter can be numeric - indicating the ordinal position of the item within the collection, or a string - indicating the text of the item.

Return Values

VVPhraseColl

The **Phrase Collection** object that contains the requested external list.

Remarks

The external list object is an instance of the **Phrase Collection** (**IVVPhraseColl**) Object. The **VVPhraseColl** Object is explained in detail on page 345.

Example

In Visual Basic:

```
Call  
VVCFGram1.ExternList.Item("Directory1").Add("C:\Temp", 100, "Temp", True)  
Or  
VVCFGram1.ExternList("Directory1").Add("C:\Temp", 100, "Temp", True)
```

In Visual C++ (MFC):

```
VARIANT va;  
VariantInit (&va);  
va.bstrVal = SysAllocString(L"Directory1");  
va.vt = VT_BSTR  
  
m_VVCFGram1.GetExternList().GetItem(va).Add(L"C:\Temp", 100, L"Temp",  
TRUE);
```

In Visual C++:

```
IVVPhraseCollGroup *pExtLists;  
pVVCFGram->get_ExtLists(&pExtLists);  
  
IVVPhraseColl *pList;  
  
VARIANT va;  
VariantInit (&va);  
va.vt = VT_BSTR;  
va.bstrVal = SysAllocString(L"DirectoryA");  
pExtList->get_Item(va, &pList);
```

See Also

None.

VVPhraseCollGroup Object Methods

The **VVPhraseCollGroup** (**IVVPhraseCollGroup**) Object supports the following methods:

- **Exists**

Exists (VVPhraseCollGroup)

Use this method to find out if a certain external list is part of the external list **VVPhraseCollGroup** Object.

Syntax

In Visual Basic:

```
[Boolean] = [VVCFGram.ExternList].Exists(ByVal Key As VARIANT)
```

In Visual C++ (MFC):

```
VARIANT_BOOL = [CVVCFPhraseCollGroup].Exists(VARIANT Key)
```

In Visual C++:

```
HRESULT [pIVVPhraseCollGroup]->Exists (VARIANT Key, VARIANT_BOOL  
*bExists)
```

Parameters

Key

VARIANT. The item identifier. This parameter can be numeric – indicating the ordinal position of the item within the group, or a string – indicating the name of the item.

Returns

VARIANT_Bool

True if the external list exists in the group; False if it does not.

Remarks

The **Exists** function does not look at the grammar file to see if the grammar contains a certain external list declaration. The purpose of this function is to find out if a certain external list has been added to the **Phrase Collection** group.

Example

In Visual Basic:

```
If VVCFGram1.ExternLists.Exists ("Animals") Then
    With VVCFGram1.ExternLists ("Animals")
        :Add "Horse", 100, "Horse", True
        :Add "Cat", 100, "Cat", True
    End With
End If
```

In Visual C++ (MFC):

```
VARIANT Va;
va.vt=VT_BSTR;
va.bstrVal=SysAllocString (L"Animals");
if (m_vVVCFGram1.GetExternLists().Exists(va)==TRUE)
{
    CVVPhraseColl PhraseColl=m_VVCFGram1.GetExternLists().GetItem(va);
    PhraseColl.Add(SysAllocString(L"Horse"), 100, SysAllocString
        (L"Horse"), TRUE);
    PhraseColl.Add(SysAllocString(L"Cat"), 100, SysAllocString (L"Cat"),
        TRUE);
}
```

In Visual C++:

```
VARIANT va;  
va.vt=VT_BSTR;  
va.bstrVal=SysAllocString (L"Animals");  
IVVPhraseCollGroup *pIVVCollGroup=NULL;  
pVVCFGram->get_ExternLists(&pIVVCollGroup);  
VARIANT_BOOL bExists;  
pIVVCollGroup->get_Exists(va,&bExists)  
if (bExists==VARIANT_TRUE)  
{  
    IVVPhraseColl *pIVVPhraseColl;  
    pIVVCollGroup->get_Item(va,&pIVVPhraseColl);  
    pIVVPhraseColl->Add (SysAllocString(L"Horse"), 100, SysAllocString  
        (L"Horse") VARIANT_TRUE);  
    pIVVPhraseColl->Add (SysAllocString(L"Cat"), 100, SysAllocString  
        (L"Cat"), VARIANT_TRUE);  
}
```

See Also

None.

Grammar Control Frequently Asked Questions

This chapter contains answers to the most frequently asked questions about the ViaVoice **Grammar Control**.

How do I create a compiled grammar file from a text-based grammar specification?

The IBM ViaVoice SDK allows you to compile a BNF grammar script into a grammar archive file. Archive files have the extension “SAR”. The command-line executable VTSAPIC.EXE, located in the ViaVoice SDK\Tools directory, creates SAR files. These can then be assigned to the **GrammarSource** property of the **VVCFGram** control.

Where can I find more information about grammar file syntax?

The BNF grammar syntax, which the VTSAPIC utility compiles into binary SAR files, is documented in the SMAPI Developer’s Guide, Part 2: SMAPI Grammars.

What kinds of grammars are supported?

The BNF grammar syntax, which generates native IBM engine SAPI archive grammar files (SAR files), is supported. Additionally GRM files, which can be compiled using the Microsoft Speech SDK, are assignable to the **GrammarSource** property of the **VVGrammar** control.

What is the purpose of annotations?

Annotations allow you to associate a phrase with data that is meaningful to your program. For instance, in a grammar file you could link the word “red” with an RGB string “255,0,0”. Then when the word “red” was recognized, you could retrieve this annotation and set the window color to the RGB value instead of having to translate “red” into its other representation through application code.

When you install the IBM ViaVoice SDK, the setup program installs a set of **Lite** controls in addition to the full-featured versions. The **Lite** controls are scaled-down versions of the full-featured controls. The ViaVoice **Lite** controls include **Dictation Lite** (**VVDictLite**), **Grammar Lite** (**VVGrammarLite**), and **Phrases Lite** (**VVPhrasesLite**) controls. The purpose of these controls is to provide you with a set of small, simple controls that have few dependencies and can be used in Web pages. The following sections explain how to use the ViaVoice **Lite** controls in HTML pages.

Note:

Although this document focuses on how to use the controls in HTML pages with Visual InterDev 1.0, the ViaVoice **Lite** controls can be used in Visual Basic, Visual C++ (with and without MFC), Delphi, Borland C++ Builder, and Visual J++ as well.

Getting Started with the Lite Controls

The following are tutorials on how to incorporate the **VVDictLite**, **VVGrammarLite**, and **VVPhrasesLite** controls into your HTML page using the Visual InterDev application. These tutorials are designed to present you with the most commonly used properties, methods, and events in these **Lite** controls.

VVDictLite Control

The ViaVoice **Dictation Lite** (**VVDictLite**) control is an ActiveX control that enables you to capture dictated speech from users. **VVDictLite** is invisible at run time. When a program creates an instance of the **VVDictLite** control at run time, the control searches the client's machine for a speech engine capable of receiving dictation. If the control does not find an engine with this capability, then it simply becomes inactive. This means that you can feel confident that your program will not crash in the absence of a speech engine. If a speech engine is present and the control's **Enabled** property is set to True, then the control listens to the user. Whenever the user speaks, the **VVDictLite** control fires the **PhraseRecognized** event for each word that it recognizes.

Using the Control

This section contains step-by-step instructions for using Visual InterDev when adding this control in an HTML page.

In Visual InterDev:

To use the **VVDictLite** control, do the following:

1. Open the HTML page in which you wish to insert the ActiveX control.
2. From the **Insert** menu, choose **Into HTML** then choose **ActiveX Control...** menu item. You will see the 'Insert ActiveX Control' dialog box as shown in Figure 25.

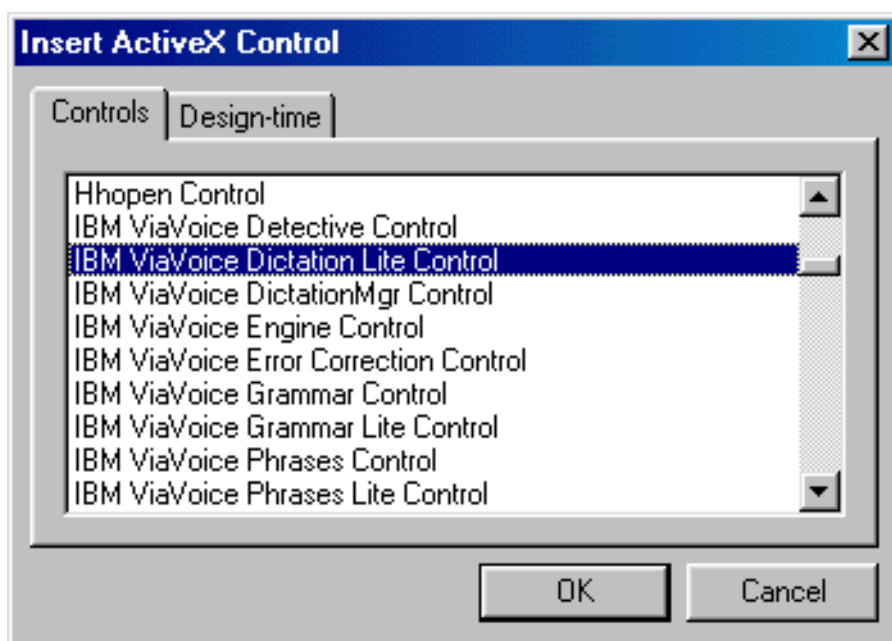


Figure 25. Insert ActiveX Control Dialog Box - VVDictLite

3. Select **IBM ViaVoice Dictation Lite Control** from the list and click **OK**. You will see the 'Properties' dialog box with a 'Control Designer Form' as shown in Figure 26.

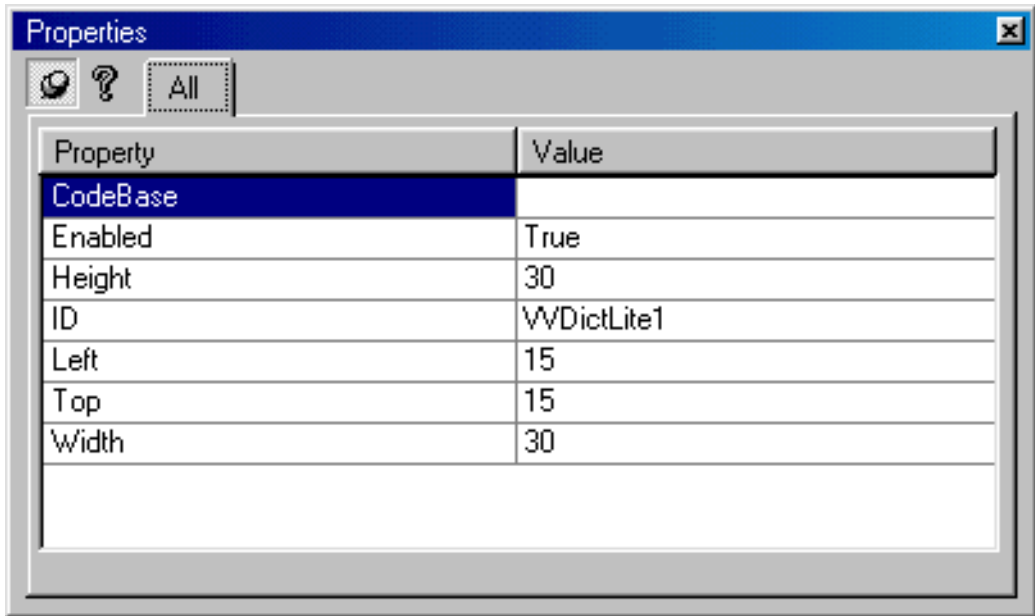


Figure 26. Control Designer Form - VVDictLite

When you are done customizing the control properties, close the control designer window. Visual InterDev will insert the following code to your page:

```
<OBJECT ID="VVDictLite1" WIDTH=40 HEIGHT=40
  CLASSID="CLSID:5AF3ED20-6A8E-11D2-A42E-002035215001">
  <PARAM NAME="Enabled" VALUE="1">
</OBJECT>
```

This is all that is necessary to begin capturing dictation from your users. Each time the control recognizes a dictated word it will fire the **PhraseRecognized** event. To enter code for this event, do the following:

1. Choose **Script Wizard** from the **View** menu. You will see the 'Script Wizard' dialog box shown in Figure 27.

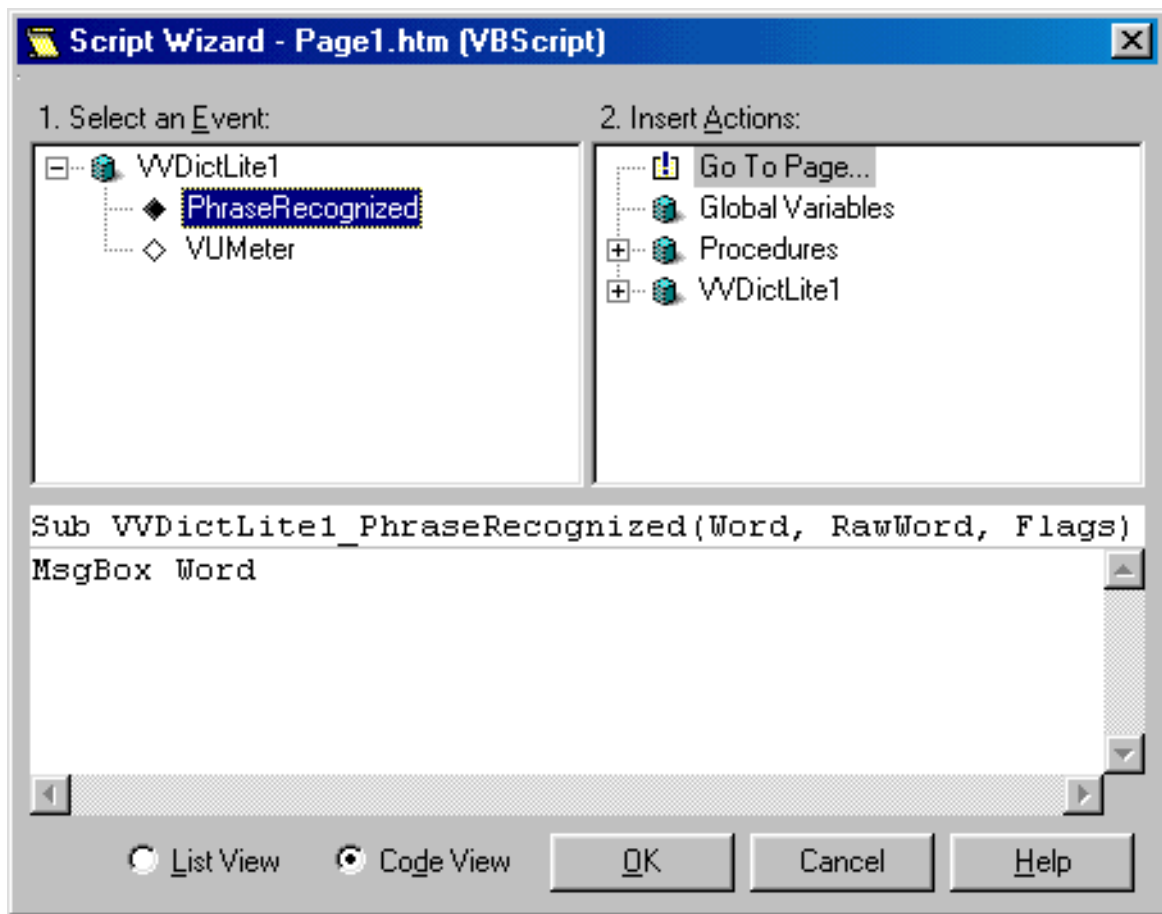


Figure 27. Script Wizard Dialog Box - VVDictLite

2. Open the VVDictLite1 branch to display the events for the **VVDictLite** control, and click on the **PhraseRecognized** event. For details on the **PhraseRecognized** event, see page 385. Then, enter code to handle this event. For example, to display a message box with the word that was recognized enter the following statement: `MsgBox Word`.
3. Click **OK** when done.

Visual InterDev will enter the following code into your HTML page:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub VVDictLite1_PhraseRecognized(Word, RawWord, Flags)
MsgBox Word
end sub
-->
</SCRIPT>
```

To test the control, first save the HTML page, then right mouse click on the editor window and choose the "Preview (DocumentName)..." from the popup menu. You should be able to dictate into your page and each word the control recognizes will be displayed in a message box.

VVGrammarLite Control

The ViaVoice **Grammar Lite** (**VVGrammarLite**) control is an ActiveX control that can recognize complex commands based on a SAPI 4.0 context-free grammar. First you supply the **VVGrammarLite** control with a string representing a grammar definition. The control compiles this grammar in memory, and listens to the user's speech. When the user speaks one of the phrases defined in the grammar, it fires the **PhraseRecognized** event. Like the **VVDictLite** control and the **VVPhrasesLite** control, when a program creates an instance of the **VVGrammarLite** control at run time, the control searches the client's machine for a speech engine capable of performing context-free command recognition. If the control does not find an engine with this capability, then it simply becomes inactive.

Using the Control

This section contains step-by-step instructions for using Visual InterDev when adding this control in an HTML page.

In Visual InterDev:

To use the **VVGrammarLite** control, do the following:

1. Open the HTML page in which you wish to insert the ActiveX control.

2. From the **Insert** menu, choose **Into HTML** then choose **ActiveX Control...** menu item. You will see the 'Insert ActiveX Control' dialog box as shown in Figure 28.

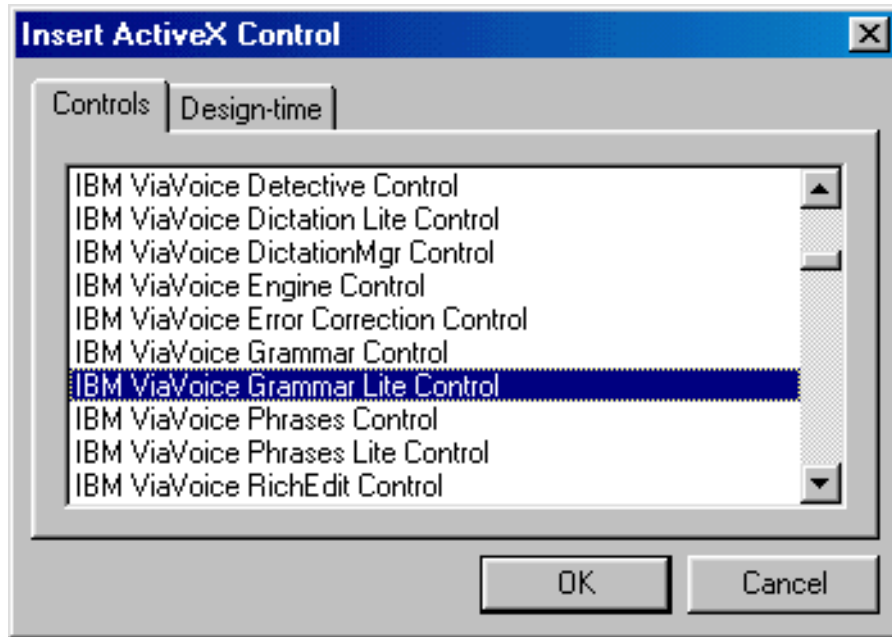


Figure 28. Insert ActiveX Control Dialog Box - VVGramLite

3. Select **IBM ViaVoice Grammar Lite Control** from the list and click **OK**. You will see the 'Properties' dialog box with a 'Control Designer Form' as shown in Figure 29.

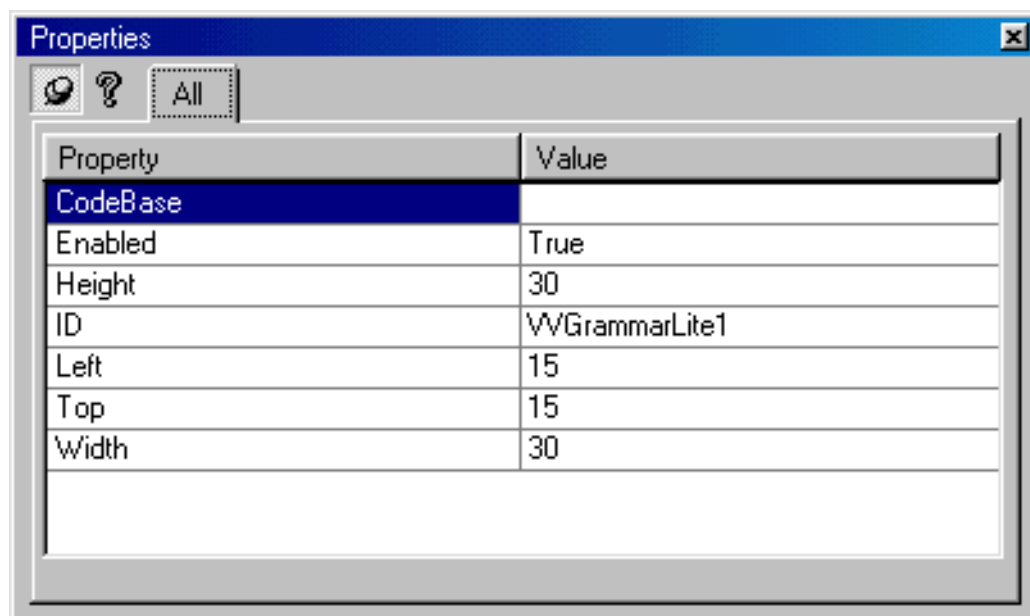


Figure 29. Control Designer Form - VVGramLite

When you are done customizing the control properties, close the control designer window. Visual InterDev will insert the following code to your page:

```
<OBJECT ID="VVGrammarLite1" WIDTH=40 HEIGHT=40
  CLASSID="CLSID:5AF3ED27-6A8E-11D2-A42E-002035215001">
  <PARAM NAME="Enabled" VALUE="1">
  <PARAM NAME="GrammarSource" VALUE="">
</OBJECT>
```

The first step in using the control is to provide the control with a SAPI 4.0 grammar. To do this simply set the control's grammar source property to the grammar text, as shown below:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub window_onload
sGrammar = "[<Start>]" & Chr(13) & Chr(10)
sGrammar = sGrammar & "<Start>=This is a test"
VVGrammarLite1.GrammarSource = sGrammar
End Sub
-->
</SCRIPT>
```

When the user speaks one of the commands in the grammar, the control will fire the **SpeechRecognized** event. To write code to handle this event, do the following.

1. Choose **Script Wizard** from the **View** menu. You will see the 'Script Wizard' dialog box shown in Figure 30.

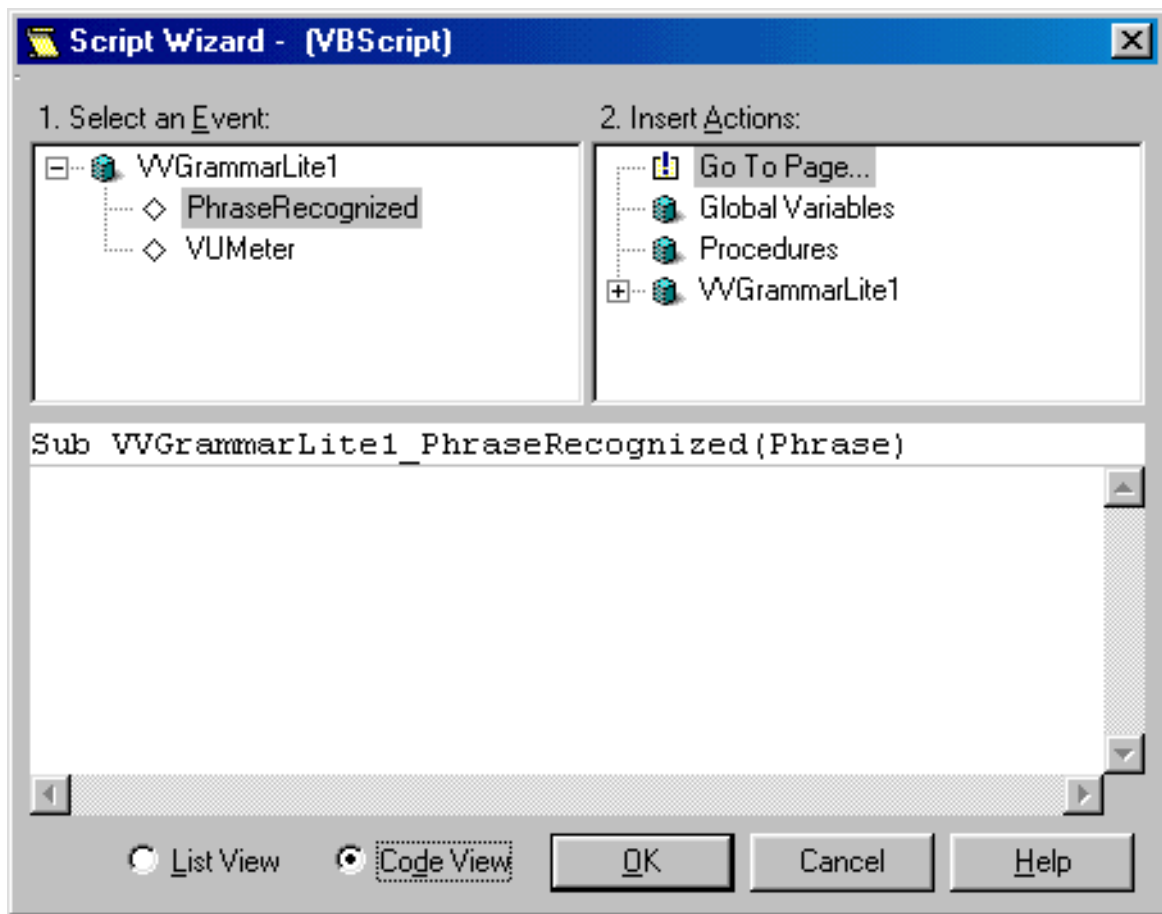


Figure 30. Script Wizard Dialog Box - VVGrammarLite

2. Open the VVGrammarLite1 branch to display the events for the **VVGrammarLite** control, and click on the **PhraseRecognized** event. For details on the **PhraseRecognized** event, see page 395. Then enter code to handle this event. For example, enter the following statement: `MsgBox "The user has issued the following command:" & Phrase`
3. Click **OK** when done.

Visual InterDev will enter the following code into your HTML page:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub VVGrammarLite1_PhraseRecognized(Phrase)
MsgBox "The user has issued the following command:" & Phrase
end sub-->
</SCRIPT>
```

To test the control first save the HTML page, then right mouse click on the editor window and choose the **Preview X...** from the popup menu. You should be able to speak any of the phrases defined in the grammar, and your page should respond with a message box displaying the phrase recognized.

VVPhrasesLite Control

The ViaVoice **Phrases Lite** (**VVPhrasesLite**) control is an ActiveX control that enables you to create a list of phrases for the control to monitor. When the user speaks one of the phrases in the list, the **VVPhrasesLite** control fires the **PhraseRecognized** event. Like the **VVDictLite** control, when a program creates an instance of the **VVPhrasesLite** control at run time, the control searches the client's machine for a speech engine capable of performing command recognition. If the control does not find an engine with this capability, then it simply becomes inactive. Normally this control is used to navigate through a page and to execute simple commands.

Using the Control

This section contains step-by-step instructions for using Visual InterDev when adding this control in an HTML page.

In Visual InterDev:

To use the **VVPhrasesLite** control, do the following:

1. Open the HTML page in which you wish to insert the ActiveX control.

2. From the **Insert** menu, choose **Into HTML** then choose **ActiveX Control...** menu item. You will see the 'Insert ActiveX Control' dialog box as shown in Figure 31.

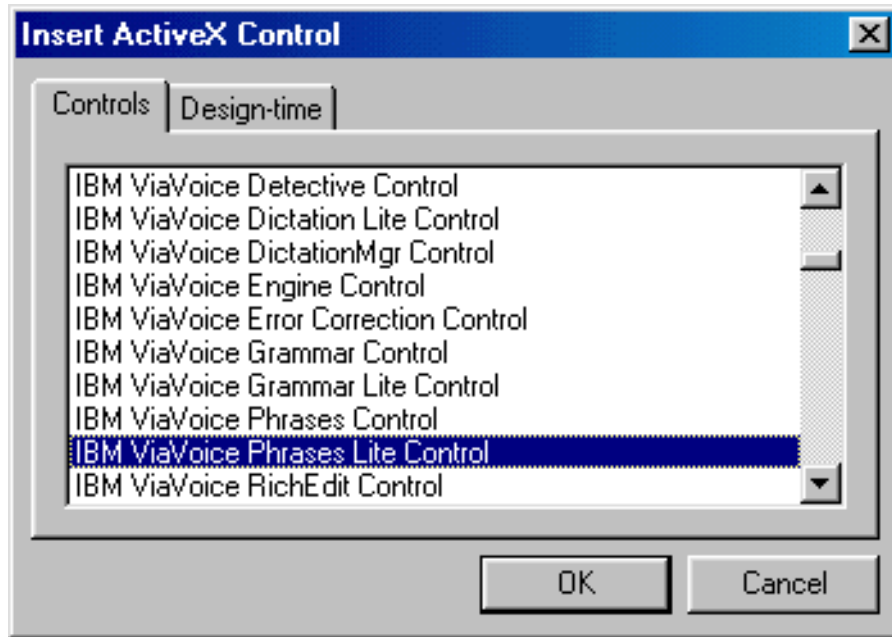


Figure 31. Insert ActiveX Control Dialog Box - VVPPhrasesLite

3. Select **IBM ViaVoice Phrases Lite Control** from the list and click **OK**. You will see the 'Properties' dialog box with a 'Control Designer Form' as shown in Figure 32.

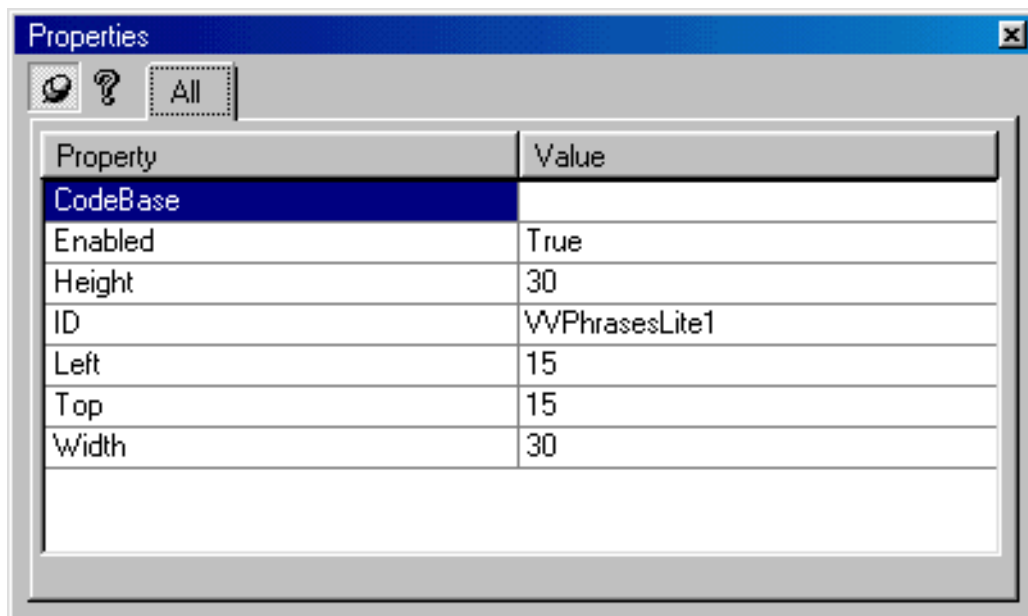


Figure 32. Control Designer Form - VVPhrasesLite

When you are done customizing the control properties, close the control designer window. Visual InterDev will insert the following code to your page:

```
<OBJECT ID="VVPhrasesLite1" WIDTH=40 HEIGHT=40
  CLASSID="CLSID:5AF3ED25-6A8E-11D2-A42E-002035215001">
  <PARAM NAME="Enabled" VALUE="1">
</OBJECT>
```


The first step in using the control is to create a list of phrases that the control will understand. To create the list of phrases use the **AddPhrase** method. A good place to create this list is in the `window_onload` event. For example, add the following code to your HTML page:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub window_onload
    Call VVPhrasesLite1.AddPhrase("Add Address",110)
    Call VVPhrasesLite1.AddPhrase("Save Address",100)
End Sub
-->
</SCRIPT>
```

Each time the control recognizes one of the phrases in the list it will fire the **PhraseRecognized** event. To enter code for this event, do the following:

1. Choose **Script Wizard** from the **View** menu. You will see the 'Script Wizard' dialog box shown in Figure 33.

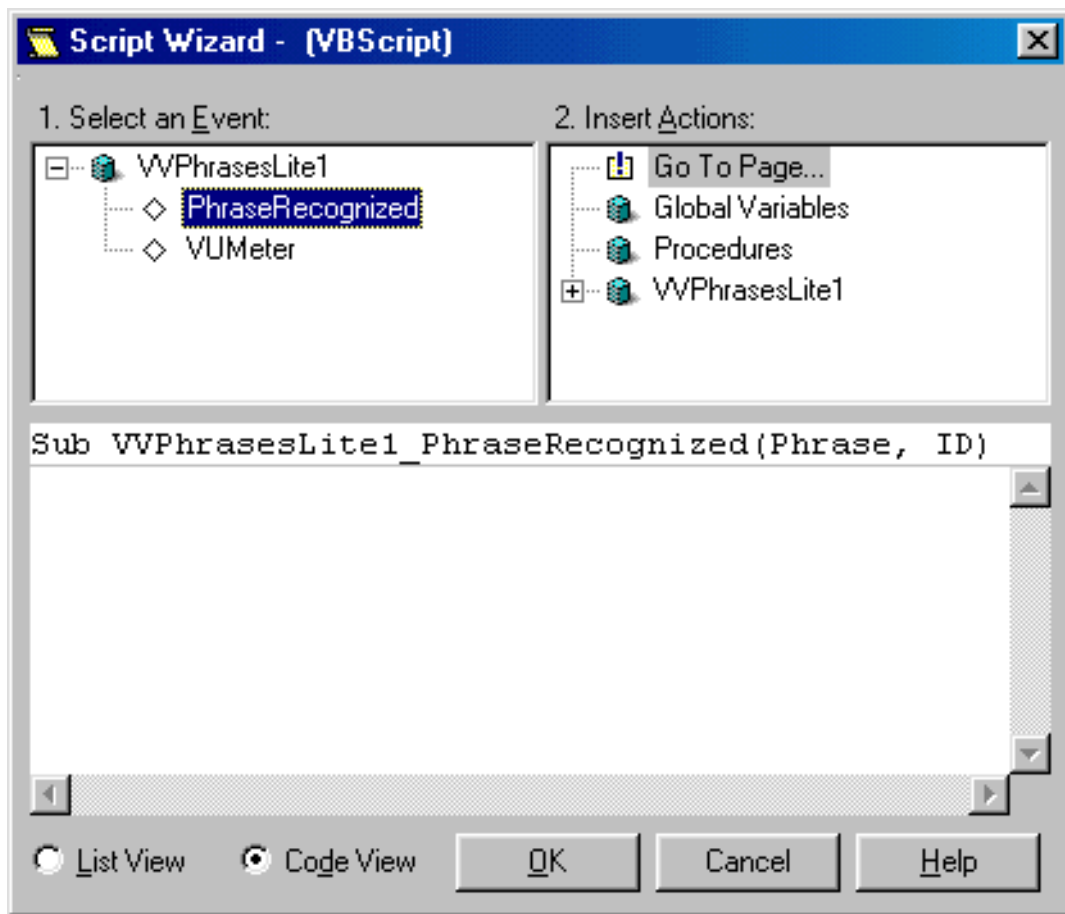


Figure 33. Script Wizard Dialog Box - VVPhrasesLite

2. Open the VVPhrasesLite1 branch to display the events for the **VVPhrasesLite** control, and click on the **PhraseRecognized** event. For details on the **PhraseRecognized** event, see page page 408. Then enter code to handle this event. For example, enter the following statement: `MsgBox "The user has issued the following command:" & Phrase`
3. Click **OK** when done.

Visual InterDev will enter the following code into your HTML page:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub VVPhrasesLite1_PhraseRecognized(Phrase, ID)
  MsgBox "The user has issued the following command:" & Phrase
end sub-->
</SCRIPT>
```

To test the control, first save the HTML page, then right mouse click on the editor window and choose the **Preview (DocumentName)...** from the popup menu. You should be able to speak any of the phrases in the list, and your page should respond with a message box displaying the phrase recognized.

Summary

At this point, you should know how to do the following:

- How to add the **VVDictLite** control into an **HTML** page.
- **How to add the VVGrammarLite control into an HTML page.**
- **How to add the VVPhrasesLite** control into an **HTML** page.

The remainder of this documentation contains a reference for all the properties, methods, and events for the **Lite** controls.

The following sections describe the properties, methods, and events for the **VVDictLite** Control, **VVGrammarLite** Control, and **VVPhrasesLite** Control.

VVDictLite Control Properties

The **VVDictLite** Control supports the following properties:

- **Enabled**
- **Index**^a
- **Name**^a
- **Object**^a
- **Parent**^a
- **Tag**^a

a. Represents a standard method in Visual Basic. For more information, refer to your Visual Basic documentation.

Enabled (VVDictLite)

Determines whether the **VVDictLite** control is listening or not.

Syntax

In Visual InterDev:

<code>[VVDictLite].Enabled = [Boolean]</code>

Parameters

??

Return Values

TRUE

The control is enabled and captures dictation speech.

FALSE

The control is disabled and does nothing.

Remarks

The **Enabled** property also reports if the **VVDictLite** control found a speech engine in the client's machine. When the control is first created at run time, it searches the client's machine for a suitable speech engine. If the control does not find a speech engine, then it automatically sets the **Enabled** property to False.

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub btnTurnOffDictation_Click()
    'This line stops dictation
    VVDictLite1.Enabled=False
End Sub
-->
</SCRIPT>
```

See Also

None.

VVDictLite Control Methods

There are no methods for this control.

VVDictLite Control Events

The **VVDictLite** Control supports the following events:

- **PhraseRecognized**
- **VUMeter**

PhraseRecognized (VVDictLite)

Event fired by the **VVDictLite** control when it recognizes speech from the user.

Syntax

In Visual InterDev:

```
PhraseRecognized (ByVal Word As String, ByVal RawWord As String, ByVal  
Flags As Long)
```

Parameters

Word

The word that the user spoke. There are certain words, however, that the engine formats before firing this event. For example, if the user says, "NEW-LINE". the word parameter will be the Carriage Return and the Line Feed characters. These phrases are called macros. In this version, the **VVDictLite** only supports the "NEW-LINE" and "NEW-PARAGRAPH" macros.

RawWord

RawWord is the unformatted word. If the user says, "NEW-LINE", the Word parameter will contain the reformatted text. However, the RawWord parameter will contain the words "NEW-LINE."

Flags

This parameter is available for future enhancements.

Return Values

??

Remarks

None.

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub btnTurnOffDictation_Click()
    'This line stops dictation
    VVDictLite1.Enabled=False
End Sub
-->
</SCRIPT>
```

See Also

None.

VUMeter (VVDictLite)

Event fired by the **VVDictLite** control when it detects a change in volume.

Syntax

In Visual InterDev:

<code>VUMeter(ByVal <i>Level</i> As Long)</code>

Parameters

Level

The audio level as a percentage, where 0 is silence, and 100 is the loudest volume the engine can support.

Return Values

??

Remarks

The **VUMeter** event returns the audio level as a percentage where 0 is silence, and 100 is the loudest volume the engine can support. Use this event to give your users indication that the control is in fact listening to their speech

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE = "VBScript">
<! --
Sub VVDictLite_VUMeter (ByVal Level As Long)
  If Level < 10 Then
    MsgBox "Please speak louder", VBOk , "Speaking too softly"
  End If
End Sub
-- >
</SCRIPT>
```

See Also

None.

VVGrammarLite Control Properties

The **VVGrammarLite** Control supports the following properties:

- **Enabled**
- **GrammarSource**
- **Index^a**
- **Name^a**
- **Object^a**
- **Parent^a**
- **Tag^a**

a. Represents a standard method in Visual Basic. For more information, refer to your Visual Basic documentation.

Enabled (VVGrammarLite)

Determines whether the **VVGrammarLite** control is listening or not.

Syntax

In Visual InterDev:

<code>[VVGrammarLite].Enabled = [Boolean]</code>

Parameters

??

Return Values

TRUE

The **VVGrammarLite** control is **Enabled** and captures dictation speech.

FALSE

The control is disabled and does nothing.

Remarks

The **Enabled** property also reports if the **VVGrammarLite** control found a speech engine in the client's machine. When the control is first created at run time, it searches the client's machine for a suitable speech engine. If the control does not find a speech engine, then it automatically sets the **Enabled** property to False.

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub btnTurnOffCommand_Click()
    'This line stops command recognition
    VVGrammarLite1.Enabled=False
End Sub
-->
</SCRIPT>
```

See Also

None.

GrammarSource (VVGrammarLite)

Enables you to specify the grammar for the **VVGrammarLite** control.

Syntax

In Visual InterDev:

<code>[VVGrammarLite].GrammarSource = [String]</code>

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub window_onload
    sGrammar = "[<Start>]" & Chr(13) & Chr(10)
    sGrammar = sGrammar & "<Start>=This is a test"
    VVGrammarLite1.GrammarSource = sGrammar
End Sub
-->
</SCRIPT>
```

See Also

None.

VVGrammarLite Control Methods

There are no methods for this control.

VVGrammarLite Control Events

The **VVGrammarLite** Control supports the following events:

- **PhraseRecognized**
- **VUMeter**

PhraseRecognized (VVGrammarLite)

event fired by the **VVGrammarLite** control when the user speaks one of the phrases defined in the control's grammar.

Syntax

In Visual InterDev:

<code>PhraseRecognized(ByVal <i>Phrase</i> As String)</code>

Parameters

Phrase

The phrase that the control recognized.

Return Values

??

Remarks

You can provide the grammar control with a SAPI 4.0 grammar source. The control automatically compiles the grammar at run time.

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub VVGrammarLite1_PhraseRecognized(Phrase)
    MsgBox "The user has issued the following command:" & Phrase
End Sub
-->
</SCRIPT>
```

See Also

None.

VUMeter (VVGrammarLite)

Event fired by the **VVGrammarLite** control when it detects a change in volume.

Syntax

In Visual InterDev:

<code>VUMeter(ByVal Level As Long)</code>

Parameters

Level

The audio level as a percentage, where 0 is silence, and 100 is the loudest volume the engine can support.

Return Values

??

Remarks

The **VUMeter** event returns the audio level as a percentage where 0 is silence, and 100 is the loudest volume the engine can support. Use this event to give your users indication that the control is in fact listening to their speech.

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE = "VBScript">
<! --
Sub VVGrammarLite_VUMeter (ByVal Level As Long)
  If Level < 10 Then
    MsgBox "Please speak louder", VBOk , "Speaking too softly"
  End If
End Sub
-->
</SCRIPT>
```

See Also

None.

VVPhrasesLite Control Properties

The **VVPhrasesLite** Control supports the following properties:

- **Enabled**
- **Index**^a
- **Name**^a
- **Object**^a
- **Parent**^a
- **Tag**^a

a. Represents a standard method in Visual Basic. For more information, refer to your Visual Basic documentation.

Enabled (VVPhrasesLite)

Determines whether the **VVPhrasesLite** control is listening or not.

Syntax

In Visual InterDev:

<code>[VVPhrasesLite].Enabled = [Boolean]</code>

Parameters

??

Return Values

TRUE

The **VVPhrasesLite** control is Enabled and listens for commands

FALSE

The control is disabled and does nothing.

Remarks

The **Enabled** property also reports if the **VVDictLite** control found a speech engine in the client's machine. When the control is first created at run time, it searches the client's machine for a suitable speech engine. If the control does not find a speech engine, then it automatically sets the **Enabled** property to False.

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub btnTurnOffCommand_Click()
    'This line stops command recognition
    VVDictLite1.Enabled=False
End Sub
-->
</SCRIPT>
```

See Also

None.

VVPhrasesLite Control Methods

The **VVPhrasesLite** Control supports the following methods:

- **AddPhrase**
- **RemoveAll**

AddPhrase (VVPhrasesLite)

This method enables you to add phrases to the list of phrases that the **VVPhrasesLite** control will recognize.

Syntax

In Visual InterDev:

<code>AddPhrase(ByVal <i>Phrase</i> As String, ByVal <i>ID</i> As Long)</code>

Parameters

Phrase

The phrase that the control will recognize.

ID

A user assigned numeric identifier for the phrase. This number does not have to be unique. You can use this number to associate phrases with the same meaning.

Return Values

??

Remarks

You can add multiple phrases with the same meaning and assign them all the same ID number. Then in the **SpeechRecognized** event take the same action for all of the phrases that have the same ID number.

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub window_onload
    Call VVPhrasesLite1.AddPhrase("Add Address",110)
    Call VVPhrasesLite1.AddPhrase("Save Address",100)
End Sub
-->
</SCRIPT>
```

See Also

None.

RemoveAll (VVPhrasesLite)

Removes all the phrases from the control's list of phrases.

Syntax

In Visual InterDev:

<code>RemoveAll ()</code>

Parameters

None.

Return Values

??

Remarks

None.

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub btnCancel_Click()
    Call VVPhrasesLite1.RemoveAll()
    Call VVPhrasesLite1.AddPhrase("Yes",110)
    Call VVPhrasesLite1.AddPhrase("No",100)
    rc = MsgBox("Are you sure you want to cancel this
        order?", "Cancel?", VBYesNo)
End Sub
-->
</SCRIPT>
```

See Also

None.

VVPhrasesLite Control Events

The **VVGrammarLite** Control supports the following events:

- **PhraseRecognized**
- **VUMeter**

PhraseRecognized (VVPhrasesLite)

The **VVPhrasesLite** control fires this event when the user speaks one of the phrases in the control's phrase list. You build the list by calling the **AddPhrase** method.

Syntax

In Visual InterDev:

<code>PhraseRecognized(ByVal <i>Phrase</i> As String, ByVal <i>ID</i> As Long)</code>

Parameters

Phrase

The phrase that the control recognized.

ID

A user assigned numeric identifier for the phrase. This number does not have to be unique. You can use this number to associate phrases with the same meaning.

Return Values

??

Remarks

None.

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE = "VBScript">
<! --
Sub VVPhraseRecognized (ByVal Phrase As String, ByVal ID As Long)
    If ID = 100 Then
        MsgBox "Are you sure you want to exit", VBYESNO, "Exit Program"
    End If
End Sub
-->
</SCRIPT>
```

See Also

None.

VUMeter (VVPhrasesLite)

Event fired when the **VVPhrasesLite** control detects a change in volume.

Syntax

In Visual InterDev:

<code>VUMeter(ByVal Level As Long)</code>

Parameters

Level

The audio level as a percentage, where 0 is silence, and 100 is the loudest volume the engine can support.

Return Values

??

Remarks

The **VUMeter** event returns the audio level as a percentage where 0 is silence, and 100 is the loudest volume the engine can support. Use this event to give your users indication that the control is in fact listening to their speech.

Example

In Visual InterDev:

```
<SCRIPT LANGUAGE = "VBScript">
<! --
Sub VVPhrasesLite_VUMeter (ByVal Level As Long)
  If Level < 10 Then
    MsgBox "Please speak louder", VBOk , "Speaking too softly"
  End If
End Sub
-->
</SCRIPT>
```

See Also

None.

Lite Controls Frequently Asked Questions

This chapter contains answers to the most frequently asked questions about the ViaVoice **Lite** controls.

How can I choose which speech engine the Lite Controls connect to?

Currently there is no way to specify the properties of the engine you wish to connect to. This functionality is available only in the full-featured controls. More information is available in the next question.

Will the Lite controls work with the other speech engines?

Yes. The Lite controls try to find the IBM ViaVoice speech engine first. If the this engine is not present, then the Lite controls will use the first suitable speech engine in the system.

If I put the Lite controls in an HTML page and someone without a speech engine wants to view the page, will they be able to?

Yes. If the **Lite** controls cannot find a speech engine, then they simply become inactive. The only requirement is that the client is using a browser that supports ActiveX technology.

How do I know if the Lite controls found a speech engine and are active?

All of the **Lite** controls have an **Enabled** property. This property allows you to activate and deactivate the **Lite** controls. However, it also notifies you if the **Lite** controls were successful in connecting to a speech engine. If the **Lite** controls were not able to find a speech engine, then they reset this property to False. So, a good way to test if there is a speech engine in the system is to set **Enabled** to True, then test the value. If it remains True, then there is a speech engine, but if it is False, then they system has no suitable speech engine.

Can I combine the Lite controls and the full-featured versions in the same application?

Yes. The **Lite** controls can be present in the same application as the full-featured versions. However, you cannot use the **Lite** controls to replace the full-featured versions whenever a full-featured version is expected. For example, you cannot use the **VVPhrasesLite** control to provide the full-featured **VVCFGram** control with an external list. The full-featured controls can only use other full-featured controls.

The ViaVoice **Error Correction Window** Control (**VVECWin**) is an ActiveX control that enables developers to utilize a common error correction dialog similar to that provided with the ViaVoice product. It provides a common, familiar user interface that users will quickly and easily become accustomed to, enabling them to correct speech recognition errors and text formatting issues, which helps the speech recognition engine to enhance and improve its speech recognition ability. The **Error Correction Window** control is also capable of understanding voice commands, which will enable the user to navigate through the contents of the **Error Correction Window** control's window with ease.

Developers will find that the **Error Correction Window** Control is easy to use and provides functionality that would otherwise be cumbersome and difficult to implement. The **Error Correction Window** control does not actually correct speech recognition errors. Instead, it provides a dialog that the programmer initializes and the user interacts with in order to identify the correction. The **Error Correction Window** informs the application through events what action the user has requested, and it is up to the application to actually perform those services.

Getting Started with the ECWin Control

The following is a tutorial on how to incorporate the **VVECWin** control into your Visual Basic or Visual C++ applications. This tutorial is designed to present you with the most commonly used properties and events in the **VVECWin** control.

Creating an Instance of the Control

In Visual Basic:

To add the **VVECWin** control to your application, do the following:

1. From the **Project** menu, choose **Components**.

The Components dialog box, Figure 34, appears. The Components dialog lists all the ActiveX Controls that you can use in your application.

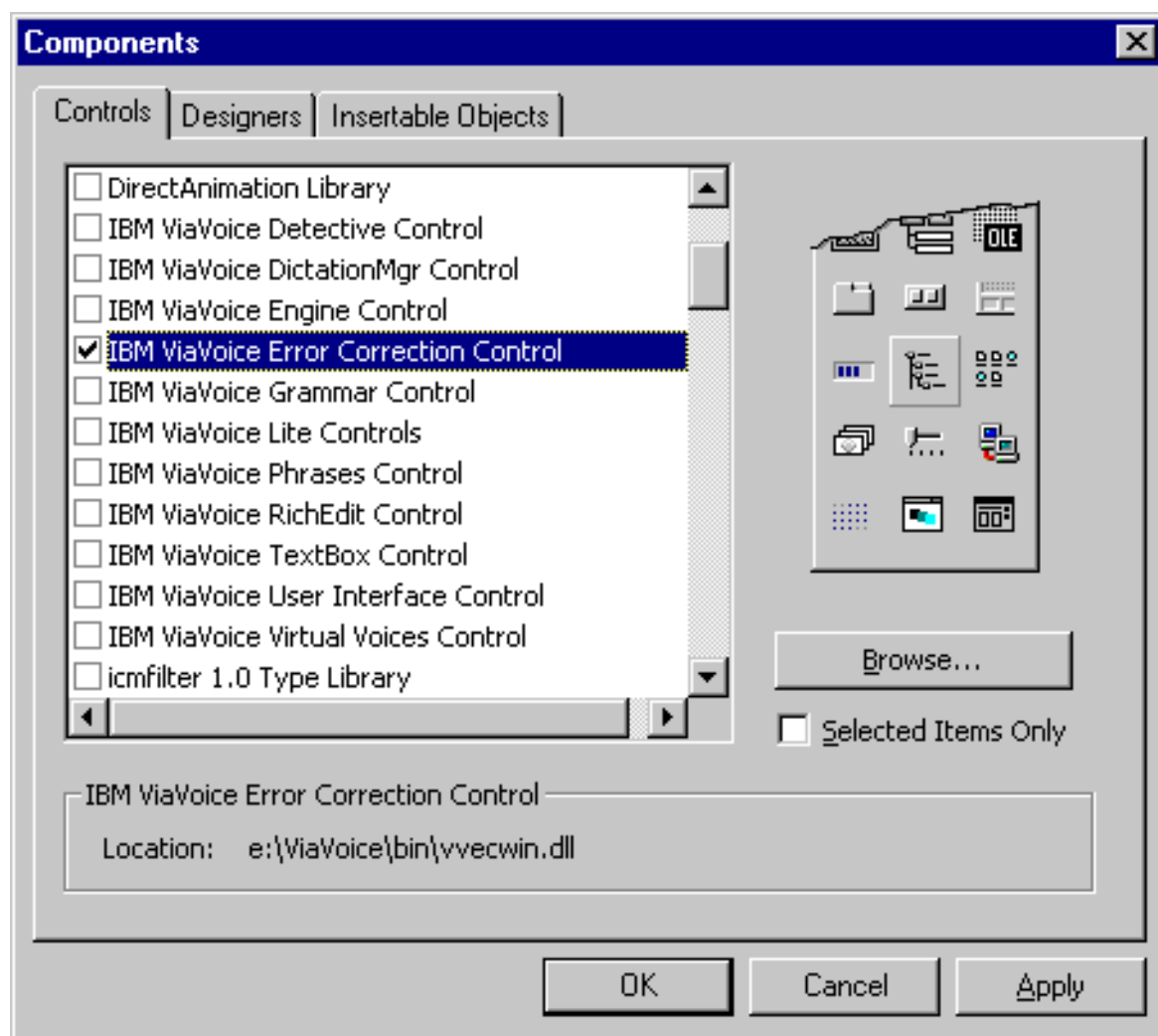


Figure 34. Component Selection Dialog - Visual Basic

2. Select **IBM ViaVoice Error Correction Control** from the list and choose **OK**.
Visual Basic adds the control to your project, and adds a new icon to the toolbar (Figure 35).



Figure 35. VVECWin Control Toolbar Icon

3. Add an instance of the **VVECWin** control to your form.
The **VVECWin** control is an invisible control at run time.

In Visual C++ (MFC):

To add the **VVECWin** control to your MFC project, do the following:

1. From the **Project** menu, select **Add To Project**, then select **Components and Controls**.
The Components and Controls Gallery dialog box, Figure 36, appears.

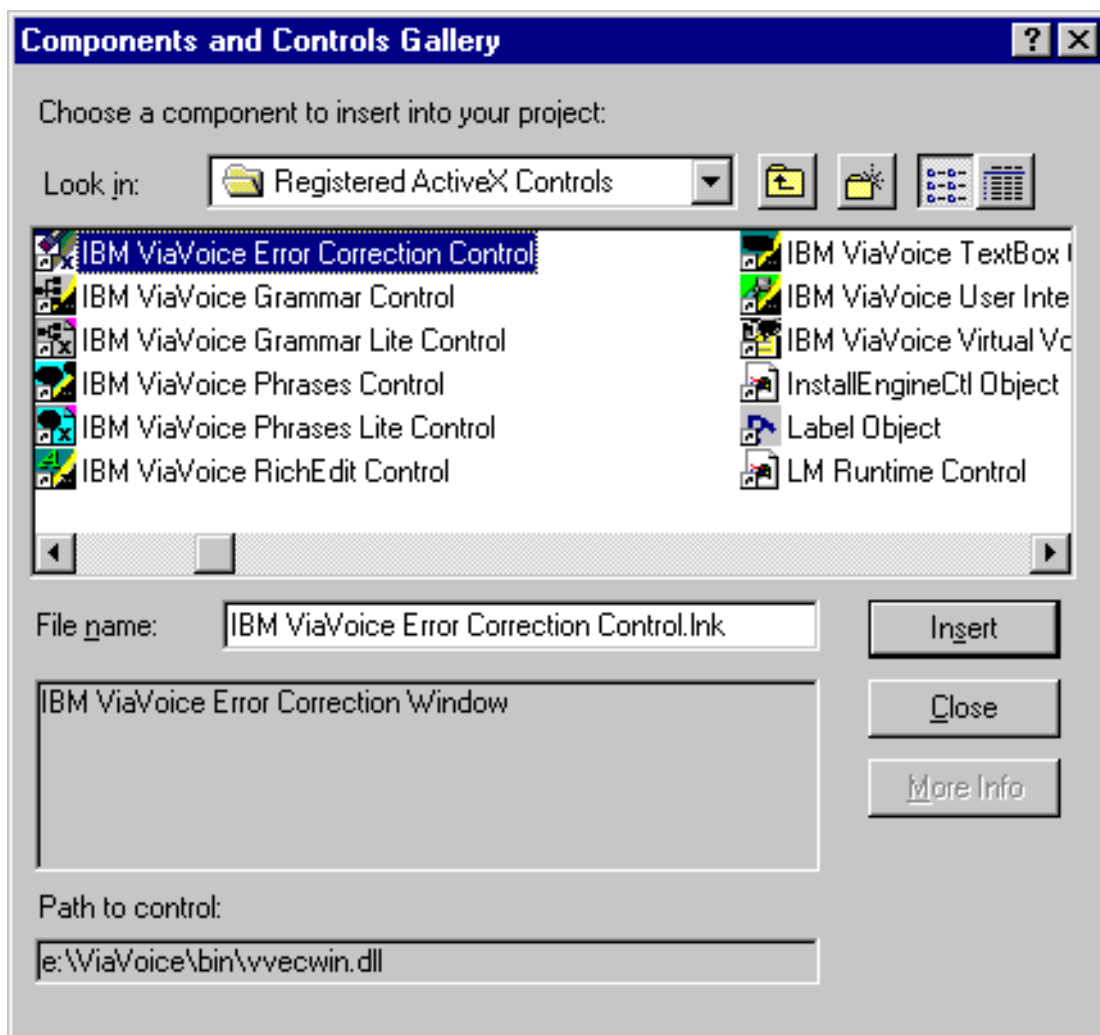


Figure 36. Insert ActiveX Control Dialog Box - Visual C++

2. Double-click the Registered ActiveX Controls folder in the dialog box.
3. Select the **IBM ViaVoice Error Correction Control** icon in the list of controls, then click **Insert**.
A confirmation message box appears, asking "Insert this component?"
4. Respond to the confirmation message box by choosing **OK**.

The Confirm Classes dialog box, Figure 37, appears listing the components in the **VVECWin** control:

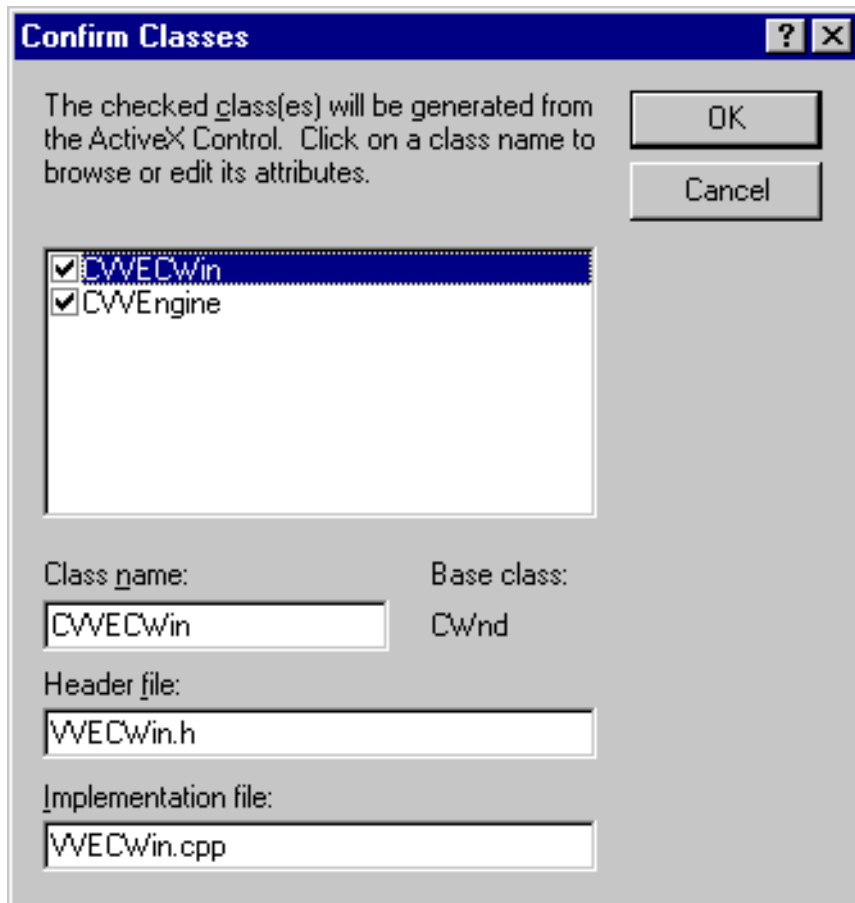


Figure 37. Confirm Classes Dialog Box

5. Click **OK** in the Confirm Classes dialog box.
6. Close the “Components and Controls Gallery” dialog box.
If you examine the Project Workspace window in the class view you will notice 2 new classes: **CVVECWin** and **CVVEngine** (assuming you accepted the default names for the class in the Confirm Classes dialog box).

7. In the resource view of your Project Workspace window double-click the dialog resource entry where you wish to insert the **VVECWin** control.

The **VVECWin** icon, Figure 38, appears in the Controls toolbar.



Figure 38. VVECWin Icon in the Controls Toolbar

8. Add an instance of the **VVECWin** control to the dialog box.

After you add the **VVECWin** control to your dialog you can invoke Class Wizard to create a member variable for your class of type **CVVECWin**. You might also decide to capture the events in the control by adding Event handlers to your dialog class. To add Event handlers, click the **VVECWin** control with the right mouse button, then select **Events** from the pop-up menu.

Initializing the Error Correction Window Control

The first step in using the **Error Correction Window** control is to initialize it. Typically an application will follow these steps in order to initialize the control and display the **Error Correction Window**.

1. · Initialize the control
2. · Set the text to be corrected
3. · Add alternate pronunciations from which the user can select
4. · Add custom menu items
5. · Display the window

To initialize the control, simply call the **Init** method giving the window handle of the parent window.

In Visual Basic:

```
VVECTWin.Init hWnd
```

In Visual C++ (MFC):

```
CVVECTWin.Init( (long)m_hWnd );
```

The next step is to set the text to be corrected. Typically this is the text returned from the speech recognition engine which was incorrectly recognized, or text that has incorrect punctuation, capitalization, etc. The **CorrectText** string will be displayed in the **Error Correction Window**'s edit field. Set the **CorrectText** property as follows:

In Visual Basic:

```
VVECWin.CorrectText = "bask"
```

In Visual C++ (MFC):

```
CVVECWin.SetCorrectText( "bask" );
```

The next step is to add alternate pronunciation strings to the alternates listbox. This listbox is used to show a list of alternate words that the speech engine recognizes that are close in pronunciation to the word the user dictated. The user can use their mouse or voice commands to select one of these alternates, or they can type their selection in the edit field.

The first parameter is the text spelling of the alternate word, which is displayed in the listbox and will be returned to the application in the **WordSelected** event if the user selects that alternate. The second parameter is an optional "Sounds Like" word, which is not displayed but will also be returned to the application in the **WordSelected** event. The last parameter determines if the window is to be repainted after adding the word, and should typically be FALSE on all additions except the last one. Alternates are shown in the listbox with a one-based index value to the left of the alternate spelling.

In Visual Basic:

```
VVECWin.AddAlternate "task", "", False  
VVECWin.AddAlternate "past", "", False  
VVECWin.AddAlternate "ask", "", True
```

In Visual C++ (MFC):

```
CVVECWin.AddAlternate( "task", "", FALSE );  
CVVECWin.AddAlternate( "past", "", FALSE );  
CVVECWin.AddAlternate( "ask", "", TRUE );
```

The next step is to optionally add custom menu items and event IDs. By default the "**Format**" menu button is not displayed, but will appear if one or more menu items have been added. Use a custom menu for special formatting options such as capitalization and punctuation. The **AddMenuItem** method takes two parameters, the text string to appear on the menu and a menu ID that will be passed back to the application in the **MenuSelected** event. Your menu ID should be unique values that your event handler will use to identify the menu item selected.

In Visual Basic:

```
VVECWin.AddItem "&Capitalize", 100
VVECWin.AddItem "&Lowercase", 200
VVECWin.AddItem "&Uppercase", 300
```

In Visual C++ (MFC):

```
CVVECWin.AddItem("&Capitalize", 100);
CVVECWin.AddItem("&Lowercase", 200);
CVVECWin.AddItem("&Uppercase", 300);
```

Finally, to make the window visible, you will call the **Show** method as follows. The only parameter is a boolean that controls whether or not the window should receive input focus when displayed.

In Visual Basic:

```
VVECWin.Show True
```

In Visual C++ (MFC):

```
CVVECWin.Show(TRUE);
```

Handling Error Correction Window Control Events

The **Error Correction Window** control can fire the following events to the calling application.

- **ButtonSelected**
- **Close**
- **ContextHelpRequest**
- **FocusChange**
- **MenuSelected**
- **WordSelected**

ButtonSelected

The **ButtonSelected** event is fired when the user selects one of the following buttons on the **Error Correction Window**. The Button ID is passed as the only parameter, and can be one of the following:

Mnemonic	ID	Description
vvecUSER_BUTTON	2	User selected the " Format " button, which contains menu items added through the AddMenuItem method. The VVECWin control will also fire the MenuSelected event with the selected menu ID.
vvecPLAY_BUTTON	3	User selected the " Play " button. The application will typically respond to this event by playing back the recorded wave file for the given " CorrectText " word.
vvecDELETE_BUTTON	4	User selected the " Delete " button. The application will typically respond to this event by deleting the given " CorrectText " word from the application.

vvecCORRECT_BUTTON	5	User selected the " Correct " button. The VVECWin control will also fire the WordSelected event with the desired spelling.
vvecADDPHRASE_BUTTON	6	User clicked the " Add words or phrases " checkbox. When this checkbox is checked, the application will typically respond by adding the text as a single phrase. If the user spoke "New York" and it was incorrectly recognized as "Newark", then the user would open the Error Correction Window , type "New York" in the edit field, check the " Add words or phrases " checkbox, and click the " Correct " button. The application would then know to update the vocabulary with the complete phrase "New York".

Close

The **Close** event is fired when the **Error Correction Window** is made invisible by a user action. This could happen in response to a voice command or the user clicking the close button on the menu. The event is not fired when application calls the **Hide** method. There are no parameters. The control remains active and the window handle remains valid.

ContextHelpRequest

The **ContextHelpRequest** event is fired when the user presses the F1 key while the **Error Correction Window** has focus. The Help ID is passed as the only parameter. The application can respond to this event by displaying its help tool.

FocusChange

The **FocusChange** event is fired whenever the **Error Correction Window** gains or loses focus. The only parameter is a boolean value that is TRUE if the window is gaining focus and FALSE if the window is losing focus. The application can respond to this event as appropriate, such as enabling/disabling its voice command support.

MenuSelected

The **MenuSelected** event is fired when the user selects one of the custom menu options added through the **AddMenuItem** method. The application specified menu ID, which was used to create the menu entry, is passed as the only parameter. The application should respond by handling the menu item as appropriate.

WordSe2224

The **WordSelected** event is fired when the user selects the word to be used as the corrected text. There are three parameters, the selected spelling, an optional sounds-like spelling, and an index identifier. The user can do this by clicking or using voice commands to select one of the alternate words in the alternates listbox, or by typing a word in the edit field and selecting the "**Correct**" button. An index value of -1 indicates that the user chose the correct text by entering it in the edit field and clicking the "**Correct**" button, and values 0 and above are the zero based indices from words selected from the alternates listbox.

Error Correction Window Control Voice Command Support

The Error Correction Window Control **VVECWin** supports a number of voice commands, which are active when the **Error Correction Window** has the input focus. Supported commands include the following:

Voice Command	Description
Pick [index number]	Selects the alternate with the specified index. The indices are displayed on the left side of each listbox entry.
Hide Correction Window	Hides the Error Correction Window .
[Added Menu Items]	Selects the spoken menu item, valid only when the menu is displayed.
Add words or phrases	Sets input focus to the " Add words or phrases " checkbox.
Format	Selects the " Format " button.
Play	Selects the " Play " button.
Delete	Selects the " Delete " button.
Correct	Selects the " Correct " button.
Alternate Word List	Sets input focus to the alternate words listbox.
Edit Word	Sets input focus to the edit field.

The following is a list of properties, methods, and events pertaining to the **Error Correction Window Control**.

Error Correction Window Control Properties

The **VVECWin** control supports the following properties:

- **AddPhraseChecked**
- **AddPhraseVisible**
- **Caption**
- **ChildEnabled**
- **CommandsEnabled**
- **CorrectText**
- **Enabled**
- **Engine**
- **hWnd**
- **LanguageUI**
- **NumVisibleAlternates**
- **StatusBarVisible**
- **StatusText**

AddPhraseChecked

Returns/sets whether or not the 'Add words or phrases' check box is checked when the **Error Correction Window** is visible.

Syntax

In Visual Basic:

```
[VVECWin].AddPhraseChecked = [Boolean]
```

In Visual C++ (MFC):

```
BOOL [CVVECWin].GetAddPhraseChecked ();  
Void [CVVECWin].SetAddPhraseChecked (BOOL fNewValue);
```

In Visual C++:

```
HRESULT [pIVVECWin]->get_AddPhraseChecked ( VARIANT_BOOL * pbValue );  
HRESULT [pIVVECWin]->put_AddPhraseChecked ( VARIANT_BOOL fNewValue );
```

Parameters

fNewValue
??

Return Values

TRUE

Add the corrected text as a single phrase.

FALSE

Add the corrected text as single words.

Remarks

You can change this property whether or not the **Error Correction Window** is actually visible. You should first set the **AddPhraseVisible** property to TRUE to ensure that the checkbox is visible. Users can only change the checkbox setting when it is enabled. When the user checks this checkbox (**AddPhraseChecked** is TRUE), the application should add the specified correction text as a single phrase.

Example

In Visual Basic:

```
VVECWin.AddPhraseChecked = False

Dim CheckState As Boolean
CheckState = VVECWin.AddPhraseChecked
```

In Visual C++ (MFC):

```
//Ensure checkbox is enabled and visible
m_CVVECWin.SetAddPhraseVisible(TRUE);
m_CVVECWin.SetChildEnabled(vvecADDPHRASE_BUTTON, TRUE);

//Ensure checkbox is not checked
m_CVVECWin.SetAddPhraseChecked(FALSE);
BOOL bValue = m_CVVECWin.GetAddPhraseChecked();
```

In Visual C++:

```
//Ensure checkbox is enabled and visible
m_pIVVEWin->put_AddPhraseVisible(VARIANT_TRUE);
m_pIVVEWin->put_ChildEnabled(vvecADDPHRASE_BUTTON, VARIANT_TRUE);

//Ensure checkbox is not checked
m_pIVVEWin->put_AddPhraseChecked(VARIANT_FALSE );
VARIANT_BOOL bValue;
m_pIVVEWin->get_AddPhraseChecked( &bValue );
```

See Also

“ChildEnabled” on page 439
“AddPhraseVisible” on page 435
“CommandsEnabled” on page 442
“Enabled” on page 446
“ButtonSelected” on page 482
“CorrectText” on page 444
“Reset” on page 477

AddPhraseVisible

Returns/sets whether or not the “Add words or phrases” check box is visible when the **Error Correction Window** is displayed

Syntax

In Visual Basic:

```
[VVECWin].AddPhraseVisible = [Boolean]
```

In Visual C++ (MFC):

```
BOOL [CVVECWin].GetAddPhraseVisible();  
void [CVVECWin].SetAddPhraseVisible(BOOL fNewValue);
```

In Visual C++:

```
HRESULT [pIVVECWin]->get_AddPhraseVisible(VARIANT_BOOL * pVal);  
HRESULT [pIVVECWin]->put_AddPhraseVisible(VARIANT_BOOL newValue);
```

Parameters

fNewValue
??

Return Values

TRUE

“Add words or phrases” check box is visible.

FALSE

“Add words or phrases” check box is not visible.

Remarks

You can change this property whether or not the **Error Correction Window** is actually visible. It will take effect when the window is shown. The check box is initially displayed unchecked. The text "Add words or phrases" is displayed next to the checkbox when visible, in the language specified by the **LanguageUI** property.

Example

In Visual Basic:

```
VVECWin.AddPhraseVisible = True
```

In Visual C++ (MFC):

```
m_CVVECWin.SetAddPhraseVisible(TRUE);
```

In Visual C++:

```
m_pIVECWin->put_AddPhraseVisible(VARIANT_TRUE);  
VARIANT_BOOL bValue;  
m_pIVECWin->get_AddPhraseVisible( &bValue );
```

See Also

“AddPhraseChecked” on page 432

“ChildEnabled” on page 439

“CommandsEnabled” on page 442

“LanguageUI” on page 452

“Enabled” on page 446

Caption

Sets/gets the text that is displayed on the title bar of the **Error Correction Window**.

Syntax

In Visual Basic:

```
[VVECWin].Caption = [String]
```

In Visual C++ (MFC):

```
CString [CVVECWin].GetCaption();  
Void [CVVECWin].SetCaption(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT [pIVVECWin]->get_Caption(BSTR* pstrCaption);  
HRESULT [pIVVECWin]->put_Caption(BSTR strCaption);
```

Parameters

None.

Return Values

None.

Remarks

By default the caption is set to "Error Correction", in the language specified by the **LanguageUI** property.

Example

In Visual Basic:

```
VVECWin.Caption = "Custom Error Correction"  
VVECWin.Show True
```

In Visual C++ (MFC):

```
m_CVVECWin.SetCaption("Custom Error Correction");  
m_CVVECWin.Show(TRUE);
```

In Visual C++:

```
m_pIVVECWin->put_Caption(SysAllocString(L"Custom Error Correction"));  
m_pIVVECWin->Show(VARIANT_TRUE);
```

See Also

“LanguageUI” on page 452

ChildEnabled

Returns/sets the enabled state of the specified child control of the **Error Correction Window**.

Syntax

In Visual Basic:

```
[VVECWin].ChildEnabled( [buttonID] ) = [Boolean]  
[Boolean] = [VVECWin].ChildEnabled( [buttonID] )
```

In Visual C++ (MFC):

```
BOOL [CVVECWin].GetChildEnabled(long lButtonID );  
void [CVVECWin].SetChildEnabled(long lButtonID, BOOL fNewValue);
```

In Visual C++:

```
HRESULT [pIVVECWin]->get_ChildEnabled(VVECBUTTONID lButtonID,  
VARIANT_BOOL * pValue );  
  
HRESULT [pIVVECWin]->put_ChildEnabled(VVECBUTTONID lButtonID,  
VARIANT_BOOL fNewValue );
```

Parameters

fNewValue
Boolean.

Return Values

TRUE
??

FALSE
??

Remarks

You can change this property whether or not the **Error Correction Window** is actually visible. You should use the following values (available in the `vvecwin.h` file distributed as part of the SDK) to specify which child control you are referring to. Disabled controls are typically grayed out and cannot receive focus.

Mnemonic	ID	Description
<code>vvecALTERNATES_LIST</code>	0	Control ID for the Alternate Words listbox.
<code>vvecRECO_EDIT</code>	1	Control ID for the edit field.
<code>vvecUSER_BUTTON</code>	2	Control ID for the user-programmable button (the "Format" button in Tool Tips).
<code>vvecPLAY_BUTTON</code>	3	Control ID for the "Play" button.
<code>vvecDELETE_BUTTON</code>	4	Control ID for the "Delete" button.
<code>vvecCORRECT_BUTTON</code>	5	Control ID for the "Correct" button.
<code>vvecADDPHRASE_BUTTON</code>	6	Control ID for the "Add words or phrases" checkbox.

Example

In Visual Basic:

```
Dim bEna As Boolean
VVECWin.ChildEnabled(vvecPLAY_BUTTON) = False
BEna = VVECWin.ChildEnabled(vvecPLAY_BUTTON)
```


In Visual C++ (MFC):

```
//Include Enumerated type for button IDs
#include "..\ViaVoice SDK\Include\vecwin.h"

//Ensure "Play" button is disabled
m_CVVECWin.SetChildEnabled(vvecPLAY_BUTTON, FALSE);
BOOL bValue = m_CVVECWin.GetChildEnabled( vvecPLAY_BUTTON );
```

In Visual C++:

```
//Ensure "Play" button is disabled

m_pIVVECWin->put_ChildEnabled(vvecPLAY_BUTTON, VARIANT_FALSE);
VARIANT_BOOL bValue;
m_pIVVECWin->get_ChildEnabled(vvecPLAY_BUTTON, &bValue );
```

See Also

“Init” on page 470

“Show” on page 479

“ButtonSelected” on page 482

CommandsEnabled

Returns/sets whether or not the **Error Correction Window** should connect to an internal **VVPhrase** control in order to provide voice navigation of the window with commands such as "Pick One" or "Hide Correction Window."

Syntax

In Visual Basic:

```
[VVECWin].CommandsEnabled = [Boolean]
```

In Visual C++ (MFC):

```
BOOL [CVVECWin].GetCommandsEnabled();  
void [CVVECWin].SetCommandsEnabled(BOOL fNewValue);
```

In Visual C++:

```
HRESULT [pIVVECWin]->get_CommandsEnabled(VARIANT_BOOL * pVal);  
HRESULT [pIVVECWin]->put_CommandsEnabled(VARIANT_BOOL newVal);
```

Parameters

fNewValue
Boolean.

Return Values

TRUE
??

FALSE
??

Example

In Visual Basic:

```
VVECWin.CommandsEnabled = True
```

In Visual C++ (MFC):

```
m_CVVECWin.SetCommandsEnabled(TRUE);  
BOOL bEnabled = m_CVVECWin.GetCommandsEnabled();
```

In Visual C++:

```
m_pIVVECWin->put_CommandsEnabled(VARIANT_TRUE);  
VARIANT_BOOL bValue;  
m_pIVVECWin->get_CommandsEnabled( &bValue );
```

See Also

“Error Correction Window Control Voice Command Support” on page 429.

CorrectText

Sets the text that the user wants to have corrected in the **Error Correction Window**.

Syntax

In Visual Basic:

```
[VVECTWin].CorrectText = [String]
```

In Visual C++ (MFC):

```
CString [CVVECTWin].GetCorrectText();  
void [CVVECTWin].SetCorrectText(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT [pIVVECTWin]->get_CorrectText(BSTR* pVal);  
HRESULT [pIVVECTWin]->put_CorrectText(BSTR newVal);
```

Parameters

None.

Return Values

None.

Remarks

Returns the text the user has currently selected or typed as a replacement for the original text to be corrected.

Example

In Visual Basic:

```
VVECWin.CorrectText = "word"
VVECWin.Show True
. . .
'after window closed
Debug.Print "User chose to replace 'word' with " & _
VVECWin.CorrectText & "."
```

In Visual C++ (MFC):

```
m_CVVECWin.SetCorrectText("word");
m_CVVECWin.Show(TRUE);
. . .
//after window closed
CString strResult = m_CVVECWin.GetCorrectText();
TRACE("The user chose to replace 'word' with '%s'", strResult);
```

In Visual C++:

```
m_pIVVECWin->put_CorrectText(SysAllocString(L"word"));
m_pIVVECWin->Show(VARIANT_TRUE);
. . .
//after window closed
BSTR bstrResult = NULL;
m_pIVVECWin->get_CorrectText(&bstrResult);
::MessageBoxW(NULL, bstrResult, L"Corrected Text:", MB_OK);
```

See Also

“WordSelected” on page 493
 “AddPhraseVisible” on page 435
 “AddPhraseChecked” on page 432

Enabled

Returns/sets whether or not the ViaVoice **Error Correction Window** appears enabled or disabled (grayed).

Syntax

In Visual Basic:

```
[VVECWin].Enabled = [Boolean]
```

In Visual C++ (MFC):

```
[BOOL] = [CVVECWin].GetEnabled();  
void [CVVECWin].SetEnabled(BOOL fNewValue);
```

In Visual C++:

```
HRESULT [pIVVECWin]->get_Enabled(VARIANT_BOOL * pbEnabled);  
HRESULT [pIVVECWin]->put_Enabled(VARIANT_BOOL bEnabled);
```

Parameters

fNewValue
Boolean.

Return Values

TRUE
??

FALSE
??

Remarks

None.

Example

In Visual Basic:

```
VVECWin.Enabled = True
```

In Visual C++ (MFC):

```
m_CVVECWin.SetEnabled(TRUE);  
BOOL bEnabled = m_CVVECWin.GetEnabled();
```

In Visual C++:

```
m_pIVVECWin->put_Enabled(VARIANT_TRUE);  
VARIANT_BOOL bEnabled;  
m_pIVVECWin->get_Enabled( &bEnabled );
```

See Also

“ChildEnabled” on page 439

“CommandsEnabled” on page 442

“Show” on page 479

“Hide” on page 468

Engine

Contains a reference to the ViaVoice **Engine** control (**VVEngine**), which is used by the **Error Correction Window** control for voice navigation.

Syntax

In Visual Basic:

```
[VVEngine] = [VVECWin].Engine
```

In Visual C++ (MFC):

```
[CVVEngine] = [CVVECWin].GetEngine();  
void [CVVECWin].SetRefEngine(LPDISPATCH newValue);
```

In Visual C++:

```
HRESULT [pIVVECWin]->get_Engine(IVVEngine * * ppVal);  
HRESULT [pIVVECWin]->putref_Engine(IVVEngine * pNewVal);
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
'Get language used by engine
Dim EngineLanguage As String
EngineLanguage = VVECWin.Engine.Language
```

In Visual C++ (MFC):

```
//Get language used by engine
CVVEngine CVVEng;
CVVEng = m_CVVECWin.GetEngine();
CString EngineLanguage = CVVEng.GetLanguage();
```

In Visual C++:

```
IVVEngine *pIVVEngine;
HRESULT hr;

hr = [pIVVECWin]->get_Engine(&pIVVEngine);
if (FAILED(hr))
    return hr;
pIVVEngine->Connect();
```

See Also

“CommandsEnabled” on page 442

hWnd

Returns the **hWnd** of the **Error Correction Window** if the **Error Correction Window** has been created (through a call to **Init**), this read-only property.

Syntax

In Visual Basic:

```
[long] = [VVECWin].hWnd
```

In Visual C++ (MFC):

```
long [CVVECWin].GetHWND();
```

In Visual C++:

```
HRESULT [pIVVECWin]->get_hWnd(long * pHWnd);
```

Parameters

??

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
Dim hWndEC As Long
VVECWin.Init hWnd
VVECWin.Show True
hWndEC = VVECWin.hWnd
```

In Visual C++ (MFC):

```
m_CVVECWin.Init( (long)m_hWnd );
m_CVVECWin.Show( TRUE );
long lhWndEC = m_CVVECWin.GetHWnd();
```

In Visual C++:

```
m_pIVVECWin->Init( (long)m_hWnd );
m_pIVVECWin->Show(VARIANT_TRUE);
long lhWndEC;
m_pIVVECWin->get_hWnd( &lhWndEC );
```

See Also

“Init” on page 470

“Show” on page 479

LanguageUI

Sets or gets the language used by the ViaVoice **Error Correction Window** for this specific client.

Syntax

In Visual Basic:

```
[VVECWin].LanguageUI = [String]
```

In Visual C++ (MFC):

```
CString = [CVVECWin].GetLanguageUI();  
void [CVVECWin].SetLanguageUI(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT [pIVVECWin]->get_LanguageUI(BSTR* pVal);  
HRESULT [pIVVECWin]->put_LanguageUI(BSTR newVal);
```

Parameters

??

Return Values

The **LanguageUI** property settings for a ViaVoice **Error Correction** control are:

Value	Description
"EN_US"	U.S. English
"EN_UK"	U.K. English
"GR_GR"	German
"IT_IT"	Italian
"ES_ES"	Spanish

Value	Description
"FR_FR"	French
"JA_JP"	Japanese

Remarks

The language affects any dialogs, menus, strings or ToolTips displayed by the control. Changing this property affects the text strings that are displayed in the control, including the Caption, the text string displayed on the "Add words or phrases" checkbox, and the tool tips displayed when the mouse moves over the format, play, delete, and correct buttons. This property defaults to the language of the installed speech recognition engine.

Example

In Visual Basic:

```
VVECWin.LanguageUI = "EN_US"
VVECWin.Init hWnd
VVECWin.Show True
```

In Visual C++ (MFC):

```
m_CVVECWin.SetLanguageUI("EN_US");
m_CVVECWin.Init((long)m_hWnd);
m_CVVECWin.Show(TRUE);
```

In Visual C++:

```
m_pIVECWin->put_LanguageUI(SysAllocString(L"EN_US"));
m_pIVECWin->Init((long)m_hWnd);
m_pIVECWin->Show(VARIANT_TRUE);
```

See Also

“Init” on page 470

“Show” on page 479

NumVisibleAlternates

Returns or sets the number of alternates (alternative spellings of the word provided by the **CorrectText** property) to initially make visible in the ViaVoice **Error Correction Window**.

Syntax

In Visual Basic:

```
[VVECWin].NumVisibleAlternates = [long]
```

In Visual C++ (MFC):

```
long [CVVECWin].GetNumVisibleAlternates();  
[CVVECWin].SetNumVisibleAlternates(long nNewValue);
```

In Visual C++:

```
HRESULT [pIVVECWin]->get_NumVisibleAlterantes( long * pVal );  
HRESULT [pIVVECWin]->put_NumVisibleAlternates(long newVal);
```

Parameters

??

Return Values

??

Remarks

For this property to take effect it must be set before calling the **Init** method. By default this property is set to five. Changing the default value affects the height of the listbox contained in the **Error Correction Window**. More alternates can still be added, but the user will have to scroll through the alternates list box or resize the **Error Correction Window** to view them.

Example

In Visual Basic:

```
VVECWin.NumVisibleAlternates = 7  
VVECWin.Init hWnd
```

In Visual C++ (MFC):

```
m_CVVECWin.SetNumVisibleAlternates(7);  
m_CVVECWin.Init( (long)m_hWnd );
```

In Visual C++:

```
m_pIVVECWin->put_NumVisibleAlternates(7);  
m_pIVVECWin->Init( (long)m_hWnd );
```

See Also

“AddAlternate” on page 461

“MoveWindow” on page 474

“Init” on page 470

StatusBarVisible

Returns/sets whether or not the status bar at the bottom of the ViaVoice **Error Correction Window** is displayed.

Syntax

In Visual Basic:

```
[VVECTWin].StatusBarVisible = [Boolean]
```

In Visual C++ (MFC):

```
BOOL [CVVECTWin].GetStatusBarVisible();  
void [CVVECTWin].SetStatusBarVisible(BOOL fNewValue);
```

In Visual C++:

```
HRESULT [pIVVECTWin]->get_StatusBarVisible(VARIANT_BOOL* pbVal);  
HRESULT [pIVVECTWin]->put_StatusBarVisible(VARIANT_BOOL fNewValue);
```

Parameters

fNewValue
Boolean.

Return Values

TRUE

FALSE

Remarks

None.

Example

In Visual Basic:

```
VVECWin.Init hWnd
VVECWin.StatusBarVisible = True
VVECWin.StatusText = "Select correct spelling"
VVECWin.Show True
```

In Visual C++ (MFC):

```
m_CVVECWin.Init( (long)m_hWnd );
m_CVVECWin.SetStatusBarVisible(TRUE);
m_CVVECWin.SetStatusText("Select correct spelling");
m_CVVECWin.Show(TRUE);
```

In Visual C++:

```
M_pIIVVECWin->Init( (long)m_hWnd );
m_pIIVVECWin->put_StatusBarVisible(VARIANT_TRUE);
m_pIIVVECWin->put_StatusText(SysAllocString(L"Select correct spelling"));
m_pIIVVECWin->Show(VARIANT_TRUE);
```

See Also

“StatusText” on page 458

StatusText

Returns/sets the text displayed in the status bar of the ViaVoice **Error Correction Window**.

Syntax

In Visual Basic:

```
[VVECWin].StatusText = [String]
```

In Visual C++ (MFC):

```
CString = [VVECWin].GetStatusText();  
void [VVECWin].SetStatusText(LPCTSTR lpszNewValue);
```

In Visual C++:

```
HRESULT [pIVVECWin]->get_StatusText(BSTR * pVal);  
HRESULT [pIVVECWin]->put_StatusText(BSTR newVal);
```

Parameters

None.

Return Values

None.

Remarks

None.

Example

In Visual Basic:

```
VVECWin.Init hWnd
VVECWin.StatusBarVisible = True
VVECWin.StatusText = "Select correct spelling"
VVECWin.Show True
```

In Visual C++ (MFC):

```
m_CVVECWin.Init( (long)m_hWnd );
m_CVVECWin.SetStatusBarVisible(TRUE);
m_CVVECWin.SetStatusText("Select correct spelling");
m_CVVECWin.Show(TRUE);
```

In Visual C++:

```
m_pIIVVECWin->Init( (long)m_hWnd );
m_pIIVVECWin->put_StatusBarVisible(VARIANT_TRUE);
m_pIIVVECWin->put_StatusText(SysAllocString(L"Select correct spelling"));
m_pIIVVECWin->Show(VARIANT_TRUE);
```

See Also

“StatusBarVisible” on page 456

Error Correction Window Control Methods

The ViaVoice **Error Correction Window** control supports the following methods:

- **AddAlternate**
- **AddMenuItem**
- **GetWindowRect**
- **Hide**
- **Init**
- **IsVisible**
- **MoveWindow**
- **Reset**
- **Show**

AddAlternate

Adds an alternate word and an optional sounds-like word to the alternate-words listbox in the ViaVoice **Error Correction Window**. Optionally redraws the window.

Syntax

In Visual Basic:

```
[VVECWin].AddAlternate AlternateWord As String, SoundsLike As String,  
Redraw As Boolean
```

In Visual C++ (MFC):

```
void [CVVECWin].AddAlternate(LPCTSTR AlternateWord, LPCTSTR SoundsLike,  
BOOL Redraw);
```

In Visual C++:

```
HRESULT [pIVVECWin]->AddAlternate(BSTR AlternateWord, BSTR SoundsLike,  
VARIANT_BOOL Redraw);
```

Parameters

AlternateWord

String. Word to be displayed in the alternate-word listbox.

SoundsLikeWord

String. Sounds-like word, which is not displayed but is returned in the WordSelected event. Can be a NULL string.

Redraw

Boolean. Redraws window after adding new string. Typically set to False while adding all alternates except the last.

Return Values

TRUE

(On last item) Clears all items in the listbox.

FALSE

Adds another alternate word.

Remarks

The **Init** method must be called before **AddAlternate** can be used. Use the **AddAlternate** method to populate the listbox with alternates that the user can select by clicking with the mouse or by voice commands (i.e., saying "Pick 2"). The application is informed of the selected word through the **WordSelected** event. The SoundsLike word is not displayed, but is internally saved in the control and returned to the caller through the **WordSelected** event. Add all alternates except the last with Redraw FALSE, and set Redraw TRUE on the final alternate. Clear all items in the listbox with the **Reset** method.

Example

In Visual Basic:

```
VVECWin.Init hWnd
VVECWin.CorrectText = "two"
VVECWin.AddAlternate "to", "two", False
VVECWin.AddAlternate "too", "", True
VVECWin.Show True
```

In Visual C++ (MFC):

```
m_CVVECWin.Init((long)m_hWnd);
m_CVVECWin.SetCorrectText("two");
m_CVVECWin.AddAlternate( "to", "two", FALSE);
m_CVVECWin.AddAlternate( "too", "", TRUE);
m_CVVECWin.Show(TRUE);
```

In Visual C++:

```
m_pIVVECWin->Init( (long)m_hWnd );
m_pIVVECWin->put_CorrectText(SysAllocString("two"));
m_pIVVECWin->AddAlternate(SysAllocString(L"to"), SysAllocString(L"two"),
VARIANT_FALSE);
m_pIVVECWin->AddAlternate(SysAllocString(L"too"), SysAllocString(L""),
VARIANT_TRUE);
m_pIVVECWin->Show( VARIANT_TRUE );
```

See Also

- “CommandsEnabled” on page 442
- “CorrectText” on page 444
- “NumVisibleAlternates” on page 454
- “Init” on page 470
- “Reset” on page 477
- “WordSelected” on page 493

AddMenuItem

Adds a custom menu item to the ViaVoice **Error Correction Window's** Format button.

Syntax

In Visual Basic:

```
[VVECTWin].AddMenuItem MenuText As String, MenuID As Long
```

In Visual C++ (MFC):

```
void [CVVECTWin].AddMenuItem(LPCTSTR MenuText, long MenuID);
```

In Visual C++:

```
HRESULT pIVVECTWin->AddMenuItem(BSTR MenuText, long MenuID);
```

Parameters

MenuText

String. Text string to appear in custom menu, when user presses the Format button.

MenuID

Long. Value to be passed to application in **MenuSelected** event, when user selects a menu item.

Return Values

??

Remarks

Use this method to add custom items to the popup menu displayed when the user presses the 'Format' button. The Format button is not displayed unless at least one menu item has been added. The **MenuSelected** event is fired with the given **MenuID** when the user selects a custom menu item.

Example

In Visual Basic:

```
VVECWin.AddItem "&Capitalize", 100  
VVECWin.AddItem "&Lowercase'", 200  
VVECWin.AddItem "&Uppercase", 300
```

In Visual C++ (MFC):

```
m_CVVECWin.AddItem("&Capitalize", 100);  
m_CVVECWin.AddItem("&Lowercase'", 200);  
m_CVVECWin.AddItem("&Uppercase", 300);
```

In Visual C++:

```
m_pIVVECWin->AddMenuItem( SysAllocString(L"&Capitalize"), 100);  
m_pIVVECWin->AddMenuItem( SysAllocString(L"&Lowercase'"), 200);  
m_pIVVECWin->AddMenuItem( SysAllocString(L"&Uppercase'"), 300);
```

See Also

“ButtonSelected” on page 482

“MenuSelected” on page 491

GetWindowRect

Returns the origin and dimensions of the **Error Correction Window** if it has been displayed.

Syntax

In Visual Basic:

```
[VVECWin].GetWindowRect px As Long, py As Long, pWidth As Long, pHeight As Long
```

In Visual C++ (MFC):

```
void [CVVECWin].GetWindowRect(long* px, long* py, long* pWidth, long* pHeight);
```

In Visual C++:

```
HRESULT [pIVVECWin]->GetWindowRect(long* px, long* py, long* pWidth, long* pHeight);
```

Parameters

px

Long. Pointer to variable to receive x coordinate.

py

Long. Pointer to variable to receive y coordinate.

pWidth

Long. Pointer to variable to receive window width.

pHeight

Long. Pointer to variable to receive window height.

Return Values

??

Remarks

The dimensions are given in screen coordinates that are relative to the upper-left corner of the screen. The **Error Correction Window** can be moved and resized with the **MoveWindow** method.

Example

In Visual Basic:

```
Dim x As Long
Dim y As Long
Dim Width As Long
Dim Height As Long
VVECWin.Init hWnd
VVECWin.Show True
VVECWin.GetWindowRect x,y,Width,Height
```

In Visual C++ (MFC):

```
long x,y,Width,Height;
m_CVVECWin.Init( (long)m_hWnd );
m_CVVECWin.Show(TRUE);
m_CVVECWin.GetWindowRect( &x, &y, &Width, &Height);
```

In Visual C++:

```
long x,y,Width,Height;
m_pIVVECWin->Init( (long)m_hWnd );
m_pIVVECWin->Show( VARIANT_TRUE );
m_pIVVECWin->GetWindowRect( &x, &y, &Width, &Height);
```

See Also

“IsVisible” on page 472

“MoveWindow” on page 474

“Show” on page 479

Hide

Hides the **Error Correction Window**, if it is currently visible.

Syntax

In Visual Basic:

```
[VVECWin].Hide
```

In Visual C++ (MFC):

```
Void [CVVECWin].Hide();
```

In Visual C++:

```
HRESULT [pIVVECWin]->Hide();
```

Parameters

None.

Return Values

None.

Remarks

Hides the **Error Correction Window**. The window handle is still valid. This method will not cause the **Close** event to be fired, since that event is meant to notify the application of a user-initiated clos

Example

In Visual Basic:

```
'first process word selected event  
. . .  
'Now hide Error Correction window until needed again  
VVECWin.Hide
```

In Visual C++ (MFC):

```
//First process word selected event  
. . .  
//Now hide Error Correction window until needed again  
CVVECWin.Hide();
```

In Visual C++:

```
//First process word selected event  
. . .  
//Now hide Error Correction window until needed again  
m_pIVVECWin->Hide();
```

See Also

“IsVisible” on page 472
“Show” on page 479
“Close” on page 485
“FocusChange” on page 489

Init

Loads, but does not show, the ViaVoice **Error Correction Window** and performs other initialization related functions.

Syntax

In Visual Basic:

```
[VVEWin].Init(hWndParent As Long)
```

In Visual C++ (MFC):

```
void [CVVEWin].Init(long hWndParent);
```

In Visual C++:

```
HRESULT [pIVVEWin]->Init(long hWndParent);
```

Parameters

hWndParent

Long. Window handle to be used as parent.

Return Values

None.

Remarks

Init must be called before **Show**.

Example

In Visual Basic:

```
VVECWin.Init hWnd
```

In Visual C++ (MFC):

```
CVVECWin.Init( (long)m_hWnd );
```

In Visual C++:

```
m_pIVVECWin->Init( (long)m_hWnd );
```

See Also

“Show” on page 479

“NumVisibleAlternates” on page 454

Is Visible

Returns whether or not the ViaVoice **Error Correction Window** is currently visible.

Syntax

In Visual Basic:

```
[Boolean] = [VVECWin].IsVisible
```

In Visual C++ (MFC):

```
BOOL [CVVECWin].IsVisible();
```

In Visual C++:

```
HRESULT [pIVVECWin]->IsVisible(VARIANT_BOOL * pVal);
```

Parameters

None .

Return Values

TRUE

Error Correction Window is currently visible.

FALSE

Error Correction Window is currently not visible.

Remarks

None.

Example

In Visual Basic:

```
Dim bVisible As Boolean  
bVisible = VVEWin.IsVisible
```

In Visual C++ (MFC):

```
BOOL bVisible;  
bVisible = CVVEWin.IsVisible();
```

In Visual C++:

```
VARIANT_BOOL bVisible;  
m_pIVVEWin->IsVisible( &bVisible );
```

See Also

“Show” on page 479

“Hide” on page 468

“Close” on page 485

“FocusChange” on page 489

MoveWindow

Moves the **Error Correction Window** to the specified absolute location and gives it the specified dimensions.

Syntax

In Visual Basic:

```
[VVEWin].MoveWindow x As Long, y As Long, Width As Long, Height As Long, AutoLocate As Boolean
```

In Visual C++ (MFC):

```
void [CVVEWin].MoveWindow(long x, long y, long Width, long Height, BOOL AutoLocate);
```

In Visual C++:

```
HRESULT [pIVVEWin]->MoveWindow(long x, long y, long Width, long Height, VARIANT_BOOL AutoLocate);
```

Parameters

x
Long. Window's x coordinate.

y
Long. Window's y coordinate.

Width
Long. Window's width.

Height
Long. Window's height.

AutoLocate
Boolean.

Return Values

TRUE

Window is located adjacent to an exclusion rectangle specified by the user.

FALSE

Window is to be moved to the specified absolute location.

Remarks

Moves the **Error Correction Window** to the given location, specified in screen coordinates. The caller can alternately specify an "exclusion rectangle," and the window will be moved adjacent to that rectangle.

Example

In Visual Basic:

```
'Try to make the window bigger
Dim x As Long
Dim y As Long
Dim Width As Long
Dim Height As Long
VVEWin.GetWindowRect x, y, Width, Height
VVEWin.MoveWindow x, y, Width * 2, Height * 2, False
```

In Visual C++ (MFC):

```
//Try to make the window bigger
long x,y,Width,Height;
CVVEWin.GetWindowRect(&x, &y, &Width, &Height);
CVVEWin.MoveWindow(x, y, Width*2, Height*2, FALSE);
```

In Visual C++:

```
//Try to make the window bigger  
long x,y,Width,Height;  
m_pIVVEWin->GetWindowRect(&x, &y, &Width, &Height);  
m_pIVVEWin->MoveWindow(x, y, Width*2, Height*2, VARIANT_FALSE);
```

See Also

“GetWindowRect” on page 466

“Show” on page 479

Reset

Clears the list of alternates and associated sounds-like spellings from the **Error Correction Window's** alternates listbox, the value stored in the **CorrectText** property, and unchecks the "Add words or phrases" checkbox.

Syntax

In Visual Basic:

```
[VVECWin].Reset Redraw As Boolean = False
```

In Visual C++ (MFC):

```
void [CVVECWin].Reset(BOOL Redraw = FALSE);
```

In Visual C++:

```
HRESULT [pIVVECWin]->Reset(VARIANT_BOOL Redraw);
```

Parameters

Redraw

Boolean. Controls whether or not the control should be redrawn upon completion of the reset activities.

Return Values

??

Remarks

None.

Example

In Visual Basic:

```
VVECTWin.Reset True
```

In Visual C++ (MFC):

```
CVVECTWin.Reset(TRUE);
```

In Visual C++:

```
m_pIVVECTWin->Reset( VARIANT_TRUE );
```

See Also

“AddAlternate” on page 461

“CorrectText” on page 444

“AddPhraseChecked” on page 432

Show

Makes the **Error Correction Window** visible, and optionally sets input focus to the **Error Correction Window**.

Syntax

In Visual Basic:

```
[VVECWin].Show TakeFocus As Boolean = True
```

In Visual C++ (MFC):

```
void [CVVECWin].Show(BOOL TakeFocus = TRUE);
```

In Visual C++:

```
HRESULT [pIVVECWin]->Show(VARIANT_BOOL TakeFocus);
```

Parameters

TakeFocus

Boolean. Optionally sets input focus to the **Error Correction Window**.

Return Values

TRUE

Input focus is given to the **Error Correction Window's** edit field, with any text set as **CorrectText** selected. (Default)

FALSE

Feature is disabled.

Remarks

Activates the **Error Correction Window** and displays it in its current size and position. **Init** must have been called prior to calling **Show**. If **TakeFocus** is TRUE, then the input focus is given to the **Error Correction Window**'s edit field, with any text set as **CorrectText** selected.

Example

In Visual Basic:

```
VVECWin.Init hWnd  
VVECWin.Show True
```

In Visual C++ (MFC):

```
CVVECWin.Init( (long)m_hWnd );  
CVVECWin.Show( TRUE );
```

In Visual C++:

```
m_pIVVECWin->Init( (long)m_hWnd );  
m_pIVVECWin->Show( VARIANT_TRUE );
```

See Also

“Hide” on page 468

“IsVisible” on page 472

“FocusChange” on page 489

“MoveWindow” on page 474

Error Correction Window Control Events

The ViaVoice **Error Correction Window** control supports the following events:

- **ButtonSelected**
- **Close**
- **ContextHelpRequest**
- **FocusChange**
- **MenuSelected**
- **WordSelected**

ButtonSelected

Event fired by **Error Correction Window** control when the user selects a button on the Error Correction window.

Syntax

In Visual Basic:

```
ButtonSelected(ByVal lButtonID As Long)
```

In Visual C++ (MFC):

```
void OnButtonSelectedVVECWin(long lButtonId);
```

In Visual C++:

```
HRESULT ButtonSelected(VVECButtonId lButtonID);
```

Parameters

lButtonId

Long. Button ID of the button that was selected:

Value	Description
vvecUSER_BUTTON	User pressed Format button.
vvecPLAY_BUTTON	User pressed Play button.
vvecDELETE_BUTTON	User pressed Delete button.
vvecCORRECT_BUTTON	User pressed Correct button.
vvecADDPHRASE_BUTTON	User pressed Add Phrase checkbox.

Return Values

None.

Remarks

The value for the button ID is passed as the only parameter. The values for the button ID's can be found in vvecwin.h located in the include directory of the ViaVoice SDK directory.

Example

In Visual Basic:

```
Private Sub ButtonSelected(ByVal lButtonID As Long)
    Select Case lButtonID
        Case vvecUSER_BUTTON
            ProcessUserButton()
        . . .
    End Sub
```

In Visual C++ (MFC):

```
void CVCTestDlg::OnButtonSelectedVVECWin(long lButtonID)
{
    switch (lButtonID)
    {
        case vvecUSER_BUTTON :
            ProcessUSER_BUTTON();
            break;
        . . .
    }
}
```

In Visual C++:

```
STDMETHODIMP CTestDlgEvents::ButtonSelected(VVECBUTTONID lButtonID)
{
    switch (lButtonID)
    {
        case vvecUSER_BUTTON :
            ProcessUSER_BUTTON();
            break;
        . . .
    }
    return S_OK;
}
```

See Also

“AddMenuItem” on page 464

Close

Event fired by the **Error Correction Window** control when the **Error Correction Window** is hidden by a user action.

Syntax

In Visual Basic:

```
Close()
```

In Visual C++ (MFC):

```
void OnCloseVVECWin();
```

In Visual C++:

```
HRESULT Close();
```

Parameters

None.

Return Values

None.

Remarks

This event is only fired when the user initiates the close, either by clicking the close button on the menu or by the voice command.

Example

In Visual Basic:

```
Private Sub Close()  
    MsgBox "Closing VVECWin"  
End Sub
```

In Visual C++ (MFC):

```
void CVCTestDlg::OnCloseVVECWin()  
{  
    ::MessageBox(m_hWnd, "Closing", "VVECWin", MB_OK);  
}
```

In Visual C++:

```
STDMETHODIMP CtestDlgEvents::Close()  
{  
    ::MessageBox(m_hWnd, "Closing", "VVECWin", MB_OK);  
    return S_OK;  
}
```

See Also

“Hide” on page 468

“Show” on page 479

“FocusChange” on page 489

“IsVisible” on page 472

ContextHelpRequest

Event fired by the **Error Correction Window** control when the user requests context sensitive help.

Syntax

In Visual Basic:

```
ContextHelpRequest(ByVal lHelpID As Long)
```

In Visual C++ (MFC):

```
void OnContextHelpRequestVVECWin(long lHelpID);
```

In Visual C++:

```
HRESULT ContextHelpRequest(long lHelpID);
```

Parameters

lHelpID

Long. Help ID.

Return Values

None.

Remarks

None.

Example

In Visual Basic:

```
Private Sub ContextHelpRequest(ByVal lHelpID As Long)
    . . .
End Sub
```

In Visual C++ (MFC):

```
Void CVCTestDlg::OnContextHelpRequestVVECWin(long lHelpID)
{
    . . .
}
```

In Visual C++:

```
STDMETHODIMP CTestDlgEvents::ContextHelpRequest(long lHelpID)
{
    . . .
    return S_OK;
}
```

See Also

None.

FocusChange

Event fired by the **Error Correction Window** control when the Error Correction Window loses or gains focus.

Syntax

In Visual Basic:

```
FocusChange(ByVal fHasFocus As Boolean)
```

In Visual C++ (MFC):

```
void OnFocusChangeVVECWin(BOOL fHasFocus);
```

In Visual C++:

```
HRESULT FocusChange(VARIANT_BOOL fHasFocus);
```

Parameters

fHasFocus
Boolean.

Return Values

TRUE
The **Error Correction Window** is gaining focus.

FALSE
The window is not gaining focus.

Remarks

None.

Example

In Visual Basic:

```
Private Sub FocusChange(ByVal bHasFocus As Boolean)
    . . .
End Sub
```

In Visual C++ (MFC):

```
void CVCTestDlg::OnFocusChangeVVECWin(BOOL bHasFocus)
{
    . . .
}
```

In Visual C++:

```
STDMETHODIMP CTestDlgEvents::FocusChange(VARIANT_BOOL bHasFocus)
{
    . . .
    return S_OK;
}
```

See Also

“Show” on page 479

“Hide” on page 468

“IsVisible” on page 472

MenuSelected

The **Error Correction Window** control fires this event when the user selects a menu item on the Error Correction Window's custom menu.

Syntax

In Visual Basic:

```
MenuSelected(ByVal lMenuID As Long)
```

In Visual C++ (MFC):

```
void OnMenuSelectedVVECWin(long lMenuID);
```

In Visual C++:

```
HRESULT MenuSelected(long lMenuID);
```

Parameters

lMenuID

Long. ID of the menu item selected. This value was provided by the application when the menu item was added with the **AddMenuItem** method.

Return Values

None.

Remarks

None.

Example

In Visual Basic:

```
Private Sub MenuSelected(ByVal lMenuID As Long)
    . . .
End Sub
```

In Visual C++ (MFC):

```
void CVCTestDlg::OnMenuSelectedVVECWin(long lMenuID)
{
    . . .
}
```

In Visual C++:

```
STDMETHODIMP CTestDlgEvents::MenuSelected(long lMenuID)
{
    . . .
    return S_OK;
}
```

See Also

“AddMenuItem” on page 464

“ButtonSelected” on page 482

WordSelected

The **Error Correction Window** control fires this event when the user chooses a word to correct their current selection.

Syntax

In Visual Basic:

```
WordSelected(ByVal sWord As String, ByVal sSoundsLike As String, ByVal  
lIndex As Long)
```

In Visual C++ (MFC):

```
void OnWordSelectedVVECWin(LPCTSTR sWord, LPCTSTR sSoundsLikeWord, long  
lIndex);
```

In Visual C++:

```
HRESULT WordSelected(BSTR sWord, BSTR sSoundsLikeWord, long lIndex);
```

Parameters

sWord

String. Spelling of the selected word.

sSoundsLike

String. Spelling of the "Sounds Like" word, optionally added when application added the given spelling with the **AddAlternate** method.

lIndex

Long. Zero-based index of the alternate word selected.

Return Values

None.

Remarks

None.

Example

In Visual Basic:

```
Private Sub WordSelected(ByVal sWord As String, ByVal sSoundsLike As  
String, ByVal lIndex As Long)  
    . . .  
End Sub
```

In Visual C++ (MFC):

```
void CVCTestDlg::OnWordSelectedVVECWin(LPCTSTR sWord, LPCTSTR  
sSoundsLikeWord, long lIndex)  
{  
    . . .  
}
```

In Visual C++:

```
STDMETHODIMP CTestDlgEvents::WordSelected BSTR sWord, BSTR  
sSoundsLikeWord, long lIndex)  
{  
    . . .  
    return S_OK;  
}
```

See Also

“AddAlternate” on page 461

ECWin Control Frequently Asked Questions

This chapter contains answers to the most frequently asked questions about the ViaVoice **Error Correction Window** Control.

Does the ViaVoice Error Correction Window Control correct errors?

No. The control is to be used by an application that actually performs the error correction. The **Error Correction Window** control does not know enough about the application context to interact with the speech recognition engine, or how to modify text in an application's window. The control is instead meant to provide an easy programming interface for an application, that users can interact with in order to inform the application what the user wants. The control informs the application through events and makes available through properties and methods the settings chosen by the user.

When I invoke Correction, nothing shows up in the list box?

The application is responsible for populating the list box with alternate by calling the **AddAlternate** method once for each item. Additionally, the alternates will not be visible until the window is repainted. The application can force repainting once they have finished populating the listbox by passing a value of TRUE for the "Repaint" parameter to **AddAlternate**.

I press the Correction button in the Error Correction Window and it doesn't correct the text?

Pressing any of the buttons in the correction window causes a **ButtonSelected** event to be fired to the application. It is the responsibility of the application to react to these events as appropriate.

Why can't I dictate into the edit field in the Correction Window?

The **Error Correction Window** control does not allow dictation into the edit field. The most likely reason for the user invoking the error correction dialog is that the engine misrecognized a word or phrase, and dictating that same phrase to the correction control would most likely generate the same misrecognition.

What is the purpose of the "Add Words or Phrases" checkbox?

When this checkbox is checked, the application will typically respond by adding the text as a single phrase. If the user spoke "New York" and it was incorrectly recognized as "Newark", then the user would open the error correction window, type "New York" in the edit field, check the Add

words or phrases" checkbox, and click the "**Correct**" button. The application would then update the vocabulary with the complete phrase "New York".

What voice commands are supported by the Error Correction window?

Refer to the section entitled "Error Correction Control Voice Command support" in the chapter entitled "Getting Started with the **Error Correction Window** Control."

Where can I find the ID values for the Error Correction buttons?

Refer to the file VVECWIN.H that was installed into the Include directory of the ViaVoice SDK install directory.

Why isn't the Close event fired when I hide the error correction window?

This event is meant to notify you only when the user closes the window with a voice command or by selecting the **Close** button on the title bar. It is assumed that the application can call their **OnClose** method when they programmatically close the window with the "**Hide**" method.

Introduction to the User Interface Control

The ViaVoice **User Interface Client Control (UIClient)** is an ActiveX control that enables developers to manipulate the IBM ViaVoice **User Interface Server (UIServer)**. The **UIServer** provides a common interface for speech-enabled applications using the IBM ViaVoice SDK. It is capable of presenting speech-related information in a number of ways, including a Taskbar View, a Docked View, a Minimal View and an Agent View. Figure 39 shows you how the **UIServer** appears in Taskbar View.



Figure 39. ViaVoice User Interface Server - Taskbar View

It is important to realize that the **UIServer** is used only to provide the user with visual information; it does not directly control any functions in the speech recognition engine or in the text-to-speech engine. For example, if you look at Figure 39, you will notice a button with a microphone icon. This button reports the state of the speech input device (on, off, or asleep). However, the **UIServer** does not automatically set the state of the device to on, off, or asleep when the user clicks the microphone button, nor does it automatically read the state of the device directly from the engine in order to display the appropriate icon. It merely provides a mechanism for you, the developer, to provide a visual interface that is common among speech-enabled applications. It will notify you (through one of the events in the **UIClient** control in Visual Basic or through a callback interface in Visual C++) when the user clicks one of its buttons or selects one of its menu options. It is up to you to implement the user's request, as well as to set the state of the various elements in the **UIServer**.

Something else to keep in mind is that there may be a number of speech-enabled applications running simultaneously, all sharing the same **UIServer**. Therefore, the **UIClient** control will never allow you to take complete control over the **UIServer**. This means, for example, that the **UIClient** control will not allow you to change the view of the **UIServer**, as this is set by the user.

The main benefit of using the **UIClient** control is that you can quickly incorporate the functionality of the **UIServer** into your application, and make your application look and feel like other speech-enabled applications in the market. This guide shows you how to display and manipulate the **UIServer** using the **UIClient** control. It gives you a tutorial on how to use the **UIClient** control in your projects, and then it will give you a complete reference of all the properties, methods, and events in the **UIClient** control.

To completely speech-enable your application, you will also need to incorporate other controls in this SDK. The ViaVoice **User Interface Client** control only provides you with a way to interact with your users.

Getting Started with the User Interface Control

The following is a tutorial on how to incorporate the **UIClient** control into your Visual Basic or Visual C++ applications. This tutorial is designed to present you with the most commonly used methods in the **UIClient** control, and the order in which you must execute them.

The following sections contain instructions to help you begin working with the **User Interface Control**.

Creating an Instance of the Control

This section contains examples for using the **UIClient** control with Visual Basic and Visual C++.

In Visual Basic:

To add the **UIClient** control to your application, you can:

1. Choose **Components** from the **Project** menu.
The Components dialog box, Figure 20, appears.

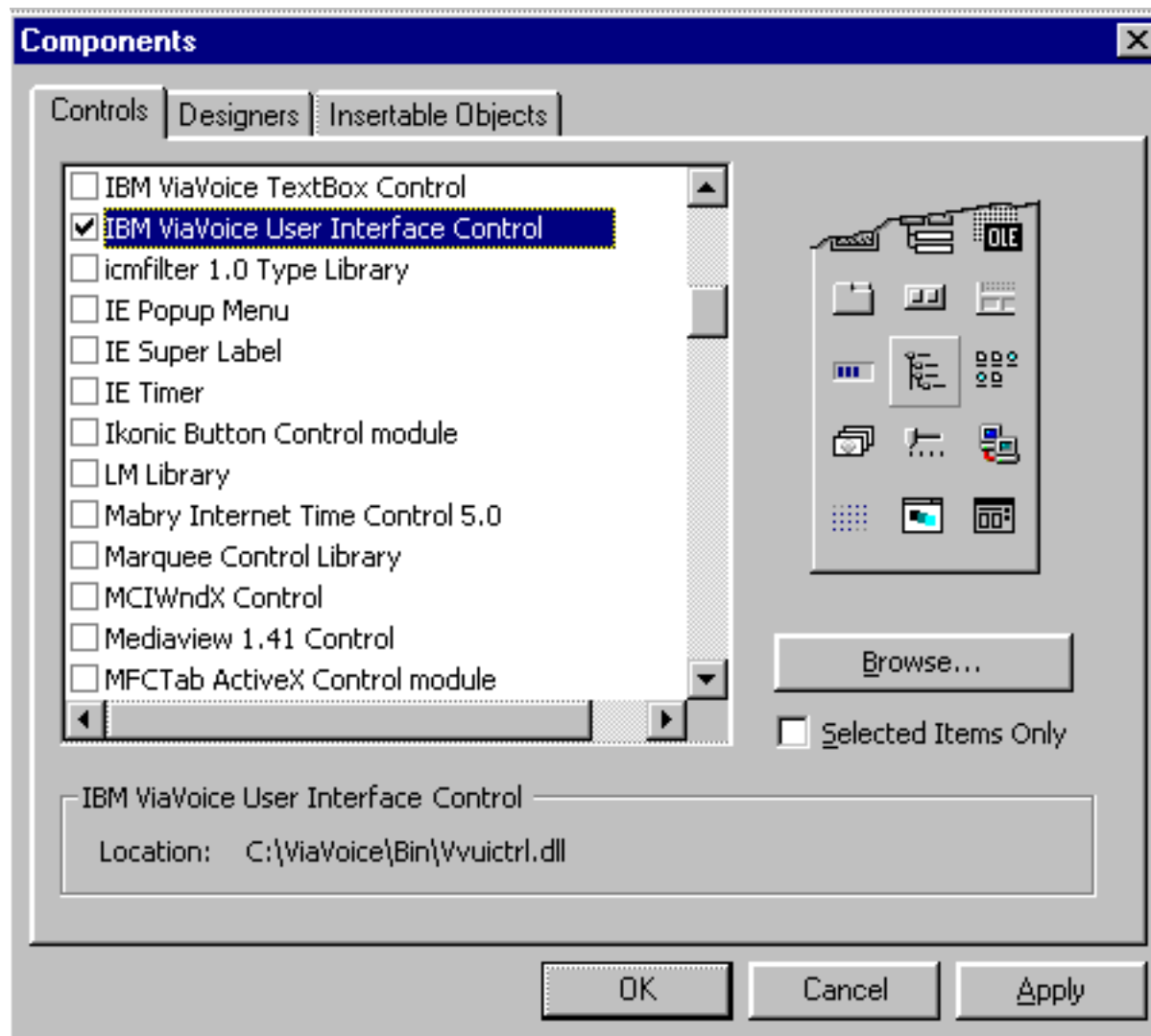


Figure 40. Component Selection Dialog Box - Visual Basic

The Components dialog box lists all the ActiveX Controls that you can use in your application.

2. Select **IBM ViaVoice User Interface Control** from the list and click **OK**.

Visual Basic adds the control to your project and adds a new icon to the toolbar (Figure 41).



Figure 41. UIClient Control Toolbar Icon

3. Add an instance of the **UIClient** control to your form.
The **UIClient** control is designed to enable you to display and interact with the **UIServer** (Figure 40). However, the **UIClient** control itself is invisible at run time, and does not add any visual enhancements to the Visual Basic form that contains it.

In Visual C++ (MFC):

There are various ways of using the **UIClient** in C++ programs. One way is to use it as an ActiveX control in a dialog box in an MFC application.

To use the **UIClient** as an ActiveX control in an MFC application, do the following:

1. From the Project menu, select **Add to Project**, and then select **Components and Controls** from the cascading menu.

The Components and Controls Gallery dialog, Figure 22, appears.

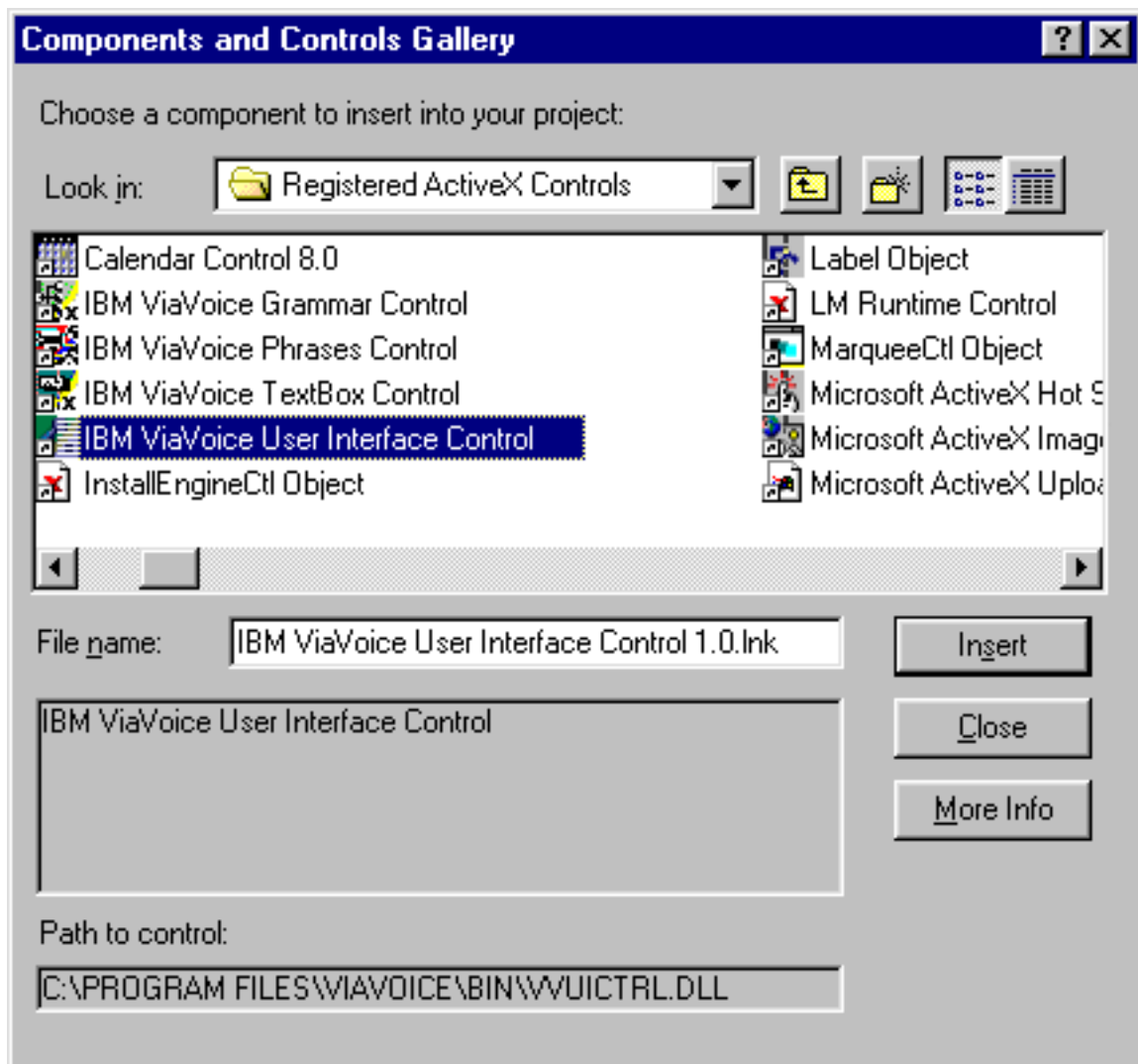


Figure 42. Insert ActiveX Control Dialog Box - Visual C++

2. Double-click the Registered ActiveX Controls folder in the dialog box.
3. In the list of controls, select the **IBM ViaVoice User Interface Control** icon; then choose **Insert**.
A prompt appears asking if you want to insert this component.

4. If you wish to insert the component, choose **OK**.

The Confirm Classes dialog box, Figure 43, appears listing the dual interface of the **UIClient** control (CVVUIClientDual).

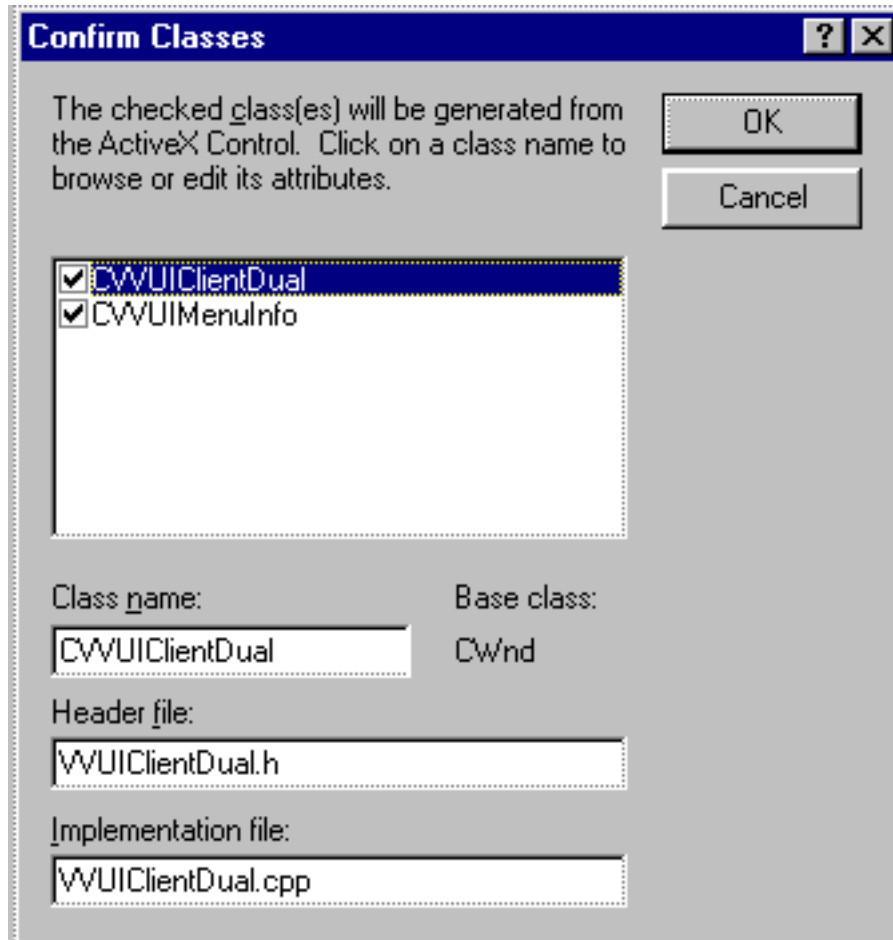


Figure 43. Confirm Classes Dialog

5. To confirm your selection, choose **OK**.
6. Close the Components and Controls Gallery dialog box.

If you examine the Project Workspace window in the class view you will notice two new classes: CVVUIClientDual and CVVUIMenuInfo (assuming you accepted the default names for the class in the Confirm Classes Dialog).

7. In the resource view of your Project Workspace window double-click the dialog box resource entry where you wish to insert the **UIClient** control.

The **UIClient** icon, Figure 44, appears in the Controls toolbar.



Figure 44. UIClient Icon in the Controls Toolbar

8. Add an instance of the **UIClient** control to the dialog box.

After you add the **UIClient** control to your dialog box you can invoke Class Wizard to create a member variable for your class of type CVVUIClientDual.

You might also decide to capture the events in the control by adding Event handlers to your dialog box class. This is done by clicking on the **UIClient** control with the right mouse button and selecting **Event** from the popup menu.

The ViaVoice SDK ships with two Include files for Visual C++ users that provide constant declarations as well as structure definitions. These files are:

- VVUITYPE.H
- VVUICNST.H

These two files are included automatically when you include the file VVUICTRL.H (#include "vvuictrl.h").

In Visual C++ (Custom Interface):

Whether you are using MFC or not in your Visual C++ project you can use the functionality in the **UIClient** control without using the **VVUIClient** dual interface (i.e. without having to insert an instance of the control in your dialogs). The **UIClient** control provides two COM custom interfaces: **IVVUIClient** and **IVVUIEventSink**. To use the custom interfaces make sure to include the files **VVUICTRL.H** and **VVUICTRL.C** into your project. These files contain definitions for the interface, IIDs and CLSIDs respectively.

To create an instance of the **IVVUIClient** interface, do the following:

1. Declare a variable of type **IVVUIClient*** as follows:
`IVVUIClient* pIVVUIClient;`
2. Make sure to initialize the COM libraries with the **CoInitialize** function.

Note:

Call the **CoUninitialize** function after releasing all COM interfaces and before exiting your application.

3. Use the **CoCreateInstance** function, as shown below:

```
CoCreateInstance(CLSID_VVUIClient,
    NULL,
    CLSCTX_ALL,
    IID_IVVUIClient,
    (void **)& pIVVUIClient);
```

After this call, `pIVVUIClient` will point to an instance of the **IVVUIClient** interface.

Initializing the UIClient

The first step in using the control is to initialize it. The **Initialize** step causes the **UIServer** to become visible. To initialize the control simply call the control's **Initialize** method as shown in the code segment below:

In Visual Basic:

```
Sub InitializeUIServer()  
    On Error Resume Next  
    Dim lRetVal As UIRC  
  
    lRetVal = VVUIClient1.Initialize  
  
    If Err.Number = 0 Then  
  
        If lRetVal <> UIAPIRC_OK Then  
            MsgBox "Initialization Failed"  
            Exit Sub  
        End If  
  
    Else  
        'Insert error code here!  
    End If  
  
End Sub
```

In Visual C++ (MFC):

```
void InitializeUIServer()
{
    UIRC rc;

    try
    {
        rc = (UIRC) m_vvuiMain.Initialize();

        if (rc != UIAPIRC_OK)
        {
            MessageBox("Initialization Failed",
                "VVUI Initialize",
                MB_OK);
            return;
        }
    }

    catch (...)
    {
        //Insert exception fault code here
    }
}
```

In Visual C++ (Custom Interface):

```
void InitializeUIServer()
{
    UIRC rc;
    HRESULT hr;

    hr = m_pIVVUIClient->Initialize(&rc);

    if (SUCCEEDED(hr))
    {
        if (rc != UIAPIRC_OK)
        {
            MessageBox(NULL,
                "Initialization Failed",
                "VVUI Initialize",
                MB_OK);
            return;
        }
    }
}
```

The **UIClient** control (Custom Interface) does not have any custom properties, and all of its functionality is accessed through methods. All of the methods in the control are functions that report the success or failure of the method to execute. When you issue one of the methods in the control the method returns an enumerated value. For a complete list of return code, refer to “UIRC (Enum)” on page 559.

When you issue the **Initialize** method the **UIServer** displays the last view the user selected. The first time the **UIServer** is initialized in the user’s machine, the **UIServer** displays its taskbar view by default (See Figure 39 on page 497). If another speech-enabled application has already called the **Initialize** method prior to your program issuing the method, your **Initialize** will have no visible effects. Internally, the **UIServer** keeps a counter of all the applications that issued the **Initialize** method. When all the applications that issued the **Initialize** method are closed, the **UIServer** automatically disappears.

If you issue the **Initialize** method twice for the same control you will receive a trappable error in Visual Basic: &H8000FFFF (ViaVoice **UIClient** is already Initialized). MFC throws an exception

through the MFC wrappers and the custom interface returns an HRESULT of E_UNEXPECTED. You must call the **Initialize** method for each instance of the control.

After the **UIServer** is visible you need to inform the server which language to use when displaying menu options. To do this you must issue the **SetLanguageByID** method or the **SetLanguageByString** method. The **SetLanguageByID** method accepts as a parameter, the language identifier (**LangID**), of the language you wish the **UIServer** to use. The **LangID** is a unique number that Windows uses to specify a foreign language and it is derived from a primary language ID and sublanguage ID. Use the **SetLanguageByString** method instead of a string representation of the number. Table 3 contains a list of possible **LangID** values and their string representations. In the **User Interface** Control (dual interface) there is also a **LanguageUI** property that can be set/queried at design time or run time.

Table 3. LangID string names supported by UIServer and their corresponding values

"EN_US"	1033 (Hex 409)	MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US)
"EN_UK"	2057 (Hex 809)	MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_UK)
"FR_FR"	1036 (Hex 40C)	MAKELANGID(LANG_FRENCH, SUBLANG_FRENCH)
"GR_GR"	1031 (Hex 407)	MAKELANGID(LANG_GERMAN, SUBLANG_GERMAN)
"IT_IT"	1040 (Hex 410)	MAKELANGID(LANG_ITALIAN, SUBLANG_ITALIAN)
"ES_ES"	1034 (Hex 40A)	MAKELANGID(LANG_SPANISH, SUBLANG_SPANISH)
"JA_JP"	1041 (Hex 411)	MAKELANGID(LANG_JAPANESE, SUBLANG_DEFAULT)

When using the **SetLanguageByString** method you can also use the last two characters of the LangID string. For example, you can use **SetLanguageByString**("US").

The following code segment shows you how to use the **SetLanguageByString** method:

In Visual Basic:

```
Sub InitializeUIServer()  
    On Error Resume Next  
    Dim lRetVal As UIRC  
  
    '*****  
    'Previous code goes here  
    '*****  
  
    lRetVal = VVUIClient1.SetLanguageByString("EN_US")  
    If Err.Number = 0 Then  
        If lRetVal <> UIAPIRC_OK Then  
            MsgBox "Language not found"  
            Exit Sub  
        End If  
    Else  
        'Insert error code here!  
    End If  
End Sub
```

In Visual C++ (MFC):

```
void InitializeUIServer()
{
    UIRC rc;
    try
    {

        //*****
        //Previous code goes here
        //*****

        rc = (UIRC)m_vvuiMain.SetLanguageByString("EN_US");
        if (rc != UIAPIRC_OK)
        {
            MessageBox("Language not found",
                "VVUI SetLanguageByString",
                MB_OK);
            return;
        }
    }
    catch (...)
    {
        //Insert exception fault code here
    }
}
```

In Visual C++ (Custom Interface):

```
void InitializeUIServer()
{
    UIRC rc;
    HRESULT hr;

    //*****
    //Previous code goes here
    //*****

    hr = pIVVUIClient->SetLanguageByString("EN_US", &rc);

    if (SUCCEEDED(hr))
    {
        if (rc != UIAPIRC_OK)
        {
            MessageBox(NULL,
                "Language not found",
                "VVUI SetLanguageByString",
                MB_OK);
            return;
        }
    }
}
```

Entering an invalid Language name results in UIRC code UIAPIRC_ERROR_INVALIDPARAM.

Programming the ViaVoice User Interface

The **UIClient** control is able to communicate actions that occur in the **UIServer** to your program. However, you must turn on this communication manually. You can use the **SetClientCallbackFlags** method in Visual Basic or in Visual C++ (MFC) as follows:

In Visual Basic:

```
Sub InitializeUIServer()
    On Error Resume Next
    Dim lRetVal As UIRC
    '*****
    'Previous code goes here
    '*****

    lRetVal = VVUIClient1.SetClientCallbackFlags(vvUIEVENT_ALL)
    If Err.Number = 0 Then

        If lRetVal <> UIAPIRC_OK Then
            MsgBox "Unable to turn on messaging"
            Exit Sub
        End If

    Else
        'Insert error code here!
    End If

End Sub
```

In Visual C++ (MFC):

```
void InitializeUIServer()
{
    UIRC rc;
    try
    {

        //*****
        //Previous code goes here
        //*****

        rc = (UIRC)m_vvuiMain.SetClientCallbackFlags(UIEVENT_ALL);
        if (rc != UIAPIRC_OK)
        {
            MessageBox("Unable to turn on messaging",
                "VVUI SetClientCallbackFlags",
                MB_OK);
            return;
        }
    }
    catch (...)
    {
        //Insert exception fault code here
    }
}
```

The parameter in the **SetClientCallbackFlags** method informs the **UIServer** which messages you wish to capture. To find all possible values for this parameter, refer to “SetClientCallbackFlags” on page 597.

For example, you can issue the **SetClientCallbackFlags** method with **UIEVENT_ALL** to receive all messages, or with **UIEVENT_VIEW_QUERYFLAGS** to receive the query view flags event (the **EventQueryViewFlags**) that the control will fire when your application has the focus. If the user changes the view of the **UIServer** from Taskbar to Docked (selecting **Docked** from the **Appearance** menu) you must write code in this event to allow the **UIServer** to dock to your dialog box. Here is an example:

In Visual Basic:

```
Private Sub VVUIClient1_EventQueryViewFlags (phwndWindow As Long,
pdwDockFlags As Long, pResult As VVUICtrlCtl.UIEVENTRC)
    If phwndWindow = VVUIClient1.Parent.hwnd Then
        pdwDockFlags = vvDVAF_ALLOW_TOPMOST_DOCK
        'Inform the control that the message was processed.
        pResult = UIEVENTRC_PROCESSED
    End If
End Sub
```

In Visual C++ (MFC):

```
void CTestDlg::OnEventQueryViewFlags(long FAR* phwndWindow, long FAR*
pdwDockFlags, long FAR* pResult)
{
    if ( (HWND)*phwndWindow == m_hWnd )
    {
        *pdwDockFlags = (long)(DVAF_ALLOW_TOPMOST_DOCK);
        //Inform the control that the message was processed.
        *pResult = (long)(UIEVENTRC_PROCESSED);
    }
}
```

If you are using the **IVVUIClient** custom interface, the procedure is slightly different. You must implement the **IVVUIEventSink** event sink interface by creating a class inherited from the **IVVUIEventSink** class; you can find example code to accomplish this in the \SAMPLES directory. The following code shows how to capture messages from **UIServer** by assuming that your class name is **CVVUIClientEvents**:

In Visual C++ (Custom Interface):

```
//Make sure to declare this variable prior to calling
//your procedure
CVVUIClientEvents  m_IVVUIClientEventSink;

void InitializeUIServer()
{
    UIRC rc;
    HRESULT hr;
    //*****
    //Previous code goes here
    //*****

    hr = m_pIVVUIClient->SetClientCallback(IID_IVVUIEventSink,
        &m_IVVUIClientEventSink,
        UIEVENT_ALL,
        &rc);
    if (SUCCEEDED(hr))
    {
        if (rc != UIAPIRC_OK)
        {
            MessageBox(NULL,
                "Unable to Set Callback methods",
                "VVUI SetClientCallback",
                MB_OK);
            return;
        }
    }
}
```

Before you can receive messages, however, there is one more step you need to perform. The ViaVoice SDK enables you to add user interface capabilities not only to your application but also to virtually any application in the system, because the **UIServer** does not really belong to any one program. It monitors user interaction and can broadcast messages to your application even if the interaction occurred while the user was working with another application. For this reason you need to tell the **UIClient** control which programs you wish to monitor events for by using either the **AddApplicationByWindow** or the **AddApplicationByName** method. If you are adding the ViaVoice User Interface support for your application you can pass the window handle of your dialog box or the

name of your executable. You can also pass the window handle or executable name of another application.

The following example shows you how to use the **AddApplicationByWindow** and the **AddApplicationByName** methods:

In Visual Basic:

```
Sub InitializeUIServer()  
    On Error Resume Next  
    Dim lRetVal As UIRC  
    /*****  
    'Previous code goes here  
    *****/  
    lRetVal = VVUIClient1.AddApplicationByWindow(VVUIClient1.Parent.hWnd)  
    If Err.Number = 0 Then  
        If lRetVal <> UIAPIRC_OK Then  
            MsgBox "Unable to add application"  
            Exit Sub  
        End If  
    Else  
        'Insert error code here!  
    End If  
End Sub
```

In Visual C++ (MFC):

```
void InitializeUIServer()
{
    UIRC rc;
    try
    {
        //*****
        //Previous code goes here
        //*****

        rc = (UIRC)m_vvuiMain.AddApplicationByWindow((long)m_hWnd);
        if (rc != UIAPIRC_OK)
        {
            MessageBox("Unable to add application",
                "VVUI AddApplicationByWindow",
                MB_OK);
            return;
        }
    }
    catch (...)
    {
        //Insert exception fault code here
    }
}
```

In Visual C++ (Custom Interface):

```
//Make sure to declare this variable prior to calling
//your procedure
CVVUIClientEvents m_IVVUIClientEventSink;
void InitializeUIServer()
{
    UIRC rc;
    HRESULT hr;

    //*****
    //Previous code goes here
    //*****

    hr = m_pIVVUIClient->AddApplicationByWindow((HWND_t)hWndApp,&rc);
    if (SUCCEEDED(hr))
    {
        if (rc != UIAPIRC_OK)
        {
            MessageBox(NULL,
                "Unable to add application",
                "VVUI AddApplicationByWindow",
                MB_OK);
            return;
        }
    }
}
```

The next section contains other examples of handling the different events in the **UIClient** control.

Getting and Setting User Interface Characteristics

The **UIServer** contains five components that you can use to inform the user of the current speech state of your application. The components are:

Microphone

Gets/Sets the state of the microphone component. When the **UIServer** is in Taskbar or Docked view, this component corresponds to the appearance of the microphone button.

Volume

Gets/Sets the level of the volume level displayed on the **UIServer**. The level can be a number between 0 and 100 percent.

User Information

Gets/Sets the user name displayed on the **UIServer**, as well as the user's full name, enrollment profile, and current vocabulary file.

Word History

Gets/Sets the word history text (next to the Volume meter when the **UIServer** is in Taskbar or Docked view.) This component usually provides the text representation of the last voice command issued. It can also be used to display the status of the recognition engine.

Custom

Gets/Sets the state of buttons in the **UIServer**. When in Taskbar or Docked view, these buttons are shortcuts to menu options.

You can set and get the characteristics of these components by using the **SetNumberValue**, **SetStringValue**, **GetNumberValue**, **GetStringValue** functions. The reference section will discuss these functions along with each component in detail. For now let's take a look at the following lines of code:

In Visual Basic:

```
Dim NewValue As Long
Dim rc As UIRC

'Set the state of the microphone component to "ON" state
rc = m_VVUIClient.SetNumberValue(COMPID_MICROPHONE, vvUIMSF_ON)

'Get the state of the microphone component
rc = m_VVUIClient.GetNumberValue(COMPID_MICROPHONE, NewValue)
```

In Visual C++ (MFC):

```
long NewValue;
UIRC rc;

//Set the state of the microphone component
rc = (UIRC)m_VVUIClient.SetNumberValue(COMPID_MICROPHONE,
    UIMSF_ON,
    UIMICINDEX_MICSTATE);

//Get the state of the microphone component
rc = (UIRC)m_VVUIClient.GetNumberValue(COMPID_MICROPHONE,
    &NewValue,
    UIMICINDEX_MICSTATE);
```

In Visual C++ (Custom Interface):

```
UIRC rc;  
HRESULT hr;  
DWORD NewValue;  
  
//Set the state of the microphone component  
hr = pIVVUIClient->SetNumberValue(COMPID_MICROPHONE,  
    UIMSF_ON,  
    UIMICINDEX_MICSTATE,  
    &rc);  
  
//Get the state of the microphone component  
hr = pIVVUIClient->GetNumberValue(COMPID_MICROPHONE,  
    &NewValue,  
    UIMICINDEX_MICSTATE,  
    &rc);
```

The code segment above illustrates how to set and get the state of the microphone component in the **UIServer**. The **SetNumberValue** method accepts a component ID as its first parameter, and a long value specifying the value to which you wish to modify the component. As you will see in the reference section, the **SetNumberValue** function (as well as its counterparts: **SetStringValue**, **GetNumberValue**, and **GetStringValue**) has a third parameter (optional in Visual Basic). This third parameter is used in some of the components to specify a specific value within the component.

To get the state of the microphone, use the **GetNumberValue** method. The **GetNumberValue** method has the same parameter list as the **SetNumberValue** method, except that in the **GetNumberValue** the second parameter is used to receive the property value from the object rather than to set the property value as it occurs in the **SetNumberValue** method.

Since there might be multiple speech-enabled applications, which change the characteristics of the various components individually, the **UIClient** control will fire events whenever a change to the properties occurs. For example, in the above code segment, whenever the state of the microphone changes, the **UIClient** control fires the **EventComponentUpdated** method. The first parameter in this event, **ciComponentID**, specifies which component was changed. You will learn more about each component in Chapter 28 “Properties, Methods, and Events” of this book.

Creating Custom Menus

When using the ViaVoice **UIServer** it may be useful at times to customize the menu bar to provide extra functionality or help for your users. The **VVUIClient** control enables you to customize the ViaVoice **UIServer**'s menu in several ways:

- You can add an application menu group – a set of menu options that appears when your application becomes active. These options appear between the Tools menu option and the Help menu option.
- You can add custom menu items to the Help menu.
- Or you can enable or disable various menu options in the main menu.

The **UIClient** control has several functions for adding, modifying, or removing custom menus:

- **AppendMenuItem**
- **InsertMenuItem**
- **GetMenuItemInfo**
- **DeleteMenuItem**
- **SetMenuItemInfo**

These functions enable you to create menu options and assign them a group name. After you add a menu option, the option will not automatically appear in the ViaVoice **UIServer** menu; it is simply saved to the registry. When the user clicks the ViaVoice menu button, the **UIClient** control fires the event **EventQueryViewMenuInfo** just before the menu appears. This event enables the current client to specify which menu group you wish to display to your users. The control then reads the menu options from the registry and displays the options that belong to the group you specified.

The previous paragraph describes how to add dynamic menus to the **UIServer**'s menu. Dynamic menus are the menus that change depending on the application that currently has the focus. However, it is possible to add static menu options to the Help menu of the **UIServer**. These options will be available regardless of the application that has the focus. For these options you must specify ahead of time which help file and topic to display when the user selects the menu item. The **UIServer** will then automatically invoke WinHelp with the appropriate file and topic.

Note:

Use static menu items only in rare occasions, and only if absolutely necessary. The application is responsible for removing status menu items before being uninstalled.

The following example code shows you how to add an application menu group to the ViaVoice main menu.

In Visual Basic:

```
Dim lRetVal As UIRC
Dim oNewMenu As New VVUIMenuInfo

oNewMenu.Type = UIMFT_CLIENT
oNewMenu.ID = 100
oNewMenu.Enabled = True
oNewMenu.Visible = True
oNewMenu.Checked = False
oNewMenu.Caption = "Menu Item #1"

lRetVal = m_VVUIClient.AppendMenuItem(UIMFG_DYNAMIC_APPLICATION, _
    App.EXENAME, _
    oNewMenu)
```

In Visual C++ (MFC):

If you have already added the **VVUIClient** control to your project and corresponding class wrappers for the interfaces, you should already have a class wrapper for the **IVVUIMenuInfo** interface named **CVVUIMenuInfo** (assuming you accepted the default values).

Prior to using the class wrapper, make sure to include VVUICTRL.H in your dialog box source file (#include "vvuictrl.h").

You can now create menu options as follows:

```

VVUIMenuInfo NewMenuItem;
NewMenuItem.CreateDispatch(CLSID_VVUIMenuInfo);
NewMenuItem.SetType(UIMFT_CLIENT);
NewMenuItem.SetID(100);
NewMenuItem.SetCaption("Menu Item #1");
NewMenuItem.SetEnabled(TRUE);
NewMenuItem.SetVisible(TRUE);
m_VVUIClient1.AppendMenuItem(UIMFG_DYNAMIC_APPLICATION,
    AfxGetAppName(),
    NewMenuItem);

```

The basic procedure implemented in the previous example creates a new instance of the **VVUIMenuInfo** class. The purpose of the class is for you to set the characteristics of the menu option you wish to add. Some of the settings in the **VVUIMenuInfo** class will be explained in more detail in Chapter 27 “Classes, Structures, and Enumerations”. The most important members in the class are Id, Caption, Enabled, Visible, and Checked.

You can add a separator bar menu option in Visual Basic and Visual C++ (MFC) by setting the Caption member of the class to a dash (“-”).

After the **VVUIMenuInfo** class is populated, you must issue the **AppendMenuItem** method to save the menu option to the registry. If you examine the line of code that follows you will notice that the first parameter is used to specify where the menu option will appear.

```

"lRetVal = m_VVUIClient.AppendMenuItem
(UIMFG_DYNAMIC_APPLICATION, App.ExeName, oNewMenu) "

```

In this case the menu will be added as an application menu. The second parameter is used to specify the name of the group to which the menu option belongs. The third parameter is simply the instance of the **VVUIMenuInfo** class that you created.

In Visual C++ (Custom Interface):

```
HRESULT hr;
UIRC rc;

UIMENUITEMINFO  uimiiItem;

ZeroMemory(&uimiiItem, sizeof(uimiiItem));

uimiiItem.m_uType  = UIMFT_CLIENT;
uimiiItem.m_dwID   = m_uMenuID;
lstrcpy(uimiiItem.m_szText, "Menu Item #1");
uimiiItem.m_uState=MFS_ENABLED;

hr = m_pIVVUIClient->AppendMenuItem(UIMFG_DYNAMIC_APPLICATION, _
    "TestMenu",
    &uimiiItem,
    &rc);
```

After adding the menu option, when the user clicks the ViaVoice **UIServer** menu button or clicks on the **UIServer** tool tray icon, the **UIClient** control will fire the **EventQueryViewMenuInfo** event. The following code shows you how to handle this event.

In Visual Basic:

```
Private Sub VVUIClient1_EventQueryViewMenuInfo( _
    vtViewType As TVIEWTYPE, _
    hwndWindow As Long, _
    ApplicationTitle As String, _
    MainMenuName As String, _
    ApplicationMenuName As String, _
    HelpMenuName As String, _
    pResult As UIEVENTRC)
    ApplicationTitle = App.Title
    ApplicationMenuName = App.EXENAME
    pResult = UIEVENTRC_PROCESSED
End Sub
```

In Visual C++ (MFC):

```
void OnEventQueryViewMenuInfo(
    long vtViewType,
    long hwndWindow,
    BSTR FAR* ApplicationTitle,
    BSTR FAR* MainMenuName,
    BSTR FAR* ApplicationMenuName,
    BSTR FAR* HelpMenuName,
    long FAR* pResult)
{
    CString sAppName = AfxGetAppName();
    *ApplicationTitle = sAppName.AllocSysString();
    *MainMenuName = sAppName.AllocSysString();
    *ApplicationMenuName = sAppName.AllocSysString();
    *HelpMenuName = sAppName.AllocSysString();
    *pResult = UIEVENTRC_PROCESSED;
}
```

In Visual C++ (Custom Interface):

```
STDMETHODIMP CUVUIClientEvents::EventQueryViewMenuInfo (
    TVIEWTYPE vtViewType,
    HWND_t hwndWindow,
    LPSTR lpszApplicationTitle,
    LPSTR lpszMainMenuName,
    LPSTR lpszApplicationMenuName,
    LPSTR lpszHelpMenuName,
    UIEVENTRC * pResult)
(
    char szAppName[256];
    lstrcpy(szAppName, "MyAppName");
    lstrcpy(lpszApplicationTitle, "MyApplication");
    lstrcpy(lpszMainMenuName, szAppName);
    lstrcpy(lpszApplicationMenuName, szAppName);
    lstrcpy(lpszHelpMenuName, szAppName);
    *pResult = UIEVENTRC_PROCESSED;
    return S_OK;
)
```

The parameters of this event will be explained in detail in Chapter 28 “Properties, Methods, and Events”. For now it is sufficient to understand the purpose of two parameters: **ApplicationTitle** and **ApplicationMenuName**. Remember that the intent of this event is to inform you that the user wishes to see the main menu, and to request from you the name of the menu group you wish to display. The parameter **ApplicationTitle** lets you specify a display name for the custom application menu group (Figure 45).

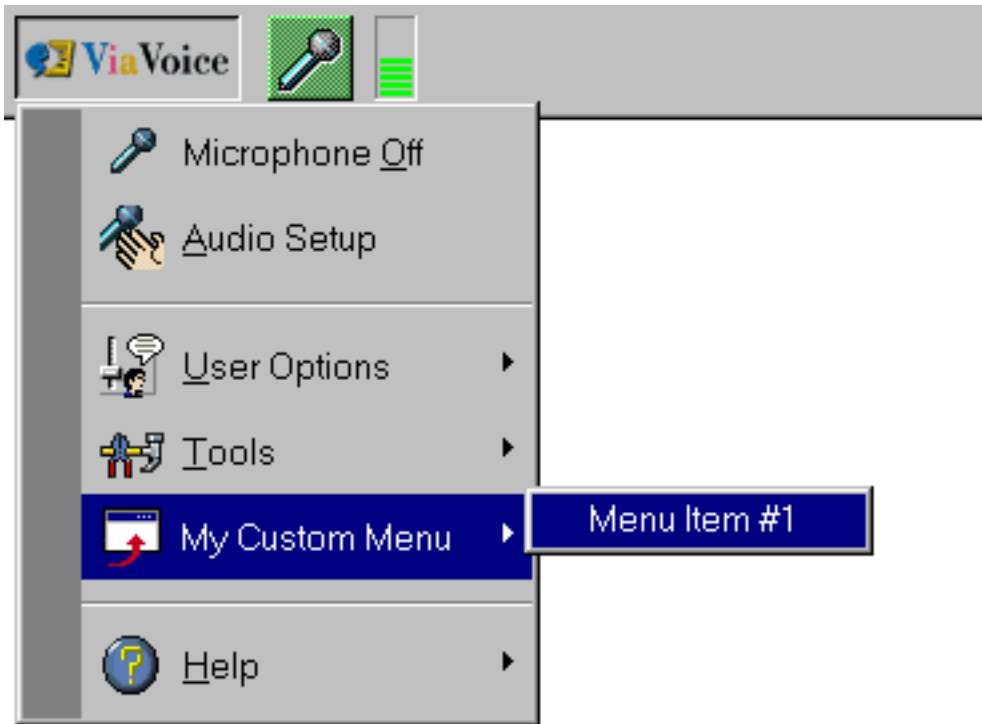


Figure 45. Application Menu Application Title

The parameter **ApplicationMenuName** enables you to specify the name of the group of menu options you wish to display. In the previous code example the **AppendMenuItem** method used the application's EXE name as the group name. The **ApplicationMenuName** parameter needs to be set to the same name in order to display the menu group previously created.

So far you have learned how to add custom menu options and how to ensure that the **UIServer** displays them when your application has the focus, but how do you know when the user actually selects the custom menu option? When the user clicks on one of the custom menu options the **UIClient** control fires the **EventMenuItemSelected** event. This event has the following syntax:

```
Private Sub m_VVUIClient_EventMenuItemSelected(dwMenuItemId As Long,
hwndTarget As Long, pResult As UIEVENTRC)
```

The first parameter will provide you with the ID number of the menu item the user selected. You can use the same ID number for various menu items if you wish to handle the menu items in the same way. For example, if you wish to add two menu options and both of them will bring up the same page in the help file, you can assign the same ID number to both. Then, in the **EventMenuItemSelected** event you would just implement one set of code for the particular ID number. However, if you decide to use the same ID number for multiple menu options, you must realize that the **GetMenuInfo** method will always return information for the first item with the specified ID.

Summary

At this point you should know how to do the following:

- How to display the ViaVoice **UIServer**, and set its menu language.
- How to specify which messages you wish to capture from the **UIServer**.
- How to change the appearance of the buttons in the **UIServer**, and capture user interaction with these buttons.
- How to create and monitor custom menu options.

The following chapters contain reference sections for all the properties, methods, and events for the **User Interface** control.

Classes, Structures, and Enumerations

This chapter gives you detailed information about the ActiveX classes, structure, constants, and enumerations included with the **UIClient** control. For more information about the properties, methods, and events of the **UIClient** control, refer to “Properties, Methods, and Events” on page 561.

The ViaVoice **UIClient** control supports the following classes, structures, constants and enumerations:

- **VVUIMenuInfo** (Class)
- **UIMENUITEMINFO** (Structure)
- **vvUIDockingAlgorithmConstants**
- **vvUIDockingStyleConstants**
- **vvUIEventCallbackFlags**
- **vvUIExtendedMenuFlags**
- **vvUIMaxConstants**
- **vvUIMenuItemConstants**
- **Component Index Constants**
- **vvUIRemoveClientConstants**
- **MICROPHONE_STATES** (Enum)
- **TCID** (Enum)
- **TVIEWTYPE** (Enum)
- **UIEVENTRC** (Enum)
- **UIMENUGROUP** (Enum)
- **UIMENUTYPE** (Enum)
- **UIRC** (Enum)

User Interface Control Classes

The ViaVoice **UIClient** control supports the following class:

- **VVUIMenuInfo**

vvUIMenuInfo (Class - Visual Basic and MFC Only)

The purpose of the **vvUIMenuInfo** class (Visual Basic and MFC only) is to provide information about custom menu options. Use this class when adding or modifying custom menu options.

Field	Type	Description
Type	UIMENUTYPE	Specifies the menu type. For more information, refer to “UIMENUTYPE” page 558.
ID	Integer	Specifies the ID number of the menu option. You can use this ID number to get information about the option with the GetMenuItemInfo method. The control uses this ID number to let you know which menu option the user selected. Note: The ID must be between 0-9,999 when adding dynamic applications, dynamic help or static help menu options. When adding dynamic main menu options, you must use one of the constants in the vvUIMenuItemConstants enumeration. For more information, refer to “vvUIMenuItemConstants” on page 547.
Checked	Boolean	Indicates whether the menu option has a check mark (True) or not (False).
Visible	Boolean	Indicates whether the menu option is visible (True) or invisible (False).
Enabled	Boolean	Indicates whether the menu option is enabled (True) or grayed (False).
Caption	String	The menu item text.
ExePathName	String	The path of the help file the UIServer will use when invoking a static help menu option

User Interface Control Structures

The ViaVoice **UIClient** control supports the following structure:

- **UIMENUITEMINFO**

UIMenuItemInfo Structure (Custom Interface Only)

The purpose of the **UIMENUITEMINFO** structure is to provide information about custom menu options for the Custom Interface. Use this structure when adding or modifying custom menu options.

Field	Type	Description
m_uType	UIMEN UTYPE	Specifies the menu type. For more information, refer to “UIMENUTYPE” page 558.
m_dwID	DWORD	Specifies the ID number of the menu option. You can use this ID number to get information about the option with the GetMenuItemInfo method. The control uses his ID number to let you know which menu option the user selected. Note: The ID must be between 0-9,999 when adding dynamic applications, dynamic help or static help menu options. When adding dynamic main menu options, you must use one of the constants in the vvUIMenuItemConstants enumeration. For more information, refer to “vvUIMenuItemConstants” on page 547.
m_uState	UINT	Indicates the state of the menu item. This member is identical to the fState member in the WIN32 SDK MENUITEMINFO structure. For more information, refer to the WIN32 SDK documentation.

Field	Type	Description
m_szText [MAX_MENU_STRING]	CHAR	Menu item text.
m_exeItem	struct UIMEN ITEMI NFO_E XE	The path of the help file the UIServer will use when invoking a static help menu option. For structure information, refer to Table 6 on page 539.

The **UIMENUITEMINFO_EXE** structure is similar to the parameters in the ShellExecute WIN32 SDK API function. For more information, refer to the WIN32 SDK documentation.

Field	Type	Description
m_szOperation [MAX_MENU_OPERATION]	CHAR	Specifies the operation to perform (“open,” “print,” “help” and so on).
m_dwOpData;	DWORD	Operation-specific data. For “open” and “print” operations it is ignored. For “help” operations, it is the topic ID to display.
m_szFile [_MAX_PATH]	CHAR	File to perform operation on (*.EXE, *.HLP, and so on).
m_szParameters [_MAX_PATH]	CHAR	File parameters (if any).
m_szWorkDirectory [_MAX_PATH]	CHAR	File working directory (if any)
m_nShowCmd	INT	Specifies how the application is to be shown.

User Interface Control Constants

The ViaVoice **UIClient** control supports the following constants:

- **vvUIDockingAlgorithmConstants**
- **vvUIDockingStyleConstants**
- **vvUIEventCallbackFlags**
- **vvUIExtendedMenuFlags**
- **vvUIMaxConstants**
- **vvUIMenuItemConstants**
- **Component Index Constants**
- **vvUIRemoveClientConstants**

Component Index Constants

Use **Component Index** constants when getting and setting the properties of a **UIServer** component by calling the **SetNumberValue**, **SetStringValue**, **GetNumberValue**, or **GetStringValue**. The index constants enable you to specify the property within the component that you wish to query or modify.

Component Index constants are specific to a particular component. The following list contains the index constants for each component in the **UIServer** that you can query or modify.

Table 4. vvUIMICINDEX

Constant Name	Value	Description
vvUIMICINDEX_MICSTATE	0	The microphone state property. For example, On or Off.
vvUIMICINDEX_WAITSTATE	1	The wait state property. You can place the microphone component in a wait state regardless of the current microphone state.

Table 5. vvUIUserInfoIndex (COMPID_USERINFORMATION)

Constant Name	Value	Description
vvUIUIINFOINDEX_USERID	0 (default)	Specifies the User ID. ID name of the speech engine's current user, which is usually the user's full name.
vvUIUIINFOINDEX_ENROLLID	1	Specifies the Enrollment ID, which specifies the language the engine uses for the current user.
vvUIUIINFOINDEX_TASKID	2	Specifies the vocabulary the speech engine uses for the particular user.
vvUIUIINFOINDEX_USER_DESCRIPTION	3	The current user's description.

Table 5. vvUIUserInfoIndex (COMPID_USERINFORMATION)

Constant Name	Value	Description
vvUIINFOINDEX_ENROLL_DESCRIPTION	4	The complete description of the enrollment ID.
vvUIINFOINDEX_TASK_DESCRIPTION	5	The complete description of the vocabulary ID.

Table 6. vvUIVolumeIndex (COMPID_VOLUME)

Constant Name	Value	Description
vvUIVOLINDEX_VOLLEVEL	0 (default)	The volume level. In taskbar and docked mode, the UIServer displays a volume meter.

Table 7. vvUIWordHistoryIndex (COMPID_WORDHISTORY)

Constant Name	Value	Description
vvUIWHINDEX_TAGGEDTEXT	0 (default)	The word history text. Applications use this text to show users the last command the engine recognized, or to give users help information, such as, a list of the commands they can say in the current state.

vvUIDockingAlgorithmConstants

You can use these constants when you write code to handle the **EventQueryViewFlags** event, specifically to set the value of the `pdwDockFlags` parameter. The Control fires this event whenever the user requests to change the view mode of the **UIServer** to “docked,” or when the view mode is already docked, but another application has had the focus and your application is about to receive the focus. These constants enable you to grant or deny the users’ request.

Table 8. vvUIDocking Algorithm Constants

Constant Name	Value	Description
vvDVAF_ALLOW_DOCK	33554432 (Hex 2000000)	Allows the UIServer to dock to your window.
vvDVAF_ALLOW_TOPMOST_DOCK	67108864 (Hex 4000000)	Allows the UIServer to dock to the top-most window in this application.
vvDVAF_DEFAULT	67108864 (Hex 4000000)	Uses default. Note: Although the default is currently <code>vvDVAF_ALLOW_TOPMOST_DOCK</code> , you should use this constant instead of <code>vvDVAF_ALLOW_TOPMOST_DOCK</code> when you want to use the default. By using this constant, you guarantee that your program always uses the default docking style, even if the default changes in future releases.

Table 8. vvUIDocking Algorithm Constants

Constant Name	Value	Description
vvDVAF_NEVER_DOCK	16777216 (Hex 1000000)	Prevents the UIServer from docking to your window. If the UIServer is unable to dock to your window, it will switch to a minimal mode tray icon.
vvDVAF_STAY_DOCK_TO_PREVIOUS	134217728 (Hex 8000000)	If the UIServer was previously docked to another window, and your application receives the focus, this constant tells the UIServer to remain in the previous application, and not to attempt to switch to yours.

vvUIDockingStyleConstants

These constants work in conjunction with **vvUIDockingAlgorithmConstants**. When you use the **vvDVAF_ALLOW_DOCK** or the **vvDVAF_ALLOW_TOPMOST_DOCK** constants to grant permission to the **UIServer** to dock to your window, you can set the style the **UIServer** uses to dock. For more information about granting docking permissions, refer to “EventQueryViewFlags” on page 621.

Table 9. vvUIDocking Style Constants

Constant Name	Value	Description
vvDVSF_DEFAULT	0	Uses default. Note: Although the default is vvDVSF_NORMAL_BACKGROUND , you should use this constant instead of vvDVSF_NORMAL_BACKGROUND when you want to use the default. By using this constant, you guarantee that your program always uses the default docking style, even if the default changes in future releases.
vvDVSF_NORMAL_BACKGROUND	0	Uses the standard windows system color for 3D objects as the background color.
vvDVSF_TRANSPARENT_BACKGROUND	1	Uses a transparent background.
vvDVSF_ADJUST_ORIGIN	4	Adjusts the origin of the docked view into the current window. For more information, refer to the VVUICNST.H file.
vvDVSF_ADJUST_WIDTH	8	Adjusts the width of the docked view into the current window. For more information, refer to the VVUICNST.H file.

vvUIEventCallbackFlags

These constants define the possible events that the **UIClient** control can receive from the **UIServer**. Use these constants when using the **SetClientCallbackFlags** function.

Table 10. vvUIEventCallbackFlags

Constant Name	Value	Description
vvUIEVENT_ACTIVEAPP_CHANGED	16777220 (Hex 1000004)	Receives application changed notifications. The UIClient control fires this application if one of the applications you added with the AddApplicationByName or AddApplicationByWindow command receives or loses the focus.
vvUIEVENT_ALL	16711681 (Hex FFFFFFFF)	Captures all events.
vvUIEVENT_BUTTON_PRESSED	33554448 (Hex 2000010)	Receives button pressed notifications. This occurs when the user clicks one of the buttons in the UIServer .
vvUIEVENT_COMPONENT_UPDATE D	134221824 (Hex 8001000)	Occurs when one of the components is changed. For more information, refer to “SetNumberValue” on page 606 and “SetStringValue” on page 609.
vvUIEVENT_MENUITEM_SELECTED	67109120 (Hex 4000100)	Occurs when the user selects a custom menu option.
vvUIEVENT_NONE	0	Do not receive events. The control will not fire any events.

Table 10. vvUIEventCallbackFlags

Constant Name	Value	Description
vvUIEVENT_VIEW_QUERYFLAGS	16777217 (Hex 1000001)	Occurs every few seconds and asks the client for permission to dock to the application’s window.
vvUIEVENT_VIEW_QUERYMENUINFO	16777218 (Hex 1000002)	Occurs just before the UIServer displays the main menu.

vvUIExtendedMenuFlags

The **vvUIExtendedMenuFlags** constant contains menu styles that complement the standard Win32 API **MFS_*** flags. Use these flags when you set the **m_uState** of the **UIMENUITEMINFO** structure.

Table 11. **vvUIExtendedMenuFlags**

Constant Name	Value	Description
vvVVUI_MFS_HIDDEN	16777216 (Hex 1000000)	Hides a menu item, but does not physically remove it. The menu item still exists, but it is invisible. You can do GetMenuItemInfo and SetMenuItemInfo calls on these hidden menus

vvUIMaxConstants

These constants define the maximum size for string value properties in various components in the UIServer.

Table 12. vvUIMaxConstants

Constant Name	Value	Description
vvMAX_MENU_OPERATION	32 (Hex 20)	The maximum number of characters allowed in the m_szOperation member of the UIMENUITEMINFO_EXE structure. For more information, refer to UIMENUITEMINFO_EXE .
vvMAX_MENU_STRING	64 (Hex 40)	The maximum number of characters the menu item caption (or text) can be.
vvMAX_WORDHISTORY_TEXT	128 (Hex 80)	The maximum number of characters you use when getting/setting the text in the word history component.
vvMAX_USERINFO_ID_LEN	32 (Hex 20)	The maximum number of characters you use when getting/setting the text in the user information component.
vvMAX_USERINFO_DESC_LEN	80 (Hex 50)	The maximum number of characters you can get/set from user information components.

vvUIMenuItemConstants

Each built-in menu option in the **UIServer** menu has a unique ID. For some of the items, you can use this ID to turn on/off the menu option (with the **AppendMenuItem** or **InsertMenuItem**, and **DeleteMenuItem** methods). Some menu options, such as Microphone On/Off, cannot be removed except where noted. You can also use these constants when handling the **EventMenuItemSelected** event - they indicate the ID of the menu item the user selected.

Table 13. vvUIMenuItemConstants

Constant Name	Value	Description
vvVIAVOICEUI_IDMENU_BEGIN_DICTATION	50200 (Hex C418)	Begin Dictation menu option. Users normally select this option when they want to enter dictation mode.
vvVIAVOICEUI_IDMENU_BEGIN_READING	50275 (Hex C463)	Begin Reading menu option. Users normally select this option when they want the program to turn selected text into speech.
vvVIAVOICEUI_IDMENU_SHOW_CORRECTION_WINDOW	50250 (Hex C44A)	Show Correction Window menu option. Users normally select this option when they want to correct a word or phrase. The application usually responds by invoking the correction dialog box.

Table 13. vvUIMenuItemConstants

Constant Name	Value	Description
vvVIAVOICEUI_IDMENU_EXIT	50900 (Hex C6D4)	Exit menu option. Users select this option when they wish to “end speech support” for the current speech application.
vvVIAVOICEUI_IDMENU_MICROPHONE	50025 (Hex C369)	Microphone menu option. You cannot remove this menu option and you will receive notification that the users selected this option.
vvVIAVOICEUI_IDMENU_STOP_DICTATION	50225 (Hex C431)	Stop Dictation menu option. This menu option is the opposite of Begin Dictation. Users select this option when they wish to exit dictation mode.

Table 13. vvUIMenuItemConstants

Constant Name	Value	Description
vvVIAVOICE_IDMENU_STOP_READING	50300 (14B4)	Stop Reading menu option. This menu option is the opposite of Begin Reading. Users select this option when they wish to stop the reading mode.
vvVIAVOICEUI_IDMENU_WHAT_CAN_I_SAY	50500 (Hex C544)	What Can I Say menu option. You will receive notification that users selected this option; you can remove, append or modify this option.

vvUIRemoveClientConstants

Use these constants when issuing the **RemoveClient** method (only available for Visual C++ Custom Interface users). These constants enable you to specify how the **UIServer** should respond to the client shutting down.

Table 14. vvUIRemoveClientConstants

Constant Name	Value	Description
vvUIRCF_NO_CLOSE	0	Do not close the UIServer , even if it is the last client to use it.
vvUIRCF_CLOSE	1	Closes the UIServer regardless of what other clients may be using it. Note: <i>You should never use this option unless it is absolutely necessary.</i>
vvUIRCF_CLOSE_IF_LAST_CLIENT	2	Closes the UIServer if this client is the last client only.
vvUIRCF_DEFAULT	2	Uses the default option. Currently the default is vvUIRCF_CLOSE_IF_LAST_CLIENT . However, you should use the vvUIRFC_DEFAULT flag instead when you want to close the UIServer in your program as this constant might change in a future release.
vvUIRCF_CLOSE_IF_LAST_CLIENT_DELAY	4	In rare instances, the UIServer cannot shut down before the client application shuts down completely. For those cases where you wish the UIServer to shut down when your client is the last one, but you would like the client to have enough time to shut down, use this constant. The UIServer will wait a few seconds after your program shuts down.

User Interface Control Enumerations

The ViaVoice **UIClient** control supports the following enumerations:

- **MICROPHONE_STATES** (Enum)
- **TCID** (Enum)
- **TVIEWTYPE** (Enum)
- **UIEVENTRC** (Enum)
- **UIMENUGROUP** (Enum)
- **UIMENUTYPE** (Enum)
- **UIRC** (Enum)

MICROPHONE_STATES (Enum)

You can use these constants to set the state of the UIServer’s Microphone component. Use these constants when using the **SetNumberValue** or **GetNumberValue** methods with the **COMPID_MICROPHONE** object, as shown below.

```
SetNumberValue (COMPID_MICROPHONE, lSetValue, Index)
```

Keep in mind that setting the state of the microphone component has no direct effect on the state of the system’s audio input device in the speech engine; it simply sets the appearance of the microphone component.

When you use the following constants, you must set the Index parameter to **vvUIMICINDEX_MICSTATE (0)** or **UIMICINDEX_MICSTATE (Custom)** as follows:

```
SetNumberValue (COMPID_MICROPHONE, UIMSF_ON, vvUIMICINDEX_MICSTATE)
```

(In Visual Basic, the Index parameter is optional and defaults to zero).

Table 15. Microphone State Constants for the vvUIMICINDEX_MICSTATE

Constant Name	Value	Description
UIMSF_ON	1	Sets the state of the microphone object to “on.”
UIMSF_OFF	2	Sets the state of the microphone object to “off.”
UIMSF_SLEEP	3	Sets the state of the microphone object to “asleep.”
UIMSF_DISABLED	4	Sets the state of the microphone object to “disabled.”
UIMSF_ERROR	5	Sets the state of the microphone object to a “no” sign.

When using the following constants you must set the Index parameter to **vvUIMICINDEX_WAITSTATE** as follows:

```
SetNumberValue (COMPID_MICROPHONE, UIMSF_ADDWAIT, vvUIMICINDEX_WAITSTATE)
```


Table 16. Microphone State Constants for the `vvUIMICSTATE_WAITSTATE`

Constant Name	Index Parameter	Description
UIMSF_ADDWAIT	4096 (&H1000)	Changes the state of the microphone object to a wait state. Each time you call the SetNumberValue method with this value the UIServer increments a counter. To return to a ready state you must call this method with the UIMSF_REMOVEWAIT flag the same number of times the UIMSF_ADDWAIT flag was used. You could also use the UIMSF_CLEARWAIT to remove all instances of the wait state.
UIMSF_CLEARWAIT	4097 (&H1001)	Removes all instances of the wait state from the microphone object. See the description on UIMSF_ADDWAIT for details.
UIMSF_REMOVEWAIT	4098 (&H1002)	Removes one instance of the wait state. See the description of the UIMSF_ADDWAIT constant for details.

TCID (Enum)

TCID stands for component ID type. This enumerated type contains the ID numbers for each of the components you can modify in the **UIServer** using the **SetNumberValue** or **SetStringValue** methods. You can also use these constants when querying the state of one of the components in the **UIServer** by using **GetNumberValue** or **GetStringValue** methods.

Table 17. Component ID Type Constants

Constant Name	Value	Description
COMPID_MICROPHONE	1	Gets/Sets the state of the microphone component.
COMPID_VOLUME	2	Gets/Sets the level of the volume component.
COMPID_WORDHISTORY	3	Gets/Sets the word history text. Word history text appears in the center of the UIServer when in Taskbar/Docked view.
COMPID_USERINFORMATION	4	Gets/Sets user information such as the displayable user name.
COMPID_CUSTOM	5	Gets/Sets the state of toolbar buttons in the UIServer .
COMPID_MAINMENU	6	Represents the main menu. You cannot use this ID with SetNumberValue or GetNumberValue ; however, you may use it when interpreting the ComponentID parameter in the EventButtonPressed event.

TVIEWTYPE (Enum)

This enumerated type is issued when the **UIClient** control fires the **EventQueryViewMenuInfo** event. The **EventQueryViewMenuInfo** event gets fired when the user requests to see the **UIServer** menu, prior to the **UIServer** displaying the menu options. This enumerated type contains constants that describe the current appearance of the **UIServer**.

Table 18. UI Server Type Constants

Constant Name	Value	Description
UIVIEW_SYSTRAY	1	Means that the UIServer appears as an icon in the Windows tool tray or system tray
UIVIEW_TASKBAR	2	Means that the UIServer is acting as a taskbar similar to the Windows taskbar on the desktop.
UIVIEW_DOCKED	3	Means that the UIServer is currently docked to the application that has the focus.
UIVIEW_AGENT	4	Means that the UIServer is currently acting as an agent character.

UIEVENTRC (Enum)

If you study the events in the **UIClient** control you will find that the last parameter in each of the events is pResult of type **UIEVENTRC**. The **UIEventRC** enumerated type is used to inform the **UIClient** control of the action taken in one of its events.

Table 19. UIEVENTRC Type Constants

Constant Name	Value	Description
UIEVENTRC_PROCESSED	0	Notifies the UIClient control that your code has processed the event, and it should not be forwarded to other clients.
UIEVENTRC_NOTPROCESSED	1	Notifies the UIClient control that your code has not processed the event, and it should be forwarded to other clients.

UIMENUGROUP (Enum)

UIMenuGroup specifies the location where you wish to add custom menu options to the ViaVoice **UIServer** menu. The word Dynamic means that the menu option will change according to which program has the focus, and Static means the options will not. The Static help menus remain even if the application is shut down and restarted until they are explicitly removed by a call to **DeleteMenu**. The **UIClient** control only informs you of interaction with Dynamic menu items.

Table 20. Dynamic Menu Group Constants

Constant Name	Value	Description
UIMFG_DYNAMIC_MAIN	1	Refers to menu options that appear in the main menu. These options are: Begin Dictation Stop Dictation Show Correction Window Begin Reading Stop Reading What Can I Say? Exit
UIMFG_DYNAMIC_APPLICATION	2	Refers to custom application menu options.
UIMFG_DYNAMIC_HELP	3	Refers to custom help menu options.
UIMFG_STATIC_HELP	4	Refers to static help menu options, which always appear regardless of which application has the focus.

UIMENUTYPE (Enum)

The **UIMENUTYPE** enumeration specifies how the **UIServer** handles the menu option. It can either notify the **UIClient** control or it can execute the menu option automatically.

Table 21. Dynamic Menu Type Constants

Constant Name	Value	Description
UIMFT_CLIENT	1	The menu option will be handled by the UIClient .
UIMFT_EXECUTE	2	The menu option will be executed automatically.

UIRC (Enum)

All the methods in the **UIClient** control return error information due to logical errors through a return code. (Critical errors are reported through trappable errors in Visual Basic or exception faults in Visual C++ (MFC) or a non-successful HRESULT in Visual C++ (Custom).)

Table 22. UIRC Type Constants

Constant Name	Value	Description
UIAPIRC_OK	0	No Error. Call was successful.
UIAPIRC_ERROR_NOSERVER	1	The server was unable to initialize.
UIAPIRC_ERROR_SERVERBUSY	2	The server was unable to execute your method within a certain amount of time.
UIAPIRC_ERROR_OUTOFMEMORY	3	Not enough memory to complete the call. You should terminate the application.
UIAPIRC_ERROR_FAILED	4	General failure.
UIAPIRC_ERROR_INVALIDCLIENT	5	The UIClient control has become unstable.
UIAPIRC_ERROR_INVALIDPARAM	7	This method had an invalid parameter.
UIAPIRC_ERROR_ALREADYINITIALIZED	8	This result code occurs when attempting to Initialize the UIServer twice.
UIAPIRC_ERROR_NOTCURRENTCLIENT	9	This result code may occur when attempting to change the state of a component when your application is not the current client.

User Interface Control Properties

The **UIClient** control, invisible at run time, does not have any custom properties, but has the following standard properties in Visual Basic:

In Visual Basic:

The **User Interface Client** control supports the following standard properties:

- **Index**^a
- **LanguageUI**
- **Left**^a
- **Tag**^a
- **Top**^a

In Visual C++ (MFC):

- **LanguageUI**

In Visual C++ (Custom Interface):

There are no standard properties available.

a. Represents a standard method in Visual Basic. For more information, refer to your Visual Basic documentation.

LanguageUI

Sets or gets the language used by the **UIServer** for this specific client.

Syntax

In Visual Basic:

```
[VVUIClient].LanguageUI = [String]
```

In Visual C++ (MFC):

```
CString = [VVUIClient].GetLanguageUI()  
[VVUIClient].SetLanguageUI(LPTSTR)
```

In Visual C++:

```
HRESULT [pIVVUIClient]->get_LanguageUI(BSTR *)  
HRESULT [pIVVUIClient]->put_LanguageUI(BSTR)
```

Parameters

Return Values

The **LanguageUI** property settings for a ViaVoice **User Interface** control are:

Language	Property Value
U.S. English	“EN_US”
U.K. English	“EN_UK”
German	“GR_GR”
Italian	“IT_IT”
Spanish	“ES_ES”
French	“FR_FR”
Japanese	“JA_JP”

Remarks

The language affects the menus, dialogs, and ToolTips displayed by the **UIServer**.

Example

In Visual Basic:

```
' Sets UI language to U.S. English
VVUIClient1.LanguageUI = "EN_US"
' Gets UI language and displays it in a message box
MsgBox VVUIClient1.LanguageUI
```

In Visual C++ (MFC):

```
// Sets UI language to U.S. English
m_VVUIClient.SetLanguageUI("EN_US");
CString sLangUI;
// Gets UI language and copies it into variable
sLangUI = m_VVUIClient.GetLanguageUI();
```

In Visual C++:

```
HRESULT hr;
BSTR bstrLangUI;

bstrLangUI = SysAllocString(OLESTR("EN_US"));
// Sets UI language to U.S. English
hr = pIVVUIClient->put_LanguageUI(bstrLangUI);
SysFreeString(bstrLangUI);

// Gets UI language into BSTR variable
hr = pIVVUIClient->get_LanguageUI(&bstrLangUI);
// Use language string now and when done free BSTR.
SysFreeString(bstrLangUI);
```

See Also

“SetLanguageByID” on page 599

“SetLanguageByString” on page 601

Table 3 on page 509

User Interface Control Methods

The **User Interface Control** supports the following methods:

- **About^a**
- **AddApplicationByName**
- **AddApplicationByWindow**
- **AppendMenuItem**
- **DeleteMenuItem**
- **GetMenuItemInfo**
- **GetNumberValue**
- **GetStringValue**
- **Initialize**
- **InsertMenuItem**
- **RemoveApplicationByName**
- **RemoveApplicationByWindow**
- **SetClientCallback**
- **SetClientCallbackFlags**
- **SetLanguageByID**
- **SetLanguageByString**
- **SetMenuItemInfo**
- **SetNumberValue**
- **SetStringValue**

a. Represents a standard method in Visual Basic. For more information, refer to your Visual Basic documentation.

AddApplicationByName

The **UIClient** control enables you to interact with the ViaVoice **UIServer**; however, in order to get messages from the **UIServer**, the **UIServer** needs to know which programs it is interacting with. To specify a program by name, you can use the **AddApplicationByName** method.

Syntax

In Visual Basic:

```
Function AddApplicationByName(ApplicationName As String) As UIRC
```

In Visual C++ (MFC):

```
long AddApplicationByName(LPCTSTR ApplicationName);
```

In Visual C++ (Custom Interface):

```
HRESULT AddApplicationByName(LPSTR lpszApplicationName, UIRC *pResult);
```

Parameters

ApplicationName

String / LPSTR. The executable name of the application and its extension (for example, CALC.EXE) or a fully qualified path plus the application name and its extension (for example, C:\WINDOWS\notepad.exe).

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

In Visual Basic the executable name is different at design time than it is at run time. It is recommended that you use the **AddApplicationByWindow** method instead for adding your program, and use the **AddApplicationByName** method when adding other programs.

The program added does not need to be running. The **UIServer** will simply not send any notifications to the **UIClient** control until the application starts and is active.

This method will not work if you omit the “.EXE” extension from the name. Remember, to receive events and interact with the **UIServer** when you are active, you must add your own application.

Example

In Visual Basic:

```
Dim rc As UIRC  
rc = VVUIClient1.AddApplicationByName('NOTEPAD.EXE')
```

In Visual C++ (MFC):

```
UIRC rc = (UIRC) m_VVUIClient1.AddApplicationByName("NOTEPAD.EXE");
```

In Visual C++ (Custom):

```
UIRC rc;  
HRESULT hr = pIVVUIClient->AddApplicationByName("NOTEPAD.EXE",&rc);
```

See Also

“AddApplicationByWindow” on page 568

“RemoveApplicationByName” on page 591

“RemoveApplicationByWindow” on page 593

AddApplicationByWindow

The **UIClient** control enables you to interact with the **UIServer**; however, in order to get messages from the **UIServer**, the **UIServer** needs to know which programs it is interacting with. To specify a program by its window handle, you can use **AddApplicationByWindow** method.

Syntax

In Visual Basic:

```
Function AddApplicationByWindow (hwndApplication As Long) As UIRC
```

In Visual C++ (MFC):

```
long AddApplicationByWindow(long hwndApplication);
```

In Visual C++ (Custom):

```
HRESULT AddApplicationByWindow(HWND_t hwndApplication, UIRC *pResult);
```

Parameters

hwndApplication

Long / HWND_t. The window handle of the main form in the application.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

In Visual Basic the executable name is different at design time that it is at run time. It is recommended that you use the **AddApplicationByWindow** method instead for adding your program, and use the **AddApplicationByName** method when adding other programs.

Remember, to receive events and interact with the **UIServer** when you are active, you must add your own application.

Example

In Visual Basic:

```
Dim rc As UIRC  
rc = VVUIClient1.AddApplicationByWindow Me.hWnd)
```

In Visual C++ (MFC):

```
UIRC rc = (UIRC)m_VVUIClient1.AddApplicationByWindow((long)m_hWnd);
```

In Visual C++ (Custom):

```
UIRC rc;  
HRESULT hr = pIVVUIClient->AddApplicationByWindow((HWND_t)hWnd,&rc);
```

See Also

“AddApplicationByWindow” on page 568

“RemoveApplicationByName” on page 591

“RemoveApplicationByWindow” on page 593

AppendMenuItem

Adds custom menu items in either of two ways: as an application-dependent menu group, or as a help menu item.

Syntax

In Visual Basic:

```
Function AppendMenuItem(uUIMenuGroup As UIMENUGROUP, MenuName As String,  
pIMenuInfo As VVUIMenuInfo) As UIRC
```

In Visual C++ (MFC):

```
long AppendMenuItem(long uUIMenuGroup, LPCTSTR MenuName, LPDISPATCH  
pIMenuInfo);
```

In Visual C++ (Custom):

```
HRESULT AppendMenuItem(UIMENUGROUP uUIMenuGroup, LPSTR lpzMenuName,  
UIMENUITEMINFO* lpuimiiItem, UIRC* pResult);
```

Parameters

uUIMenuGroup

UIMENUGROUP. This parameter specifies the location where you wish to add the menu item. For more information, refer to Chapter 27 “Classes, Structures, and Enumerations”.

MenuName

String / LPSTR. Use this parameter to specify the menu group name.

pIMenuInfo

VVUIMenuInfo / UIMENUITEMINFO. A menu information structure. For more information, refer to “User Interface Control Structures” on page 535. This structure contains the characteristics of the menu item.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

You can also use this method to turn on main menu items such as “Begin Dictation” and “Show Correction Window.”

The **AppendMenuItem** item is similar to the **InsertMenuInfo** item, except that **AppendMenuItem** adds the menu item to the end of the menu list, while **InsertMenuInfo** enables you to specify the location of the new menu item.

When adding a menu item you must specify an ID number for the item. You can do this by setting the ID member of the **VVUIMenuInfo** structure.

When the user clicks on the custom menu item, the **UIClient** control will fire the **EventMenuItemSelected** event. For more information, refer to “EventMenuItemSelected” on page 619.

If you use the **AppendMenuInfo** method multiple times with the same MenuID for the menu items, the **UIClient** control will not generate an error – it will simply add the menu item multiple times. This is useful if you wish to handle menu items with different text in the same way.

Example

In Visual Basic:

```
Dim lRetVal As UIRC
Dim oNewMenu As New VVUIMenuInfo

oNewMenu.Type = UIMFT_CLIENT
oNewMenu.ID = 100
oNewMenu.Caption = "Menu Item #1"
oNewMenu.Checked = True

rc = VVUIClient1.AppendMenuItem(UIMEF_DYNAMIC_APPLICATION, _
App.EXENAME, _
oNewMenu)

End Sub
```

In Visual C++ (MFC):

```
CVVUIMenuInfo NewMenuItem;
NewMenuItem.CreateDispatch(CLSID_VVUIMenuInfo);
NewMenuItem.SetType(UIMFT_CLIENT);
NewMenuItem.SetID(100);
NewMenuItem.SetCaption("Menu Item #1");
NewMenuItem.SetChecked(TRUE);

m_VVUIClient1.AppendMenuItem(UIMFG_DYNAMIC_APPLICATION,
AfxGetAppName(),
NewMenuItem);
```

In Visual C++ (Custom):

```
UIRC rc;
UIMENUITEMINFO NewMenuItem;

ZeroMemory(&NewMenuItem, sizeof(NewMenuItem));
NewMenuItem.m_uType = UIMFT_CLIENT;
NewMenuItem.m_dwID = 100;
lstrcpy(NewMenuItem.m_szText, "Menu Item #1");
NewMenuItem.m_uState = MFS_ENABLED | MFS_CHECKED;

HRESULT hr=pIVVUIClient->AppendMenuItem(UIMFG_DYNAMIC_APPLICATION,
    "MyAppName"
    &NewMenuItem,
    &rc);
```

See Also

“InsertMenuItem” on page 587

“SetClientCallback (Custom Interface)” on page 595

DeleteMenuItem

Removes a custom menu entry added as an application menu item or as a help menu item.

Syntax

In Visual Basic:

```
Function DeleteMenuItem(uUIMenuGroup As UIMENUGROUP, MenuName As String,  
uItem As Long, fByPosition As Long) As UIRC
```

In Visual C++ (MFC):

```
long DeleteMenuItem(long uUIMenuGroup, LPCTSTR MenuName, long uItem, long  
fByPosition);
```

In Visual C++ (Custom):

```
HRESULT DeleteMenuItem(UIMENUGROUP uUIMenuGroup, LPSTR lpszMenuName,  
UIint uItem, BOOL fByPosition, UIRC* pResult);
```

Parameters

uUIMenuGroup

UIMENUGROUP. One of the menu group flags described in Chapter 27 “Classes, Structures, and Enumerations”.

MenuName

String / LPSTR. The name of the menu group from which the menu item will be deleted.

uItem

Long / UINT. The menu item’s ID number, or the menu item’s position within the group. This parameter changes meanings depending on the value of the fByPosition parameter. When the fByPosition parameter is zero, then the uItem parameter indicates the menu ID number. If the fByPosition parameter is set to one, then the uItem parameter indicates the 1-based position from the top within the group of the menu item.

fByPosition

Long / BOOL. Set this parameter to zero to indicate the ulItem parameter represents a menu ID number. Set it to one to indicate that the ulItem parameter represents a menu item's position.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

If you are unsure that the menu item exists, use the **GetMenuInfo** menu item prior to using this method; otherwise the method will return an error HRESULT generating a trappable error in Visual Basic and Visual C++ (MFC).

Example

In Visual Basic:

```
Dim rc As UIRC
rc = VVUIClient1.DeleteMenuItem(UIMFG_DYNAMIC_APPLICATION, _
    App.EXENAME, _
    100, _
    False)
```

In Visual C++ (MFC):

```
UIRC rc;
rc = (UIRC) m_VVUIClient1.DeleteMenuItem(UIMFG_DYNAMIC_APPLICATION,
    AfxGetAppName(),
    100,
    FALSE);
```

In Visual C++ (Custom):

```
UIRC rc;  
HRESULT hr = pIVVUIClient->DeleteMenuItem(UIMFG_DYNAMIC_APPLICATION,  
    "MyAppName",  
    100,  
    FALSE,  
    &rc);
```

See Also

“AppendMenuItem” on page 570

“InsertMenuItem” on page 587

GetMenuItemInfo

Obtains information about a custom menu item, or about one of the main menu items.

Syntax

In Visual Basic:

```
Function GetMenuItemInfo(uUIMenuGroup As UIMENUGROUP, MenuName As
String, uItem As Long, fByPosition As Long, pIMenuInfo As VVUIMenuInfo)
As UIRC
```

In Visual C++ (MFC):

```
long GetMenuItemInfo(long uUIMenuGroup, LPCTSTR MenuName, long uItem,
long fByPosition, LPDISPATCH pIMenuInfo);
```

In Visual C++ (Custom):

```
HRESULT GetMenuItemInfo(UIMENUGROUP uUIMenuGroup, LPSTR lpszMenuName,
int uItem, BOOL fByPosition, UIMENUITEMINFO* lpuimiiItem, UIRC* pResult);
```

Parameters

uUIMenuGroup

UIMENUGROUP. One of the UIMENUGROUP flags described in Chapter 27 “Classes, Structures, and Enumerations”.

MenuName

String / LPSTR. The name of the menu group from which you are requesting information.

uItem

Long / UINT. The menu item’s ID number, or the menu item’s position within the group. This parameter changes meanings depending on the value of the *fByPosition* parameter. When the *fByPosition* parameter is zero then the *uItem* parameter indicates the menu ID number. If the *fByPosition* parameter is set to one then the *uItem* parameter indicates the 1-based position from the top within the group of the menu item.

fByPosition

Long / BOOL. Set this parameter to zero to indicate the ultem parameter represents a menu ID number. Set it to one to indicate that the ultem parameter represents a menu item's position.

pIMenuInfo

VVUIMenuInfo / UIMENUITEMINFO. An instance of the **VVUIMenuInfo** class or of the **UIMENUITEMINFO** structure. You must declare a new instance of this class or of the structure prior to calling the method.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

The information is retrieved through the **pIMenuInfo** parameter. This parameter will contain an instance of the **VVUIMenuInfo** class in Visual Basic and Visual C++ (MFC), or an instance of the **UIMENUITEMINFO** structure in Visual C++ (Custom).

If you call the **GetMenuItemInfo** method with an invalid ID number the function will return UIRC code UIAPIRC_ERROR_FAILED (4). To enumerate through all of the menu items, issue this method using the “by position” flag until the method returns the failed error code.

Example

In Visual Basic:

```
Dim rc As UIRC
Dim oMenuInfo As New VVUIMenuInfo
rc = VVUIClient1.GetMenuItemInfo(UIMFG_DYNAMIC_APPLICATION, _
                                sMenuGroup, _
                                iCounter, _
                                1, _
                                oMenuInfo)
```

In Visual C++ (MFC):

```
CVVUIMenuInfo MenuItem;  
MenuItem.CreateDispatch(CLSID_VVUIMenuInfo);  
UIRC rc = (UIRC)  
m_vvUIClient1.GetMenuItemInfo(UIMFG_DYNAMIC_APPLICATION,  
    "MyAppName",  
    iCounter,  
    TRUE,  
    MenuItem);
```

In Visual C++ (Custom):

```
UIRC rc;  
UIMENUITEMINFO MenuItem;  
  
HRESULT hr=pIVVUIClient->GetMenuItemInfo(UIMFG_DYNAMIC_APPLICATION,  
    "MyAppName",  
    iCounter,  
    TRUE,  
    &MenuItem,  
    &rc);
```

See Also

“SetMenuItemInfo” on page 603

“vvUIMenuInfo (Class - Visual Basic and MFC Only)” on page 534

“User Interface Control Structures” on page 535.

GetNumberValue

Obtains numeric information on the state of a certain component in the **UIServer** (for example, the microphone component, `COMPID_MICROPHONE`).

Syntax

In Visual Basic:

```
Function GetNumberValue(ciComponent As TCID, pdwValueData As Long,  
[nIndex As Long]) As UIRC
```

In Visual C++ (MFC):

```
long GetNumberValue(long ciComponent, long* pdwValueData, long nIndex);
```

In Visual C++ (Custom):

```
HRESULT GetNumberValue(TCID ciComponent, DWORD* pdwValueData, int  
nIndex, UIRC* pResult);
```

Parameters

ciComponent

TCID. One of the UIServer's object IDs. You will find a complete list of objects in Chapter 27 "Classes, Structures, and Enumerations".

pdwValueData

Long / DWORD. The `pdwValueData` parameter will contain the numeric value after issuing the method. In Visual Basic, you must declare a long variable for this parameter.

nIndex

Long (Optional) / int. This value enables you to access extended information about the object. Its meaning changes depending on which object you request information from. For a description of each of the possible values for this parameter, refer to "Component Index Constants" on page 538.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

If you call the **GetMenuItemInfo** method with an invalid ID number the function will return

Example

In Visual Basic:

```
Dim rc As UIRC
Dim NewValue as Long
rc = VVUIClient1.GetNumberValue(COMPID_MICROPHONE, NewValue)
```

In Visual C++ (MFC):

```
long NewValue;
UIRC rc = (UIRC) m_VVUIClient1.GetNumberValue(COMPID_MICROPHONE,
    &NewValue,
    UIMICINDEX_MICSTATE);
```

In Visual C++ (Custom):

```
DWORD NewValue;
UIRC rc;
HRESULT hr = pVVUIClient->GetNumberValue(COMPID_MICROPHONE,
    &NewValue,
    UIMICINDEX_MICSTATE,
    &rc);
```

See Also

“GetNumberValue” on page 580

“GetStringValue” on page 582

“SetNumberValue” on page 606

GetStringValue

Obtains string information on the state of a certain component in the **UIServer** (for example, the word history component, COMPID_WORDHISTORY).

Syntax

In Visual Basic:

```
Function GetStringValue(ciComponent As TCID, ValueData As String, [nIndex  
As Long]) As UIRC
```

In Visual C++ (MFC):

```
long GetStringValue(long ciComponent, BSTR* ValueData, long nIndex);
```

In Visual C++ (Custom):

```
HRESULT GetStringValue(TCID ciComponent, int nDataSize, LPSTR  
lppszValueData, int nIndex, UIRC* pResult);
```

Parameters

ciComponent

TCID. One of the UIServer's object IDs. You will find a complete list of objects in Chapter 27 "Classes, Structures, and Enumerations".

nDataSize (*Visual C++ Custom Only*)

Long / int. The int size of the string buffer that will hold the string value. The constant MAX_WORDHISTORY_TEXT defines the maximum length of the word history string, and MAX_USERINFO_ID_LEN/MAX_USERINFO_DESC_LEN defines the maximum lengths for the user information strings.

ValueData

String / LPSTR. The *pdwValueData* parameter will contain the string value after issuing the method. In Visual Basic, you must declare a string variable to use for this parameter.

nIndex

Long (Optional) / int. This value enables you to access extended information about the object. Its meaning changes depending on which object you request information from. For a detailed description of each of the possible values for this parameter, refer to “Component Index Constants” on page 538. For more information about this parameter, refer to the Remarks section below.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

In Visual Basic, the ValueData parameter can be a variable length string or a fixed-size string.

If using a fixed-size, use the MAX_* constants in Visual C++ (MFC or Custom) or vvMAX_* constants in Visual Basic.

Example

In Visual Basic:

```
Dim rc As UIRC
Dim NewValue As String
rc = If VVUIClient1.GetStringValue(COMPID_WORDHISTORY, NewValue) =
UIAPIRC_OK Then
```

In Visual C++ (MFC):

```
BSTR bstrNewValue=NULL;
UIRC rc = (UIRC) m_vvUIClient1.GetStringValue(COMPID_WORDHISTORY,
    &bstrNewValue,
    UIWHINDEX_TAGGEDTEXT);
SysFreeString(bstrNewValue);
```

In Visual C++ (Custom):

```
UIRC rc;
char szNewValue[MAX_WORDHISTORY_TEXT];
HRESULT hr = pIVVUIClient->GetStringValue(COMPID_WORDHISTORY,
    MAX_WORDHISTORY_TEXT,
    szNewValue,
    &rc,
    UIWHINDEX_TAGGEDTEXT);
```

See Also

“GetNumberValue” on page 580

“GetStringValue” on page 582

“SetNumberValue” on page 606

Initialize

Causes the **UIServer** to initialize (if it is not already visible, it also causes the **UIServer** to appear).

Syntax

In Visual Basic:

```
Function Initialize() As UIRC
```

In Visual C++ (MFC):

```
long Initialize();
```

In Visual C++ (Custom):

```
HRESULT Initialize(UIRC* pResult);
```

Parameters

None.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

You cannot call this method twice for the same control. However, each instance of the control must call this method. The **UIServer** keeps track of all the clients (controls) using it so that it can properly shut down when there are no clients using it.

The **UIServer**’s menu will not function until you call the **SetLanguageByString** or the **SetLanguageByID** method.

Example

In Visual Basic:

```
Dim rc As UIRC  
rc = m_VVUIClient1.Initialize()
```

In Visual C++ (MFC):

```
UIRC rc = (UIRC) m_VVUIClient1.Initialize();
```

In Visual C++ (Custom):

```
UIRC rc;  
HRESULT hr = pIVVUIClient->Initialize(&rc);
```

See Also

“SetLanguageByID” on page 599

“SetLanguageByString” on page 601

InsertMenuItem

Adds custom menu entries in either of two ways: as an application-dependent menu group or as a help menu item.

Syntax

In Visual Basic:

```
Function InsertMenuItem(uUIMenuGroup As UIMENUGROUP, MenuName As String,
    uItem As Long, fByPosition As Long, pIMenuInfo As VVUIMenuInfo) As UIRC
```

In Visual C++ (MFC):

```
long InsertMenuItem(long uUIMenuGroup, LPCTSTR MenuName, long uItem, long
    fByPosition, LPDISPATCH pIMenuInfo);
```

In Visual C++ (Custom):

```
HRESULT InsertMenuItem(UIMENUGROUP uUIMenuGroup, LPSTR lpzMenuName,
    UINT uItem, BOOL fByPosition, UIMENUITEMINFO* lpuiiiiItem, UIRC*
    pResult);
```

Parameters

uUIMenuGroup

UIMENUGROUP. This parameter specifies the location where you wish to add the menu item.

Valid values are: One of the UIMENUGROUP described in Chapter 27 “Classes, Structures, and Enumerations”.

MenuName

String / LPSTR. Use this parameter to specify the menu group name.

uItem

Long / int. The menu item’s ID number, or the menu item’s position within the group. This parameter changes meanings depending on the value of the *fByPosition* parameter. When the *fByPosition* parameter is zero, then the *uItem* parameter indicates the menu ID number. If the

fByPosition parameter is set to one, then the ulItem parameter indicates the 1-based position from the top within the group of the menu item.

fByPosition

Long / BOOL. Set this parameter to zero to indicate the ulItem parameter represents a menu ID number. Set it to one to indicate that the ulItem parameter represents a menu item's position.

pMenuInfo

VVUIMenuInfo / UIMENUITEMINFO. A menu information class or structure. For more information, refer to “User Interface Control Structures” on page 535. This structure contains the characteristics of the menu item.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

You can also use this method to enable main menu items such as “Begin Dictation”, and “Show Correction Window”. This method will position the new menu item before the item specified by the ulItem and fByPosition parameters.

Example

In Visual Basic:

```
Dim lRetVal As UIRC
Dim oNewMenu As New VVUIMenuInfo

oNewMenu.Type = UIMFT_CLIENT
oNewMenu.ID = 150
oNewMenu.Caption = "Menu Item #1"
oNewMenu.Checked = True

lRetVal = VVUIClient1.InsertMenuItem(UIMFG_DYNAMIC_HELP, _
    App.EXENAME, _
    200, _
    False, _
    oMenuInfo)
```

In Visual C++ (MFC):

```
CVVUIMenuInfo NewMenuItem;
NewMenuItem.CreateDispatch(CLSID_VVUIMenuInfo);
NewMenuItem.SetType(UIMFT_CLIENT);
NewMenuItem.SetID(150);
NewMenuItem.SetCaption("Menu Item #1");
NewMenuItem.SetChecked(TRUE);

m_VVUIClient1.InsertMenuItem(UIMFG_DYNAMIC_APPLICATION,
    AfxGetAppName(),
    200,
    FALSE,
    NewMenuItem);
```

In Visual C++ (Custom):

```
UIRC rc;
UIMENUITEMINFO NewMenuItem;

ZeroMemory(&NewMenuItem, sizeof(NewMenuItem));
NewMenuItem.m_uType = UIMFT_CLIENT;
NewMenuItem.m_dwID = 150;
lstrcpy(NewMenuItem.m_szText, "Menu Item #1");
NewMenuItem.m_uState = MFS_CHECKED: MFS_ENABLED;

HRESULT hr=pIVVUIClient->InsertMenuItem(UIMFG_DYNAMIC_APPLICATION,
    "MyAppName",
    200,
    FALSE,
    &NewMenuItem,
    &rc);
```

See Also

“AppendMenuItem” on page 570

“DeleteMenuItem” on page 574

RemoveApplicationByName

Removes a program from the list using its name.

Syntax

In Visual Basic:

```
Function RemoveApplicationByName(ApplicationName As String) As UIRC
```

In Visual C++ (MFC):

```
long RemoveApplicationByName(LPCTSTR ApplicationName);
```

In Visual C++ (Custom):

```
HRESULT RemoveApplicationByName(LPSTR lpzApplicationName, UIRC*  
pResult);
```

Parameters

ApplicationName

String / LPSTR. The executable name of the application and its extension, for example, CALC.EXE, or a fully qualified path plus the application.

Returns

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

The **UIClient** control enables you to interact with the **UIServer**; however, in order to get messages from the **UIServer**, the **UIServer** needs to know which programs it is interacting with. To specify a program by name, you can use the **AddApplicationByName** method. The **RemoveApplicationByName** function enables you to remove a program from the list using its name.

In Visual Basic the executable name is different at design time than it is at run time. It is recommended that you use the **RemoveApplicationByWindow** method instead for removing your program by name, and use the **RemoveApplicationByName** method when removing other programs.

This method will not work if you omit the “.EXE” extension from the name.

Example

In Visual Basic:

```
Dim rc As UIRC  
rc = VVUIClient1.RemoveApplicationByName("CALC.EXE")
```

In Visual C++ (MFC):

```
UIRC rc = (UIRC) m_VVUIClient1.RemoveApplicationByName("CALC.EXE");
```

In Visual C++ (Custom):

```
UIRC rc;  
HRESULT hr = pVVUIClient->RemoveApplicationByName("CALC.EXE", &rc);
```

See Also

“AddApplicationByName” on page 566

“AddApplicationByWindow” on page 568

“RemoveApplicationByWindow” on page 593

RemoveApplicationByWindow

Removes a program from the list using its window handle.

Syntax

In Visual Basic:

```
Function RemoveApplicationByWindow(hwndApplication As Long) As UIRC
```

In Visual C++ (MFC):

```
long RemoveApplicationByWindow(long hwndApplication);
```

In Visual C++ (Custom):

```
HRESULT RemoveApplicationByWindow(HWND_t hwndApplication, UIRC* pResult);
```

Parameters

hwndApplication

Long/ HWND_t. The window handle of the application you wish to remove from the list.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

The **UIClient** control enables you to interact with the **UIServer**; however, in order to get messages from the **UIServer**, the **UIServer** needs to know which programs it is interacting with. To specify a program by name, you can use the **AddApplicationByName** method. The **RemoveApplicationByWindow** function enables you to remove a program from the list using its window handle.

In Visual Basic the executable name is different at design time than it is at run time. It is recommended that you use the **RemoveApplicationByWindow** method instead for removing your program, and use the **RemoveApplicationByName** method when removing other programs.

Example

In Visual Basic:

```
Dim rc As UIRC  
rc = VVUIClient1.RemoveApplicationByWindow(VVUIClient1.Parent.hWnd)
```

In Visual C++ (MFC):

```
UIRC rc = (UIRC) m_VVUIClient1.RemoveApplicationByWindow((long)m_hWnd);
```

In Visual C++ (Custom):

```
UIRC rc;  
HRESULT hr = pVVUIClient->RemoveApplicationByWindow((HWND_t)hWnd,&rc);
```

See Also

“AddApplicationByName” on page 566

“AddApplicationByWindow” on page 568

“RemoveApplicationByName” on page 591

SetClientCallback (Custom Interface)

Turns on communication actions that occur on the **UIServer** to a C++ program.

Syntax

In Visual C++ (Custom):

```
HRESULT SetClientCallback(GUID* riid, IUnknown* pIClientEventSink, DWORD dwFlags, UIRC* pResult);
```

Parameters

riid

GUID. A pointer to the client's implementation of the event sink to be used when handling notifications from the **UIServer**.

pIClientEventSink

IUnknown. The Interface Id (IID) at the event interface implementation. At this time, the only valid value is IID_IVVUIEventSink.

dwFlags

DWORD. The dwFlags parameter specifies the messages the **UIClient** control receives from the **UIServer**. For more information about values, refer to “vvUIEventCallbackFlags” on page 543.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

This method is also known as **SetClientCallbackFlags** in the Visual Basic and C++ (MFC).

The **UIClient** control is able to communicate actions that occur on the **UIServer** to your C++ program. However, you must turn on this communication manually. To do so, use the **SetClientCallback** method.

When the user interacts with the **UIServer**, the **UIServer** notifies the **UIClient** control which in turn notifies your program by firing one of its events. However, each notification sent from the **UIServer** to the control affects the performance of your program. For that reason only call this method when you wish to receive messages from the **UIServer**.

Example

In Visual C++ (Custom):

```
UIRC rc;  
HRESULT hr = pIVVUIClient->SetClientCallback(IID_IVVUIEventSink,  
    &m_IVVUIClientEventSink,  
    UIEVENT_ALL,  
    &rc);
```

See Also

None.

SetClientCallbackFlags

Communicates actions that occur in the **UIServer** to a Visual Basic program.

Syntax

In Visual Basic:

```
Function SetClientCallbackFlags(dwFlags As Long) As UIRC
```

In Visual C++ (MFC):

```
long SetClientCallbackFlags(long dwFlags);
```

Parameters

dwFlags

Long / DWORD. The *dwFlags* parameter specifies the messages the **UIClient** control receives from the **UIServer**. For more information about values, refer to “vvUIEventCallbackFlags” on page 543.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

This method is also known as **SetClientCallback** in the Custom interface.

The **UIClient** control is able to communicate actions that occur in the **UIServer** to your Visual Basic program. However, you must turn on this communication manually. To do so, use the **SetClientCallbackFlags** method.

When the user interacts with the **UIServer**, the **UIServer** notifies the **UIClient** control which in turn notifies your program by firing one of its events. However, each notification sent from the **UIServer** to

the control affects the performance of your program. For that reason only call this method when you wish to receive messages from the **UIServer**.

Example

In Visual Basic:

```
Dim rc As UIRC  
rc = VVUIClient1.SetClientCallbackFlags(vvUIEVENT_ALL)
```

In Visual C++ (MFC):

```
UIRC rc = (UIRC)m_vvUIClient1.SetClientCallbackFlags(vvUIEVENT_ALL);
```

See Also

None.

SetLanguageByID

Specifies the language the **UIServer** will use for displaying its menu items.

Syntax

In Visual Basic:

```
Function SetLanguageByID(wLangID As Integer) As UIRC
```

In Visual C++ (MFC):

```
long SetLanguageByID(short wLangID);
```

In Visual C++ (Custom):

```
HRESULT SetLanguageByID(LANGID wLangID, UIRC* pResult);
```

Parameters

wLangID

Integer / WORD. The language ID according to Table 3, “LangID string names supported by UIServer and their corresponding values,” on page 509.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

Without calling this method the **UIServer** will not display a menu when the user clicks the ViaVoice menu.

Example

In Visual Basic:

```
Dim rc As UIRC  
rc = VVUIClient1.SetLanguageByID(1033) 'for US English
```

In Visual C++ (MFC):

```
UIRC rc =  
m_VVUIClient1.SetLanguageID(MAKELANGID(LANG_ENGLISH,SUBLANG_US)); //for  
US English.
```

In Visual C++ (Custom):

```
UIRC rc;  
HRESULT hr = pVVUIClient->  
SetLanguageID(MAKELANGID(LANG_ENGLISH,SUBLANG_US),&rc); // for US  
English
```

See Also

“MAKELANGID” in the Win32 SDK documentation.

“SetLanguageByString” on page 601

SetLanguageByString

Specifies the language the **UIServer** will use for displaying its menu items.

Syntax

In Visual Basic:

```
Function SetLanguageByString(LangStr As String) As UIRC
```

In Visual C++ (MFC):

```
long SetLanguageByString(LPCTSTR LangStr);
```

In Visual C++ (Custom):

```
HRESULT SetLanguageByString(LPSTR lpzLangStr, UIRC* pResult);
```

Parameters

LangStr

String / LPSTR. A 2- or 5-character string representing the language that the **UIServer** will use when displaying menu items according to Table 3 on page 509:

Returns

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

Without calling this method the **UIServer** will not display a menu when the user clicks the ViaVoice menu.

Example

In Visual Basic:

```
Dim rc As UIRC  
rc = VVUIClient1.SetLanguageByString("EN_US")
```

In Visual C++ (MFC):

```
UIRC rc = (UIRC) m_VVUIClient1.SetLanguageByString("EN_US");
```

In Visual C++ (Custom):

```
UIRC rc;  
HRESULT hr = pVVUIClient->SetLanguageByString("EN_US", &rc);
```

See Also

“SetLanguageByID” on page 599

SetMenuItemInfo

Sets the characteristics of a certain menu item.

Syntax

In Visual Basic:

```
Function SetMenuItemInfo(uUIMenuGroup As UIMENUGROUP, MenuName As String, uItem As Long, fByPosition As Long, pIMenuInfo As VVUIMenuInfo) As UIRC
```

In Visual C++ (MFC):

```
long SetMenuItemInfo(long uUIMenuGroup, LPCTSTR MenuName, long uItem, long fByPosition, LPDISPATCH pIMenuInfo);
```

In Visual C++ (Custom):

```
HRESULT SetMenuItemInfo(UIMENUGROUP uUIMenuGroup, LPSTR lpzMenuName, UINT uItem, BOOL fByPosition, UIMENUITEMINFO* lpuimiiItem, UIRC* pResult);
```

Parameters

uUIMenuGroup

UIMENUGROUP. This parameter specifies the location where you wish to add the menu item. Valid values are: One of the "UIMENUGROUP (Enum)" flags described in Chapter 27 "Classes, Structures, and Enumerations".

MenuName

String / LPCTSTR. Use this parameter to specify the menu group name.

uItem

Long / UINT. The menu item's ID number, or the menu item's position within the group. This parameter changes meanings depending on the value of the *fByPosition* parameter. When the *fByPosition* parameter is zero then the *uItem* parameter indicates the menu ID number. If the *fByPosition* parameter is set to one then the *uItem* parameter indicates the 1-based position from the top within the group of the menu item.

fByPosition

Long / BOOL. Set this parameter to zero to indicate the ultem parameter represents a menu ID number. Set it to one to indicate that the ultem parameter represents a menu item's position.

pIMenuInfo

VVUIMenuInfo / UIMENUITEMINFO. A menu information class in Visual Basic and Visual C++ (MFC) or a structure in Visual C++ (Custom). For more information refer to “vvUIMenuInfo (Class - Visual Basic and MFC Only)” on page 534 and “User Interface Control Structures” on page 535, which contains the characteristics of the menu item.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

You must add the menu item with the **AppendMenuItem** or **InsertMenuItem** prior to using this method. If you attempt to modify a menu item that does not exist the method will return an error code of UIAPIRC_ERROR_FAILED(4).

Example

In Visual Basic:

```
Dim rc As UIRC
Dim MenuInfo As New VVUIMenuInfo
MenuInfo.Checked = True
rc = VVUIClient1.SetMenuItemInfo(UIMFG_DYNAMIC_MAIN,
    App.EXEName,
    100,
    False,
    oNewMenuInfo)
```

In Visual C++ (MFC)

```
CVVUIMenuInfo MenuInfo;
MenuInfo.CreateDispatch(CLSID_VVUIMenuInfo);
MenuInfo.SetChecked(TRUE);
UIRC rc = (UIRC) m_VVUIClient1.SetMenuItemInfo(UIMFG_DYNAMIC_MAIN,
    AfxGetAppName(),
    100,
    FALSE,
    MenuInfo);
```

In Visual C++ (Custom):

```
UIRC rc;
UIMENUITEMINFO MenuInfo;

ZeroMemory(&NewMenuItem, sizeof(NewMenuItem));
HRESULT hr = pVVUIClient->GetMenuItemInfo(UIMFG_DYNAMIC_MAIN,
    "MyAppName",
    100,
    FALSE,
    &MenuInfo,
    &rc);

MenuInfo.m_uState = MFS_CHECKED | MFS_ENABLED;
HRESULT hr = pVVUIClient->SetMenuItemInfo(UIMFG_DYNAMIC_MAIN,
    "MyAppName",
    100,
    FALSE,
    &MenuInfo,
    &rc);
```

See Also

“GetMenuItemInfo” on page 577

SetNumberValue

Modifies the state of a certain component in the **UIServer**.

Syntax

In Visual Basic:

```
Function SetNumberValue(ciComponent As TCID, dwValueData As Long, [nIndex  
As Long]) As UIRC
```

In Visual C++ (MFC):

```
long SetNumberValue(long ciComponent, long dwValueData, long nIndex);
```

In Visual C++ (Custom):

```
HRESULT SetNumberValue(TCID ciComponent, DWORD dwValueData, int nIndex,  
UIRC* pResult);
```

Parameters

ciComponent

TCID. One of the UIServer's object IDs. You will find a complete list of objects in the Classes, Structures, Enumerations part of this reference section.

dwValueData

Long / DWORD. The *pdwValueData* parameter will contain the numeric value that the control will use to set the object's characteristic.

nIndex

Long (Optional) / int. This value enables you to set extended information about the object. Its meaning changes depending on which object you request information from. See the Remarks section below for more information.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

This method enables you to change characteristics represented with numeric values, such as the volume level in the volume meter component (COMPID_VOLUME). In Visual Basic, the dwValueData parameter must be a long value. You should convert all your values to be long with the CLng() function prior to issuing this method.

Example

In Visual Basic:

```
Dim lSetValue As Long
Dim lRetVal As UIRC

lSetValue = CLng(75)
lRetVal = VVUIClient1.SetNumberValue(COMPID_VOLUME, lSetValue)
```

In Visual C++ (MFC):

```
UIRC rc = (UIRC) m_VVUIClient1.SetNumberValue(COMPID_VOLUME,
75,
UIVOLINDEX_VOLLEVEL);
```

In Visual C++ (Custom):

```
UIRC rc;
HRESULT hr = pVVUIClient->SetNumberValue(COMPID_VOLUME,
75,
UIVOLINDEX_VOLLEVEL,
&rc);
```

See Also

“GetNumberValue” on page 580

“GetStringValue” on page 582

“SetNumberValue” on page 606

SetStringValue

Changes characteristics represented with string values, such as the text in the word history object (COMPID_WORDHISTORY).

Syntax

In Visual Basic:

```
Function SetStringValue(ciComponent As TCID, ValueData As String, [nIndex  
As Long]) As UIRC
```

In Visual C++ (MFC):

```
long SetStringValue(long ciComponent, LPCTSTR ValueData, long nIndex);
```

In Visual C++ (Custom):

```
HRESULT SetStringValue(TCID ciComponent, LPSTR lpzValueData, int  
nIndex, UIRC* pResult);
```

Parameters

ciComponent

TCID. One of the UI Server's object IDs. You will find a complete list of objects in the Classes, Structures, Enumerations part of this reference section.

ValueData

String / LPSTR. The ValueData parameter will contain the numeric value that the control will use to set the object's characteristic.

nIndex

Long (Optional) / int. This value enables you to set extended information about the object. Its meaning changes depending on which object you request information from. For more information, refer to "Component Index Constants" on page 538.

Return Values

UIRC

For more information, refer to “UIRC (Enum)” on page 559.

Remarks

None.

Example

In Visual Basic:

```
Dim lRetVal As UIRC
lRetVal = VVUIClient1.SetStringValue(COMPID_WORDHISTORY, _
    "Say 'Stop Dictation' to stop.")
```

In Visual C++ (MFC):

```
UIRC rc = (UIRC) m_VVUIClient1.SetStringValue(COMPID_WORDHISTORY,
    "Say 'Stop Dictation' to stop.",
    UIWHINDEX_TAGGEDTEXT);
```

In Visual C++ (Custom):

```
UIRC rc;
HRESULT hr = pVVUIClient->SetStringValue(COMPID_WORDHISTORY,
    "Say 'Stop Dictation' to stop.",
    UIWHINDEX_TAGGEDTEXT,
    &rc);
```

See Also

“GetNumberValue” on page 580

“GetStringValue” on page 582

“SetNumberValue” on page 606

User Interface Control Events

The **UIClient** control triggers events when the user interacts with the **UIServer**, but only if you specify that you wish the control to receive messages from the **UIServer** by issuing the **SetClientCallbackFlags** or **SetClientCallback** methods.

All of the events in the control have a **pResult** parameter, which enables you to inform the control of how you handled the event. This helps the control determine if it should communicate the event to other clients or not. The User Interface Client control supports the following events:

- **EventActiveApplication**
- **EventButtonPressed**
- **EventComponentUpdated**
- **EventMenuItemSelected**
- **EventQueryViewFlags**
- **EventQueryViewMenuInfo**

EventActiveApplication

Event fired by the **UIClient** control whenever the **UIServer** detects a change in registered application activation.

Syntax

In Visual Basic:

```
EventActiveApplication(bActive As Boolean, hwndApplication As Long,  
ApplicationName As String, pResult As UIEVENTRC)
```

In Visual C++ (MFC):

```
void OnEventActiveApplication(BOOL bActive, long hwndApplication, SPCSTR  
lpstrApplicationName, long FAR* pResult);
```

In Visual C++ (Custom):

```
HRESULT EventActiveApplication(BOOL bActive, HWND_t hwndApplication,  
LPSTR lpstrApplicationName, UIEVENTRC* pResult);
```

Parameters

bActive

Boolean /BOOL. Flags indicating if the application has the focus (Active = True) or doesn't (Active = False).

hwndApplication

Long / HWND_t. The window handle of the application window that has gained focus or has lost focus.

ApplicationName

String / LPSTR. The executable name of the application window that has gained or lost focus.

pResult

UIEVENTRC. The result code you wish to pass back to the control. The control uses this to determine if the message should be passed on to other controls.

Return Values

TRUE

Sets the state of the components.

FALSE

Saves the state.

Remarks

The control fires this event when one of the applications, added through the **AddApplicationByWindow** or **AddApplicationByName** method, either gains or loses the **UIServer** activation. Please note that since the **UIServer** has to detect the change in order to notify the client, certain quick lose/gain/lose focus change scenarios might go unnoticed by the **UIServer**. This event is useful for saving/restoring **UIServer** component states (i.e. Microphone state, custom buttons, etc.).

Since there may be other speech-enabled applications using the same **UIServer**, this event enables you to reset the state of the various components in the **UIServer** whenever your application receives the focus, and to save the state before it is changed by another application. Set the state of the components when bActive=True and save the state when bActive=False.

Example

In Visual Basic:

```
EventActiveApplication(bActive As Boolean, hwndApplication As Long,
pResult As UIEVENTRC)
    If bActive = True Then
        lRetVal = VVUIClient1.SetNumberValue(COMPID_MICROPHONE, vvUIMSF_ON)
    End If
    pResult = UIEVENTRC_PROCESSED
End Sub
```

In Visual C++ (MFC):

```
void CCVUICtrDlg::OnEventActiveApplication(BOOL bActive, long
hwndApplication, LPCTSTR pszApplicationName, long FAR* pResult)
{
    if (bActive)
    {
        UIRC rc = (UIRC) m_vvUIClient1.SetNumberValue(COMPID_MICROPHONE,
            UIMSF_ON,
            UIMICINDEX_MICSTATE);
    }
    *pResult = UIEVENTRC_PROCESSED;
}
```

In Visual C++ (Custom):

```
STDMETHODIMP CVVUIEvents::EventActiveApplication (
    BOOL bActive,
    HWND_t hwndApplication,
    LPSTR lpszApplicationName,
    UIEVENTRC * pResult )
{
    if (bActive)
    {
        UIRC rc;
        HRESULT hr = pvvUIClient->SetNumberValue(COMPID_MICROPHONE,
            UIMSF_ON,
            UIMICINDEX_MICSTATE,
            &rc);
    }
    *pResult = UIEVENTRC_PROCESSED;
    return S_OK;
}
```

See Also

None.

EventButtonPressed

Event fired by the **UIClient** control whenever the user clicks one of the buttons in the **UIServer**.

Syntax

In Visual Basic:

```
EventButtonPressed(ComponentID As TCID, hwndTarget As Long, pResult As UIEVENTRC)
```

In Visual C++ (MFC):

```
void OnEventButtonPressed (long ComponentID, long hwndTarget, long FAR* pResult)
```

In Visual C++ (Custom):

```
HRESULT EventButtonPressed(TCID ciComponentID, HWND_t hwndTarget, UIEVENTRC* pResult);
```

Parameters

ciComponent

TCID. One of the UIServer's component IDs (COMPID_*). You will find a complete list of objects in the Classes, Structures, Enumerations part of this reference section.

hwndTarget

Long / HWND_t. The handle of the window that had the focus before the user clicked the button in the **UIServer**.

pResult

UIEVENTRC. The result code you wish to pass back to the control. The control uses this to determine if the message should be passed on to other controls.

Return Values

None.

Remarks

None.

Example

None.

See Also

None.

EventComponentUpdated

Event fired by the **UIClient** control whenever a client changes the characteristics of one of the components in the **UIServer**.

Syntax

In Visual Basic:

```
EventComponentUpdated(ciComponentID As TCID, pResult As UIEVENTRC)
```

In Visual C++ (MFC):

```
void OnEventComponentUpdated (long ciComponentID, long FAR* pResult)
```

In Visual C++ (Custom):

```
HRESULT EventComponentUpdated(TCID ciComponentID, UIEVENTRC* pResult);
```

Parameters

ciComponent

TCID. One of the UIServer's component IDs (COMPID_*). You will find a complete list of objects in Chapter 20, "Classes, Structures, and Enumerations" on page 533.

pResult

UIEVENTRC. The result code you wish to pass back to the control. The control uses this to determine if the message should be passed on to other controls.

Return Values

None.

Remarks

This can be done with the **SetNumberValue**, or **SetStringValue** APIs.

Example

None.

See Also

None.

EventMenuItemSelected

Event fired by the **UIClient** control event whenever the user selects one of the following menu items in the **UIServer**:

VIAVOICEUI_IDMENU_MICROPHONE
 VIAVOICEUI_IDMENU_BEGIN_DICTATION
 VIAVOICEUI_IDMENU_STOP_DICTATION
 VIAVOICEUI_IDMENU_BEGIN_READING
 VIAVOICEUI_IDMENU_STOP_READING
 VIAVOICEUI_IDMENU_SHOW_CORRECTION_WINDOW
 VIAVOICEUI_IDMENU_WHATCANISAY
 VIAVOICEUI_IDMENU_EXIT

or application-defined dynamic menu item ID between 0 and 9,999.

Syntax

In Visual Basic:

```
EventMenuItemSelected(dwMenuItemId As Long, pResult As UIEVENTRC)
```

In Visual C++ (MFC):

```
void OnEventMenuItemSelected (long dwMenuItemId, long hwndTarget, long  
FAR* pResult);
```

In Visual C++ (Custom):

```
HRESULT EventMenuItemSelected(DWORD dwMenuItemId, HWND_t hwndTarget,  
UIEVENTRC* pResult);
```

Parameters

dwMenuItemId

Long / DWORD. The ID number of the menu item. This value depends on whether the menu item is a built-in menu item (see VIAVOICEUI_IDMENU*) or an application-defined dynamic menu

item ID (between 0-9,999). For more information, refer to “AppendMenuItem” on page 570 and “InsertMenuItem” on page 587.

hwndTarget

Long / HWND_t. The handle of the window that had the focus before the user clicked the button in the **UIServer**.

pResult

UIEVENTRC. The result code you wish to pass back to the control. The control uses this to determine if the message should be passed on to other controls.

Return Values

None.

Remarks

None.

Example

None.

See Also

“AppendMenuItem” on page 570

“InsertMenuItem” on page 587

EventQueryViewFlags

Event fired by the **UIClient** control whenever the **UIServer** needs specific view information that can be used with current appearance.

Syntax

In Visual Basic:

```
EventQueryViewFlags(phwndWindow As Long, pdwDockFlags As Long, pResult As UIEVENTRC)
```

In Visual C++ (MFC):

```
void OnEventQueryViewFlags(long FAR* phwndWindow, long FAR*  
pdwDockFlags, long FAR* pResult);
```

In Visual C++ (Custom):

```
HRESULT EventQueryViewFlags(HWND_t* phwndWindow, DWORD* pdwDockFlags,  
UIEVENTRC* pResult);
```

Parameters

phwndWindow

Long / HWND_t. The window that currently has focus. This parameter is changeable if you want the **UIServer** to use a different window.

pdwDockFlags

Long / DWORD. A combined (OR'ed) value from the **VVUIDockingAlgorithmConstants** and **VVUIDockingStyleConstants**. You must set this value to DVAF_ALLOW_TOPMOST_DOCK (or DVAF_ALLOW_DOCK) if you wish to allow the **UIServer** to dock to one of the clients.

pResult

UIEVENTRC. The result code you wish to pass back to the control. The control uses this to determine if the message should be passed on to other controls.

Return Values

None.

Remarks

None.

Example

In Visual Basic:

```
Private Sub m_VVUIClient_EventQueryViewFlags(phwndWindow As Long,
pdwDockFlags As Long, pResult As VVUICtrlCtl.UIEVENTRC)

    pdwDockFlags = vvDVAF_ALLOW_TOPMOST_DOCK
    pResult = UIEVENTRC_PROCESSED

End Sub
```

In Visual C++ (MFC):

```
void CVVUICtrlDlg::OnEventQueryViewFlags(long FAR* phwndWindow, long
FAR* pdwDockFlags, long FAR* pResult)
{
    *pdwDockFlags = vvDVAF_ALLOW_TOPMOST_DOCK;
    *pResult = UIEVENTRC_PROCESSED;
}
```

In Visual C++ (Custom):

```
STDMETHODIMP CVUIClientEvents::EventQueryViewFlags (
    HWND_t * phwndWindow,
    DWORD * pdwDockFlags,
    UIEVENTRC *pResult)
{
    *pdwDockFlags = DVAF_ALLOW_TOPMOST_DOCK;
    *pResult = UIEVENTRC_PROCESSED;
    return S_OK;
}
```

See Also

“AppendMenuItem” on page 570

“InsertMenuItem” on page 587

EventQueryViewMenuInfo

Event fired by the **UIClient** control whenever the user requests to view the **UIServer**'s menu, just before it actually displays the menus.

Syntax

In Visual Basic:

```
EventQueryViewMenuInfo(vtViewType As TVIEWTYPE, hwndWindow As Long,  
ApplicationTitle As String, MainMenuName As String, ApplicationMenuName  
As String, HelpMenuName As String, pResult As UIEVENTRC)
```

In Visual C++ (MFC):

```
void OnEventQueryViewMenuInfo(long vtViewType, long hwndWindow, BSTR  
FAR* ApplicationTitle, BSTR FAR* MainMenuName, BSTR FAR*  
ApplicationMenuName, BSTR FAR* HelpMenuName, long FAR* pResult);
```

In Visual C++ (Custom):

```
HRESULT EventQueryViewMenuInfo(TVIEWTYPE vtViewType, HWND_t hwndWindow,  
LPSTR lpszApplicationTitle, LPSTR lpszMainMenuName, LPSTR  
lpszApplicationMenuName, LPSTR lpszHelpMenuName, UIEVENTRC* pResult);
```

Parameters

vtViewType

TVIEWTYPE. The current view state of the **UIServer**.

hwndWindow

Long / HWND_t. The last window to have the focus before the user requested to view the menus.

ApplicationTitle

String / LPSTR. This parameter lets you specify the menu caption for a dynamic application menu group.

MainMenuName

String / LPSTR. This parameter lets you specify the group name you wish to display for a dynamic main menu

ApplicationMenuName

String / LPSTR. This parameter lets you specify the menu group you wish to use when displaying dynamic application menus.

HelpMenuName

String / LPSTR. This parameter lets you specify the menu group you wish to use when displaying dynamic help menus.

pResult

UIEVENTRC. The result code you wish to pass back to the control. The control uses this to determine if the message should be passed on to other controls.

Return Values

None.

Remarks

The user does this by clicking the ViaVoice menu button when the **UIServer** is in Taskbar or Docked views, or right-click the system tray icon.

It is not necessary to set the value of all the “menu name” parameters, only the ones that your application uses. In other words, you do not have to set the HelpMenuName parameter if your application does not use custom help menu items.

Example

In Visual Basic:

```
Private Sub m_VVUIClient_EventQueryViewMenuInfo(ByVal vtViewType As
VVUICtrlCtl.TVIEWTYPE, ByVal hwndWindow As Long, ApplicationTitle As
String, MainMenuName As String, ApplicationMenuName As String,
HelpMenuName As String, pResult As VVUICtrlCtl.UIEVENTRC)
    Dim sAppName As String
    sAppName = App.EXENAME
    ApplicationTitle = sAppName
    MainMenuName = sAppName
    ApplicationMenuName = sAppName
    HelpMenuName = sAppName
    pResult = UIEVENTRC_PROCESSED
End Sub
```

In Visual C++ (MFC):

```
void CVVUICtrlDlg::OnEventQueryViewMenuInfo(long vtViewType, long
hwndWindow, BSTR FAR* ApplicationTitle, BSTR FAR* MainMenuName, BSTR
FAR* ApplicationMenuName, BSTR FAR* HelpMenuName, long FAR* pResult)
{
    CString sAppName = AfxGetAppName();
    *Application Title = sAppName.AllocSysString(),
    *MainMenuName = sAppName.AllocSysString();
    *ApplicationMenuName = sAppName.AllocSysString();
    *HelpMenuName = sAppName.AllocSysString();
    *pResult = UIEVENTRC_PROCESSED;
}
```

In Visual C++ (Custom):

```

STDMETHODIMP CVVUIClientEvents::EventQueryViewMenuInfo(
    TVIEWTYPE vtViewType,
    HWND_t * hwndWindow,
    LPSTR lpszApplicationTitle,
    LPSTR lpszMainMenuName,
    LPSTR lpszApplicationMenuName,
    LPSTR lpszHelpMenuName,
    UIEVENTRC *pResult)
{
    char sAppName[256];
    lstrcpy(sAppName, "MyAppName");
    lstrcpy(lpszApplicationTitle, sAppName);
    lstrcpy(lpszMainMenuName, sAppName);
    lstrcpy(lpszApplicationMenuName, sAppName);
    lstrcpy(lpszHelpMenuName, sAppName);
    *pResult = UIEVENTRC_PROCESSED;
    return S_OK;
}

```

See Also

[“AddApplicationByName” on page 566](#)
[“AddApplicationByWindow” on page 568](#)
[“AppendMenuItem” on page 570](#)
[“InsertMenuItem” on page 587](#)
[“SetClientCallback \(Custom Interface\)” on page 595](#)
[“SetClientCallbackFlags” on page 597](#)
[“TVIEWTYPE \(Enum\)” on page 555.](#)

User Interface Control Frequently Asked Questions

This chapter contains answers to the most frequently asked questions about the ViaVoice **User Interface Control**.

How can I make the ViaVoice UIServer appear in docked view when my application first becomes visible?

You can't. Your program cannot set the view appearance of the **UIServer**; only the user can change the view by selecting the **Appearance** option from the ViaVoice menu. Your application, however, can allow or deny the user's request to dock the **UIServer** to your specific application. See "Programming the ViaVoice User Interface" on page 513 of this manual for an example.

Why is the microphone button in the ViaVoice UIServer disabled?

The Microphone component is disabled as default. You can change the state of the microphone component using the **SetNumberValue** method. See "Getting and Setting User Interface Characteristics" on page 520 for details.

Why is the UIServer not displaying a menu when I click the ViaVoice button?

The most likely explanation is that you have not set the language of the **UIServer**. To set the language use the **SetLanguageByString** or **SetLanguageByID** method. For more information, refer to "Initializing the UIClient" on page 506.

Why is the UIClient control not firing events when an action occurs in the ViaVoice UIServer?

You need to turn on event messaging with the **SetClientCallbackFlags** or the **SetClientCallback (Custom Interface)** method. Refer to "Programming the ViaVoice User Interface" on page 513 for an example.

Why is the UIServer NOT docking to my application windows?

For the **UIServer** to dock to your application windows, you must:

1. Enable event firing in the **UIClient** control by calling **SetClientCallbackFlags** in Visual Basic or Visual C++ (MFC) applications, or **SetClientCallback** in Visual C++ applications using the custom interface.
2. Add your application to the UIServer's list of supported applications by calling **AddApplicationByName** or **AddApplicationByWindow**.

3. Write code to handle the **EventQueryViewFlags** event. Make sure to set the parameter **pdwDockFlags** to **DVAF_ALLOW_TOPMOST_DOCK** or **DVAF_ALLOW_DOCK**. Also, make sure to set the parameter **pResult** to **UIEVENTRC_PROCESSED** to tell the control that you have processed the event.

Introduction to the DictationMgr Control

The ViaVoice **Dictation Manager** Control (**VVDictationMgr**) is a moderately high level control which provides much of the functionality a client needs to add dictation to an application. However, in order to use this control, clients must be able to synchronize the **VVDictationMgr** with their application user interface through zero (0) based character indices. **VVDictationMgr** is a full ActiveX Control, which means that it can be "dropped" onto a form and configured at "design-time" in most high-level language environments. Using **VVDictationMgr** allows the user to manage both typed and dictated text, get wave data for playback of dictated text, and perform correction of dictated text.

An application using **VVDictationMgr** will receive events when speech has been recognized including the text of the recognition, the character index where the text should be placed, the character index where the cursor should be after the replacement, and (potentially) the length of text to be replaced. All this is necessary because **VVDictationMgr** keeps up with bookmarks and handles advanced formatting features when running on the IBM engine. See the **VVDictation** documentation, starting on page 699, for more information on bookmarks and advanced formatting features.

Getting Started with the DictationMgr Control

The following is a tutorial on how to incorporate the **VVDictationMgr** control into your Visual Basic or Visual C++ applications. This tutorial is designed to present you with the most commonly used properties and events in the **VVDictationMgr** control.

The following sections contain information to help you write code to create an instance of the **DictationMgr** Control, and capture speech.

Creating an Instance of the Control

This section contains step-by-step instructions for using Visual Basic or Visual C++ (MFC) to create an instance of the control.

In Visual Basic:

To add the **VVDictationMgr** control to your application, do the following:

1. From the **Project** menu, choose **Components**.

The Components dialog box, Figure 46, appears. The Components dialog lists all the ActiveX Controls that you can use in your application.

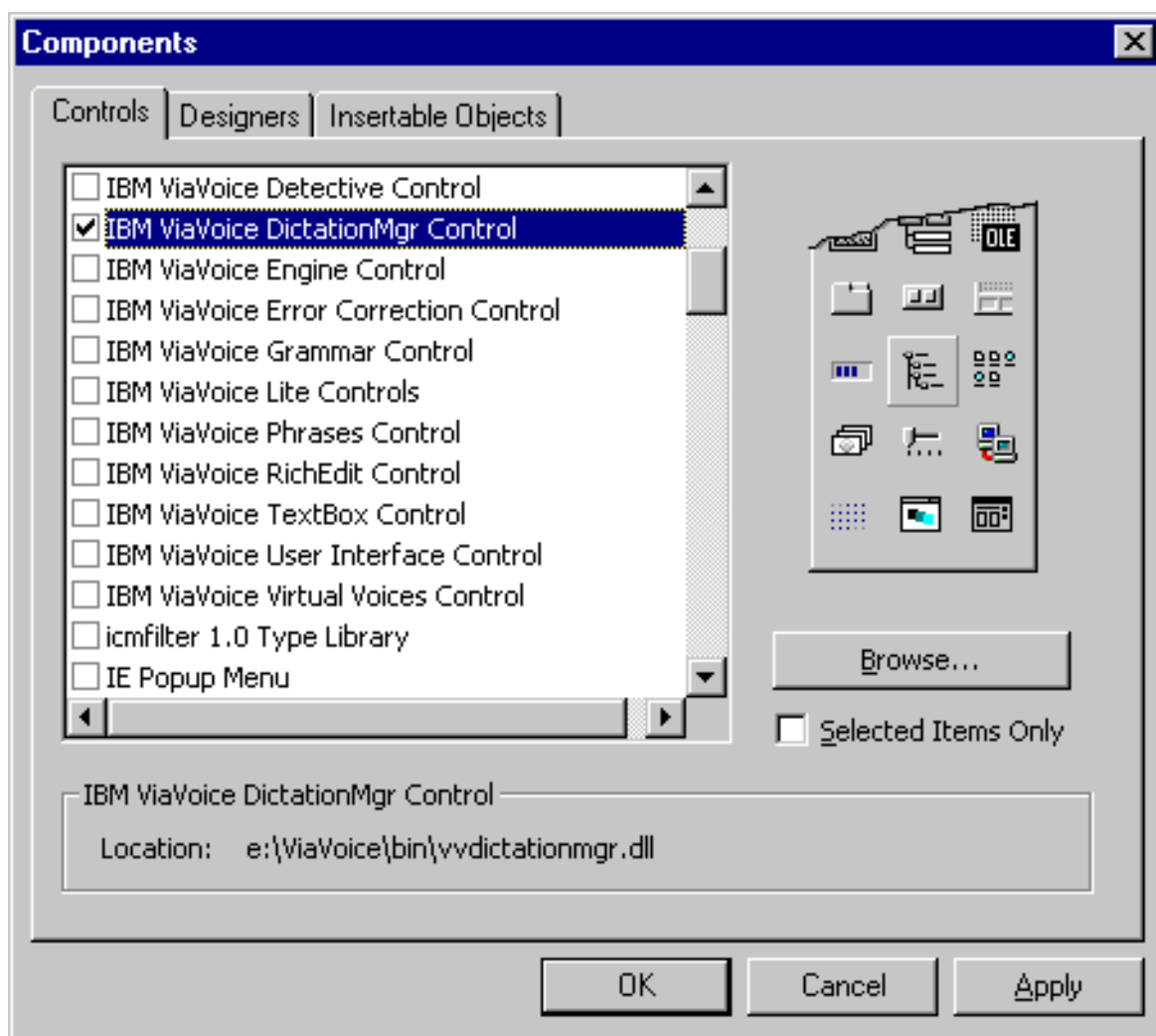


Figure 46. Component Selection Dialog - Visual Basic

- 2. Select IBM ViaVoice DictationMgr Control from the list and click OK.**

Visual Basic adds the control to your project, and adds a new icon to the toolbar (Figure 47).



Figure 47. VVDictationMgr Control Toolbar Icon

3. Add an instance of the **VVDictationMgr** control to your form.

The **VVDictationMgr** control is invisible at run-time and uses an icon at design-time much like the Visual Basic native **Timer** control.

It is also helpful to understand the way Visual Basic will respond to error codes returned from ActiveX control methods. When an error code is returned, Visual Basic will convert it into a "trappable" error which can be handled with the "On Error" syntax. If you do not use "On Error" when invoking ActiveX control methods that happen to return an error code, such as E_INVALIDARG, E_FAIL, E_OUTOFMEMORY, and others, then your application will exit with the message about run time errors. This is probably not something you want your users to see. Since these errors often happen at the most inconvenient time (i.e., during demos to your boss), it is strongly advised that you provide "On Error" handling when making any calls to ActiveX control methods.

In Visual C++ (MFC):

To add the **VVDictationMgr** to your MFC project, do the following:

1. From the **Project** menu, select **Add To Project**, then select **Components and Controls**.
The 'Components and Controls Gallery' dialog box, Figure 48, appears.

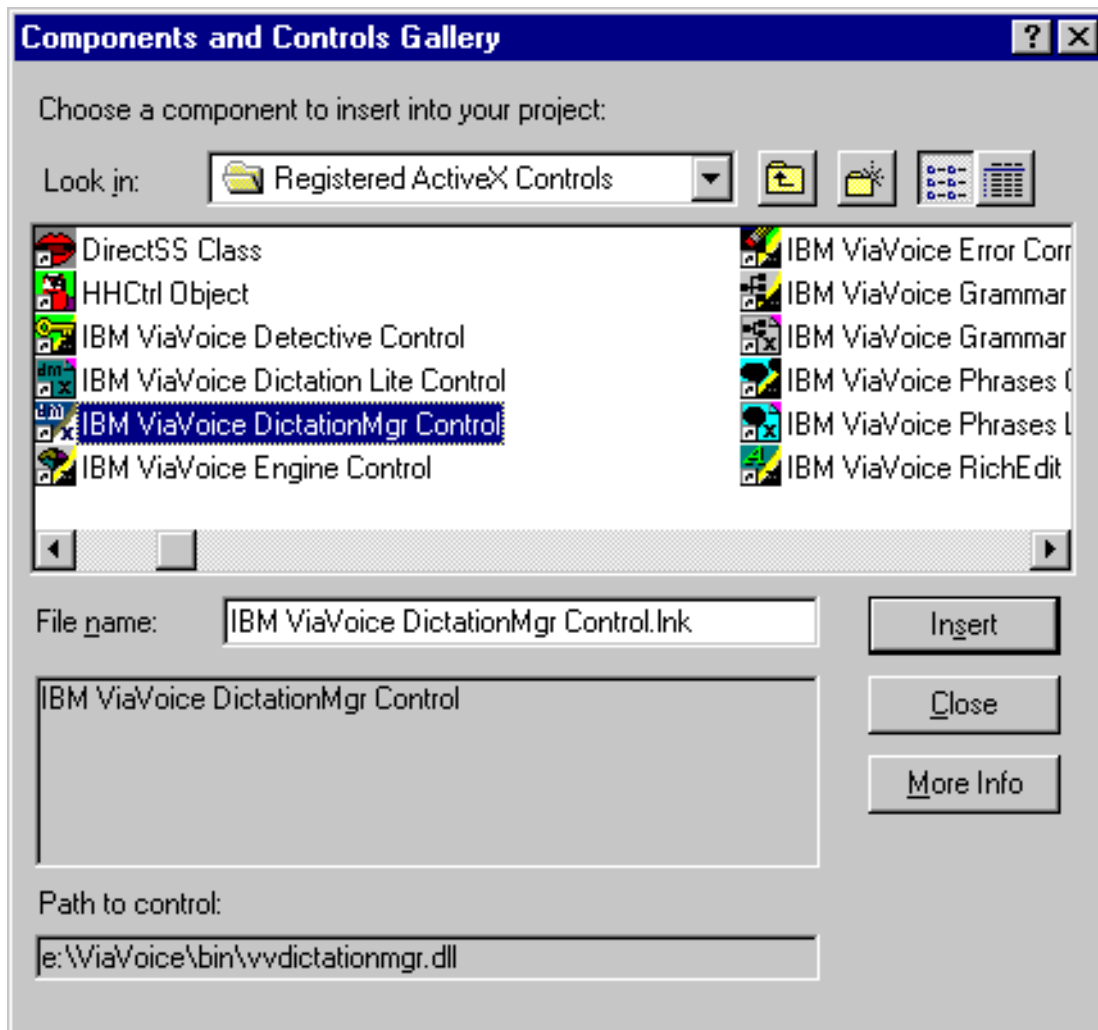


Figure 48. Insert ActiveX Control Dialog Box - Visual C++

2. Double-click the 'Registered ActiveX Controls' folder in the dialog box.
3. Select the **IBM ViaVoice DictationMgr Class** icon in the list of controls, then click **Insert**.
A confirmation message box appears, asking "Insert this component?"
4. Respond to the confirmation message box by clicking **OK**.
The 'Confirm Classes' dialog box, Figure 49, appears listing the Dual interface of the Dictation Manager control (CVVDictationMgr) and the Engine (CVVEngine) interface.

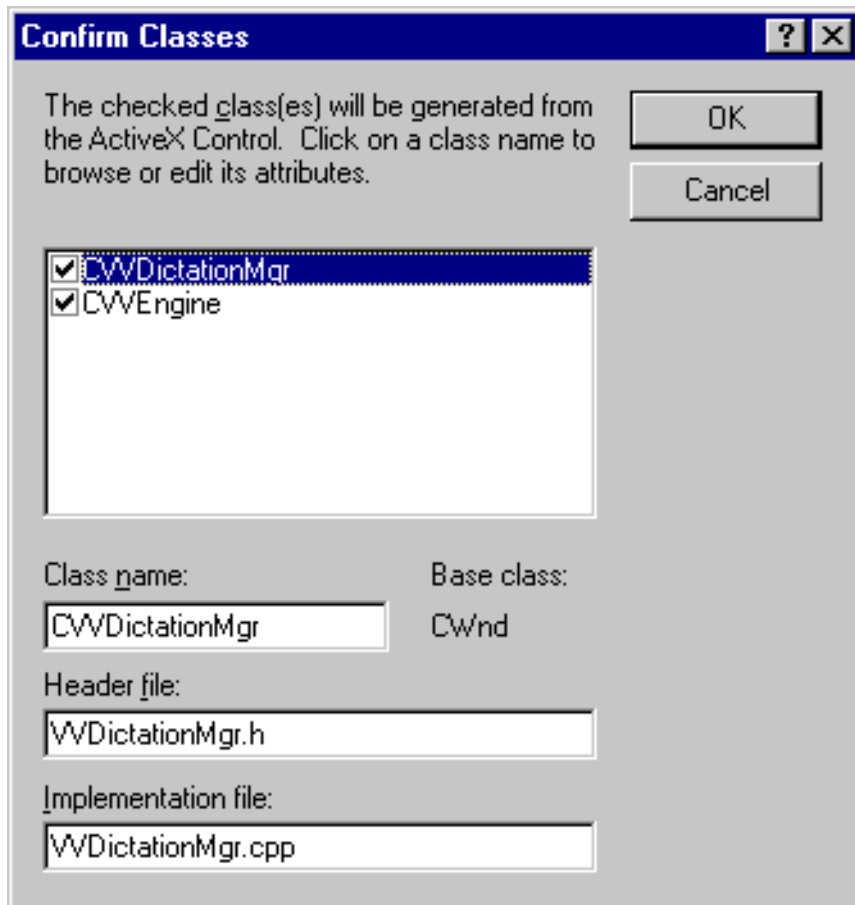


Figure 49. Confirm Classes Dialog Box

5. Click **OK** in the 'Confirm Classes' dialog box.

6. Close the 'Components and Controls Gallery' dialog box.

If you examine the Project Workspace window in the class view, you will notice four classes:

CVVDictationMgr and **CVVEngine** (assuming you accepted the default names for the class in the 'Confirm Classes' dialog box).

7. In the resource view of your Project Workspace window, double-click the dialog resource entry where you wish to insert the **VVDictationMgr** control.

The **VVDictationMgr** icon, Figure 50, appears in the Controls toolbar.



Figure 50. VVDictationMgr Icon in the Controls Toolbar

8. Add an instance of the **VVDictationMgr** control to the dialog box.

After you add the **VVDictationMgr** control to your dialog you can invoke Class Wizard to create a member variable for your class of type **CVVDictationMgr**. You might also decide to capture the events in the control by adding Event handlers to your dialog class. To add Event handlers, you can use the Class Wizard just like adding notification message handlers for a non-speech controls.

You should also understand the way MFC wrappers respond to error codes returned from ActiveX control methods. When an error code is returned, the MFC wrapper class will convert the error HRESULT into a COleException or COleDispatchException and "throw" it. When an error code such as E_INVALIDARG, E_FAIL, or E_OUTOFMEMORY is returned, your user may see an MFC error message. Or, your application may even exit with a message about "unhandled exceptions", depending on where the failing call occurs. Probably not something you want your users to see. Since these errors often happen at the most inconvenient time (i.e., during demos to your boss for instance), it is strongly advised that you wrap all calls to MFC wrapped ActiveX controls with exception handling.

Capturing Speech

The **VVDictationMgr** object converts speech input into text using an internal **VVDictation** object (see **VVDictation** documentation, starting on page 699, for more information). This text is formatted and supplied to the client in the **PutText** event. This event will provide the text to display, the location where the text should be placed, the cursor index after replacement occurs, and (potentially) the length of text to be replaced at the insertion point. Handling this event is as simple as applying the presented information to the client UI.

In order for **DictationMgr** to stay synchronized with the client UI it is the client's responsibility to inform **DictationMgr** of any changes to the UI state such as:

- The cursor is moved.
- A selection is made.
- Characters are inserted or removed (typed, pasted, or deleted)

By keeping this information updated as necessary, **DictationMgr** will be able to properly update the client when speech is recognized. This includes support for many complex operations such as, setting bookmarks to synchronize relatively real-time UI with latent speech recognition, updating engine context as needed to improve recognition accuracy, and (when running on the IBM engine) multi-phrase macro resolution including advanced numeric and date formatting. **DictationMgr** also provides easy access to information necessary to provide other high-level operations such as speech playback and correction.

Summary

At this point, you should know how to do the following:

- How to incorporate the **VVDictationMgr** control into your project.
- How to receive speech input.

The remainder of this documentation contains a reference for all the properties, methods, and events in the **VVDictationMgr** control.

Dictation Manager Control Properties

The ViaVoice **DictationMgr** control supports the following properties:

- **AutoDictationWindow**
- **CursorIndex**
- **DictationOn**
- **Engine**
- **ExpandMacros**
- **Locked**
- **ProcessingMacro**
- **UppercaseOn**

AutoDictationWindow (Run Time Only)

Controls the scope in which dictation is available.

Syntax

In Visual Basic:

```
Property AutoDictationWindow As Long
```

In Visual C++ (MFC):

```
long GetAutoDictationWindow();  
void SetAutoDictationWindow(long nNewValue);
```

In Visual C++:

```
HRESULT get_AutoDictationWindow(long * pVal);  
HRESULT put_AutoDictationWindow(long newVal);
```

Parameters

nNewValue
??

Return Values

Any valid “top-most^a” window handle.

Dictation is available only when the indicated window is “active” as indicated by it, or one of its children, having the focus. **Note:** There can only be one dictation object active for the same window (**DictationOn** is True) at any one time.

a. A “top-most” window is defined as any window without a parent. For more information on issues dealing with finding the “top-most” window, see the Microsoft Knowledge Base article Q84190.

NULL

Dictation mode is always available and must be controlled manually by setting the **DictationOn** property to True or False. **Note:** There can only be one global dictation object active (**DictationOn** is True) at any one time!

Remarks

The default value of this property is NULL (0), which will enable dictation globally. *However, please note that there can only be one global dictation object active (DictationOn is True) at any one time in the entire system (including other applications)!* For this reason, it is strongly suggested that you avoid global dictation objects if at all possible. Alternatively, you can set this property to any valid "top-most^a" window handle, which maps dictation availability to that window's activation state (it or one of its children having focus).

Remember, if you use NULL be aware that there can only be one global dictation object active (**DictationOn** is True) at any one time. This includes your own or any other application running on the system. For this reason, global dictation objects should be used with extreme care and should be avoided, unless absolutely necessary. Regardless of the value of this property, finer granularity of control can always be achieved by changing the state of **DictationOn** appropriately.

Example

In Visual Basic:

```
'Assumes this form is the top-most form!  
VVDict.AutoDictationWindow = hWndd
```

a.

In Visual C++ (MFC):

```
// Makes no assumptions about m_hWnd
HWND Hwnd = m_hWnd;
// Due to the Win32 implementation of GetParent, this is necessary
// to find the "Foreground" window for SAPI grammar activation
// For more information see MS Knowledge Base article Q84190
while ( ::GetParent ( hwnd ) != NULL &&
        ! ( ::GetWindowLong( hwnd ,GWL_STYLE ) & WS_POPUP ) )
{
    hwnd = ::GetParent ( hwnd );
}
m_VDDictationMgr.SetAutoDictationWindow ( (long)hwnd );
```

In Visual C++:

```
// Makes no assumptions about m_hWnd
HWND Hwnd = m_hWnd;
// Due to the Win32 implementation of GetParent, this is necessary
// to find the "Foreground" window for SAPI grammar activation
while ( ::GetParent ( hwnd ) != NULL &&
        ! ( ::GetWindowLong( hwnd ,GWL_STYLE ) & WS_POPUP ) )
{
    hwnd = ::GetParent ( hwnd );
}
HRESULT hr = S_OK;
hr = m_pIVVDictationMgr->put_AutoDictationWindow ( (long)hwnd );
```

See Also

“DictationOn” on page 647

“DictationStateChange” on page 685

CursorIndex

Indicates changes to the UI cursor location.

Syntax

In Visual Basic:

```
Property CursorIndex As Long
```

In Visual C++ (MFC):

```
long GetCursorIndex();  
void SetCursorIndex(long nNewValue);
```

In Visual C++:

```
HRESULT get_CursorIndex(long * pVal);  
HRESULT put_CursorIndex(long newVal);
```

Parameters

nNewValue
??

Return Values

The **CursorIndex** is 0 based and can be set to any value greater than or equal to 0 but less than or equal to the number of characters held by **DictationMgr**.

Remarks

In order for **DictationMgr** to correctly manage speech input it must know where to place new text in the UI. To do this, the cursor index property of **DictationMgr** must be set appropriately any time there are changes to the UI cursor location. This might be an explicit cursor location change (the user moves the cursor with the mouse or keyboard) or an implicit change (characters inserted or deleted at or

before the current cursor location). If an implicit change occurs due to speech input (see the **PutText** event on page 687) it is not necessary to update the **CursorIndex** property of **DictationMgr**.

It is not necessary to update the **DictationMgr** cursor for actions such as typing (resulting in a call to **PutText** method) and speech input (**PutText** events) since they implicitly update the cursor much like text user interfaces do when typing. In fact, updating the cursor index arbitrarily in response to every cursor change will ultimately lead to degraded performance since explicit cursor changes (arrow key navigation, mouse clicks, etc.) require setting a bookmark (see **VVDictation**) in order to synchronize the latent speech recognition with the relatively real-time UI update.

Example

In Visual Basic:

```
VVDictationMgr.CursorIndex = CurrentCursorIndex
```

In Visual C++ (MFC):

```
m_VVDictationMgr.SetCursorIndex ( CurrentCursorIndex );
```

In Visual C++:

```
HRESULT hr = S_OK;  
hr = m_pIVVDictationMgr->put_CursorIndex ( CurrentCursorIndex );
```

See Also

“PutText” on page 678

“PutText” on page 687

DictationOn

Returns or sets the desired state of the dictation mode.

Syntax

In Visual Basic:

```
Property DictationOn As Boolean
```

In Visual C++ (MFC):

```
BOOL GetDictationOn();  
void SetDictationOn(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_DictationOn(VARIANT_BOOL * pVal);  
HRESULT put_DictationOn(VARIANT_BOOL newVal);
```

Parameters

fNewValue

Boolean.

Return Values

TRUE

The control can receive dictation input when dictation is available, based on the **AutoDictationWindow**.

FALSE

The control ignores dictation input that occurs after making this setting. Note that any speech input which occurred before setting **DictationOn** false but which is not yet completed will still be processed and passed to the client through the **PhraseReco** event.

Remarks

You can think of this property semantically as "Client want's dictation on". What this means is that if dictation is available (i.e. the **AutoDictationWindow** is active), then the user will be able to dictate into the control.

When the state of the dictation mode changes, the control fires the **DictationStateChange** event. You should not set the value of this property in the **DictationStateChange** event, as this will cause the event to fire again.

Example

In Visual Basic:

```
VVDictationMgr.DictationOn = True
```

In Visual C++ (MFC):

```
m_VVDictationMgr.SetDictationOn( TRUE );
```

In Visual C++:

```
HRESULT hr = S_OK;  
hr = m_pIVVDictationMgr->put_DictationOn( VARIANT_TRUE );
```

See Also

“AutoDictationWindow (Run Time Only)” on page 642

“DictationStateChange” on page 685

Engine (Run Time Only)

Sets or gets a reference to the ViaVoice **Engine** control (**VVEngine**), which is used by the **VVDictationMgr** control.

Syntax

In Visual Basic:

```
Property Engine As IVVEngine
```

In Visual C++ (MFC):

```
LPDISPATCH GetEngine();  
void SetRefEngine(LPDISPATCH newValue);
```

In Visual C++:

```
HRESULT get_Engine(IVVEngine * * pVal);  
HRESULT putref_Engine(IVVEngine * newVal);
```

Parameters

None.

Return Values

None.

Remarks

The **Engine** property is actually holding an implicitly created ActiveX control (**VVEngine**), which can also be created separately. Inserting a **VVEngine** control in a project enables you to set the engine properties on this control, and then assign the engine to multiple ViaVoice ActiveX controls.

Example

In Visual Basic:

```
VVDictationMgr.Engine = VVEngine
```

In Visual C++ (MFC):

```
m_VVDictationMgr->SetRefEngine( & m_pIVVEngineDispatch );
```

In Visual C++:

```
HRESULT hr = S_OK;  
hr = m_VVDictationMgr->putref_Engine( & m_pIVVEngineDispatch );
```

See Also

Refer to the ViaVoice Engine Control Guide for more information.

ExpandMacros

Indicates whether the **VVDictationMgr** control should expand and format speech input when possible.

Syntax

In Visual Basic:

```
Property ExpandMacros As Boolean
```

In Visual C++ (MFC):

```
BOOL GetExpandMacros();  
void SetExpandMacros(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_ExpandMacros(VARIANT_BOOL * pVal);  
HRESULT put_ExpandMacros(VARIANT_BOOL newVal);
```

Parameters

fNewValue

Boolean.

Return Values

TRUE

(Default) **VVDictationMgr** will expand macros and provide advanced, multi-phrase, number formatting if running on the IBM engine.

FALSE

VVDictationMgr does not do any advanced formatting or macro expansion.

Remarks

This property affects dictation only when used with the IBM engine. The following examples show how you might see the macros, however, this depends on regional settings:

False	True
"one thousand two hundred and thirty four"	"1,234"
"January first two thousand"	"January 1, 2000"
"my macro"	<whatever "my macro" is defined as>

If the value of this property is set to false you will get dictation exactly as interpreted by the engine. If set to true, and running on the IBM engine, all dictation will be checked for possible macro expansion (see Dictation Macro Editor application included with the SDK). This property also controls the availability of IBM advanced, multi-phrase, numeric formatting. For instance, setting this property to true would cause the following speech input to "expand" differently.

The setting of **ExpandMacros** has no effect if not running on the IBM engine.

Example

In Visual Basic:

```
VVDictationMgr.ExpandMacros = True
```

In Visual C++ (MFC):

```
m_VVDictationMgr.SetExpandMacros ( TRUE );
```

In Visual C++:

```
HRESULT hr = S_OK;  
hr = m_pIVVDictationMgr->put_ExpandMacros ( VARIANT_TRUE );
```

See Also

None.

Locked

Stops speech input immediately.

Syntax

In Visual Basic:

```
Property Locked As Boolean
```

In Visual C++ (MFC):

```
BOOL GetLocked();  
void SetLocked(BOOL bFewValue);
```

In Visual C++:

```
HRESULT get_Locked(VARIANT_BOOL * pVal);  
HRESULT put_Locked(VARIANT_BOOL newVal);
```

Parameters

fNewValue

Boolean.

Return Values

TRUE

All speech input will be disregarded.

FALSE

Speech input will be processed normally based on the setting of **DictationOn**.

Remarks

Because of the inherent latency of speech processing, setting **DictationOn** to false will not always prevent further speech input from being provided. However, it is sometimes necessary for a UI to

instantaneously stop ALL input immediately (**MaxText** and the **Locked** property of standard edit controls for instance). In order to facilitate this (and prevent the internal state of **DictationMgr** from becoming inconsistent with the UI) the Locked property can be used to stop speech input immediately, regardless of pending speech input.

Due to re-entrancy issues with COM STA controls, you should always set value of **Locked** to true before setting **DictationOn** false if you want to guarantee that no further speech input will be processed.

Example

In Visual Basic:

```
VVDictationMgr.Locked = True
```

In Visual C++ (MFC):

```
m_VVDictationMgr.SetLocked( TRUE );
```

In Visual C++:

```
HRESULT hr = S_OK;  
hr = m_pIVVDictationMgr->put_Locked( VARIANT_TRUE );
```

See Also

“DictationOn” on page 647

ProcessingMacro (Run Time Only)

Determines if the **VVDictationMgr** object is currently processing a multi-phrase macro expansion.

Syntax

In Visual Basic:

```
Property ProcessingMacro As Boolean
```

In Visual C++ (MFC):

```
BOOL GetProcessingMacro();  
void SetProcessingMacro(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_ProcessingMacro(VARIANT_BOOL * pVal);  
HRESULT put_ProcessingMacro(VARIANT_BOOL newVal);
```

Parameters

fNewValue
Boolean.

Return Values

TRUE

VVDictationMgr displays is currently processing a multi-phrase macro. This property will never be true unless running on the IBM engine. **This value cannot be set.**

FALSE

VVDictationMgr is not processing a multi-phrase macro. This value can be set to force completion of multi-phrase macro processing.

Remarks

If the value of **ProcessingMacro** is false, then all phrases are complete. If the value is true, then **VVDictationMgr** is currently processing a multi-phrase macro. If the value is true and you set it to false, all infirm phrases are then considered complete and any new speech input will be a "new" phrase. You can never set the value of **ProcessingMacro** to true. For more information on infirm phrases and multi-phrase macro expansion, see **PhraseReco** event on page 739.

The value of the **ProcessingMacro** property will never be true when not using the IBM speech engine.

Example

In Visual Basic:

```
VVDictationMgr.ProcessingMacro = False
```

In Visual C++ (MFC):

```
m_VVDictationMgr.SetProcessingMacro ( FALSE );
```

In Visual C++:

```
HRESULT hr = S_OK;  
hr = m_pIVVDictationMgr->put_ProcessingMacro ( VARIANT_FALSE );
```

See Also

“PutText” on page 687

UppercaseOn

Locks the speech input in upper case.

Syntax

In Visual Basic:

```
Property UppercaseOn As Boolean
```

In Visual C++ (MFC):

```
BOOL GetUppercaseOn();  
void SetUppercaseOn(BOOL fNewValue);
```

In Visual C++:

```
HRESULT get_UppercaseOn(VARIANT_BOOL * pVal);  
HRESULT put_UppercaseOn(VARIANT_BOOL newVal);
```

Parameters

fNewValue
Boolean.

Return Values

TRUE
Speech input will be uppercase.

FALSE
Speech input will be cased normally.

Remarks

None.

Example

In Visual Basic:

```
VVDictationMgr.UppercaseOn = True
```

In Visual C++ (MFC):

```
m_VVDictationMgr.SetUppercaseOn ( TRUE );
```

In Visual C++:

```
HRESULT hr = S_OK;  
hr = m_pIVVDictationMgr->put_UppercaseOn ( VARIANT_TRUE );
```

See Also

None.

DictationMgr Control Methods

The ViaVoice **DictationMgr** control supports the following methods:

- **About^a**
- **Command**
- **Correct**
- **DeleteText**
- **GetAlternate**
- **GetText**
- **GetWordInfo**
- **Playback**
- **PlaybackEx2**
- **PutText**
- **SetSelection**

a. Represents a standard method in Visual Basic. For more information, refer to your Visual Basic documentation.

Command

Issues a variety of commands that modify internal text held by **DictationMgr** without lowering the fidelity of information associated with the modified text.

Syntax

In Visual Basic:

```
Sub Command(Command As VVDM_Command)
```

In Visual C++ (MFC):

```
void Command(long Command);
```

In Visual C++:

```
HRESULT Command(VVDM_Command Command);
```

Parameters

Command
VVDM_Command. An identifier indicating the command to be executed. They include the following:

Constant	Value	Description
VVDM_Capitalize	0	Uppercase the first character of the selection. If there is no selection this command will modify the word containing the cursor.
VVDM_Uppercase	1	Uppercase the entire selection. If there is no selection this command will modify the word containing the cursor.

Constant	Value	Description
VVDM_Lowercase	2	Lowercase the entire selection. If there is no selection this command will modify the word containing the cursor.
VVDM_ScratchThat	3	Issuing this command will cause DictationMgr to "undo" the last speech input "phrase" that was received. A maximum of 10 phrases can be "undone" with VVDM_ScratchThat.

Return Values

??

Remarks

For instance, without this facility, capitalizing an existing word would require that the client delete the existing word and then add it back in the modified form. If that word happened to be dictated then the ability to correct or playback the word would be lost since it is now effectively typed text. These commands allow the client to uppercase, lowercase, and capitalize existing words or to remove the last dictated "phrase" recognized.

Altering the state of **DictationMgr** through any means other than speech input will eliminate the possibility of using "ScratchThat."

Example

In Visual Basic:

```
VVDictationMgr.Command VVDM_Capitalize
```

In Visual C++ (MFC):

```
m_VVDictationMgr.Command( VVDM_Capitalize );
```

In Visual C++:

```
HRESULT hr = S_OK;  
hr = m_pIVVDictationMgr->Command( VVDM_Capitalize );
```

See Also

None.

Correct

Corrects any misrecognized dictated words.

Syntax

In Visual Basic:

```
Sub Correct(CorrectText As String, _
    SoundsLike As String,
    AddAsSingleWord As Boolean, _
    StartIndex As Long, _
    IncorrectTextLength As Long)
```

In Visual C++ (MFC):

```
void Correct( LPCTSTR CorrectText,
    LPCTSTR SoundsLike,
    BOOL AddAsSingleWord,
    long StartIndex,
    long IncorrectTextLength);
```

In Visual C++:

```
HRESULT Correct( BSTR CorrectText,
    BSTR sSoundsLike,
    VARIANT_BOOL AddAsSingleWord,
    long StartIndex,
    long IncorrectTextLength );
```

Parameters

CorrectText

String. The correct interpretation of the indicated word.

SoundsLike

String. The phonetic spelling of the word (necessary for some languages, most notably Asian Pacific). An empty string ("") may be passed if no "sounds like" spelling is required.

AddAsSingleWord

Boolean. This parameter allows addition of multiple words to be corrected as a single phrase.

StartIndex

Long. This is the 0 based starting index of the word to be corrected. An error will be returned if the start index is not on a word boundary. The starting index of a word can be determined using the **GetWordInfo** method.

IncorrectTextLength

Long. This is the length of the misrecognized text to be corrected. An error will be returned if the specified length does not end on a word boundary. The length of a word can be determined using the **GetWordInfo** method.

Return Values

??

Remarks

By passing in the correct text, the 0 based starting index for the incorrect text, and the length of the incorrect text, you will be able to update the engine in order to improve accuracy for the same word when used in the future. Correcting **VVDictationMgr** will keep it synchronized with the UI and also insure that information on the word provided by **VVDictationMgr** will be accurate (see **GetWordInfo** method on page 673).

When calling the **Correct** method do not update the UI in the client. The **PutText** event will be fired with the appropriate information when any and all formatting issues have been resolved.

Example

In Visual Basic:

```
VVDictationMgr.Correct CorrectText, "", False, _  
    StartIdx, IncorrectLen
```

In Visual C++ (MFC):

```
m_VVDictationMgr.Correct( szCorrect, _T(""), FALSE,  
    StartIdx, IncorrectLen );
```

In Visual C++:

```
HRESULT hr = S_OK;  
BSTR s = SysAllocString ( OLESTR("") );  
hr = m_pIVVDictationMgr->Correct( sCorrect, s, FALSE,  
    StartIdx, IncorrectLen );  
SysFreeString ( s );  
s = NULL;
```

See Also

“GetWordInfo” on page 673

“PutText” on page 687

DeleteText

Remove any or all of the text within **VVDictationMgr**.

Syntax

In Visual Basic:

```
Sub DeleteText(StartIndex As Long, TextLength As Long)
```

In Visual C++ (MFC):

```
void DeleteText(long StartIndex, long TextLength);
```

In Visual C++:

```
HRESULT DeleteText(long StartIndex, long TextLength);
```

Parameters

StartIndex

Long. The 0 based index of the first character to be deleted.

TextLength

Long. The number of characters to be deleted. If the sum of the *StartIndex* and *TextLength* are greater than the number of characters stored in the **DictationMgr** an error (E_INVALIDARG) will be returned.

Return Values

??

Remarks

Simply call **DeleteText** with the 0 based starting index of the text to be deleted and the length to delete. Using a starting index of 0 and specifying the distinguished constant VV_EOT (-2) for the length will remove all text from **DictationMgr**.

When it is necessary to delete text (delete key, backspace, replacing a selection, etc.) always remember to update the UI before **DictationMgr**. This is necessary because deletion of text may change certain formatting characteristics triggering a **PutText** event to update the UI. For instance, deleting a dictated period may remove the capitalization from the following word if it is also dictated forcing an update of the UI. If you have not yet updated the UI then the information in the **PutText** event will be incorrect.

Example

In Visual Basic:

```
VVDictationMgr.DeleteText TextStart, TextLength
```

In Visual C++ (MFC):

```
m_VVDictationMgr.DeleteText ( lTextStart, lTextLen );
```

In Visual C++:

```
HRESULT hr = S_OK;  
hr = m_pIVVDictationMgr->DeleteText ( lTextStart, lTextLen );
```

See Also

“PutText” on page 687

GetAlternate

Gets alternative interpretations for dictation processed by the speech engine.

Syntax

In Visual Basic:

```
Function GetAlternate(StartIndex As Long, _  
    TextLength As Long, _  
    Rank As Long, _  
    SoundsLike As String) As String
```

In Visual C++ (MFC):

```
CString GetAlternate(long StartIndex,  
    long TextLength,  
    long Rank,  
    BSTR* SoundsLike);
```

In Visual C++:

```
HRESULT GetAlternate( long StartIndex,  
    long TextLength,  
    long Rank,  
    BSTR * SoundsLike,  
    BSTR * Alternate);
```

Parameters

TextLength ??
??

StartIndex

Long. The 0 based index of the first character of the word for which alternates are required.

Reserved

Long. This value is reserved for future use and must be 0.

Rank

Long. The 1 based rank of the desired alternate. A rank of 1 will provide the most likely alternative to the original word presented in the **PutText** method or **PutText** events. A rank of 0 will provide the most likely interpretation of the speech input and is the text that was used to construct the text provided in **PutText** event.

SoundsLike

String. In some languages (most notably Asian Pacific) many words have the same spelling but different pronunciations and meanings based on the context in which they are used. This is an output parameter that provides information necessary for accurate correction in these languages. It may also prove useful for getting the "spoken text" in other languages when acronyms and macros are in use. This parameter can be NULL if "sounds like" spelling is not required.

Alternate

String. Output parameter (return value in VB and MFC wrappers) which contains the alternate interpretation requested. If no alternate of the requested rank is available, this parameter will contain an empty string ("").

Return Values

??

Remarks

When the speech engine analyzes audio input, there are many possible choices. Based on various weighting algorithms, one possibility is chosen as most likely, and that "phrase" is given to the client. Using **GetAlternate** will allow you to access each successively less likely choice in order by rank.

What this means is that if you request the alternate of "rank 1", you will receive the engine second best choice (the original event gave you the best choice). You can continue asking for as many additional alternates as you desire. Eventually, once the engine drops below an engine specific threshold, it will no longer return any alternates of a higher rank. If you ask for an alternate which is not available you will receive an empty string. For languages that can access the HRESULT return code, S_FALSE will be returned when no alternate of the requested rank is available.

Alternates are usually presented to the user in the context of correction. Because these words are the most likely alternatives to the originally presented text, the user often finds the correct interpretation

within this list. Most error correction interfaces will allow the user to choose one of the alternatives or, since it may not be in the list, type in the correction themselves.

Example

In Visual Basic:

```
Dim Alternate as String
Alternate = VVDictationMgr.GetAlternate ( TextStart, _
    0, _
    CurRank, _
    SoundsLike )
```

In Visual C++ (MFC):

```
CString Alternate;
Alternate = m_VVDictationMgr.GetAlternate ( TextStart,
    0,
    CurRank,
    & bstrSoundsLike);
```

In Visual C++ :

```
BSTR bstrAlternate = NULL;
HRESULT hr = S_OK;
hr = m_pIVVDictationMgr->GetAlternate ( TextStart,
    0,
    CurRank,
    & bstrSoundsLike,
    & bstrAlternate );
```

See Also

None.

GetText

Retrieves a copy of any or all of the text held within **DictationMgr**.

Syntax

In Visual Basic:

```
Function GetText(StartIndex As Long, TextLength As Long) As String
```

In Visual C++ (MFC):

```
CString GetText(long StartIndex, long TextLength);
```

In Visual C++:

```
HRESULT GetText(long StartIndex,  
                long TextLength,  
                BSTR* Text);
```

Parameters

StartIndex

Long. The 0 based index of the first character required.

TextLength

Long. The length of text required. Use the distinguished constant VV_EOT (-2) to retrieve all text from the start index to the end.

Text

String. Output parameter (return value in VB and MFC wrappers) which contains the text requested.

Return Values

??

Remarks

The client need only specify the 0 based index of the first character and the length of text required. If a start index of 0 and a length of VV_EOT (-1) are specified, the entire text contained in **DictationMgr** can be retrieved.

One potential (and often valuable) use of this method occurs when debugging a potential index synchronization problem. By extracting the text at critical points during debug sessions you can more easily determine when, where, or if the cursor index has gotten out of sync.

Example

In Visual Basic:

```
Dim CurText as String
CurText = VVDictationMgr.GetText( StartIndex, TextLength )
```

In Visual C++ (MFC):

```
CString CurText;
CurText = m_VVDictationMgr.GetText( lStartIndex, lTextLength );
```

In Visual C++ :

```
BSTR bstrCurText = NULL;
HRESULT hr = S_OK;
hr = m_pIVVDictationMgr->GetText( lStartIndex, lTextLength,
    & bstrText );
```

See Also

None.

GetWordInfo

Determines information about the "word" at any valid character index.

Syntax

In Visual Basic:

```
Sub GetWordInfo(Index As Long, _  
    Length As Long, _  
    Flags As Long, _  
    SoundsLike As String)
```

In Visual C++ (MFC):

```
void GetWordInfo(long* Index,  
    long* Length,  
    long* Flags,  
    BSTR* SoundsLike);
```

In Visual C++:

```
HRESULT GetWordInfo(long * Index,  
    long * Length,  
    long * Flags,  
    BSTR * psSoundsLike );
```

Parameters

Index

Long. When the call is made, this parameter can be any valid 0 based index. If successful, this parameter will hold the index of the first character of the word containing the input index on return.

Length

Long. This parameter will hold the length of the word on return.

Flags

Long. These flags are a bit mask indicating how the word is combined with the words around it and also whether or not it is an expanded macro. Valid values are found in “VVDictation Phrase Formatting Flags” on page 741.

SoundsLike

String. On return this parameter will hold the "sounds like" spelling of the word. This is particularly useful for Asian Pacific languages but may also be useful for retrieving the "spoken text" when acronyms or macros are in use. This parameter can be NULL if the "sounds like" spelling is not required.

Remarks

This information includes the index of the first character, the length of the "word", formatting flags, and the "spoken text" or "sounds like text" for the word. This information can be used to implement commands like "Select This", "Next Word", and "Previous Word". It can also be used to find the exact word boundaries necessary for some of the other methods.

In many languages the "word length" will include the trailing space.

Example

In Visual Basic:

```
VVDictationMgr.GetWordInfo Index , Length, Flags, 0
```

In Visual C++ (MFC):

```
m_VVDictationMgr.GetWordInfo ( & lIndex, & lLength, & lFlags, NULL );
```

In Visual C++:

```
HRESULT hr = S_OK;  
    hr = m_pIVVDictationMgr->GetWordInfo ( & lIndex, & lLength, & lFlags,  
        NULL );
```

See Also

None.

Playback

See Also

Chapter 7, “Playback” on page 178

PlaybackEx2

See Also

Chapter 7, “PlaybackEx2” on page 182

PutText

Keeps **DictationMgr** synchronized with any non-dictated text in the UI.

Syntax

In Visual Basic:

```
Sub PutText(Text As String, _  
    TextLength As Long, _  
    [StartIndex As Long = -1])
```

In Visual C++ (MFC):

```
void PutText(LPCTSTR Text, long TextLength, long StartIndex);
```

In Visual C++:

```
HRESULT PutText(BSTR Text, long TextLength, long StartIndex);
```

Parameters

Text

String. The is the text to be added.

TextLength

Long. The length of the text to be added.

StartIndex

Long. The 0 based index where the text is to be placed. The default for this parameter (in languages like VB that use defaults) is VV_USECURSOR (-1), which causes the text to be inserted at the current cursor location.

Return Values

??

Remarks

For instance, text typed or pasted into the UI would need to be added to **DictationMgr** using **PutText**. It is not necessary to update the **CursorIndex** after calling **PutText** since it implicitly updates the cursor position to the index immediately after insertion. This is the same behavior seen when typing or pasting into a typical text UI. Remember, arbitrarily setting the cursor index when not required will degrade performance.

When it is necessary to insert text (keystroke, paste, etc.) always remember to update the UI before **DictationMgr**. This is necessary because inserting text may change certain formatting characteristics triggering a **PutText** event to update the UI. For instance, inserting text before the first word in a sentence may remove the capitalization from the word that was the first word if it is also dictated forcing an update of the UI. If you have not yet updated the UI then the information in the **PutText** event will be incorrect.

Example

In Visual Basic:

```
VVDictationMgr.PutText ( CurTextStr, Len(CurTextStr) )
```

In Visual C++ (MFC):

```
m_VVDictationMgr.PutText ( lpszCurText, strlen(lpszCurText), -1 );
```

In Visual C++:

```
HRESULT hr = S_OK;  
hr = m_pIVVDictationMgr.PutText ( bstrCurText,  
    SysStringLen (bstrCurText),  
    VV_USECURSOR );
```

See Also

None.

SetSelection

Synchronizes selections made in the UI with **DictationMgr**.

Syntax

In Visual Basic:

```
Sub SetSelection (StartIndex As Long, SelectionLength As Long)
```

In Visual C++ (MFC):

```
void SetSelection(long StartIndex, long SelectionLength);
```

In Visual C++:

```
HRESULT SetSelection (long StartIndex, long SelectionLength);
```

Parameters

StartIndex

Long. The 0 based index of the first character selected.

SelectionLength

Long. The number of characters selected. VV_EOT (-2) may be used to select to the end of existing text.

Return Values

None.

Remarks

This will allow the user to select text within the UI and then dictate a replacement in the same way that they would when typing or pasting text for a replacement. If a selection has been set when dictation is detected, the selection will be replaced in **DictationMgr** and the **DeleteText** event will be fired

directing the client to delete the selection. Setting the selection with a SelectionLength of 0 is equivalent to setting the **CursorIndex** property.

Calling **SetSelection** will implicitly update the cursor location to the beginning of the selection. As a result, calling **SetSelection** with a length of 0 is equivalent to setting the **CursorIndex** property. Like the **CursorIndex** property, calling **SetSelection** arbitrarily can impose significant overhead and should be used only when necessary. To do this effectively you must remain aware of which methods and events (i.e., **PutText**) implicitly update the cursor.

Example

In Visual Basic

```
VVDictationMgr.SetSelection StartIndex, Length
```

In Visual C++ (MFC):

```
m_VVDictationMgr.SetSelection ( lStartIdx, lLength );
```

In Visual C++:

```
HRESULT hr = S_OK;  
hr = m_pIVVDictationMgr->SetSelection ( lStartIdx, lLength );
```

See Also

“CursorIndex” on page 645

“PutText” on page 678

“PutText” on page 687

DictationMgr Control Events

The ViaVoice **DictationMgr** control supports the following events:

- **DeleteText**
- **DictationStateChange**
- **PutText**
- **QueryText**

DeleteText

Event fired when client should delete the text indicated from the UI.

Syntax

In Visual Basic:

```
Event DeleteText(StartIndex As Long, TextLength As Long
```

In Visual C++ (MFC):

```
void OnDeleteText (long StartIndex, long TextLength);
```

In Visual C++:

```
HRESULT DeleteText ( long StartIndex, long TextLength );
```

Parameters

StartIndex

Long. The 0 based index of the first character to be deleted.

TextLength

Long. The number of characters to be deleted.

Return Values

None.

Remarks

This event may be fired in response to dictation occurring after a selection is made or when the cursor is located inside an existing word. It will also be fired when the Command method is called with the "Scratch That" command.

Example

In Visual Basic:

```
Private Sub VVDictationMgr_DeleteText(ByVal StartIndex As Long, ByVal _
    TextLength As Long)
    TextBox.SelStart = StartIndex
    TextBox.SelLength = TextLength
    TextBox.SelText = ""
End Sub
```

In Visual C++ (MFC):

```
void [ClassName]::OnDeleteTextVVDictationMgr( long StartIndex,
    long TextLength)
{
    m_Edit.SetSel ( StartIndex, TextLength );
    m_Edit.ReplaceSel ( _T("") );
}
```

In Visual C++:

```
HRESULT [ClassName]::DeleteText ( long StartIndex, long TextLength )
{
    SendMessage (m_hwndEdit, EM_SETSEL,
        (WPARAM)StartIndex, (LPARAM)TextLength );
    SendMessage (m_hwndEdit, EM_REPLACESEL,
        FALSE, _T("") );
    return S_OK;
}
```

See Also

“Command” on page 660

“SetSelection” on page 680

DictationStateChange

Event fired any time there is a change in the state of dictation.

Syntax

In Visual Basic:

```
Event DictationStateChange(DictationOn As Boolean)
```

In Visual C++ (MFC):

```
void OnDictationStateChange (BOOL DictationOn);
```

In Visual C++:

```
HRESULT DictationStateChange ( VARIANT_BOOL DictationOn );
```

Parameters

DictationOn

Long. The current state of dictation.

Return Values

None.

Remarks

This might be in response to an explicit change in the value of the **DictationOn** property. Setting the **DictationOn** property will not trigger this event if the new value is the same as the old value of the property.

Example

In Visual Basic:

```
Private Sub VVDictationMgr_DictationStateChange( _  
    ByVal DictationOn As Boolean)  
    DoSomethingUseful DictationOn  
End Sub
```

In Visual C++ (MFC):

```
void [ClassName]::OnDictationStateChangeVVDictationMgr(BOOL DictationOn)  
{  
    DoSomethingUseful ( DictationOn );  
}
```

In Visual C++:

```
HRESULT [ClassName]::DictationStateChange( VARIANT_BOOL DictationOn)  
{  
    return DoSomethingUseful ( DictationOn );  
}
```

See Also

“DictationOn” on page 647

PutText

Event fired to provide all the information necessary to merge dictated text into the UI interface.

Syntax

In Visual Basic:

```
Event PutText(Text As String, _
    StartIndex As Long, _
    TextLength As Long, _
    CursorIndex As Long, _
    PhraseComplete As Boolean)
```

In Visual C++ (MFC):

```
void OnPutText (LPCTSTR Text,
    long StartIndex,
    long TextLength,
    long CursorIndex, BOOL PhraseComplete);
```

In Visual C++:

```
HRESULT PutText ( BSTR Text,
    long StartIndex,
    long DeleteLength,
    long CursorIdx,
    VARIANT_BOOL PhraseComplete );
```

Parameters

Text

String. This is the text to be displayed.

StartIndex

Long. This is the 0 based index where the text should be inserted.

DeleteLength

Long. This is the number of characters, beginning with the **StartIndex**, which need to be replaced. If the **DeleteLength** is 0 then no existing text needs to be replaced.

CursorIndex

Long. This is the 0 based index where the cursor should be located after the new text has been placed. It is necessary to explicitly set the UI cursor index in response to this event since the **Text** parameter may include more than just the new dictated text.

PhraseComplete

Boolean. This parameter will be true only when **DictationMgr** thinks it may be processing a multi-phrase macro. A multi-phrase macro is a macro which cannot be fully resolved within the context of a single spoken phrase. For instance, if a user dictated "one thousand," paused, and then dictated "five hundred," the ultimate result should be "1,500" not "1,000 500." The **DictationMgr** will take care of proper formatting and replacement of text as necessary to support this functionality without any interaction by the client. This information is provided only for clients who wish to visually indicate the potential resolution of a multi-phrase macro visually in the UI. It can also be used to allow the user to prevent multi-phrase macro resolution if the client application provides a means for the user to stop expansion of multi-phrase macros in progress (see the **ProcessingMacro** property on page 655). In the previous example, this would allow the user to dictate "1,000 500" as opposed to "1,500." Also note that if the user dictated "1,000," paused, and then dictated something like "computers," the first phrase would have **PhraseComplete** false. This is because **DictationMgr** has no way of knowing if the multi-phrase macro is complete until it processes the next phrase.

Return Values

TRUE

??

FALSE

??

Remarks

The client will receive a **PutText** event when dictation is recognized. If the **DictationMgr** has been kept correctly synchronized with the UI, this event provides all the information necessary to merge dictated text into the UI interface.

Example

In Visual Basic:

```
Private Sub VVDictationMgr_PutText(ByVal Text As String, _  
    ByVal StartIndex As Long, _  
    ByVal TextLength As Long, _  
    ByVal CursorIndex As Long, _  
    ByVal PhraseComplete As Boolean)  
    TextBox.SelStart = StartIndex  
    TextBox.SelLength = TextLength  
    TextBox.SelText = Text  
    TextBox.SelStart = CursorIndex  
End Sub
```

In Visual C++ (MFC):

```
void [ClassName]::OnPutTextVVDictationMgr(LPCTSTR Text,  
    long StartIndex,  
    long TextLength,  
    long CursorIndex,  
    BOOL PhraseComplete)  
{  
    m_Edit.SetSel ( StartIndex, TextLength );  
    m_Edit.ReplaceSel ( Text );  
    m_Edit.SetSel ( CursorIndex, 0 );  
}
```

In Visual C++:

```
void [ClassName]::PutText ( LPCTSTR Text,
    long StartIndex,
    long TextLength,
    long CursorIndex,
    BOOL PhraseComplete)
{
    SendMessage ( m_hwndEdit, EM_SETSEL,
        (WPARAM)StartIndex, (LPARAM)TextLength );
    LPTSTR pszText = ConvertBSTR2LPTSTR ( Text );
    SendMessage ( m_hwndEdit, EM_REPLACESEL,
        FALSE, pszText );
    delete[] pszText;
    SendMessage ( m_hwndEdit, EM_SETSEL,
        (WPARAM)CursorIndex, (LPARAM)0 );
    return S_OK;
}
```

See Also

“ProcessingMacro (Run Time Only)” on page 655

DictationMgr Control Frequently Asked Questions

This chapter contains answers to the most frequently asked questions about the ViaVoice **DictationMgr** Control.

When should I use the DictationMgr control?

VVDictationMgr does almost everything you need to do normal document based dictation except show it to the user. You only have to keep the user interface synchronized with **VVDictationMgr** through character indices. If you are not voice enabling a word processor or memo type application then this may well be more than you really need. **VVDictationMgr** will also not be suitable for voice enabling applications where it is not possible to synchronize using character indices.

When should I use the Dictation control?

VVDictation provides a simple means of just getting the text of what was said one phrase at a time. What you do with that text is up to you. You may or may not care about getting audio later and or doing correction in which case you could just destroy the RecoHandles provided and use the text. However, if you have or are attempting to build a word processor which is speech enabled on top of this object you will still have a great deal of work to do just to manage speech.

Getting Started with the Dictation Control

The following is a tutorial on how to incorporate the **VVDictation** control into your Visual Basic or Visual C++ applications. This tutorial is designed to present you with the most commonly used properties and events in the **VVDictation** control.

The following sections contain information to help you write code to create an instance of the **Dictation** Control and capture speech. You will also take steps to allow text correction and playback of captured speech.

Creating an Instance of the Control

This section contains step-by-step instructions for using Visual Basic or Visual C++ to create an instance of the control.

In Visual Basic:

To add the **VVDictation** control to your application, do the following:

1. From the **Project** menu, choose **References**.
The References dialog box, Figure 51, appears. The References dialog lists all the ActiveX Controls and simple COM objects that you can use in your application.

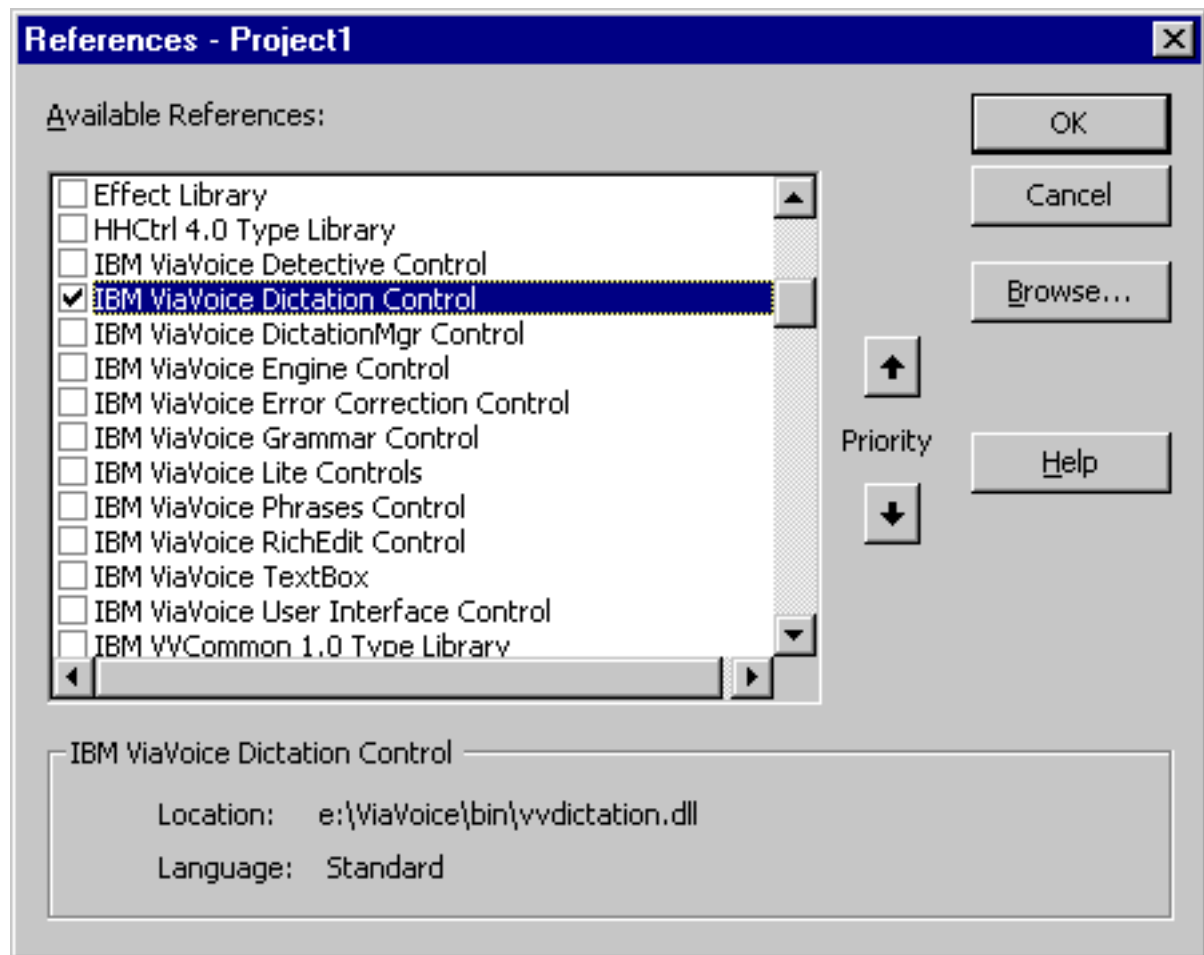


Figure 51. Reference Selection Dialog - Visual Basic

2. Select **IBM ViaVoice Dictation Control** from the list and click **OK**.
Visual Basic adds the control to your project, but does not add a new icon to the toolbar.
3. Add a global instance of the **VVDictation** control to your form code using the following syntax:

```
Dim WithEvents VVDict As VVDictation
```

In Visual C++:

To add the **VVDictation** component to your C++ project, do the following:

1. Include the VVDICTION.H. header file in any files that need access to the **VVDictation** component.
2. Create a class derived from IVVDictationEventSink (defined in VVDICTION.H) and implement all methods appropriately including the methods of **IUnknown**. This is the class that will receive notification events when speech input is processed. You can also derive your implementation class from IVVDictationEventSink and implement the methods there.
3. Use CoCreateInstance to create an instance of the **VVDictation** component using the following syntax:

```
HRESULT hr = CoCreateInstance ( CLSID_VVDictation,
                                NULL,
                                CLSCTX_INPROC_SERVER,
                                IID_IVVDictation,
                                (void**) & m_pIVVDictation );
```

4. Give your event sink implementation to the **VVDictation** instance you just created.

```
IconnectionPointContainer* pCPC = NULL;
IconnectionPoint* pCP = NULL;
m_dwSinkCookie = 0;
HRESULT hr = m_pIVVDictation ->QueryInterface
                                (IID_IConnectionPointContainer,
                                (void**) &pCPC);
if (SUCCEEDED(hr))
    hr = pCPC->FindConnectionPoint(IVVDictationEventSink, &pCP);
if (SUCCEEDED(hr))
    hr = pCP->Advise( & m_VVDictationEventSink, & m_dwSinkCookie);
if ( pCPC ) pCPC->Release();
if ( pCP ) pCP->Release();
```

5. Turn dictation on whenever you would like to start processing dictation.

Note:

Always check return codes from all method calls.

```
hr = m_pIVVDictation->put_DictationOn ( VARIANT_TRUE );
```

6. When you are done with VVDictation, you must disconnect the sink object and release VVDictation.

```
IconnectionPointContainer* pCPC = NULL;
IconnectionPoint* pCP = NULL;
HRESULT hr = m_pIVVDictation ->QueryInterface
                (IID_IConnectionPointContainer,
                (void**)&pCPC);
if (SUCCEEDED(hr))
    hr = pCPC->FindConnectionPoint(IVVDictationEventSink, &pCP);
if (SUCCEEDED(hr))
    hr = pCP->Unadvise(m_dwSinkCookie);
if ( pCPC ) pCPC->Release();
if ( pCP ) pCP->Release();
m_pIVVDictation->Release();
m_pIVVDictation = NULL;
```

Capturing Speech

The **VVDictation** object converts speech input into text suitable for display which is passed to the client as one of the parameters of the **PhraseReco** event. This event also provides the client with a **RecoHandle** and formatting flags relating to the particular phrase which is being presented. The **RecoHandle** identifies a particular phrase event when interacting with the **VVDictation** object and the flags provide information about how the current phrase should be combined with any existing or future text arranged around the current phrase. For instance, if the current phrase contains only a period ("."), you will receive flags indicating that it should merge flush with the phrase to the left (no white space left before the period) and that the next word should be capitalized. (*These flags are only available when using the IBM ViaVoice runtime.*) If you wish to be able to correct any misrecognized words, or provide playback of the speech which generated the event, you must store the **RecoHandle** in some way associated with the displayed text so that it can be retrieved when needed and given to **VVDictation** for processing. It is not necessary to store the flags since you can retrieve them at any time using the **RecoHandle**.

Summary

At this point, you should know how to do the following:

- How to incorporate the **VVDictation** control into your project.
- How to receive speech input.

The remainder of this documentation contains a reference for all the properties, methods, and events in the **VVDictation** control.

Introduction to the Dictation Control

The ViaVoice **Dictation** Control (**VVDictation**) is a low-level dictation object providing only the basics necessary for dictation, correction, and playback. This object is implemented as a simple COM object rather than a full ActiveX control and can not be "dropped" into a form and configured at design-time.

An application using **VVDictation** will receive **PhraseReco** events when speech has been recognized including the text of the recognition, and a recognition "handle". The recognition "handle" should be stored in some way associated with (i.e., mapped to) the displayed text of the recognition event. This will be necessary in order to correct the engine in the case of a "misrecognition", to do playback of dictated text.

If **VVDictation** is running on the IBM engine, the client will also be able to use the advanced formatting features available only with the IBM engine. This advanced formatting is enabled by setting **ExpandMacros** to true which will provide dates and numbers formatted according to system settings as well as expansion of user defined macros. If the property **ExpandMacros** is set to false, macro conversion and formatting is the responsibility of the client.

Because of the inherent latency of the Speech Engine audio resolution, Bookmarks must be used to synchronize the relatively real-time GUI with the dictation provided from the Engine in the **PhraseReco** events. The **SetBookmark** method can be used to set a bookmark with an ID at the current location in the audio stream. (For example, in response to the user changing the cursor location using the mouse, a bookmark would be set). When that bookmarked position in the audio stream is processed, the **HitBookmark** event will be fired including a bookmark ID that matches the one set by the call to **SetBookmark**.

Note:

When a **HitBookmark** event is received, this indicates to the client application that the current recognition event, and all subsequent speech related events, occurred after the point in time that the bookmark was set. For instance, when the above mentioned bookmark (set as a result of changing the cursor position) is passed in a **HitBookmark** event the client knows that the current and all subsequent speech input should be placed based on the new cursor position indicated by the user's mouse click.

Dictation Control Properties

The ViaVoice **Dictation** control supports the following properties:

- **AutoDictationWindow**
- **DictationOn**
- **Engine**
- **ExpandMacros**
- **ProcessingMacro**

AutoDictationWindow

Controls the scope in which dictation is available.

Syntax

In Visual Basic:

```
Property AutoDictationWindow As Long
```

In Visual C++:

```
HRESULT put_AutoDictationWindow ( long hWnd );  
HRESULT get_AutoDictationWindow ( long * hWnd );
```

Parameters

None.

Return Values

Any valid “top-most” window handle.

Dictation is available only when the indicated window is “active” as indicated by it, or one of its children, having the focus. **Note:** There can only be one dictation object active for the same window (**DictationOn** is True) at any one time!

NULL

Dictation mode is always available and must be controlled manually by setting the **DictationOn** property to True or False. **Note:** There can only be one global dictation object active (**DictationOn** is True) at any one time!

Remarks

The default value of this property is NULL (0), which will enable dictation globally. *However, please note that there can only be one global dictation object active (**DictationOn** is True) at any one time in the entire system, including other applications!* For this reason, it is strongly suggested that you avoid

global dictation objects, if possible. Alternatively, you can set this property to any valid "top-most"^a window handle, which maps dictation availability to that window's activation state (it or one of its children having focus).

Remember, if you use NULL be aware that there can only be one global dictation object active (**DictationOn** is True) at any one time. This includes your own or any other application running on the system. For this reason, global dictation objects should be used with extreme care and should be avoided, unless absolutely necessary. Regardless of the value of this property, finer granularity of control can always be achieved by changing the state of **DictationOn** appropriately.

Example

In Visual Basic:

```
'Assumes this form is the top-most form!
VVDict.AutoDictationWindow = hWnd
```

In Visual C++:

```
// Makes no assumptions about m_hWnd
HWND Hwnd = m_hWnd;
// Due to the Win32 implementation of GetParent, this is necessary
// to find the "Foreground" window for SAPI grammar activation
// For more information see MS Knowledge Base article Q84190
while ( ::GetParent ( hwnd ) != NULL &&
        ! ( ::GetWindowLong( hwnd ,GWL_STYLE ) & WS_POPUP ) )
{
    hwnd = ::GetParent ( hwnd );
}
m_pIVVDictation->put_AutoDictationWindow ( (long)hwnd );
```

See Also

“DictationOn” on page 704, “DictationStateChange” on page 735

a. A “top-most” window is defined as any window without a parent. For more information on issues dealing with finding the “top-most” window, see the Microsoft Knowledge Base article Q84190.

DictationOn

Returns or sets the desired state of the dictation mode.

Syntax

In Visual Basic:

```
Property DictationOn As Boolean
```

In Visual C++:

```
HRESULT get_DictationOn(VARIANT_BOOL * DictationOn )  
HRESULT put_DictationOn(VARIANT_BOOL DictationOn )
```

Parameters

DictationOn
Boolean.

Return Values

TRUE

The control can receive dictation input, when available.

FALSE

The control ignores dictation input that occurs after making this setting. Note that any speech input which occurred before setting **DictationOn** false but which is not yet completed will still be processed and passed to the client through the **PhraseReco** event.

Remarks

You can think of this property semantically as "Client want's dictation on". What this means is that if dictation is available (i.e. the AutoDictationWindow is active), then the user will be able to dictate into the control.

When the state of the dictation mode changes, the control fires the **DictationStateChange** event. You should not set the value of this property in the **DictationStateChange** event, as this will cause the event to fire again.

Example

In Visual Basic:

```
VVDict.DictationOn = True
```

In Visual C++:

```
m_pIVVDictation->put_DictationOn( VARIANT_TRUE );
```

See Also

“AutoDictationWindow” on page 702

“DictationStateChange” on page 735

Engine

Sets or gets a reference to the ViaVoice **Engine** control (**VVEngine**), which is used by the **VVDictation** control.

Syntax

In Visual Basic:

```
Property Engine As IVVEngine
```

In Visual C++:

```
HRESULT get_Engine( IVVEngine * * IVVEngine )  
HRESULT putref_Engine( IVVEngine * IVVEngine )
```

Parameters

None.

Return Values

None.

Remarks

The **Engine** property is actually holding an implicitly created ActiveX control (**VVEngine**), which can also be created separately. Inserting a **VVEngine** control in a project enables you to set the engine properties on this control, and then assign the engine to multiple ViaVoice ActiveX controls.

Example

In Visual Basic:

```
VVDict.Engine = VVEngine1
```

In Visual C++:

```
m_pIVVDictation->putref_Engine( m_pIVVEngine );
```

See Also

Refer to the ViaVoice Engine Control Guide for more information.

ExpandMacros

Indicates whether the **VVDictation** control should expand and format speech input when possible.

Note:

This property affects dictation only when used with the IBM engine.

The following examples show how you might see the macros, however, this depends on regional settings:

False	True
"one thousand two hundred and thirty four"	"1,234"
"January first two thousand"	"January 1, 2000"
"my macro"	<whatever "my macro" is defined as> using the Dictation Macro Editor

Syntax

In Visual Basic:

```
Property ExpandMacros As Boolean
```

In Visual C++:

```
HRESULT get_ExpandMacros( VARIANT_BOOL * ExpandMacros )
HRESULT put_ExpandMacros( VARIANT_BOOL ExpandMacros )
```

Parameters

ExpandMacros
Boolean.

Return Values

TRUE

(Default) **VVDictation** will expand macros and provide advanced, multi-phrase, number formatting if running on the IBM engine.

FALSE

VVDictation does not do any advanced formatting or macro expansion.

Remarks

If the value of this property is set to false you will get dictation exactly as interpreted by the engine. If set to true, and running on the IBM engine, all dictation will be checked for possible macro expansion (see Dictation Macro Editor application, DME.EXE, included with the SDK Dictation Runtime). This property also controls the availability of IBM advanced, multi-phrase, numeric formatting. For instance, setting this property to true would cause the following speech input to "expand" differently.

The setting of **ExpandMacros** has no effect if not running on the IBM engine.

Example

In Visual Basic:

```
VVDict.ExpandMacros = True
```

In Visual C++:

```
m_pIVVDictation->put_ExpandMacros ( VARIANT_TRUE );
```

See Also

None.

ProcessingMacro

Determines if the **VVDictation** object is currently processing a multi-phrase macro expansion.

Syntax

In Visual Basic:

```
Property ProcessingMacro As Boolean
```

In Visual C++:

```
HRESULT get_ProcessingMacro(VARIANT_BOOL * ExpandMacros )  
HRESULT put_ProcessingMacro(VARIANT_BOOL ExpandMacros )
```

Parameters

??

Return Values

TRUE

VVDictation displays is currently processing a multi-phrase macro. This property will never be true unless running on the IBM engine. **This value can not be set.**

FALSE

VVDictation is not processing a multi-phrase macro. This value can be set to force completion of multi-phrase macro processing.

Remarks

If the value of **ProcessingMacro** is false, then all phrases are complete. If the value is true, then **VVDictation** is currently processing a multi-phrase macro. If the value is true and you set it to false, all infirm phrases are then considered complete and any new speech input will be a new phrase. You can never set the value of **ProcessingMacro** to true. For more information on infirm phrases and multi-phrase macro expansion, see **PhraseReco** event on page 739, which also indicates whether or not **VVDictation** is processing a multi-phrase macro.

The value of the **ProcessingMacro** property will never be true when not using the IBM speech engine.

Example

In Visual Basic:

```
VVDict.ProcessingMacro = False
```

In Visual C++:

```
m_pIVVDictation->put_ProcessingMacro ( VARIANT_FALSE );
```

See Also

“PhraseReco” on page 739

Dictation Control Methods

The ViaVoice **Dictation** control supports the following methods:

- **Correct**
- **Destroy**
- **GetAlternatePhrase**
- **GetFlags**
- **GetWavData**
- **GetWordInfo**
- **MergeRecoPhrases**
- **SetBookMark**
- **SetContext**
- **SplitOutLeftWord**

Correct

Corrects any misrecognized words provided by the **PhraseReco** event.

Syntax

In Visual Basic:

```
Sub Correct(RecoHandle As Long, _
    lIndex As Long, _
    Reserved As Long, _
    AddAsSingleWord As Boolean, _
    CorrectText As String, _
    SoundsLike As String, _
    Phrase As String, _
    Flags As Long)
```

In Visual C++:

```
HRESULT Correct ( VV_RecoHandle RecoHandle,
    long Index,
    long Reserved,
    VARIANT_BOOL AddAsSingleWord,
    BSTR CorrectText,
    BSTR SoundsLike,
    BSTR* Phrase,
    long* Flags);
```

Parameters

RecoHandle

Long. A 32 bit value that uniquely identifies the phrase to be corrected. The *RecoHandle* for a phrase is passed as one of the parameters of the **PhraseReco** event.

Index

Long. The index of the first character of the word to be corrected.

Reserved

Long. This parameter is reserved for future use and should always be set to 0.

AddAsSingle

Boolean. This parameter allows addition of multiple words to be corrected as a single phrase.

CorrectText

String. A string indicating the "correct" interpretation for this particular speech input (RecoHandle).

SoundsLike

String. A string indicating the "correct" interpretation for this particular speech input (RecoHandle).

Phrase

String. The modified (corrected) phrase. This string is a replacement for the original string provided in the **PhraseReco** event and should replace the original string in the UI.

Flags

Long. An output only parameter which will provide you the correct formatting flags based on the corrected text. These flags should be applied in exactly the same manner as the original flags received in the **PhraseReco** event. For instance, correct application of these flags will allow the replacement of the word "period" with "." to remove white space from the left of the "." and capitalize the following word.

Return Values

None.

Remarks

By passing in the RecoHandle for the phrase containing the word to be corrected, the character index of the first character of the word to be corrected, and the corrected text, you will be able to update the engine in order to improve accuracy for the same word when used in the future. Correcting **VVDictation** will also insure that information on the word provided by **VVDictation** will be accurate (see **GetWordInfo** method on page 723).

After correction, a corrected phrase and formatting flags will be returned to the caller using the "Phrase" parameter. This corrected phrase should replace the original text provided in **PhraseReco**.

Example

In Visual Basic:

```
VVDict.Correct ( RecoHandle, WordIndex, 0, _  
    False, CorrectText, SoundsLike, _  
    NewPhrase, Flags )
```

In Visual C++:

```
m_pIVVDictation->Correct(hReco, lWordIdx, 0, VARIANT_FALSE,  
    bstrCorrectText, bstrSoundsLike,  
    &bstrNewPhrase, &lFlags);
```

See Also

“PhraseReco” on page 739

“VVDictation Phrase Formatting Flags” on page 741

GetAlternatePhrase

Gets alternative interpretations of a particular speech input (represented by a RecoHandle).

Syntax

In Visual Basic:

```
Function GetAlternatePhrase (RecoHandle As Long, _  
    Index As Long, _  
    Reserved As Long, _  
    Rank As Long, _  
    SoundsLike As String) As String
```

In Visual C++:

```
HRESULT GetAlternatePhrase ( VV_RecoHandle RecoHandle,  
    long Index,  
    long Reserved,  
    long Rank,  
    BSTR * SoundsLike,  
    BSTR * PhraseText)
```

Parameters

RecoHandle

Long. A 32-bit value that uniquely identifies the phrase (word) for which alternates are needed. The RecoHandle for a phrase is passed as one of the parameter to the **PhraseReco** event or returned from the **SplitOutLeftWord** or **MergeRecoPhrases** method.

Index

Long. The character index of the word for which an alternate is desired.

Reserved

Long. This parameter is reserved for future use.

Rank

Long. The rank of desired alternate.

SoundsLike ??

??

PhraseText

String. The alternate of the indicated rank or an empty string if no alternate of the indicated rank was available.

Return Values

??

Remarks

These alternatives are retrieved based on their rank, which represents the likelihood, according to the engine, that the particular alternate is a correct interpretation of the speech input. Alternates are typically presented to the user for selection in the context of a potential correction.

Ranks begin with 0 and the total number of alternatives available depends on the word and the underlying speech engine. For any given speech input, the alternate of rank 0 is what the engine thinks was said. This is the text originally provided by the **PhraseReco** event. An alternative of rank 1 is, according to the engine, the most likely alternative to the text originally provided. Each succeeding rank (higher numerically) alternate is correspondingly less likely. If an alternative of a given "rank" is not available, an empty string will be returned. An empty string for a given rank also implies that no alternates of higher rank are available. The number of alternates available for any speech input is dependent both on the speech input itself and on the underlying speech engine implementation. The recommended approach to retrieve all alternates for a given speech input is to use forward iteration, beginning with rank 1, until an empty string is returned. In languages which give access to the actual return code (HRESULT) the return code can be tested for S_FALSE to determine if no alternative was available for the requested rank.

Example

In Visual Basic:

```
VVDict.GetAlternatePhrase( RecoHandle, Index, 0, Rank, AlternateText )
```

In Visual C++:

```
m_pIVVDictation->GetAlternatePhrase(hReco, lIndex, 0, lRank,  
    &bstrAltText);
```

See Also

None.

GetFlags

Retrieves the formatting flags originally provided with speech input in the **PhraseReco** event.

Syntax

In Visual Basic:

```
Function GetFlags(RecoHandle As Long) As Long
```

In Visual C++:

```
HRESULT GetFlags(VV_RecoHandle RecoHandle, long * Flags)
```

Parameters

RecoHandle

Long. A 32 bit value that uniquely identifies the phrase for which flags are being requested. The *RecoHandle* for a phrase is passed as one of the parameter to the **PhraseReco** event.

Flags

Long. Storage where the flags associated with the indicated **RecoHandle** will be placed.

Return Values

None.

Remarks

None.

Example

In Visual Basic:

```
Flags = VVDict.GetFlags ( RecoHandle)
```

In Visual C++:

```
m_pIVVDictation->GetFlags ( hReco, &lFlags );
```

See Also

“VVDictation Phrase Formatting Flags” on page 741

GetWavData

Gets the actual audio associated with a given speech input packaged in a BSTR.

Syntax

In Visual Basic:

```
Function GetWavData(RecoHandle As Long, _
    Index As Long, _
    Reserved As Long, _
    PlaySound As Boolean) As String
```

In Visual C++:

```
HRESULT GetWavData( VV_RecoHandle RecoHandle,
    long Index,
    long Reserved,
    VARIANT_BOOL PlaySound,
    BSTR* WavData)
```

Parameters

RecoHandle

Long. A 32 bit value that uniquely identifies the phrase for which audio (WAV data) is being requested. The *RecoHandle* for a phrase is passed as one of the parameter to the **PhraseReco** event.

Index

Long. The character index of the word for which audio data is desired.

Reserved

Long. This parameter is reserved for future use and should always be 0.

PlaySound

Boolean. Indicates whether the caller would like **VVDictation** to play the sound or just return the audio data. If this value is true, then **VVDictation** will play the audio. If false, the data will be returned without playing.

WavData

String (BSTR). This is the actual audio data. If you do not wish to use the actual audio data (perhaps when **PlaySound** is True) you may pass NULL for this parameter.

Remarks

The data is in standard RIFF WAV format and may be saved for playback in a *.wav file or sent directly to any API (for example, sndPlaySound) capable of playing RIFF WAV audio. If **PlaySound** is True, **VVDictation** can also play the audio for you.

While the BSTR type may seem a strange choice for transfer of audio data, it was chosen for its relative ease of access and manipulation from all development environments. However, you must be aware that some environments and frameworks (VB, MFC CString, etc.) will truncate a BSTR at the first NULL when performing certain operations. If you have difficulties playing the resultant data, please refer to the relevant documentation.

Example

In Visual Basic:

```
WavString = VVDict.GetWavData ( RecoHandle, 0, 0,True )
```

In Visual C++:

```
m_pIVVDictation->GetWavData ( hReco, 0, 0, VARIANT_FALSE, &bstrWavData );
```

See Also

None.

GetWordInfo

Gets information on individual words within a phrase.

Syntax

In Visual Basic:

```
Sub GetWordInfo(RecoHandle As Long, _
    Index As Long, _
    Length As Long, _
    Flags As Long, _
    SoundsLike As String)
```

In Visual C++:

```
HRESULT GetWordInfo ( VV_RecoHandle RecoHandle,
    long * Index,
    long * Length,
    long * Flags,
    BSTR * SoundsLike )
```

Parameters

RecoHandle

Long. A 32 bit value that uniquely identifies the phrase for which word information is being requested. The *RecoHandle* for a phrase is passed as one of the parameter to the **PhraseReco** event.

Index

Long. Input: A zero based character index into the phrase represented by the *RecoHandle*.

Output: A zero based index of the beginning of the word containing the input index.

Length

Long. The length of the word containing the input index.

Flags

Long. The flags associated with the word containing the input cursor.

SoundsLike

String. The "sounds-like" spelling, if any, for the indicated word.

Example

In Visual Basic:

```
VVDict.GetWordInfo ( RecoHandle, Index, Length, Flags, SoundsLikeText)
```

In Visual C++:

```
m_pIVVDictation->GetWordInfo ( hReco, &lIndex, &lLength, &lFlags,  
    &bstrSoundsLike );
```

Remarks

By passing in a zero based character index and a RecoHandle, the client can get information regarding the word containing the input index. This includes the starting index and length of the word, the flags associated with the word, and the sounds-like text (if any) for the word.

See Also

“VVDictation Phrase Formatting Flags” on page 741

MergeRecoPhrases

Merges any two adjacent phrases (represented by their RecoHandles) into a single phrase (RecoHandle).

Syntax

In Visual Basic:

```
Sub MergeRecoPhrases(LeftRecoHandle As Long, _
    RightRecoHandle As Long, _
    MergedText As String, _
    Flags As Long)
```

In Visual C++:

```
HRESULT MergeRecoPhrases(VV_RecoHandle LeftRecoHandle,
    VV_RecoHandle RightRecoHandle,
    BSTR* MergedText,
    long* Flags)
```

Parameters

LeftRecoHandle

Long. A 32 bit value that uniquely identifies the left phrase to be merged. If this method call is successful, then this RecoHandle will represent the combined phrase. This RecoHandle **MUST** be the RecoHandle from the last **PhraseReco** before the RightRecoHandle. That is, the left and right RecoHandles **MUST** be both temporally adjacent and sequential. The RecoHandle for a phrase is passed as one of the parameter to the **PhraseReco** event.

RightRecoHandle

Long. A 32 bit value that uniquely identifies the phrase to be corrected. The RecoHandle for a phrase is passed as one of the parameter to the **PhraseReco** event. This RecoHandle is invalid after a successful call.

MergedText

String. If the merge is successful, this parameter will hold the text that results from evaluating the merged speech input.

Flags

Long. The flags associated with the merged phrase (RecoHandle).

Return Values

None.

Remarks

To merge a pair of phrases, they must be both temporally adjacent and sequential.

Example

In Visual Basic:

```
VVDict.MergeRecoPhrases ( LeftRecoHandle, RightRecoHandle, MergedText, _  
    Flags )
```

In Visual C++:

```
m_pIVVDictation->MergeRecoPhrases( hLeftReco, hRightReco,  
    &bstrMergedText, &lFlags);
```

See Also

“VVDictation Phrase Formatting Flags” on page 741

SetBookMark

Synchronizes the relatively "real-time" UI events with the inherent latency of speech recognition.

Syntax

In Visual Basic:

```
Sub SetBookMark(BookMarkId As Long)
```

In Visual C++:

```
HRESULT SetBookMark(long BookMark)
```

Parameters

BookMarkId

Long

A number that uniquely identifies the bookmark being set.

Remarks

This is done by passing in an identifying BookMarkId which will be used to identify the synchronization point being set. When the speech engine begins processing the point in the audio stream where the bookmark was set, **VVDictation** will fire the **HitBookMark** event with BookMarkId which was passed into **SetBookMark**. All subsequent **PhraseReco** events will have occurred after the point in time where the bookmark was set.

Example

In Visual Basic:

```
VVDict.SetBookMark ( CurrentBookMarkId )
```

In Visual C++:

```
m_pIVVDictation->SetBookMark ( m_lCurBookMarkId );
```

Remarks

As stated previously, bookmarks are used to synchronize UI events with speech input. An example where this synchronization is needed would be the user changing the cursor location with the mouse while the engine is still processing speech.

If the client does not use bookmarks, the speech input being processed (which occurred before the cursor location changed) would be placed at the new cursor location--probably not what the user had in mind. This confusion can be avoided by setting a bookmark when the cursor location changes and then continuing to place speech input based on the previous cursor location until the **HitBookMark** event (with the correct BookMarkId) is received.

If no there is no unresolved audio remaining to be processed, the **HitBookMark** event will fire immediately with the indicated BookMarkId. In this example the client would also need to call **SetContext** after receiving the **HitBookMark** event to prevent degradation of speech recognition accuracy.

See Also

“HitBookMark” on page 737

“SetContext” on page 729

SetContext

Sets the "context" in which the speech input is to be evaluated.

Syntax

In Visual Basic:

```
Sub SetContext(LeftText As String, RightText As String)
```

In Visual C++:

```
HRESULT SetContext(BSTR LeftText, BSTR RightText)
```

Parameters

LeftText

String. One or more words (if any) on the left side of the current speech input location. If there are no words to the left, pass in an empty ("") string. For best results, pass in at least two words when possible.

RightText

String. One or more words (if any) on the right side of the current speech input location. If there are no words to the right, pass in an empty ("") string. For best results, pass in at least two words when possible.

Return Values

None.

Remarks

This maximizes speech recognition accuracy. In this case the context we are referring to is the words surrounding the current speech input location (usually one or more words, if any, on either side of the cursor). This allows sentences such as "I have *two* pencils" and "I went *to* work" to be resolved correctly. For maximum speech recognition accuracy the engine context must be updated when:

- The cursor location changes.
- Existing text is deleted from the current document.
- New text is typed or pasted into the current document.
- When a new document is opened or made current.

No context update is necessary if cursor movement is a result of normal dictation speech input (i.e. the cursor moves from n to n+5 when the word "test " is recognized and inserted). The number of words which should be provided for optimal accuracy will vary depending on the underlying speech engine installed. In most cases 2 words on either side will provide the necessary context, although, more may be passed without problem. The engine will simply use what it can and discard the rest. However, you must keep in mind that depending on engine implementation and configuration there may be unacceptable memory and processing overhead associated with excessively large blocks of text used for context.

Example

In Visual Basic:

```
VVDict.SetContext ( LeftText, RightText )
```

In Visual C++:

```
m_pIVVDictation->SetContext ( LeftText, RightText );
```

See Also

“SetBookMark” on page 727

SplitOutLeftWord

Splits a phrase into its component words or to isolate a single word within a phrase (for deletion perhaps).

Syntax

In Visual Basic:

```
Sub SplitOutLeftWord(RecoHandle As Long,
    RightText As String,
    RightFlags As Long,
    LeftRecoHandle As Long,
    LeftText As String,
    LeftFlags As Long)
```

In Visual C++:

```
HRESULT SplitOutLeftWord(VV_RecoHandle RecoHandle, BSTR* RightText,
    long* RightFlags,
    VV_RecoHandle* LeftRecoHandle,
    BSTR* LeftText,
    long* LeftFlags)
```

Parameters

RecoHandle

Long. A 32 bit value that uniquely identifies the phrase to be manipulated. The *RecoHandle* for a phrase is passed as one of the parameter to the **PhraseReco** event. This existing *RecoHandle* will represent the right side of the phrase after a successful split occurs.

RightText

String. The text of the remaining portion of the phrase after the first word is removed.

RightFlags

Long. The formatting flags indicating how this phrase should be combined with any surrounding text.

LeftRecoHandle

Long. A 32 bit value that uniquely identifies the new phrase consisting of the left word of the original phrase.

LeftText

String. The text of the new single word "phrase" (the left word of the original phrase).

LeftFlags

Long. The formatting flags indicating how this phrase should be combined with any surrounding text.

Return Values

None.

Remarks

SplitOutLeftWord will allow you to take an existing phrase of more than one word and convert it into two phrases. The "left phrase" will consist of the first word of the original phrase. The "right phrase" will consist of the remainder of the original phrase.

If the original "phrase" consists of only a single word, the new LeftRecoHandle will be NULL and the LeftText will be an empty string (""). For languages where the client can make use of the true HRESULT return value, a value of S_FALSE will be returned.

Example

In Visual Basic:

```
VDict.SplitOutLeftWord ( OldRecoHandle, RightText, RightFlags, _  
    NewRecoHandle, LeftText, LeftFlags )
```

In Visual C++:

```
m_pIVDictation->SplitOutLeftWord ( hOld, &bstrRight, &lRightFlags,  
    HNew, &bstrLeft, &lLeftFlags);
```

See Also

“VVDictation Phrase Formatting Flags” on page 741

Dictation Control Events

The ViaVoice **Dictation** control supports the following events:

- **DictationStateChange**
- **HitBookMark**
- **PhraseReco**

DictationStateChange

Event fired when the ability to receive dictation input changes.

Syntax

In Visual Basic:

```
Event DictationStateChange(DictationOn As Boolean)
```

In Visual C++:

```
HRESULT DictationStateChange ( VARIANT_BOOL DictationOn )
```

Parameters

DictationOn

Long. The current state of dictation.

Return Values

TRUE

The control is ready to receive speech input and turn it into text.

FALSE

The control is not able to receive speech input.

Remarks

You can explicitly change the state of dictation by setting the value of the **DictationOn** property in the control. If the new state set is different from the previous state, the **DictationStateChange** event will be fired upon successful transition to the new state. This does not imply anything about dictation availability based on the **AutoDictationWindow**.

Example

In Visual Basic:

```
Private Sub VVDict_DictationStateChange(ByVal DictationOn As Boolean)
    ProcessDictationStateChange ( DictationOn )
End Sub
```

In Visual C++:

```
Void CVVDictEvents::DictationStateChange( VARIANT_BOOL DictationOn )
{
    m_pClient->ProcessDictationStateChange ( DictationOn );
}
```

See Also

“AutoDictationWindow” on page 702

“DictationOn” on page 704

HitBookMark

Event fired when the engine begins to process audio occurring after a bookmark was successfully set using the **SetBookMark** method.

Syntax

In Visual Basic:

```
Event HitBookMark(BookMarkId As Long)
```

In Visual C++:

```
HRESULT HitBookMark ( long BookMarkId )
```

Parameters

BookMarkId

Long. The value which was passed as an argument to **SetBookMark** when the bookmark being processed was set.

Return Values

None.

Remarks

In many cases the context will need updating when a bookmark is hit.

Example

In Visual Basic:

```
Private Sub VVDict_HitBookMark( BookMarkId As Long )  
    ProcessBookMark ( BookMarkId )  
End Sub
```

In Visual C++:

```
void CVVDictEvents::HitBookMark( long BookMarkId )  
{  
    ProcessBookMark ( BookMarkId );  
}
```

See Also

“SetBookMark” on page 727

“SetContext” on page 729

PhraseReco

Event fired when dictation speech input is recognized to provide the client with the dictated text along with flags indicating how this phrase should be combined with existing text.

Syntax

In Visual Basic:

```
Event PhraseReco(Text As String, RecoHandle As Long, RecoFlags As Long)
```

In Visual C++:

```
HRESULT PhraseReco ( BSTR Text,  
    VV_RecoHandle RecoHandle,  
    VV_RecoFlags RecoFlags )
```

Parameters

Text

Long. This is the text that was recognized.

RecoHandle

Long. A 32 bit value that uniquely identifies the current phrase. In most cases this "RecoHandle" should be stored in some way mapped to the displayed text. This will allow you to ask **VVDictation** to perform various manipulations of the current phrase in the future such as correction and playback. For more information on how the RecoHandle is used by **VVDictation** you see the **VVDictation** methods documentation.

RecoFlags

Long. Flags indicating how the current phrase should be combined with existing text.

Return Values

None.

Remarks

For instance, flags could indicate that the current text should replace the last "phrase". It might also cause the next word to be capitalized if this phrase ends in a period.

Example

In Visual Basic:

```
Private Sub VVDict_PhraseReco(ByVal Text As String, _  
    ByVal RecoHandle As Long, _  
    ByVal RecoFlags As Long)  
    'Do something useful  
End Sub
```

In Visual C++:

```
void CVVDictEvents::PhraseReco( BSTR Text,  
    VV_RecoHandle RecoHandle,  
    VV_RecoFlags RecoFlags )  
{  
    // Do something useful  
}
```

See Also

“VVDictation Phrase Formatting Flags” on page 741

VVDictation Phrase Formatting Flags

Phrase formatting flags indicated how a given phrase should be combined with existing text. This might include replacing the previous phrase (if multi-phrase formatting is enabled) or capitalizing the next word if the given phrase is the end of a sentence. Each of the phrase formatting flags is listed below with a short description of their meaning and use.

- **FF_EXPANDED_MACRO** (0x00000001)
This phrase is not the actual text that was spoken. The spoken text has been replaced with text specified in a macro or through advanced formatting. These services are available only with the IBM speech recognition engine.
- **FF_JOIN_LEFT** (0x00000002)
This phrase should be merged flush against the word to the left with no white space separating them. For instance this flag will be used if the current phrase is a period (". ").
- **FF_JOIN_RIGHT** (0x00000004)
This phrase should be merged flush against the word to the right with no white space separating them.
- **FF_CAPITALIZE_NEXT** (0x00000008)
This flag indicates that the word following should be capitalized. For instance this flag will be used if the current phrase is the end of a sentence.
- **FF_INFIRM_PHRASE** (0x00000010)
This flag is used only when advanced formatting is available (i.e. **ExpandMacros** is true AND the IBM speech engine is being used). It indicates that the client may need to replace the current phrase with the next and should store any information necessary to perform the replacement as indicated by the flags of the next phrase. Each subsequent phrase that should replace the "infirm" phrase will also include the **FF_INFIRM_PHRASE** flag but not the **FF_NEW_PHRASE** flag. A phrase with the **FF_NEW_PHRASE** flag indicates that any previous multi-phrase formatting is completed (i.e. a phrase with the **FF_NEW_PHRASE** flag should not replace a previous phrase having the **VV_INFIRM_PHRASE** flag). The **FF_INFIRM_PHRASE** flag may be combined with the **FF_NEW_PHRASE** flag if the current phrase is both a new phrase in its own right AND an infirm phrase as well.
- **FF_NEW_PHRASE** (0x00000020)
This flag indicates the current phrase is an entirely new phrase and any previous "infirm" phrases (having the **FF_INFIRM_PHRASE** flag) are now complete. This flag may be combined with the **FF_INFIRM_PHRASE** flag if the current phrase is both a new phrase in its own right AND an infirm phrase as well.

- **FF_UPPERCASE (0x00000040)**
This flag indicates the word following this phrase should be displayed using all uppercase characters.
- **FF_LOWERCASE (0x00000080)**
This flag indicates the word following this phrase should be displayed using all lowercase characters.

Dictation Control Frequently Asked Questions

This chapter contains answers to the most frequently asked questions about the ViaVoice **Dictation Control**.

When should I use the Dictation control?

VVDictation provides a simple means of just getting the text of what was said one phrase at a time. What you do with that text is up to you. You may or may not care about getting audio later and or doing correction in which case you could just destroy the RecoHandles provided and use the text. However, if you have or are attempting to build a word processor which is speech enabled on top of this object you will still have a great deal of work to do just to manage speech.

When should I use the DictationMgr control?

VVDictationMgr does almost everything you need to do normal document based dictation except show it to the user. You only have to keep the user interface synchronized with **VVDictationMgr** through character indices. If you are not voice enabling a word processor or memo type application then this may well be more than you really need. **VVDictationMgr** will also not be suitable for voice enabling applications where it is not possible to synchronize using character indices.

Getting Started with the Virtual Voices Control

This chapter contains basic information to help you get started using the **Virtual Voices** ActiveX control.

Overview

Virtual Voices is an ActiveX control that enables developers to incorporate personality into their applications. A personality is represented through a voice (using text-to-speech or prerecorded audio wave file) and an animated face. The voice and face become the spokesperson through which the user interacts with the application.

The **Virtual Voices** Control can be used within applications to provide many useful functions as well as to enhance the usability and overall appeal of the application. For example, an e-mail application could use the **Virtual Voices** Control to read unopened mail to the user. In an Internet browser, the **Virtual Voices** Control could read selected text. In a textbox, the **Virtual Voices** Control is used to read back dictated text. What's more, using speech recognition and text-to-speech, an application could "talk" to the user in the context of an ongoing dialog.

Virtual Voices includes a text-to-speech engine, **ViaVoice Outloud** that converts plain text to audible speech. **Virtual Voices** also includes an engine that animates the face. The animated face is synchronized with the text-to-speech or audio output.

Note:

The animated face is optional.

The **Virtual Voices** Control is built on Microsoft's Speech Application Programming Interface (SAPI) to provide text-to-speech within client applications. It runs with SAPI-compliant text-to-speech engines, but hides this interface layer from the application developer. With the **Virtual Voices** Control, developers have access to state-of-the-art text-to-speech capabilities without learning SAPI. They can manipulate the **Virtual Voices** Control using either high-level or low-level tools, and they can use it in Visual Basic, as well as in Visual C++ applications.

Since the **Virtual Voices** Control is an ActiveX control, it can, for example, be placed inside a Lotus Notes document, a Word document, or an Excel spreadsheet. The user can drop some highlighted text or audio on it, and the control will read it back to the user. Inserting the control into container applications enables text-to-speech to be used without having the applications to be rewritten specifically to handle text-to-speech.

As an example, to insert the **Virtual Voices** Control into a Lotus Note (you must have this version of the ViaVoice SDK installed), you can:

Start Lotus Notes (Release 4 or higher) and create a new memo. Position the cursor within the body of the memo.

- Select **Create**.
- Select **Object**.
- Select **Virtual Voices Control** and then click **OK**.

This procedure places the **Virtual Voices** Control in your memo at the current insertion point. You can set its properties while editing the memo, then send it to a colleague (who must also have the same version of the SDK installed on his or her machine). The person receiving the memo can hear the control speak by double-clicking it.

How the Virtual Voices Control Works

Since the **Virtual Voices** Control is an ActiveX control, it appears within a container (application) window. It can take on any of three forms: an icon, a face, or neither (that is, it can be invisible). As an icon, the **Virtual Voices** Control looks like this:

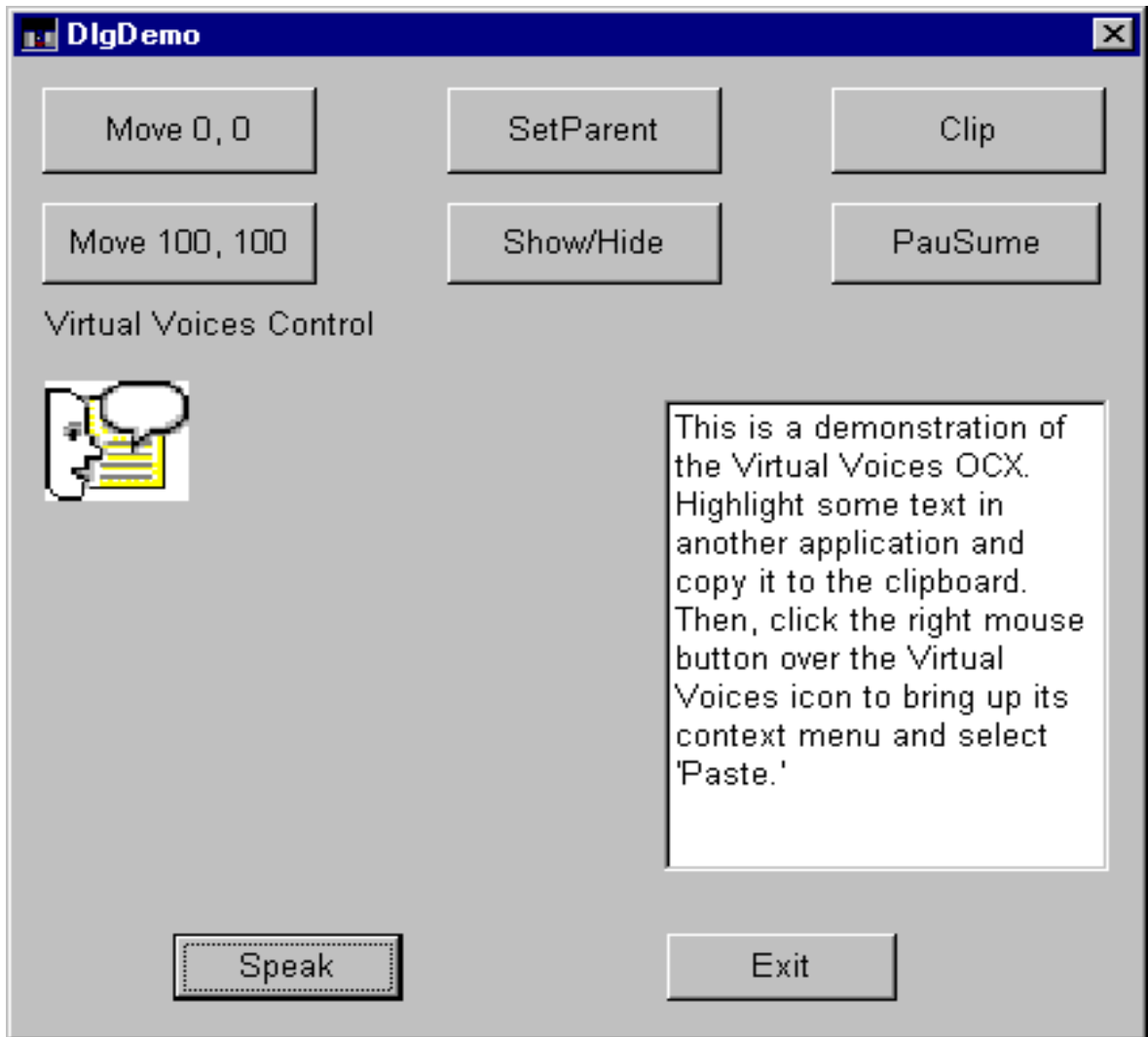


Figure 52. Virtual Voices as an Icon

Displayed as a face, the Virtual Voices Control looks like this:

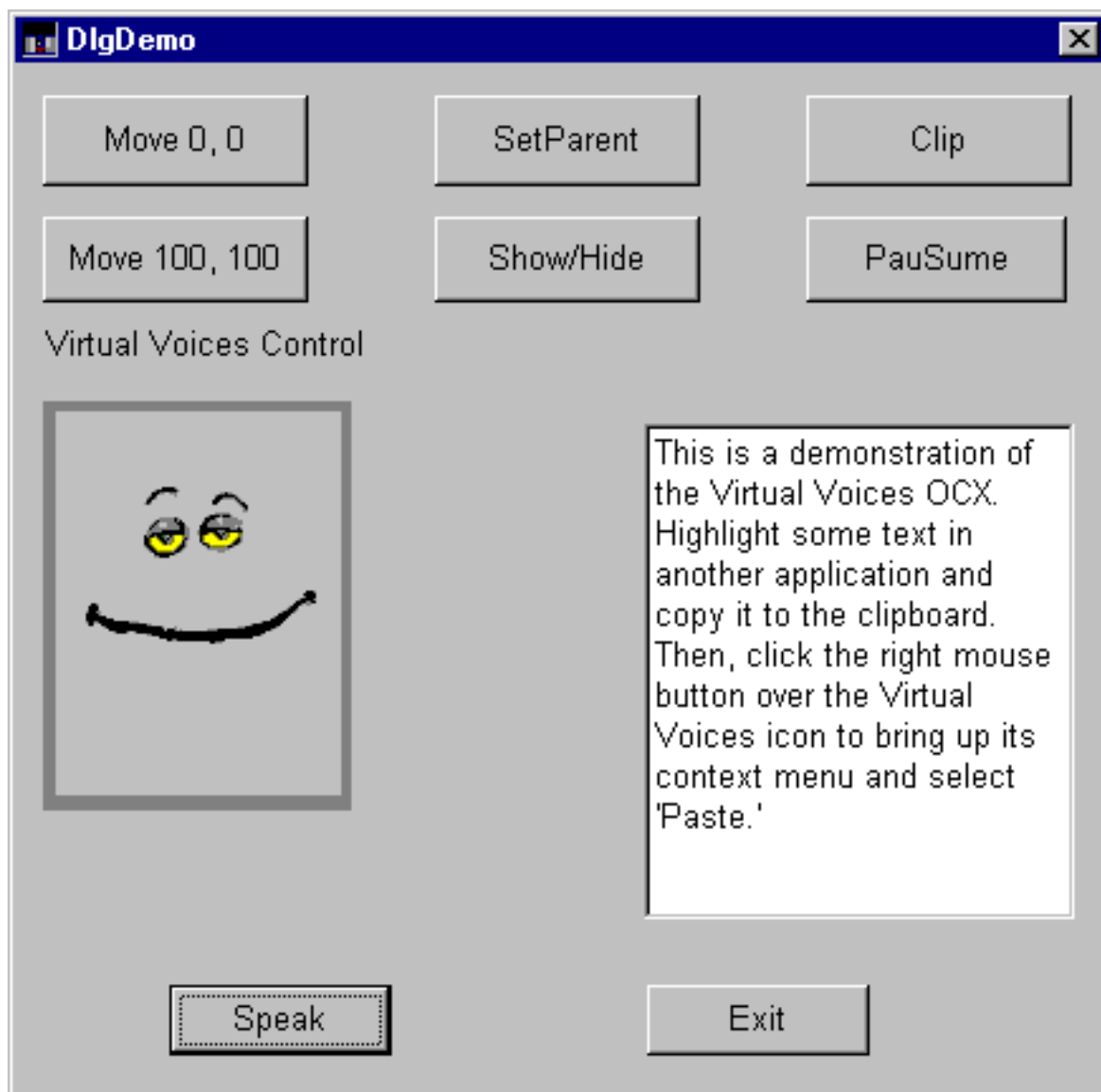


Figure 53. Virtual Voices as a Face

The face can also be clipped; that is, instead of appearing within a rectangular window, the window in which the face is displayed is “clipped” to the actor shape. If the control is clipped, and made a child of the desktop, it functions as a sprite. (A sprite is an animated, non-rectangular object that can be moved around the desktop.)

Faces are implemented in two different styles: as vector drawings or as bitmaps. There are eight faces provided with **Virtual Voices**. Four of them are implemented using vector drawings (Benny, Betty, Charlie, and Woodrow); the other four are implemented as bitmaps (Computer, Curly, Kincaid, and Kingsley).

To cause the **Virtual Voices** Control to speak, the user double-clicks the icon or the face. The control will speak the text specified in the **SpeakText** property (if **UseWave** is set to False), or the wave file specified in the **WaveFileName** property (if **UseWave** is set to True). The Control synchronizes the face animation with the speech output. The user can drag-and-drop text or audio over the control to cause it to speak.

The functions of the **Virtual Voices** Control are also available to the end user through a context menu. When the user places the pointing device cursor over the control (over its icon or, if a face is selected, over its face), and clicks the right mouse button, the context menu appears:

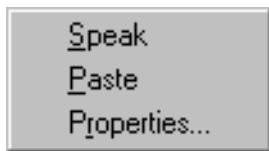


Figure 54. Virtual Voices Context Menu

The context menu contains three options: Speak, Paste, and Properties.

Speak

The first option, **Speak**, causes one of two things to happen: If the **UseWave** property is set to False, then the text stored in the control is converted to wave audio and is played back through the default wave audio device (usually, the system speaker). If the **UseWave** property is set to True, then the audio wave file pointed to by the **WaveFileName** property in the control is played back through the default

wave audio device. In either case, if a face has been enabled for the control, then the face is animated while speaking.

Paste

The second option, **Paste**, retrieves text or audio (if any) from the system clipboard, sets the corresponding property, and speaks it.

The end user can also place text or audio in the control by dragging and dropping it on the control. When text or audio is dropped on the control, it is spoken without further action required from the end user.

Properties

The third option, **Properties**, displays the **Virtual Voices Control Properties** pages. If the **Virtual Voices Control** property **AllowProperties** is set to False at design time, this option does not appear on the context menu. If **AllowProperties** is set to True, the end user can set the properties of the **Virtual Voices Control** through the property pages at run time. You can make the property pages appear in your code by issuing the **DoProperties** method. For more information, refer to “DoProperties” on page 793.

There are three tabs in the **Virtual Voices** properties dialog box: Voice Models, Text, and Actor Gallery.

Voice Models Page

Use the Voice Models page to select and configure the voice that will be used as shown below:

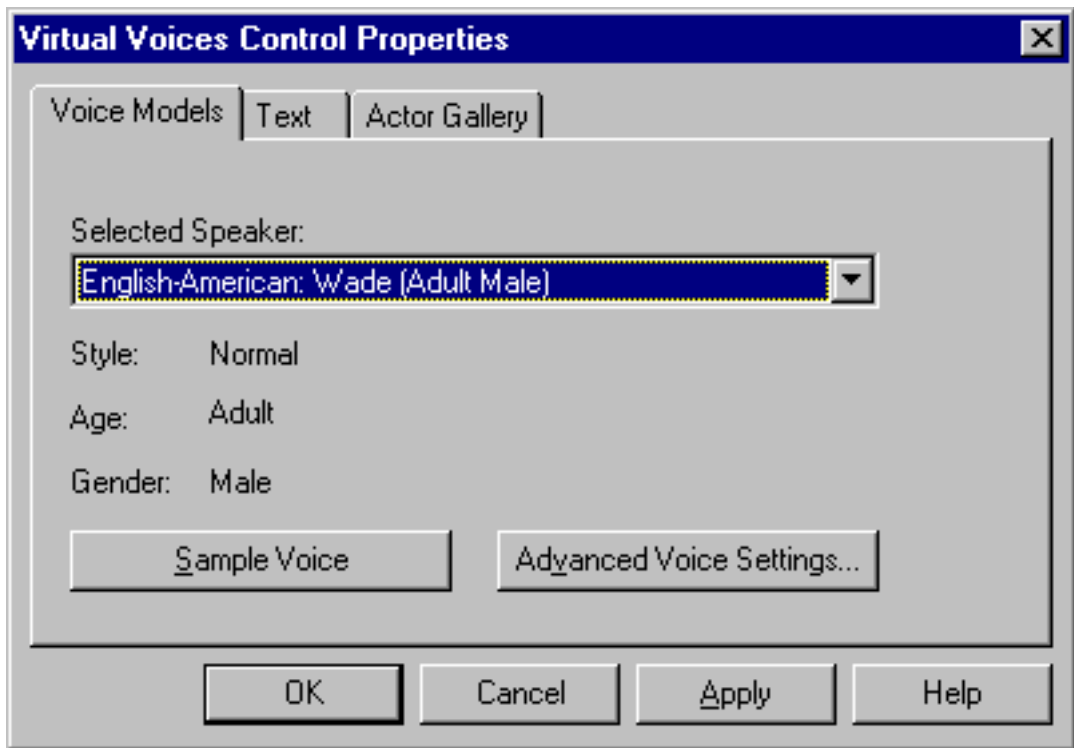


Figure 55. Virtual Voices Properties - Voice Models Page

The available voices are listed in the Selected Speaker list box.

Note:

The Style, Age, and Gender characteristics of the voice are defined by the **ViaVoice Outloud** (text-to-speech) engine and are provided for informational purposes only. They cannot be set by the developer or the end user.

Click **Sample Voice** button to hear a short sample of the selected voice. To customize the characteristics of the selected voice, click **Advanced Voice Settings**.

At design time, the Voice Models page also lets you set whether the Properties option will be available from the context menu (Allow Property Settings) and whether the context menu will be shown at run time (Show Context Menu).

Note:

These options are not visible to the end user.

Text Page

Use the Text page to set the type of speech output used, either text-to-speech (**Use Text**) or audio wave file (**Use Wave**). The two selections are mutually exclusive. You can set the text that will be spoken as shown below:

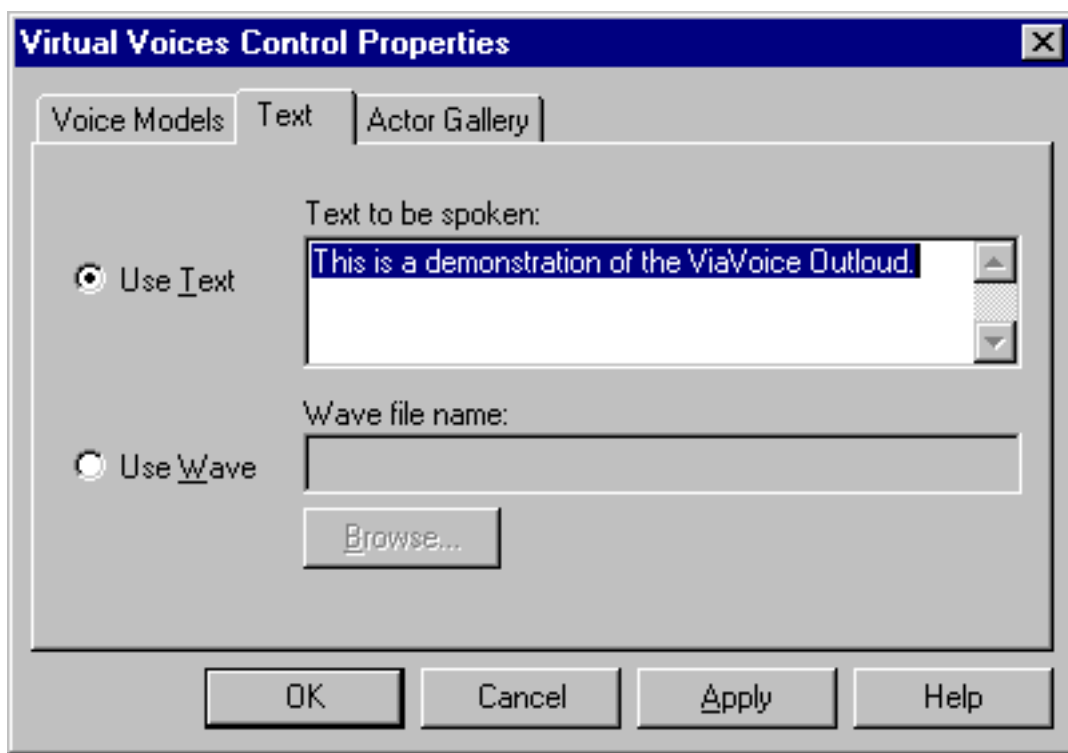


Figure 56. Virtual Voices Properties - Text Page Selecting Text

Or, you can specify the name of an audio wave file to be played as shown below:

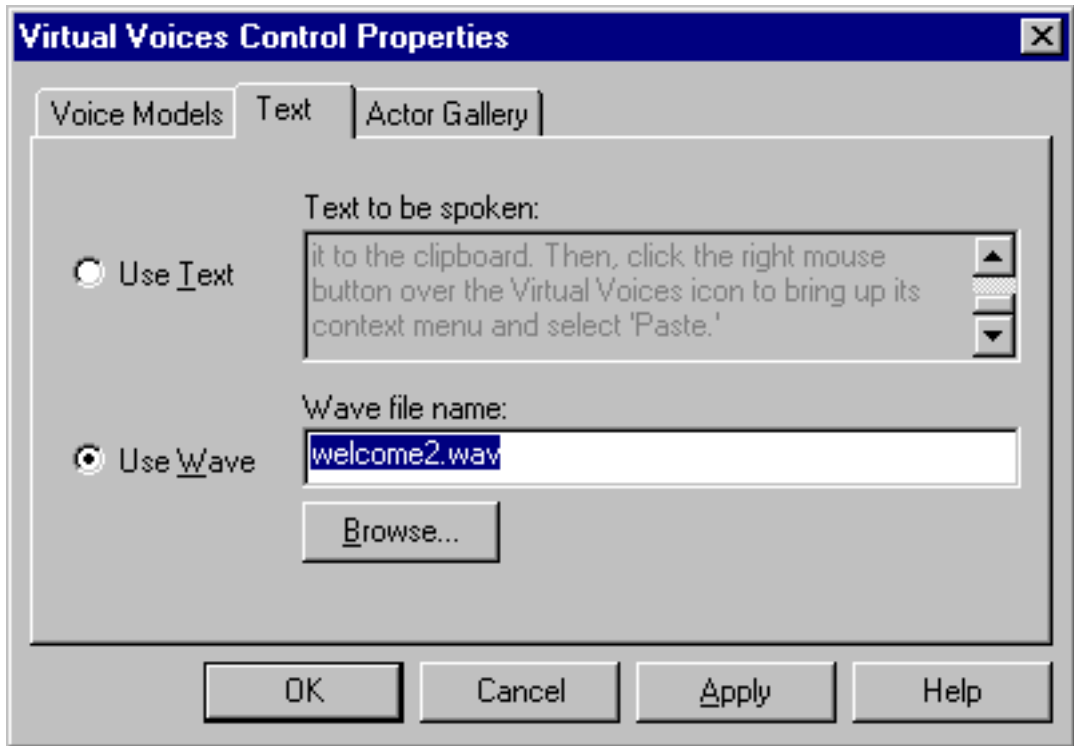


Figure 57. Virtual Voices Properties - Text Page Selecting Wave File

Note:

The **Virtual Voices** Control uses the **UseWave** property to determine which type of speech output to perform. If the **UseWave** property is set to True (that is, if it is selected), the control will play the specified audio wave file, even if there is text in the Text-to-Be-Spoken text box. Conversely, if the **UseWave** property is set to False (that is, if it is not selected), the control will convert the text (if any) typed in the Text-to-Be-Spoken text box to audible speech, even if there is an audio wave file specified in the wave file name text box.

Actor Gallery Page

Use the Actor Gallery page to select an actor face for the control as shown below:

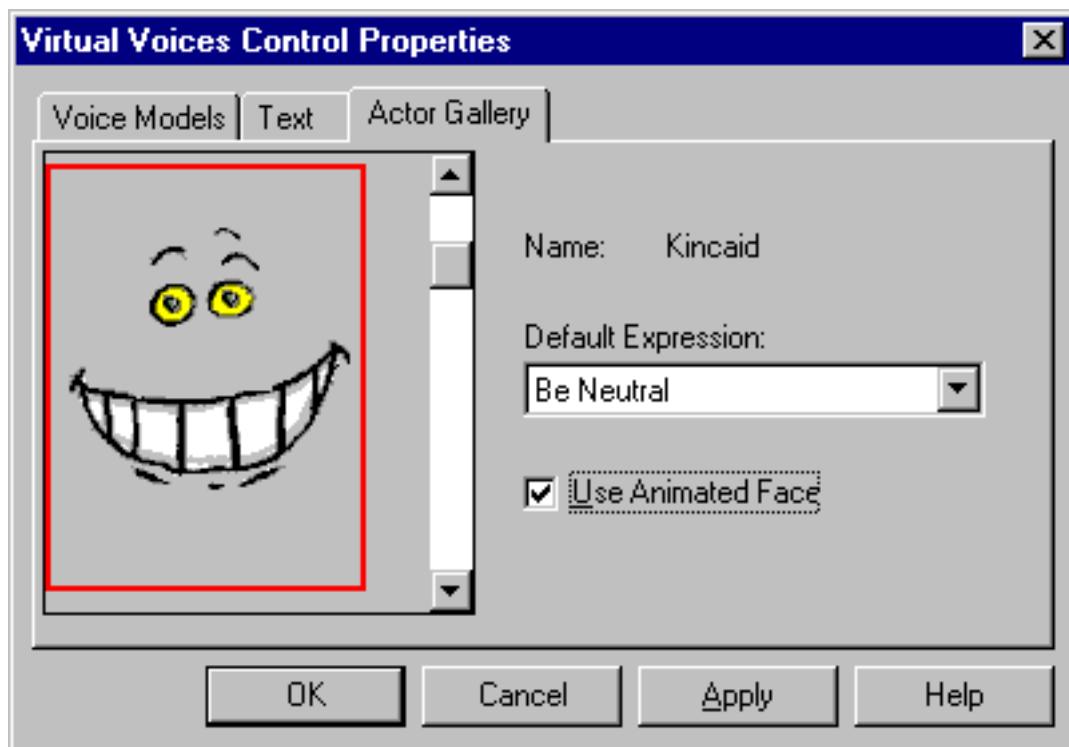


Figure 58. Virtual Voices Properties - Actor Gallery Page

On this page, you can also set the default expression (that is, the expression that will be used when the actor is not speaking). Note that the name cannot be set by the user or the developer.

To specify whether a face will be displayed, click **Use Animated Face**. If this box is not checked, the **Virtual Voices** icon will be shown instead of a face.

Programming Interfaces

The **Virtual Voices** Control provides a robust set of programming interfaces through which developers can incorporate text-to-speech into their applications. Most of these interfaces are available in both Visual C++ and Visual Basic.

There are three types of interfaces provided by the **Virtual Voices** Control:

- Methods to query and set **Virtual Voices** properties
- Methods to invoke **Virtual Voices** functions
- Events that signify relevant conditions have occurred in the **Virtual Voices** control

The following chapters describe these interfaces in detail and provide programming examples to illustrate how they are used.

Introduction to Virtual Voices Control

Files and Directories that Support Virtual Voices

The files necessary to support the **Virtual Voices** Control are installed with this version of the ViaVoice SDK. The following additional directories and files are included for **Virtual Voices**:

Bin	This directory is created under the run time directory. It includes the Virtual Voices Control and DLLs.
Data	This directory is created under the run time directory. It contains the data files used to define the actor faces.
Docs	This directory is created under the SDK directory. The documentation for this control is included in the ActiveX Developer's Guide.

Virtual Voices Control Properties

A **Virtual Voices** Control contains many properties through which the end user or the developer can customize its appearance and behavior. All **Virtual Voices** properties can be queried using a “Get” method. Properties that are not read-only can be set using a “Set” method. The following topics are provided for each of the **Virtual Voices** property:

- How a developer can query and set a **Virtual Voices** property
- Sample codes that show how to set a **Virtual Voices** property
- Other properties inherent to **Virtual Voices** because it is an ActiveX control. These properties can also be manipulated by the developer, causing interesting behaviors in the **Virtual Voices** Control.

The syntax for each property is offered for both Visual C++ and Visual Basic.

In Visual Basic:

The “Get” and “Set” methods for a property use the following format:

<property name> Returns the current value of the property

<property name>=value Sets the property to value

In Visual C++:

The “Get” and “Set” methods for a property use the following convention:

Get<property name> Returns the current value of the property

Set<property name>(value) Sets the property to value

ActorName

Gets/Sets the **ActorName** property of the control (that is, it selects an actor by name.)

Syntax

In Visual Basic:

```
ActorSvr1.ActorName="String of actor name"
```

In Visual C++:

```
void SetActorName(LPCTSTR)/LPCTSTR GetActorName(void)
```

Remarks

Each actor (or face) has a name. The names are listed in the Actor Gallery page of the **Virtual Voices** property pages. This method selects the actor by name.

Valid actor names are:

- Benny
- Betty
- Charlie
- Computer
- Curly
- Kincaid
- Kingsley
- Woodrow

Age (Read Only)

Returns a short Integer conforming to the TTSAGE_XXX attribute in the **TTSMODEINFO** structure of the SAPI specification.

Syntax

In Visual Basic:

```
ActorSvr1.Age
```

In Visual C++:

```
short GetAge(void)
```

Remarks

The TTSAGE_XXX values are as follows (from the SAPI 3.0 include the SPEECH.H file):

TTSAGE_BABY	1
TTSAGE_TODDLER	3
TTSAGE_CHILD	6
TTSAGE_ADOLESCENT	14
TTSAGE_ADULT	30
TTSAGE_ELDERLY	70

For more information, please see “Implementing Text-to-Speech in Applications” in the Microsoft Speech API Developer's Guide.

AllowProperties

Sets or gets the **AllowProperties** property.

Syntax

In Visual Basic:

```
ActorSvr1.AllowProperties=True
```

In Visual C++:

```
void SetAllowProperties(BOOL)/BOOL GetAllowProperties(void)
```

Parameters

AllowProperties

Boolean.

Return Values

TRUE

The end user can access the **Virtual Voices** property pages and change the properties of the control. (The user accesses the property pages through the context menu, if it is enabled.)

FALSE

The end user can access the **Virtual Voices** property pages only if the container calls the **DoProperties** method of the control.

Remarks

The developer can always access the **Virtual Voices** property pages.

BackColor

Sets or gets the background color of the control.

Syntax

In Visual Basic:

```
ActorSvr1.BackColor=vbBlack
```

In Visual C++:

```
void SetBackColor( OLE_COLOR )
```

Return Values

Returns the background color of the control.

Remarks

The animated faces are implemented in two different styles: as vector drawings or as bitmaps. Of the eight faces provided with **Virtual Voices**, four of them are implemented using vector drawings (Benny, Betty, Charlie, and Woodrow) and the others are implemented as bitmaps (Computer, Curly, Kincaid, and Kingsley). Therefore, if the control has an animated face, and it uses a vector model to draw the face (such as Woodrow), then the face background area adopts the background color. If the control uses a bitmap face (such as Kincaid), or if it does not use a face at all, then setting the background color has no visible effect.

Note:

The vector model face, by default, adopts the background color of the container in which it is placed, even though it does not show at design time. If you want the control to show a different color at run time, then use this property to set it.

OLE_COLOR is very similar to COLORREF, which is a 32-bit value of the form 0x00bbggrr. See the Microsoft OLE Automation Reference for more detailed information.

Clipping

Sets or gets the value of the control's **Clipping** property.

Syntax

In Visual Basic:

```
ActorSvr1.Clipping=True
```

In Visual C++:

```
void SetClipping(BOOL)/BOOL GetClipping(void)
```

Parameters

Return Values

TRUE

FALSE

Remarks

This property specifies whether the actor appears within a rectangular window, or whether the window is clipped to the actor shape.

The **Clipping** property interacts with the **UseFace** and **ActorName** properties. If **Clipping** is set to FALSE (the default value for new instances of the control), the control appears within a window. When visible, the control appears either as an actor or an icon within a borderless, rectangular window.

If **Clipping** is set to TRUE, and **UseFace** is FALSE, it has no effect on the icon. If **Clipping** and **UseFace** are both TRUE (and if the control is visible), the control's window is clipped to the shape of the actor. The clipping algorithm finds the transparent color at (0,0) within the actor dataset. It then clips out, or makes invisible, all adjacent pixels with the same value. If a selectable object is "beneath" the control's window, it is visible and selectable.

The **Clipping** property is made persistent by the container. Its value can be set by the developer at design time and changed at run time through `SetClipping()`. The end user does not have access to the **Clipping** property (that is, there is no property page control for it.)

Clipping can be set at any time. When the **ActorName** or **UseFace** properties are changed, the current **Clipping** value is applied to the new face. For example, if one actor is being shown with clipping when another actor is selected, the new actor is clipped. **Clipping** is supported in any screen color depth (8-, 16-, and 24-bit color) and display resolution.

Computer, Kincaid, and Kingsley do not support clipping.

DefaultExpression

Sets or gets the **DefaultExpression** property.

Syntax

In Visual Basic:

```
ActorSvr1.DefaultExpression=Integer
```

In Visual C++:

```
void SetDefaultExpression(short)/Short GetDefaultExpression(void)
```

Remarks

The default expression is shown whenever the control returns from its Speak method. The default expression is a short Integer with the following values:

Neutral	0
Happy	1
Thoughtful	2
Surprised	3
Asleep	4

Expression

Sets or gets the **Expression** property.

Syntax

In Visual Basic:

```
ActorSvr1.Expression=Integer
```

In Visual C++:

```
void SetExpression(short)/short GetExpression(void)
```

Remarks

If an animated face is visible at the time this method is called, the requested expression is shown. The face continues to display this expression while the control is speaking. When the control finishes speaking, it displays the **DefaultExpression**.

The expression is a short Integer with the following values:

Neutral	0
Happy	1
Thoughtful	2
Surprised	3
Asleep	4

Gender (Read Only)

Returns a short Integer conforming to the GENDER_XXX attribute in the **TTSMODEINFO** structure of the SAPI specification.

Syntax

In Visual Basic:

```
ActorSvr1.Gender
```

In Visual C++:

```
short GetGender(void)
```

Remarks

The GENDER_XXX values are as follows (from the SAPI 3.0 include the SPEECH.H file):

GENDER_NEUTRAL	0
GENDER_FEMALE	1
GENDER_MALE	2

For more information, please see “Implementing Text-to-Speech in Applications” in the Microsoft Speech API Developer’s Guide.

ModeGuid

Sets or gets the **ModeGuid** property in this instance of the control.

Syntax

In Visual Basic:

```
ActorSvr1.ModeGuid="A valid GUID string"
```

In Visual C++:

```
void SetModeGuid(LPCTSTR)/LPCTSTR GetModeGuid(void)
```

Remarks

A GUID is sometimes referred to as a CLSID. The definition of a GUID may be found in the Microsoft OLE Automation Reference.

Under SAPI, text-to-speech engines identify each of their “voice modes” with a separate globally unique identifier, or GUID. Each voice mode has different audible characteristics. (For instance, a vendor might support separate voice modes and languages for male and female voices.)

A GUID is a 128-bit number. These methods require a string representation of that number. Note that each GUID string begins and ends with curly brackets ({}). The following GUIDs are valid for the control:

English-American: Wade (Adult Male)	{BF5EAD40-9F65-11CF-8FC8-0020AF14F271}
English-American: Flo (Adult Female)	{BF5EAD41-9F65-11CF-8FC8-0020AF14F271}
English-American: Grandpa (Elderly Male)	{BF5EAD42-9F65-11CF-8FC8-0020AF14F271}
English-American: Grandma (Elderly Female)	{BF5EAD43-9F65-11CF-8FC8-0020AF14F271}
English-American: Bobbie (Child)	{BF5EAD44-9F65-11CF-8FC8-0020AF14F271}

English-American: Wade-Tel (Adult Male for Telephone)	{BF5EAD45-9F65-11CF-8FC8-0020AF14F271}
English-American: Flo-Tel (Adult Female for Telephone)	{BF5EAD46-9F65-11CF-8FC8-0020AF14F271}

If the **ModeGuid** property is not set before the **Speak** method is called, the control uses the first voice mode it finds on the end-user’s machine.

Pitch

Sets or gets the **Pitch** property.

Syntax

In Visual Basic:

```
ActorSvr1.Pitch=Integer
```

In Visual C++:

```
void SetPitch(long)/long GetPitch(void)
```

Remarks

Sets the baseline frequency of the text-to-speech voice to the pitch, in Hertz, that you specify. Allowable values for pitch are:

Male	33 to 150 Hertz
Female	60 to 200 Hertz

To determine the minimum pitch for a particular voice, set pitch to 0 and then query it. To find out the maximum pitch for a particular voice, set pitch to 0xffff and then query it.

ShowMenu

Sets or gets the **ShowMenu** property for the control.

Syntax

In Visual Basic:

```
ActorSvr1.ShowMenu=True
```

In Visual C++:

```
void SetShowMenu(BOOL)/BOOL GetShowMenu(void)
```

Remarks

If the **ShowMenu** property is set to TRUE, the end user can access the context menu by clicking the alternate select button over the control. Otherwise, the context menu does not appear, and the properties and behavior of the control maintain their design-time settings until they are changed by the container application (by setting a property).

SpeakText

Sets or gets the **SpeakText** property in this instance of the control.

Syntax

In Visual Basic:

```
ActorSvr1.SpeakText="text string"
```

In Visual C++:

```
void SetSpeakText(LPCTSTR)  
LPCTSTR GetSpeakText(void)
```

Remarks

The LPCTSTR is a pointer to an array of characters that are “spoken” by the **ViaVoice Outloud** (text-to-speech) engine when the **Speak** method is invoked or when the ActiveX verb “Edit” is invoked.

The **SpeakText** property is also set by dropping a text file on the control or by pasting text from the clipboard using the context menu.

The default value is “I don't have anything to say.” The maximum length of the text string is 64K, or 65535 bytes.

Speed

Sets or gets the **Speed** property.

Syntax

In Visual Basic:

```
ActorSvr1.Speed=Integer
```

In Visual C++:

```
void SetSpeed(long)/long GetSpeed(void)
```

Remarks

Sets the baseline average speed of the text-to-speech voice to the speed, in words per minute, that you specify. Valid values for speed range from 30 to 300. To find the minimum speed for a particular voice, set speed to 0 and then immediately query the speed. To determine the maximum speed for a voice, set speed to 0xffffffff and then immediately query the speed.

UseFace

Sets and gets the **UseFace** property so that an animated face can be used.

Syntax

In Visual Basic:

```
ActorSvr1.UseFace=True
```

In Visual C++:

```
void SetUseFace(BOOL)  
/BOOL GetUseFace(void)
```

Remarks

The Virtual Voices Control detects whether the text-to-speech engine supports the Visual method in the **ITTSNotify COM** interface. If it does, and this property is **TRUE**, then the selected actor's face is animated when the control is speaking. If it does not, then the face is visible in the control, but is not animated.

If this property is **FALSE**, then the control displays its icon.

The default value is **FALSE**.

UseWave

Sets the type of speech output to be used by the control.

Syntax

In Visual Basic:

```
ActorSvr1.UseWave=True
```

In Visual C++:

```
void SetUseWave(BOOL) /BOOL GetUseWave(void)
```

Remarks

If the **UseWave** property is set to TRUE, the control uses its wave audio when the **Speak** method is called. Otherwise, it uses its text.

The default value is FALSE.

Volume

Sets the **Volume** property.

Syntax

In Visual Basic:

```
ActorSvr1.Volume=Integer
```

In Visual C++:

```
void SetVolume(long) /long GetVolume(void)
```

Remarks

Sets the baseline speaking volume of the text-to-speech voice to the volume that you specify. The low-order word is the left channel, and the high-order word is the right channel. Volume can range from 0 to 0xffff for each channel, left and right.

WaveFileName

Sets or gets the name of the **Virtual Voices** Control audio wave file.

Syntax

In Visual Basic:

```
ActorSvr1.WaveFileName="file name"
```

In Visual C++:

```
void SetWaveFileName(LPCTSTR)  
/ LPCTSTR GetWaveFileName(void)
```

Remarks

The control does not “Speak” the wave audio unless its **UseWave** property is set to TRUE.

If the control can find and open the file, and if the file is a valid RIFF format wave file, it reads the contents of this file and closes the file. The maximum size of the wave file is limited by the amount of available memory, or DWORD, whichever is less.

The control makes no further use of this name. In particular, the control can be moved to other machines where this wave file might not exist, since the control uses the stored audio wave file to speak wave audio.

This property is also set when an audio wave file is dropped onto the control, or when a wave file name is entered on the property page.

ViaVoice Outloud (Text-To-Speech) Engine Attributes

Certain attributes of the **ViaVoice Outloud** (text-to-speech) engine are supported as persistent properties of the **Virtual Voices** Control. When modified at design time in Visual Basic or Visual C++, these attributes are stored with the application and restored at run time through property persistence. The end user can change these properties through the **Virtual Voices** Properties pages, and the control honors the new values if they are applied. The client application can also query and set these properties as though they are **Virtual Voices** properties through “Get” and “Set” methods. The text-to-speech attributes and their respective “Get” and “Set” methods are described in detail below.

The text-to-speech properties interact with the selected voice mode (**ModeGuid** property.) If the voice mode is changed, whether through the **SetModeGuid** method or through the **Virtual Voices** Properties pages, all the attributes of the new voice, including those of the **ViaVoice Outloud** engine-specific, are read from the SAPI engine and applied to the control's properties.

Breathiness

Sets or gets the **Breathiness** property of the IBM **ViaVoice Outloud** (text-to-speech) engine.

Syntax

In Visual Basic:

```
ActorSvr1.Breathiness Integer
```

In Visual C++:

```
void SetBreathiness(short) /short GetBreathiness(void)
```

Remarks

This property controls the amount of breathiness in the voice. A low value produces a voice with no breathiness. A high value adds significant breathiness. A value of 100 produces a whisper. If the current voice mode is not one of the IBM **ViaVoice Outloud** (text-to-speech) engine modes, the property is set, but has no effect.

Values range from 0 to 100, with 0 being minimum, and 100 being maximum, or a whisper.

HeadSize

Sets or gets the **HeadSize** property of the IBM text-to-speech engine.

Syntax

In Visual Basic:

```
ActorSvr1.HeadSize=Integer
```

In Visual C++:

```
void SetHeadSize(short)/ short GetHeadSize(void)
```

Remarks

This property controls the size of the head for the speaker, changing the pitch and acoustics of the voice. A large number indicates a large head and a deeper voice.

If the current voice mode is not one of the IBM **ViaVoice Outloud** (text-to-speech) engine modes, the property is set, but has no effect.

Values range from 0 to 100, with larger values indicating a larger head and thus a deeper voice.

PitchFluctuation

Sets or gets the **PitchFluctuation** property of the IBM **ViaVoice Outloud** (text-to-speech) engine.

Syntax

In Visual Basic:

```
ActorSvr1.PitchFluctuation=Integer
```

In Visual C++:

```
void SetPitchFluctuation(short)/ short GetPitchFluctuation(void)
```

Remarks

This property controls the amount of pitch change in the voice. A value of zero produces a voice with no pitch fluctuation, resulting in monotone speech. A high value produces a voice with large pitch fluctuation, which is typical of excited speech.

If the current voice mode is not one of the IBM **ViaVoice Outloud** (text-to-speech) engine modes, the property is set, but it has no effect.

Values range from 0 to 100, with 0 being minimum, or monotone, and 100 being maximum, or excited. There is no direct correlation to Hertz.

Roughness

Sets the **Roughness** property when running on the IBM **ViaVoice Outloud** (text-to-speech) engine.
Syntax

In Visual Basic:

```
ActorSvr1.Roughness=Integer
```

In Visual C++:

```
void SetRoughness(short)
```

Remarks

This property adds roughness to the voice, a quality of the vocal chords. A low value produces a smooth voice, while a high value produces a rough, or scratchy, voice.

If the current voice mode is not one of the IBM **ViaVoice Outloud** (text-to-speech) engine modes, the property is set, but has no effect.

Values range from 0 to 100, with 0 being minimum, or smooth, and 100 being maximum, or scratchy.

Example - Setting a Property

In Visual Basic:

The following example illustrates setting the **Expression** property in Visual Basic. The actor's expression is set to surprised (3) when speaking is canceled.

```
Private Sub Command2_Click()  
  
    ActorSvr1.Cancel  
  
    ActorSvr1.Expression = 3  
  
End Sub
```

Figure 59. Setting the Expression Property in Visual Basic

In Visual C++:

The following example illustrates setting the **Expression** property in Visual C++. In this example, the actor's expression is set to surprised (3) when speaking is interrupted, and to happy (1) when the actor speaks.


```
void CExample::OnSpeak()
{
    // TODO: Add your control notification handler code here...
    // When the Speak button is clicked, call the Virtual Voices
    // Speak method

    if ( m_bIsSpeaking ) // If busy speaking when clicked
    {
        switch (AfxMessageBox(LPCTSTR("Virtual Voices control is
        busy speaking."), MB_ABORTRETRYIGNORE | MB_ICONSTOP) )
        {
            case IDRETRY:        // Try to speak again
                OnSpeak();
                break;
            case IDIGNORE:
                m_VVCtrl.SetExpression(3); // Surprised when
                // interrupted
                m_VVCtrl.Speak(); // Interrupt busy control
                break;
        }                        // Abort cancels request
    }
    else
    {
        m_VVCtrl.SetExpression(1):    // Happy
        m_VVCtrl.Speak(); // Control is not busy,
        // so honor user's
        // request to speak
    }
}
```

Figure 60. Setting the Expression Property in Visual C++

Other Useful Properties

Because the **Virtual Voices** Control is an ActiveX control, you can manipulate some of the standard properties for the control, which can result in some interesting and desirable behaviors. For example, you might find it useful to set some of these properties for the **Virtual Voices** Control:

Visibility

By hiding the control's window, you can provide speech output within your application without displaying an additional interface element. The **Virtual Voices** Control can operate invisibly.

Note:

If the control is hidden when it comes up, the **InitDone** event is not triggered.

In Visual Basic:

Use the custom control **Visible** property:

```
If ActorSvr1.Visible = False Then
    ActorSvr1.Visible = True      \show it
Else
    ActorSvr1.Visible = False    \hide it
End If
```

Figure 61. Hiding and Showing the Virtual Voices Control Window in Visual Basic

In Visual C++:

Use the superclass method **SetWindowPlacement** as follows:

```

WINDOWPLACEMENT wndpl;
static BOOL showState=TRUE;

showState = !showState;

if ( m_VVCtrl.GetWindowPlacement( &wndpl ) )
{
    if ( showState )
        wndpl.showCmd = SW_SHOW;
    else
        wndpl.showCmd = SW_HIDE;
    if ( !m_VVCtrl.SetWindowPlacement( &wndpl ) )
        TRACE( "Error showing/hiding VV window\n" );
}

```

Figure 62. Hiding and Showing the Virtual Voices Control Window in Visual C++

Setting the Parent Window

The control's parent window can be set by the client application. Because the control is a subclass of `CWnd`, methods on the base class can be called by the client application. When the control is clipped and made a child of the desktop, it functions as a sprite. (A sprite is an animated, non-rectangular object that can be moved around the desktop.)

In Visual Basic:

To accomplish the same task in Visual Basic, you must first declare the following APIs:

GetDesktopWindow

GetFocus

SetParent

Then write the following:

```
Dim hwndActor As Long
ActorSvr1.SetFocus ` where ActorSvr1 is an instance of the actor control
hwndActor=GetFocus()
Call SetParent(hwndActor, GetDesktopWindow())
```

Figure 63. Setting the Parent Window in Visual Basic

In Visual C++:

To set the parent of the control, write the code as shown below:

```
CActorSvr m_VVCtrl;
m_VVCtrl.SetParent( GetDesktopWindow() ); // puts control on
        // desktop
m_VVCtrl.SetParent( this ); // returns control to
        // client window
```

Figure 64. Setting the Parent Window in Visual C++

Moving the Control on the Desktop

The control can be moved around on the desktop. Because the control is a subclass of CWnd, methods on the base class can be called by the client application.

In Visual Basic:

First declare the following:

MoveWindow

After you declare the API, write the following code:

```
`Add to the previous code segment
`you must save the window handle of
`the object before doing a SetParent, then:
Call MoveWindow(hwndActor,0,0,120,160,True)
```

In Visual C++:

To move the control around on the desktop, call the superclass **Move** method:

```
RECT rect;  
rect.left = rect.top = 0;  
rect.right = 120;  
rect.bottom = 160;  
m_VVCtrl.Move( &rect ); // moves the control
```

Figure 65. Moving the Control in Visual C++

The only size supported for an actor is 120 by 160 pixels.

Virtual Voices Control Methods

The **Virtual Voices** Control encapsulates several functions that developers can use to incorporate text-to-speech into their applications. These functions are invoked through methods called on the control and are available in both Visual Basic and Visual C++.

The syntax of each method is represented using the following conventions:

In Visual Basic:

type <MethodName>(parameters)

In Visual C++:

<MethodName>

Example code in both Visual C++ and Visual Basic are provided at the end of the section to illustrate how to call these methods from a client application.

AboutBox

Calls the **AboutBox** method in the control.

Syntax

In Visual Basic:

```
type <MethodName>(parameters)
```

In Visual C++:

```
<MethodName>
```

Remarks

This method causes the Virtual Voices Control “About” box to be shown.

Cancel

Calls the **Cancel** method in the control.

Syntax

In Visual Basic:

```
ActorSvr1.Cancel
```

In Visual C++:

```
BOOL Cancel(void)
```

Remarks

This method returns TRUE if the control was speaking at the time this method was called, and if the speaking was successfully stopped. It also triggers the **Reset** event, and resets the control's internal **Pause/Resume** state. Otherwise, the **Cancel** method returns FALSE.

DoProperties

Calls the **DoProperties** method in the control.

Syntax

In Visual Basic:

```
ActorSvr1.DoProperties
```

In Visual C++:

```
void DoProperties(void)
```

Remarks

This method causes the **Virtual Voices** properties dialog to appear, regardless of the values of **ShowMenu** and **AllowProperties**. The end user can then interactively change the values of many of the Virtual Voices properties.

Pause

Calls the **Pause** method in the control.

Syntax

In Visual Basic:

<code>ActorSvr1.Pause</code>

In Visual C++:

<code>BOOL Pause(void)</code>

Remarks

If the control was speaking either text-to-speech or wave audio, this method pauses playback and returns TRUE. Otherwise, it returns FALSE.

Use the **Resume** method to continue.

Resume

Calls the **Resume** method in the control.

Syntax

In Visual Basic:

```
ActorSvr1.Resume
```

In Visual C++:

```
BOOL Resume(void)
```

Remarks

If the control was paused, this method resumes either text-to-speech or wave audio and returns TRUE. Otherwise, it returns FALSE.

Speak

Calls the **Speak** method in the control.

Syntax

In Visual Basic:

```
ActorSvr1.Speak
```

In Visual C++:

```
BOOL Speak(void)
```

Remarks

This method causes one of two things to occur: If the **UseWave** property is set to FALSE, the stored text (if any) is converted to wave audio and played through the default system audio output device; if the **UseWave** property is set to TRUE, the stored audio (if any) is played through the default system audio output device.

Speak returns a Boolean value. TRUE indicates that the request was successful. FALSE indicates that an error has occurred. If **Speak** is successful, the **StartSpeaking** and **StopSpeaking** events are triggered.

Speak appends **SpeakText** to any outstanding speaking by the same instance of the **Virtual Voices** Control. If another instance of the control is speaking, or if some other multimedia application is using the audio output device, **Speak** causes a “busy” message to appear, and the application (or end user) must retry the **Speak** request. In this case, **Speak** returns FALSE.

Example - Using a Method

In Visual Basic:

The following example illustrates using the **Pause** and **Resume** methods in a Visual Basic program. In this example, Command1 is a button object which, when clicked, alternatively pauses and resumes speaking.

```
Private Sub Command1_Click()  
  
    If Paused Then  
        ActorSvr1.Resume  
    Else  
        ActorSvr1.Pause  
    End If  
  
    Paused = Not Paused  
End Sub
```

Figure 66. Calling the Pause and Resume methods in Visual Basic

In Visual C++:

Using the **Speak** method, write the following:

```
void CExample::OnSpeak()
{
    // TODO: Add your control notification handler code here...
    // When the "Speak" button is clicked, call the Virtual Voices
    // Speak method

    if ( m_bIsSpeaking ) // If busy speaking when clicked
    {
        switch ( AfxMessageBox(LPCTSTR("Virtual Voices control is busy
            speaking."), MB_ABORTRETRYIGNORE | MB_ICONSTOP) )
        {
            case IDRETRY: // Try to speak again
                OnSpeak();
                break;
            case IDIGNORE:
                m_VVCtrl.Speak(); // Interrupt
                break;
        } // Abort cancels request
    }
    else
    {
        m_VVCtrl.Speak();           // Control is not busy, so honor
            // user's request to speak
    }
}
```

Figure 67. Calling the Speak() method in Visual C++

Virtual Voices Control Events

The **Virtual Voices** Control triggers events when certain conditions are detected. The developer can implement event handlers for those events in which the application is interested.

A typical event handler sets a state variable to some value and returns. For instance, an event handler for **StartSpeaking** might set a Boolean for “busy” to TRUE, and an event handler for **StopSpeaking** might set the same variable to FALSE. The client application consults the value of this variable to detect whether the control is busy speaking or not.

In general, the event handlers should not make calls back into the control, nor should they execute a lot of time-consuming code.

This chapter describes the **Virtual Voices** events and includes examples illustrating how to handle these events within a client application. The syntax for each event is offered for both Visual C++ and Visual Basic. For Visual C++ developers, events take on the following format:

In Visual Basic:

```
Private Sub ActorSvr1_<EventName>( parameters )
```

In Visual C++:

```
type on<EventName>( parameters )
```

BookMark

Notifies the client application that a bookmark has been encountered in the text to be spoken.

Syntax

In Visual Basic:

```
Private Sub ActorSvr1_BookMark(ByVal dwMarkNum As Long)
```

In Visual C++:

```
void BookMark(long dwMarkNum)
```

Remarks

You can embed bookmarks in the text to be spoken by following the SAPI rules for bookmark tags. Reference the Microsoft SAPI Developer's Guide for more information on bookmarks.

This event is generated only when speaking text, not wave audio.

Example

In Visual C++:

Handling the **BookMark** event in Visual C++, write the following:


```

// This example detects a bookmark in a text string and displays it // in
a message box. To run this example, first create an input // area on
your form. In the input area, type "This is the first // day of the rest
of \Mrk=9876\ your life." (You can use any number // for the bookmark).
// Select this sentence and copy it to the clipboard. Bring up the
// Virtual Voices context menu and select Paste. The sentence is
// spokenby the control, after which a message box displays the
// bookmark.
// Note: For this example, declare textstr in your object as
// protected, char textstr[100].
void CExample::OnBookMark(long dwMarkNum)
{
    // TODO: Add your control notification handler code here...
    // Create a text string that displays the bookmark number
    char myBuf[10];
    itoa(dwMarkNum, myBuf, 10);
    strcpy(textstr, "A bookmark has been encountered. It is ");
    strcpy(textstr+strlen("A bookmark has been encountered. It is "),
        myBuf);
    strcpy(textstr+strlen("A bookmark has been encountered. It is ")
        +strlen(myBuf), "\n");
}
// OnStopSpeaking, show the text string using a message box
// with this code...
void CExample::OnStopSpeaking()
{
    // TO DO: Add your control notification handler here...
    // Display the text string if there is one
    if (textstr != "")
    {
        MessageBox(textstr);
        strcpy(textstr, "");
    }
}

```

Figure 68. Handling the BookMark Event in Visual C++

InitDone

Notifies the container application.

Syntax

In Visual Basic:

```
Private Sub ActorSvr1_InitDone()
```

In Visual C++:

```
void InitDone(void)
```

Remarks

When the control has been initialized and is ready to speak, it uses the **InitDone** event to notify the container application. Your application should wait for this event before calling the **Speak** method for the first time.

This event is not triggered if the control is invisible when it comes up.

Example

In Visual C++:

Handling the **InitDone** event, write the following:

```
// This event is triggered by the Virtual Voices control, when it
// has completed its initialization and is ready to speak

void CExample::OnInitDone()
{
    // TODO: Add your control notification handler code here...
    CString lpszString;

    // Get the text out of the edit box control
    m_EditBox.GetWindowText( lpszString );

    // Set the text into the Virtual Voice control's speak text property
    m_VVCtrl.SetSpeakText( (LPCTSTR) lpszString.GetBuffer(0) );

    // Call the Virtual Voices Speak method
    m_VVCtrl.Speak();
}
```

Figure 69. Handling the InitDone Event in Visual C++

KeyPress

Event fired when the user pauses or resumes speaking by pressing a key on the keyboard.

Syntax

In Visual Basic:

```
Private Sub ActorSvr1_KeyPress(KeyAscii As Integer)
```

In Visual C++:

```
void KeyPress(short *KeyAscii)
```

Remarks

If the control has the keyboard focus, the end user can pause and resume speaking by pressing a key on the keyboard. If your application wants to be aware of such events, you can write a method that is fired when this event occurs. Your method receives the ASCII value of the pressed key. If the end user presses the **Esc** key when the control is speaking, speaking is aborted and the **StopSpeaking** event is fired. **KeyPress** is not returned for system keys, such as the function keys.

Example

In Visual C++:

The following example illustrates how to handle the **KeyPress** event. To run this sample, highlight some text and copy it to the clipboard. Bring up the Virtual Voices Control context menu and select Paste. Press a key (other than Esc or a function key) while the control is speaking. The control says “A key has just been pressed.”

```
// This example causes the control to speak a message when a key has
// been pressed.

void CExample::OnKeyPress(short FAR* KeyAscii)
{
    // TODO: Add your control notification handler code here...
    m_VVCtrl.SetSpeakText("A key has just been pressed.");
    m_VVCtrl.Speak();
}
```

Figure 70. Handling the KeyPress Event in Visual C++

Pause

Notifies the client application that the control has just been paused.

Syntax

In Visual Basic:

```
Private Sub ActorSvr1_Pause()
```

In Visual C++:

```
void Pause(void)
```

Example

In Visual C++:

To handle the **Pause** event, write the following:

```
// This example displays a message box when the control is paused

void CExample::OnPause()
{
    // TODO: Add your control notification handler code here...
    MessageBox("The Virtual Voices control has been paused.");
}
```

Figure 71. Handling the Pause Event in Visual C++

Reset

Notifies the client application that the control has just been reset.

Syntax

In Visual Basic:

```
Private Sub ActorSvr1_Reset()
```

In Visual C++:

```
void Reset(void)
```

Remarks

Reset is generated if the user presses the **Esc** key while the control is speaking, if the Properties dialog is brought up when the control is speaking, whenever the **SetUseFace** property is set to TRUE, and whenever speaking is interrupted for any reason.

Example

In Visual C++:

To handle the **Reset** event, write the following:

```
// This example displays a message box when the control is reset.

void CExample::OnReset()
{
    // TO DO: Add your control notification handler here...
    MessageBox("The Virtual Voices control has been reset.");
}
```

Figure 72. Handling the Reset Event in Visual C++

Resume

Notifies the client application that the control has just been resumed.

Syntax

In Visual Basic:

```
Private Sub ActorSvr1_Resume()
```

In Visual C++:

```
void Resume(void)
```

Remarks

Resume works with longer text. If the control is paused while speaking and there is very little remaining to be said, the **ViaVoice Outloud** (text-to-speech) engine does not resume.

Example

In Visual C++:

```
// This example displays a message box when the control is resumed.

void CExample::OnResume()
{
    // TO DO: Add your control notification handler here...
    MessageBox("The Virtual Voices control has been resumed.");
}
```

Figure 73. Handling the Resume Event in Visual C++

StartSpeaking

Notifies the container application when the control starts speaking.

Syntax

In Visual Basic:

```
Private Sub ActorSvr1_StartSpeaking()
```

In Visual C++:

```
void StartSpeaking(void)
```

Remarks

Whenever the control starts speaking, it uses the **StartSpeaking** event to notify the container application. It is recommended that you maintain a Boolean variable to indicate whether the control is speaking. Your **StartSpeaking** method sets this variable to TRUE, while the **StopSpeaking** and **Reset** methods set it to FALSE. Don't forget to initialize it to FALSE.

Example

In Visual C++:

```
// This event is triggered by the Virtual Voices Control, when it
// starts to speak

void CExample::OnStartSpeaking()
{
    // TODO: Add your control notification handler code here...
    // Set the busy speaking flag
    m_bIsSpeaking = TRUE;
}
```

Figure 74. Handling the StartSpeaking Event in Visual C++

StopSpeaking

Notifies the container application when the control stops speaking.

Syntax

In Visual Basic:

```
Private Sub ActorSvr1_StopSpeaking()
```

In Visual C++:

```
void StopSpeaking(void)
```

Remarks

Whenever the control stops speaking, it uses the **StopSpeaking** event to notify the container application. See the discussion of the **StartSpeaking** event for recommendations on how to use this event in your application.

Example

In Visual C++:

```
// This event is triggered by the Virtual Voices Control, when it has
// stopped speaking

void CExample::OnStopSpeaking()
{
    // TODO: Add your control notification handler code here...
    // Set the (not) busy speaking flag
    m_bIsSpeaking = FALSE;
}
```

Figure 75. Handling the StopSpeaking Event in Visual C++

WordPosition

Notifies the client application that a byte offset has been reached in the text to be spoken.

Syntax

In Visual Basic:

```
Private Sub ActorSvr1_WordPosition(ByVal dwByteOffset As Long)
```

In Visual C++:

```
void WordPosition(long dwByteOffset)
```

Remarks

This event is generated only when speaking text, not wave audio.

Example

In Visual C++:

```
// This example displays the index number of the first letter of the
// last word spoken. Note: For this example, declare textstr in your
// object as protected, char textstr[100] and wordindex as protected
// long.
void CExample::OnWordPosition(long dwByteOffset)
{
    // TODO: Add your control notification handler code here...
    // Store the index of the word spoken in wordindex
    wordindex=dwByteOffset;
}

void CExample::OnStopSpeaking()
{
    // TODO: Add your control notification handler code here...
    // Display the index of the last word in a message box
    char myBuf[20];
    itoa(wordindex, myBuf,10);
    strcpy(textstr,"The index of the last word said is ");
    strcpy(textstr+strlen("The index of the last word said is "),
        mybuf);
    strcpy(textstr+strlen("The index of the last word said is ")
        +strlen(myBuf), "\n");
    MessageBox(textstr);
}
```

Figure 76. Handling the WordPosition Event in Visual C++

Running multiple versions of the **Virtual Voices** Control at the same time is not supported. If you want to try this, be aware that each control competes for the audio device when speaking. Also, multiple instances of the control may interfere with the color palette when using animation and running in 256-color mode. Three is the maximum number of actors allowed per application.

Visual Basic Notes

If you want to give keyboard control of the **Virtual Voices** Control to the end user, add the following code to the click handler for the Speak button:

```
ActorSvr1.SetFocus
```

This directs keyboard focus to the control, even when it is animated.

Visual C++ Notes

If you use Class Wizard to produce a message map for **Virtual Voices** events, Class Wizard appends the control's ID to the default function names it generates. For example, if your control is identified as IDC_VVCTRL1 and you map the **InitDone** event, the default function name generated by Class Wizard is OnInitDoneVvctrl1(). You can change the function names if you don't like the default names generated by Class Wizard.

Face Customization Notes

The **Virtual Voices** Control enables developers to incorporate personality into their applications. A personality is represented through a voice (using text-to-speech or prerecorded audio waveforms) and an (optional) animated face. The voice and face become the spokesperson through which the user can interact with the application or system.

Virtual Voices includes an engine that animates the face. The animated face can convey several expressions and emotions, and it is synchronized with the text-to-speech or audio output.

Faces are implemented in two different styles: as vector drawings or as flip books. Eight faces are provided with **Virtual Voices**. Four of the characters provided are implemented using vector drawings (Benny, Betty, Charlie, and Woody); the other four characters are implemented using the flip-book style (Computer, Curly, Kincaid, and Kingsley).

This document describes how to create additional flip-book style faces for **Virtual Voices**. First, some background information on the flip-book style is in order.

A flip-book face is represented through a series of bitmaps. These bitmaps, when superimposed over each other in particular sequences, convey the appearance of motion and thus, animation. Consider a child's picture book of individual, yet subtly distinct cartoon drawings. When you flip through the pages quickly, the drawings, even though they are still, appear to animate. This same concept is used to provide the animation for the flip-book, or bitmap, faces. The **Virtual Voices** animation engine handles displaying ("flipping through") the appropriate series of bitmaps to convey expression and emotion as the face speaks. It also handles synchronizing the face with the spoken text.

Resources

A flip-book face is represented through the following resources:

- A series of bitmap (.BMP) drawings
- A face (.FAC) file
- A parameter (.PAR) file
- An entry in the Windows 95 registry

As the creator of a new face, you must provide all of these resources so that **Virtual Voices** can recognize and use your face.

To know how to test your new face, please see the “Testing Your Face” on page 831 and “Style Considerations” on page 833.

Bitmaps

A face is represented as a series of Windows bitmap (.BMP) files, which must be 120 pixels wide by 160 pixels high and must be drawn using a 256-color palette. However, it does not mean that you need to use all 256 colors in your bitmaps. In fact, you should not use more than 236 colors. (You should ensure that the system colors occupy the upper and lower 10 palette positions.)

Note:

The fewer colors you actually use in your bitmap, the better it can adapt to the other color applications on the desktop.

The animation engine uses double buffering to display the bitmaps for a face. It composes the next image in an offscreen buffer, then moves it to the display. It first puts bitmaps at plane 0 into the buffer, then bitmaps at plane 1, and so on. If images at different planes overlap, the one at the higher plane number will obscure the image below it.

The bitmap at plane 0 is the base bitmap for the face. Typically, the base bitmap is a full-face drawing. This bitmap is displayed unconditionally (always), and other bitmaps are superimposed over it to create the animation.

Bitmap faces are not sizable; however, they can be displayed as a non-rectangular window by setting a property in the Control at run time. If you choose to use the non-rectangular window feature, the background around the face must be all the same color.

Each face is assumed to have eyes and a mouth, which, however is not required. You may design your face to represent whatever you want: a car, a pet rock, or a rainbow. The position and shape of the eyes and mouth are used to animate the face. Even if your character does not have eyes or a mouth, the bitmaps you provide should include these parameters. The variation in the bitmaps (it could be a change in color instead of a change in eye position) will convey the expression for your face. You may want to review the resource files for the Computer chip face (ia01*.*) for an implementation of a face without eyes or a mouth. Keep in mind, though, that the face is used to speak to the end user, and will be more vivid and convincing if you give it eyes and a mouth.

Virtual Voices supports five expressions for each face: neutral, happy, surprised, thoughtful, and asleep. You should develop bitmaps for all five expressions. For each expression, you can specify several eye and mouth positions. For example, you can show the eyes looking straight ahead, as if paying attention; you can show the eyes looking up and to the character's right, as if thinking; and you can show the eyes closed, as if blinking. You can show the mouth shaped as if to speak certain sounds,

such as “oo,” “m,b,p” or “f,v,” or you can position it in several degrees of openness as well as closed. These mouth positions add realism to the animation as the face is synchronized with its text-to-speech or audio output.

You can provide as few bitmaps as you like. Do not provide more than 70 bitmaps for a single face. In some cases, you may need to reuse the same eyes and mouths for more than one expression.

The more eye and mouth positions you support (that is, for which you provide bitmaps), the more realistic and smooth the animation will appear (since the transition from one expression to another will occur in smaller increments).

In summary, you should provide the following bitmaps for each face:

Full face (for use in the Actor Gallery and as the base face):

- Mouth making “oo” sound
- Mouth making “f,v” sound
- Mouth making “m,b,p” sound

For each expression:

- Eyes looking straight ahead
- Eyes looking up
- Eyes looking down
- Eyes looking left
- Eyes looking right
- Eyes $\frac{1}{2}$ open
- Eyes $\frac{3}{4}$ open
- Eyes closed
- Mouth full open
- Mouth $\frac{1}{4}$ open
- Mouth $\frac{2}{4}$ open
- Mouth $\frac{3}{4}$ open
- Mouth closed

The only full-face bitmap you provide as a resource to **Virtual Voices** is the one to use at plane 0 (the base face or background). The rest of the bitmaps are cutouts of the eyes and mouth. All of the cutouts for the eyes should be in the same size (that is, they should have the same x, y, cx, and cy values), as should all the cutouts for the mouth.

Note:

The mouth positions for “oo”, “m,b,p” and “f,v” are independent of expression. You need only provide one bitmap for each of these sounds (for a total of 3 bitmaps).

An Example

Let's use one of the existing characters, Kincaid, as an example. First, your artist should create a full-face drawing of your character. This is usually the face drawn with a neutral or happy expression. This bitmap is used by Virtual Voices in the Actor Gallery (from the Properties dialog) where the user selects your face. It can also be used as the base bitmap (at plane 0) for the animation engine. Working with Kincaid, the bitmap displayed in the Actor Gallery is:



Your artist should develop full-face poses for all five expressions (neutral, happy, surprised, thoughtful, and asleep). Each pose should be created in the same size using the same palette. In each pose, the areas surrounding the eyes and mouth should be as small as possible (for better run-time performance), and they should be in the same size and should cover the same area. After creating these full-face drawings, the artist should cut out the areas around the eyes and mouth and save them as separate bitmaps. For example, the following bitmaps for Kincaid represent the different cutouts of the eyes for the thoughtful expression:



Eyes open looking straight ahead



Eyes open looking up



Eyes open looking up



Eyes open looking left



Eyes open looking right



Eyes $\frac{1}{2}$ open



Eyes $\frac{3}{4}$ open



Eyes closed

Notice that the eyes are represented by the same basic expression (eye shape, arched eyebrows, and scrunched forehead). The differences in each bitmap are reflected in the eye position.

The mouth positions for the thoughtful expression are also specified for Kincaid. These are represented in the following bitmaps:



Mouth full open



Mouth $\frac{3}{4}$ open



Mouth $\frac{1}{2}$ open



Mouth $\frac{1}{4}$ open



Mouth closed

Let's look at the differences between expressions for a particular eye position. Your artist can depict the expressions by varying such attributes as eye shape, eye size, eyebrow position and arch, forehead scrunched, etc.



Eyes open looking straight ahead - Surprised



Eyes open looking straight ahead - Neutral



Eyes full open, looking straight ahead - Happy



Eyes open looking straight ahead - Thoughtful

Expressions can also be depicted by the position and shape of the mouth. For example, let's look at the full open mouth position across expressions for Kincaid.



Mouth full open - Neutral



Mouth full open - Happy



Mouth full open - Thoughtful



Mouth full open - Surprised

For Kincaid, there are 62 bitmaps provided. The following tables illustrate how the bitmaps were defined for this particular character. Notice how some bitmaps are used for more than one eye or mouth position.

Table 23. General bitmaps (not related to specific expressions):

Character ID	Ia02
Background	Ia02-bas
“oo”	Ia02mnoo
“f,v”	Ia02mnf
“m,b,p”	Ia02mnm

Table 24. Eye positions:

	Neutral	Happy	Surprised	Thoughtful
Straight Ahead	Ia02enop	Ia02enop	Ia02ewop	Ia02ecop
Up	ie02endn	ia02ehup	ia02ewup	ia02ecup
Down	ie02endn	ia02ehdn	ia02ewdn	ia02ecdndn
Left	ia02enlt	ia02ehlt	ia02ewlt	ia02eclt
Right	ia02enrt	ia02ehrt	ia02ewrt	ia02ecrt
3/4 Open	ia02en23	ia02eh23	ia02ew23	ia02ec23
1/2 Open	ia02en12	ia02eh12	ia02ew12	ia02ec12
Close	ia02encl	ia02ehcl	ia02ewcl	ia02eccl

Table 25. Mouth positions:

	Neutral	Happy	Surprised	Thoughtful
Full Open	Ia02mn4	Ia02mh4	Ia02mw4	Ia02mc4
3/4 Open	ie02mn3	ia02mh3	ia02mw3	ia02mc3
1/2 Open	ie02mn2	ia02mh2	ia02mw2	ia02mc2
1/4 Open	ia02mn1	ia02mh1	ia02mw1	ia02mc1
Close	ia02mncl	ia02mhcl	ia02mw4cl	ia02mccl

You may want to use a similar set of tables to develop the bitmaps for your face.

Tips for the Artist

As you can see, creating all of the bitmaps for the face is a very elaborative and manual process. It is worthy to invest the time and effort that can help you narrow down the possibilities. The face design and creation process should be iterative and should involve customer feedback on an ongoing basis to validate the design and acceptability of the face.

First, create full-face drawings of many different characters or designs. You may want to provide faces which vary in form (e.g., human/mechanical, male/female, and anthropomorphic) and style (e.g., cartoonish, photorealistic, and stylistic). These initial renderings can be brought to customers to evaluate their acceptability.

When the set of faces is narrowed down to a few (from user testing and marketing evaluations), you should create pose sheets for these faces. A pose sheet is a set of 5 full-face drawings, one for each expression. Again, the pose sheets can be shown to customers to get feedback.

When you've decided which face you want to create, you can start creating the individual bitmaps. The best way to do this is to work off copies of the base (full-face) bitmap. You can use a tool like PhotoShop to make a copy of the base bitmap, and to make the appropriate changes to the copy (for example, to create a happy face with eyes fully open and mouth fully open). When you've finished with that bitmap, make another copy of the base bitmap, and start again making changes to the bitmap for other expressions, and eye and mouth positions. If you work off copies of the base bitmap, you are guaranteed that everything will be in the right position and be the right size.

Don't cut out any of the eyes or mouths until you're finished with all of your full-face bitmaps.

Once you have all of the bitmaps and cutouts, you are ready to create the face (.FAC) file, which tells the **Virtual Voices** when and how each bitmap should be displayed.

Face (.FAC) File

Now that you have created your bitmaps, you need to tell **Virtual Voices** when to display each bitmap. The face (.FAC) file provides the animation engine with this information. It specifies the bitmap file name and the condition(s) under which the bitmap is to be displayed - for instance, the expression, eye, or mouth position.

The face file is comprised of a series of text statements. The types of text statements within a face file are:

- Comments
- Style definition (MUST BE THE FIRST NON-COMMENT STATEMENT)
- Number of image fragments (MUST BE THE SECOND NON-COMMENT STATEMENT)
- Image definition statements
- Condition definition statements
- Parameter file definition (MUST BE THE LAST NON-COMMENT STATEMENT)

Syntax for the statements in a face file is:

- Any line beginning with # is considered a comment.
- The first statement defines the style of the face and is specified as 2 integers (the first specifies flip book or vector; the second specifies whether the bitmap contains scenery. For flip book faces, the first integer should be 10, and the second integer should be 0.
Note: Scenery is not supported in the flip-book style.)
- The second statement defines the number of image fragments defined within the face file, and is specified as a single integer.
- Image definition statements which identify the image to be displayed. These statements are of the form:

x y w h p filename num
where:

x and y	Integers which specify that the image is displayed at this offset from the origin (lower left corner).
w and h	Integers which specify the width and height of the image fragments (Note: w must be a multiple of 4).
p	An integer which specifies the image plane, or display priority (0 is displayed first, or lowest).

filename	A text filename of the .BMP file which contains the image data.
num	An integer which specifies how many condition definition statements follow for this image. These statements are logically AND'ed together to comprise the condition under which the image is displayed.

Note:

You must define at least one image as the base bitmap at plane 0. If only one base bitmap image is defined (num is 0), it will be displayed unconditionally (always). You may want to define multiple images to be displayed at plane 0 conditionally (for example, if there is a wide variance in eye and mouth positions and sizes between expressions, you may want a full-face bitmap of each expression displayed as the base bitmap for the expression.

- Condition definition statements specify the condition under which the image is displayed. Conditions are specified in terms of the parameters that can be varied for each face (for example, eye opening and mouth width, such as:

i min max
where:

i	an integer which identifies the parameter (reference the PAR file)
min	an integer which specifies the minimum value of this parameter for which this image will be displayed
max	an integer which specifies the maximum value of this parameter for which this image will be displayed

Note:

The image is displayed if $\text{min} < \text{current value} \leq \text{max}$ for all conditions. (Multiple conditions are logically AND'ed together.)

- The final command specifies the parameter (PAR) file to be used for this face.

Let's take a look at the face file provided for Kincaid for more details:

```
...
# style 10 (flip book)
10 0(1)
# number of image fragments
70(2)
...
0 0 120 160 0 IA02-BAS.BMP 0(3)
...
# surprised or anything else(4)
# eyes wide open ahead
0 96 120 64 1 IA02ewop.BMP 4
53 6.9 11.1
30 0.9 2.0
40 -1.0 1.0
41 -0.8 0.15
...
IA02.PAR(5)
```

Notes:

- (1) This is the first statement that is not a comment. It specifies that this face file is being used for a flip-book style face (10) and that no scenery is included in the bitmaps (0).
- (2) This is the second statement that is not a comment. It specifies that there will be 70 image fragments defined for this face.
- (3) This statement defines the bitmap to be used at plane 0, which is the base bitmap. It is positioned at 0,0 and is 120 pixels wide by 160 pixels high. For Kincaid, this bitmap is a full-face with a smile.
- (4) The next 69 sets of statements identify under which conditions a bitmap is displayed. This particular set of statements is identifying how and when the bitmap IA02ewop.BMP is displayed. The first line specifies that bitmap IA02ewop.BMP will be displayed at 0, 96 (offset from the origin, which is the lower-left corner of the bitmap) and it is 120 pixels wide and 64 pixels high. The 1 indicates that the bitmap is displayed at plane 1. The 4 indicates that the next 4 statements identify the condition under which this bitmap is displayed. In this case, the IA02ewop.BMP is displayed when the expression is surprised or asleep (parameter 53 between 7 and 11) AND the eyes are full to exaggerated open (parameter 30 between 0.9 and 2.0) AND the eyes are looking left, straight ahead, or right (parameter 40 between -1.0 and 1.0) AND the eyes are at a level gaze (parameter 41 between -0.8 and 0.15).

(5) This statement identifies the parameter (PAR) file for this face; in this case, IA02.PAR.

More about Conditions and Parameters

There are over 50 parameters you can use in your face file to tell the animation engine when to display a bitmap. You will not need to use most of them (the defaults used by the engine will provide quite acceptable animation.) The four flip-book faces included with **Virtual Voices** use the following parameters to specify when individual bitmaps are displayed:

- Mouth shape (13)
- Jaw rotation (16)
- Mouth width (17)
- Eye opening (30)
- Eyes looking left or right (40)
- Level of gaze (41)
- Expression (53)

Recall that conditions are specified as a range of values for a parameter (that is, the bitmap is displayed if the minimum value specified is less than the current value which is also less than or equal to the maximum value specified ($\text{min} < \text{current} \leq \text{max}$). So, you need to specify the condition as a range rather than a distinct value. For example, if you want to specify that a bitmap is displayed when the expression (53) is happy (1), you don't just use the value for happy (1), such as:

```
53 1.0
```

You need to specify it as a range, such as:

```
53 0.9 1.1
```

This tells the engine to display a bitmap when the current value for expression is greater than 0.9 but less than or equal to 1.1 (e.g., 1.0).

Following is a list of the parameters that are used in the face files for the character included with **Virtual Voices** as well as example ranges used in the face file to trigger conditions. You may need to finetune these to the behavior you experience with your face animated through **Virtual Voices**.

13 - Mouth shape [0..2]

-0.1 to 0.1 -- Normal (0)

.9 to 1.1 -- “f,v” (1)

1.1 to 2.1 -- “m,b,p” (2)

16 - Jaw rotation [0..1]

-0.1 to .09 -- Closed

.11 to .3 -- $\frac{1}{4}$ Open

.3 to .53 -- $\frac{1}{2}$ Open

.53 to .83 -- $\frac{3}{4}$ Open

.83 to 1.0 -- Full Open

17 - Mouth width [0..1]

-.01 to .79 -- “oo” sound

.8 to 2.0 -- All other sounds

30 - Eye opening [0..1.1]

.9 to 2.0 -- Full Open

.65 to .9 -- $\frac{3}{4}$ Open

.2 to .65 -- $\frac{1}{2}$ Open

-0.1 to .2 -- Closed

40 - Eyes looking left or right [-1..1]

-4.0 to -1.0 -- Left (-1)

0 -- Straight ahead (0)

1.0 to 4.0 -- Right (1)

41 - Level of gaze [-20..20]

-100 to -0.8 -- Down

-.8 to .15 -- Level

.15 to 100 -- Up

53 - Expression [0..15]

-0.1 to .1 -- Neutral (0)

0.9 to 1.1 -- Happy (1)

5.9 to 6.1 -- Thoughtful (6)

8.9 to 9.1 -- Surprised (9)

10.9 to 11.1 -- Asleep (11)

Parameter (.PAR) File

The parameter (.PAR) file defines the default values of the various parameters that are used by the animation engine to define and display a face. It also defines the default feature (eye, mouth, eyebrow, etc.) positions to be used for each expression in the resting state.

After you create your bitmaps and face file, you should test your face using the default parameter file. Copy one of the parameter files for the bitmap faces that were included with **Virtual Voices**, and use this as your parameter file.

Registry Entry

Virtual Voices uses the Windows 95 registry to store information about each face. This information is stored under the HKEY_LOCAL_MACHINE\SOFTWARE\IBM\VirtualVoices\3.5\Actors entry.

There is a unique entry for each character. The following information must be stored in the registry for a face to be recognized and used by **Virtual Voices**:

- Path to the faces bitmap (usually ViaVoice\data)
- Long name of the character (for descriptive purposes only)
- Short name of the character (displayed in the Actor Gallery)
- A description of the character (for descriptive purposes only)
- Introductory message (the default text that is spoken by the character)
- Dataset (.FAC file)
- Bitmap (Bitmap file used in the Actor Gallery)

For example, the information registered for Kincaid is (your entry may vary):

[HKEY_LOCAL_MACHINE\SOFTWARE\IBM\VirtualVoices\3.5\Actors\Kincaid]:

Pathname=D:\Viavoice\data

LongName=Kincaid B. Funface

ShortName=Kincaid

Description=A bitmap face on a rough surface with big teeth. Male voice...

IntroMessage=I'm fun to work with! Call me Kincaid.

Dataset=IA02.FAC

Bitmap=IA02.BMP

Testing Your Face

There are two types of testing you should conduct for your face: Functional and Usability/Acceptability. For functional testing, the best approach is to use the **Virtual Voices** OCX as the test vehicle. Here's how:

- First, create all of the resources for your face (bitmaps, FAC file, PAR file).
- Edit the registry to include the appropriate information for your face.
- Run the **Virtual Voices** OCX. From the context menu, select **Properties**. On the Actor Gallery page, your face should be included in the set of faces that are available. Select your face (you may have to scroll through the list of available faces.)
- On the Voice Models page, make sure that the "Use Animated Face Engine" is selected. Click **OK**. You should now see your face as the face of **Virtual Voices**.
- Test each expression by selecting it from the Properties page. Have the face speak text that is representative of the expression (for example, a thoughtful expression might be used to ask questions of the user. So, use a question as the text to speak for the thoughtful expression.) For each expression, consider the following:
 - Is the animation smooth?
 - Were all of the bitmaps displayed?
 - Does the expression convey the emotion you intended?
 - When the face says a word with the "oo" sound, is the correct bitmap used? Same for "f,v" and "m,b,p."
 - Others...

Your face should have potential users of your product to validate its acceptability and utility. You may want to show users early renderings of face designs to narrow down possibilities and to identify preferences and expectations. You should iterate customer feedback as the design evolves. You can start with drawings on paper, but you should eventually show the face in action to your users, too (so they can see it expressing emotion and hear it speaking text).

By getting feedback from actual users and potential customers of your product, you can evaluate and validate your face on several dimensions that:

- The face is acceptable;
- The name is acceptable;
- The voice "matches" the face;

- The expressions are meaningful and recognizable; and
- Others...

Style Considerations

The type and style of faces and characters that can be used to personalize your application is virtually limitless. However, there are some things that should be considered when designing a face for your application:

- Understand the requirements of your audience (users). End users may prefer characters of different styles than power users; children have different preferences than adults; some users prefer no face at all. It is important to allow users to change the face (provide a library of faces and voices from which they can choose). It is also very important to allow users to turn the face on and off as desired.
- Consult marketing on the design of your face. The face should be consistent with the product image and marketing strategy.
- Understand that photorealistic (human) faces imply intelligence, truthfully, is not here. Also, with the current text-to-speech technology, the voice does not sound human yet, which can create cognitive dissonance for the end user.
- Consider the appropriateness of the character for the environment in which it will be used (e.g., business, game, home, and education). A cartoonish character may be quite appropriate for a game but may be too silly for business use.

Virtual Voices Control Frequently Asked Questions

This chapter contains answers to the most frequently asked questions about the ViaVoice **Virtual Voices** Control.

Can I create my own actors?

Yes. More information is available in “Face Customization Notes” on page 814. However, IBM does not support this.

I am using the Virtual Voices control and the VVTextBox control in the same form. When I click the VVTextBox, I want Virtual Voices to speak the text in the VVTextBox. However, when I issue the Speak command, I get an error: audio source busy. What am I doing wrong?

The **VVTextBox** control and the **Virtual Voices** control share the same audio source. Before you issue the **Speak** command in the Virtual Voices control, you need to tell the **VVTextBox** control to stop using the audio source. To accomplish this, set **AutoDictation** to False and **CommandEnabled** to False. These settings will cause the **VVTextBox** to stop using the audio source and allow **Virtual Voices** to use it.

I do not want my users to change actors or any other speech-related settings at run time. How do I keep the properties menu option from appearing?

To keep the properties menu option from appearing, set the property **AllowProperties** to False.

Is it better to use bitmap-based actors or vector graphic actors?

Vector-based actors have the following benefits:

- Smaller source files. (This might be a consideration for downloading the files via the Internet.)
- More motion. Vector-based actors do not have pre-drawn expressions. As the character speaks, the control reshapes the expression of the character at run time. Therefore, the control can match the expression to the text with more accuracy.

Can I have more than one actor at the same time?

Yes, however, only one can speak at a time. There are no known problems with two characters speaking if one **Virtual Voices** instance control instance waits for the other to finish before speaking.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program or service is not intended to state or imply that only that IBM product, program, or service may be used.

Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service.

The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department T01B, 3039 Cornwallis, Research Triangle Park, NC 27709-2195, USA. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

IBM

ViaVoice

VoiceType

Adobe Acrobat is a trademark or registered trademark of Adobe Systems Incorporated.

Intel and Pentium are trademarks or registered trademarks of Intel Corporation in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Other company, product, and service names may be trademarks or service marks of others.

Index

A

AboutBox method, **791**

accessing

member of a collection, **250**

object information, **580**

action

control not firing events for, **629**

description, **261**

informing the UIClient about, **556**

ActionDesc

property, **261**

ActiveX components

installing, **19**

Actor

controlling, **767**

gender, **769**

head size, **781**

shape, **764**

ActorName property, **760**

Add method, **253**

AddApplicationByName method, **566**

AddApplicationByWindow method, **568**

adding

command phrases, **212**

custom menu options, **534, 535, 570, 587**

DictLite to HTML document, **363**

DictLite to HTML page, **363**

Error Correction to application, **417**

Grammar to application, **281**

GrammarLite to HTML document, **368**

GrammarLite to HTML page, **368**

new phrase, **253**

phrases at design time, **216**

Phrases control to application, **205**

phrases to a collection, **234**

PhrasesLite to HTML document, **374**

PhrasesLite to HTML page, **374**

TextBox to application, **29**

VVDictation to application, **693**

VVDictationMgr to application, **633**

VVPhrases control to application, **205**

VVRichEdit to application, **87**

VVTextBox to application, **31, 89**

AddPhrase method, **234**

adult voices, **769**

Age property, **761**

AllowProperties property, **762**

Alternates property, **296**

amount of space to indent, **109**

animated face

controlling, **767**

enabling, **775**

Annotations property, **298**

annotations, purpose of, **359**

apostrophe, allowed in a phrase, **277**

appearance of UIServer, constants that

describe, **555**

AppendMenuItem method, **570**

application

changes, UIClient event, **612**

executable name, **566**

menu group, adding menu option, **587**

window handle parameter, **593**

ApplicationMenuName parameter, **625**

ApplicationName parameter, **566, 591, 612**

ApplicationTitle parameter, **624**

architecture, SDK, **25**

associating

word with related information, **298**

attributes

Classes, editing, **209**

ViaVoice Outloud engine, **779**

audio

wave, using, **776**

AutoConnect property, **222, 303**

AutoDictation property, **42, 46, 100, 104, 106**

AutoDictationWindow property, 642, 702
AutoLoad property, 305
automatic
 connection to speech engine, 222, 303
 connection to UIServer, 223
 loading of a grammar file, 305
 starting of dictation mode, 42, 46, 83, 100, 104, 106, 201, 642, 702
AutoUI property, 223, 307

B

BackColor property, 763
background color of the control, 763
bActive parameter, 612
baseline frequency, setting, 771
Begin Dictation, enabling in main menu, 587
beginning
 speech recognition, 240
 text, moving to, 36, 94
BeginSpeechRecognition event, 240
BeginSpeechRecognized
 method, 337
BIN directory, 755
binary grammar file
 getting/setting path, 317, 320
binary snapshot of a control, 229
BookMark event, 800
borderless,rectangular window, 764
Breathiness property, 780
bShow parameter, 77, 196
BulletIndentation property, 109
buttons
 checkmark, 39, 98
 EventButtonPressed event, 615
 microphone, 497, 520, 629
 Sample Voice, 751
 toolbar, setting state of, 554

C

Cancel method, 792
capitalizing selected text, 36, 94
Caps Lock, turning off, 37, 95
Caption property, 534
capture
 dictation speech, 382, 390, 400, 403, 405
capturing
 commands, 36, 94
 messages from the UIServer, 568
 speech, 35, 93
 VVDictation, 697
 VVDictationMgr, 639
causing UIServer to appear, 585
changing
 command words, 36, 94
 selected text to lowercase, 36, 94
 selected word to uppercase, 37, 95
 state of UIServer component, 606
characters
 LangID, using last two, 509
 MAX_MENU_OPERATION limiting, 546
 maximum allowed, 80, 199
 MaxText event, 80, 199
 menu, number allowed in, 546
 non-printing ASCII, 277
 permitted in a phrase, 277
 vvMAX_MENU_STRING constant limiting, 546
check mark indicator for menu option, 534
Checked property, 534
checkmark button, error correction dialog box, 39, 98
child's voice, 769
ciComponent parameter, 580, 582, 606, 609, 615, 617
class
 User Interface control, 534
Class Wizard, 34, 92, 638
Classes
 wrappers, 524

classes

- Confirm Classes dialog box, **285, 421**
- editing attributes, **209**
- Index constants, **538**
- vvDVAF_ALLOW_DOCK, **540**
- vvUIDockingAlgorithmConstants, **540**
- vvUIDockingStyle, **542**
- vvUIEventCallbackFlags, **543**
- vvUIExtendedMenuFlags, **545**
- vvUIMaxConstants, **546**
- vvUIMenuInfo, **534**
- vvUIMenuItemConstants, **547**
- vvUIRemoveClientConstants, **550**

client application, notifying, **806, 807**

clipboard

- cutting selected text to, **36, 94**
- pasting text from, **37, 95**

Clipping property, **764**

CmdID parameter, **70, 191**

collection

- accessing a member of, **250**
- accessing member of, **250**
- number of phrases in, **246**
- PhraseColl methods, **245, 252**
- removing a member, **257**
- removing all objects, **259**

collection object, VVPhrasesColl, **214**

collection of commands used, **49, 111**

comma, allowed in a phrase, **277**

Command event, **70, 191**

Command method, **660**

command phrases

- enabling/disabling, **214**
- turning off recognition of, **214**
- vvTBCapitalizeThis, **36, 94**
- vvTBCopy, **36, 94**
- vvTBCorrectionThis, **36, 94**
- vvTBCut, **36, 94**
- vvTBDeleteThis, **36, 94**
- vvTBHideEC, **36, 94**
- vvTBLowercaseThis, **36, 94**

- vvTBMoveBeginning, **36, 94**

- vvTBMoveEnd, **36, 94**

- vvTBNextWord, **36, 95**

- vvTBPasteThis, **37, 95**

- vvTBPreviousWord, **37, 95**

- vvTBScratchThat, **37, 95**

- vvTBSelectText, **95**

- vvTBSelectThis, **37, 95**

- vvTBShowEC, **37, 95**

- vvTBUppercaseOff, **37, 95**

- vvTBUppercaseOn, **37, 95**

- vvTBUppercaseThis, **37, 95**

command recognition

- disabling, **309**
- enabling, **309**
- turning off, **248**
- turning on, **248**

commands

- , **387**
- adding phrases, **212**
- capturing, **36, 94**
- disabling recognition, **225**
- enabling recognition, **225**
- programmer-assigned, descriptions of, **263**
- VVRichEdit recognition, **191**
- VVTextBox recognition, **70**

Commands property, **49, 111**

CommandsEnabled property, **52, 114**

communicating

- actions to UIServer, start, **597**

COMPID_CUSTOM constant, **554**

COMPID_MAINMENU constant, **554**

COMPID_MICROPHONE constant, **554**

COMPID_USERINFORMATION constant, **554**

COMPID_VOLUME constant, **554**

COMPID_WORDHISTORY constant, **554**

Confirm Classes dialog box, **209, 285, 421**

connecting

- automatically, **222**
- speech engine automatically, **303**

- constants, 546
 - COMPID_CUSTOM, 554
 - declared in VVUITYPE.h & VVUICNST.h, 504
 - Index, 538
 - UIAPIRC_ERROR_ALREADYINITIALIZE D, 559
 - UIAPIRC_ERROR_FAILED, 559
 - UIAPIRC_ERROR_INVALIDCLIENT, 559
 - UIAPIRC_ERROR_INVALIDPARAM, 559
 - UIAPIRC_ERROR_NOSERVER, 559
 - UIAPIRC_ERROR_NOTCURRENTCLIEN T, 559
 - UIAPIRC_ERROR_OUTOFMEMORY, 559
 - UIAPIRC_ERROR_SERVERBUSY, 559
 - UIAPIRC_OK, 559
 - UIMFG_DYNAMIC_APPLICATION, 557
 - UIMFG_DYNAMIC_HELP, 557
 - UIMFG_DYNAMIC_MAIN, 557
 - UIMFG_STATIC_HELP, 557
 - UIMFT_CLIENT, 558
 - UIMFT_EXECUTE, 558
 - UIMSF_ADDWAIT, 553
 - UIMSF_CLEARWAIT, 553
 - UIMSF_DISABLED, 552
 - UIMSF_ERROR, 552
 - UIMSF_OFF, 552
 - UIMSF_ON, 552
 - UIMSF_REMOVEWAIT, 553
 - UIMSF_SLEEP, 552
 - User Interface control, 537
 - vvDVAF_ALLOW_DOCK, 540
 - vvDVAF_ALLOW_TOPMOST_DOCK, 540
 - vvDVAF_DEFAULT, 540
 - vvDVAF_NEVER_DOCK, 541
 - vvDVAF_STAY_DOCK_TO_PREVIOUS, 541
 - vvDVSF_ADJUST WIDTH, 542
 - vvDVSF_ADJUST_ORIGIN, 542
 - vvDVSF_DEFAULT, 542
 - vvDVSF_NORMAL_BACKGROUND, 542
 - vvDVSF_TRANSPARENT_BACKGROUN D, 542
 - vvMAX_MENU_OPERATION, 546
 - vvMAX_MENU_STRING, 546
 - vvMAX_USERINFO_ID_LEN, 546
 - vvMAX_WORDHISTORY_TEXT, 546
 - vvUIDockingStyle, 542
 - vvUIEVENT_ACTIVEAPP_CHANGED, 543
 - vvUIEVENT_ALL, 543
 - vvUIEVENT_BUTTON_PRESSED, 543
 - vvUIEVENT_COMPONENT_UPDATED, 543
 - vvUIEVENT_MENUITEM_SELECTED, 543
 - vvUIEVENT_NONE, 543
 - vvUIEVENT_VIEW_QUERYFLAGS, 544
 - vvUIEVENT_VIEW_QUERYMENUINFO, 544
 - vvUIEventCallbackFlags, 543
 - vvUIMaxConstants, 546
 - vvUIMenuItemConstants, 547
 - vvUIMICINDEX_MICSTATE, 538
 - vvUIMICINDEX_WAITSTATE, 538
 - vvUIRCF_CLOSE, 550
 - vvUIRCF_CLOSE_IF_LAST_CLIENT, 550
 - vvUIRCF_CLOSE_IF_LAST_CLIENT_DE LAY, 550
 - vvUIRCF_DEFAULT, 550
 - vvUIRCF_NO_CLOSE, 550
 - vvUIRemoveClientConstants, 550
 - vvUIUIFOINDEX_ENROLLID, 538
 - vvUIUIFOINDEX_ENROLL_DESCRIPTORI ON, 539
 - vvUIUIFOINDEX_TASK_DESCRIPTION, 539
 - vvUIUIFOINDEX_TASKID, 538
 - vvUIUIFOINDEX_USER_DESCRIPTION, 538
 - vvUIUIFOINDEX_USERID, 538
 - vvUIVOLINDEX_VOLLEVEL, 539

- vvUIWHINDEX_TAGGEDTEXT, 539
- context menu
 - enabling, 772
 - Virtual Voices, 749
- context-free grammar file, using, 279
- control
 - Dictation
 - events, 734
 - frequent questions about, 743
 - getting started, 693
 - introduction, 699
 - methods, 712
 - properties, 701
 - DictationMgr
 - events, 682
 - frequent questions about, 691
 - getting started, 633
 - introduction, 631
 - methods, 659
 - properties, 641
 - DictLite
 - events, 384
 - getting started, 363
 - introduction, 361
 - methods, 384
 - properties, 381
 - Error Correction
 - events, 481
 - frequent questions about, 495
 - getting started, 417
 - introduction, 415
 - methods, 460
 - properties, 431
 - Grammar
 - events, 336
 - frequent questions about, 359
 - getting started, 281
 - hierarchy, 279
 - introduction, 279
 - methods, 327
 - properties, 295
 - GrammarLite
 - events, 394
 - getting started, 363
 - introduction, 361
 - methods, 394
 - properties, 389
 - Lite
 - frequent questions about, 413
 - getting started, 363
 - introduction, 361
 - properties, 381
 - Phrases
 - events, 239
 - frequent questions about, 277
 - getting started, 205
 - hierarchy, 203
 - introduction, 203
 - methods, 233
 - properties, 221
 - PhrasesLite
 - events, 407
 - getting started, 363
 - introduction, 361
 - methods, 402
 - properties, 399
 - RichEdit
 - events, 190
 - frequent questions about, 201
 - getting started, 87
 - hierarchy, 85
 - introduction, 85
 - methods, 171
 - properties, 99
 - TextBox
 - events, 69
 - frequent questions about, 83
 - getting started, 29
 - hierarchy, 27
 - introduction, 27
 - methods, 63
 - properties, 41

- User Interface
 - class, 534
 - constants, 537
 - enumerations, 551
 - events, 611
 - frequent questions about, 629
 - getting started, 499
 - introduction, 497
 - methods, 565
 - properties, 561
 - structure, 535
 - Virtual Voices, 755
 - events, 799
 - face customization notes, 814
 - frequent questions about, 835
 - getting started, 745
 - introduction, 757
 - methods, 790
 - programming notes, 813
 - properties, 759
 - tts engine attributes, 779
 - controls
 - list, 19
 - copying selected text to the clipboard, 36, 94
 - Correct method, 663, 713
 - correcting
 - selected text, 36, 94
 - correction
 - text, 38, 97
 - Count property, 246, 346
 - creating
 - control
 - VVDictLite, 363
 - VVGrammarLite, 368
 - VVPhrasesLite, 374
 - custom menus, 523
 - instance of control, 499
 - Error Correction, 417
 - Grammar, 281
 - Phrases, 205
 - TextBox, 29
 - UIClient, 499
 - VVDictation, 693
 - VVDictationMgr, 633
 - VVRichEdit, 87
 - VVTextBox, 31, 89
 - member variable for a class, 34, 92, 638
 - new phrase, 253
 - new phrase object, 234
 - current state of UIServer, 624
 - cursor
 - moving to the previous word, 37, 95
 - placing at the beginning of the next word, 36, 95
 - selecting text at, 37, 95
 - CursorIndex property, 645
 - custom designer
 - VVPhrases control, 216
 - Custom Interface methods,
 - SetClientCallback, 597
 - custom menu options
 - adding, 534, 587
 - creating, 523
 - getting information about, 577
 - modifying, 534
 - providing information about, 535
 - cutting selected text to the clipboard, 36, 94
- D**
- data
 - storing, 269
 - DATA directory, 755
 - DefaultExpression property, 766
 - defining maximum size for string value
 - properties, 546
 - DeleteMenuItem method, 574
 - DeleteText event, 683
 - DeleteText method, 666
 - deleting dictated text, 37, 95
 - description

- actions for program to take, **261**
 - Description property, **263**
 - Detective control
 - summary, **24**
 - Dictation
 - phrase formatting flags, **741**
 - summary, **23**
 - dictation
 - display current state, **61**
 - enable, **83, 201**
 - Dictation control
 - capturing speech, **697**
 - creating instance of, **693**
 - events, **734**
 - getting started, **693**
 - introduction, **699**
 - methods, **712**
 - properties, **701**
 - questions, **743**
 - summary, **697**
 - dictation mode
 - AutoDictation, **42, 46, 100, 104, 106**
 - AutoDictationWindow, **642, 702**
 - description, **35, 93, 639, 697**
 - DictationStateChange, **73, 193, 685, 735**
 - setting state, **54, 116, 647, 704**
 - starting automatically, **42, 46, 100, 104, 106, 642, 702**
 - DictationMgr control
 - capturing speech, **639**
 - creating instance of, **633**
 - events, **682**
 - getting started, **633**
 - introduction, **631**
 - methods, **659**
 - properties, **641**
 - questions, **691**
 - summary, **23, 640**
 - DictationOn
 - parameter, **73, 193**
 - property, **54, 116**
 - DictationOn property, **647, 704**
 - DictationStateChange event, **73, 193, 685, 735**
 - DictLite control
 - events, **384**
 - introduction, **361**
 - methods, **384**
 - properties, **381**
 - using, **363**
 - directory structure, **755**
 - disabled microphone button, why?, **629**
 - disabling
 - command recognition, **225**
 - command recognition, **309**
 - grammar, **290**
 - recognition
 - command phrases, **214**
 - particular phrases, **265**
 - displaying
 - UIServer, **568**
 - displays
 - state of dictation, **61**
 - docked, event to change view, **621**
 - docking
 - preventing, **541**
 - docking, view mode of UIServer, **540**
 - DOCS directory, **755**
 - DoProperties method, **793**
 - Drag-Drop-n-Go
 - Grammar control, **287, 423, 426**
 - support, **211**
 - dwFlags parameter, **597**
 - dwMenuItemId parameter, **619**
 - dwValueData parameter, **606**
- E**
- editing Class attributes, **209**
 - elderly voices, **769**
 - embedding bookmarks, **800**
 - EN_UK LangID string name, **509**
 - EN_US LangID string name, **509**
 - enable dictation, **83, 201**

- Enabled property, 225, 248, 265, 349, 382, 390, 400, 403, 405, 534
 - VVCFGram, 290
- enabling, 265
 - animated face, 775
 - Caps Lock, 37, 95
 - command recognition, 225, 309
 - context menu, 772
 - grammar, 290
 - main menu options, 587
 - recognition of command phrases, 214
 - recognition of a particular phrase, 265
- end-user
 - ability to change Virtual Voices Properties, 793
 - control via Virtual Voices context menu, 749
 - pausing and resuming speaking, 804
- Engine
 - property, 56, 118
- engine attributes, ViaVoice Outloud, 779
- Engine control
 - summary, 21
- Engine property, 227, 649, 706
- entering dictation mode, 73, 83, 193, 201, 685, 735
- enumerated types
 - microphone states, 552
 - taskbar component ID, 554
 - TVIEWTYPE, 555
 - UIEVENTRC, 556
 - UIMENUTYPE, 558
 - UIRC, 559
- enumerations
 - MICROPHONE_STATES, 552
 - TCID, 554
 - TVIEWTYPE, 555
 - UIEVENTRC, 556
 - UIMenuGroup, 557
 - UIMENUTYPE, 558
 - User Interface control, 551
- error
 - correction, hiding the dialog box, 36, 94
 - information, 559
 - message dialog box, controlling, 76, 195
- Error Correction control
 - creating instance of control, 417
 - events, 481
 - getting started, 417
 - introduction, 415
 - questions frequently asked, 495
 - summary, 22
- Error Correction dialog box
 - showing, 37, 95
- error correction window, 38, 97
- error description parameter,
 - pstrDescription, 76, 195
- Error event, 76, 195
- ES_ES LangId string name, 509
- EventActiveApplication event, 612
- EventButtonPressed event, 615
- EventComponentUpdated event, 617
- EventMenuItemSelected event, 619
- EventQueryViewFlags
 - event description, 621
 - using to handle changing view mode of UISever, 540
- EventQueryViewMenuInfo event, 624
- events
 - BeginSpeechRecognition, 240
 - BookMark, 800
 - Command, 70, 191
 - DeleteText, 683
 - DictationStateChange, 73, 193, 685, 735
 - DictLite control, 384
 - Error, 76, 195
 - Error Correction control, 481
 - EventActiveApplication, 612
 - EventButtonPressed, 615
 - EventComponentUpdated, 617
 - EventMenuItemSelected, 619
 - EventQueryViewFlags, 621
 - EventQueryViewMenuInfo, 624

- Grammar control, 336
- GrammarLite control, 394
- HitBookMark, 737
- InitDone, 802
- KeyPress, 804
- MaxText, 80, 199
- Pause, 806
- Phrase object, 275
- PhraseReco, 739
- PhraseRecognized, 385, 395, 408
- Phrases control, 239
- PhrasesLite control, 407
- PutText, 687
- Reset, 807
- Resume, 808
- SpeechRecognized, 212, 241
- StartSpeaking, 809
- StopSpeaking, 810
- supported in Dictation control, 734
- supported in DictationMgr control, 682
- supported in RichEdit control, 190
- supported in TextBox control, 69
- TrainingRequired, 243, 342
- User Interface control, 611
- Virtual Voices, 799
- VUMeter, 387, 397, 410
- WordPosition, 811
- exclamation point, allowed in a phrase, 277
- ExecuteCommand, 172
- ExecuteCommand method, 64
- ExePathName property, 534
- Exist method, 255, 355
- exiting dictation mode, 73, 193, 685, 735
- ExpandMacros property, 651, 708
- Expression property, 767
- ExtendedMenuFlags, 545
- external lists
 - grammar, 291
 - number in a group, 346
 - returning, 352
 - turning off, 349
 - turning on, 349
- ExternLists property, 314
- F**
 - face customization notes
 - Virtual Voices, 814
 - face, animated
 - controlling, 767
 - enabling, 775
 - fByPosition parameter, 574, 577, 588, 604
 - female gender, 769
 - file
 - grammar
 - setting path to, 317, 320
 - GrammarSource
 - setting path to, 324
 - finding out if external list is part of an object, 355
 - finding out if phrase is part of a collection, 255
 - focus
 - using to start dictation mode, 42, 46, 100, 104, 106, 642, 702
 - format speech input, 651, 708
 - formatting flags, 741
 - FR_FR LangID string name, 509
 - frequently asked questions
 - Dictation control, 743
 - DictationMgr control, 691
 - Error Correction control, 495
 - Grammar control, 359
 - Lite controls, 413
 - Phrases control, 277
 - RichEdit control, 201
 - TextBox control, 83
 - User Interface control, 629
 - Virtual Voices, 835
 - functions
 - AboutBox, 791
 - Cancel, 792
 - DoProperties, 793

Pause, 794
 Resume, 795
 Speak, 796

G

Gender property, 769
 GetAlternate method, 668
 GetFlags method, 719
 GetMenuItemInfo method, 577
 GetNumberValue method, 580
 GetStringValue method, 582
 GetText method, 671
 getting
 focus, UIClient event, 612
 information about custom menu option, 577
 numeric information about UIServer, 580
 path to binary grammar file, 317, 320
 properties of UIServer, and setting, 538
 string information about the UIServer, 582
 UI characteristics, 520
 getting started
 Dictation control, 693
 DictationMgr control, 633
 DictLite control, 363
 Error Correction control, 417
 Grammar control, 281
 GrammarLite control, 363
 Lite controls, 363
 Phrases control, 205
 PhrasesLite control, 363
 RichEdit control, 87
 TextBox control, 29
 User Interface control, 499
 Virtual Voices control, 745
 GetWavData method, 721
 GetWordInfo method, 673, 723
 globally unique identifier, 769
 Gr_GR LandID string name, 509
 Grammar
 loading, 288
 grammar
 enabling/disabling, 290
 external lists, 291
 Grammar control
 Alternates, 296
 Annotations, 298
 AutoConnect, 303
 AutoLoad, 305
 AutoUI, 307
 BeginSpeechRecognized, 337
 Count, 346
 creating instance of control, 281
 displaying UIServer, 307
 Drag-Drop-n-Go, 287, 423, 426
 Enabled, 309, 349
 Enabled property, 309
 Engine, 311
 Engine property, 311
 events, 336
 Exists, 355
 ExternLists, 314
 getting started, 281
 GrammarSource, 317, 320
 hierarchy, 279
 interacting with UIServer, 307
 introduction, 279
 Item, 352
 LoadFromSource, 328, 330
 questions frequently asked, 359
 RefreshUIText, 331
 Rules, 322
 ShowTrainDialog, 334
 SourceType, 324
 SpeechRecognized, 339
 summary, 21
 TrainingRequired, 342
 VVPhraseCollGroup object, 345, 354
 grammar file
 compiled
 how to create, 359
 using, 279
 loading automatically, 305

- loading manually, **328, 330**
- GrammarLite control
 - events, **394**
 - introduction, **361**
 - methods, **394**
 - properties, **389**
 - using, **368**
- grammars
 - enabling and disabling, **290**
 - kinds supported, **359**
- GrammarSource property, **317, 320, 392**
- group
 - number of external lists in, **346**

H

- handling
 - BookMark event, **800**
 - InitDone event, **802**
 - KeyPress event, **804**
 - Pause event, **806**
 - Reset event, **807**
 - Resume event, **808**
 - SpeechRecognized event, **212**
 - StartSpeaking event, **809**
 - StopSpeaking event, **810**
 - WordPosition event, **811**
- HeadSize property, **781**
- hearing a voice sample, **751**
- help menu option, adding, **587**
- HelpMenuName parameter, **625**
- hiding the error correction dialog, **36, 94**
- hiding the Virtual Voices control, **786**
- hierarchy
 - Grammar Control, **279**
 - Phrases Control, **203**
 - RichEdit control, **85**
 - TextBox control, **27**
 - VVPhrases, **218**
- hierarchy of objects for VVPhrases, **203**
- HitBookMark event, **737**
- hresult parameter, **77, 196**

- hwndApplication parameter, **568, 593, 612**
- hwndTarget parameter, **615, 620**
- hwndWindow parameter, **624**

I

- ID property, **267, 534**
- identifier for a phrase, programmer-assigned, **267**
- ignoring
 - all commands in a collection, **248**
 - all external lists in a group, **349**
 - phrases, **265**
- improving recognition, TrainingRequired event, **243**
- Include files, VVUICINST.h and VVUICTYPE.h, **504**
- Index constants
 - description, **538**
 - vvUIUserInfoIndex (COMPID_USERINFORMATION), **538**
 - vvUIVolumendex (COMPID_USERINFORMATION), **539**
 - vvUIWordHistoryIndex (COMPID_USERINFORMATION), **539**
- InitDone event, **802**
- Initialize method, **585**
- initializing UI client, **506**
- input index, **723**
- InsertMenuItem method, **587**
- installation notes, **755**
- installing
 - ViaVoice ActiveX components, **19**
- instance of control
 - creating, **29, 87, 499, 633, 693**
 - UIClient, **499**
- instance of control, creating
 - Phrases, **205**
- interacting with UI server, **513**
- introduction
 - Dictation control, **699**
 - DictationMgr control, **631**

DictLite control, **361**
Error Correction control, **415**
Grammar control, **279**
GrammarLite control, **361**
Lite controls, **361**
Phrases control, **203**
PhrasesLite control, **361**
RichEdit control, **85**
TextBox control, **27**
User Interface control, **497**
ViaVoice ActiveX controls, **19**
Virtual Voices ActiveX control, **757**
invisible operation, Virtual Voices control, **786**
invoke voice commands, **64**
IT_IT LangID string name, **509**
Item property, **250, 352**
ItemData property, **269**

K
keyboard control of Virtual Voices control, **813**
KeyPress event, **804**
kinds of grammars supported, **359**

L
Lang ID parameter, **509**
LangStr parameter, **601**
LanguageUI property, **58, 562**
Layout property, **229**
lHelpID parameter, **77, 196**
list
 ViaVoice ActiveX controls, **19**
list of phrases stored by VVPhrases, **212**
Lite controls
 introduction, **361**
 properties, **381**
 questions, **413**
 summary, **21**
LoadFromSource method, **328, 330**

loading Grammar, **288**
LoadRTF method, **174**
LoadTextFile method, **176**
location of a menu item, parameter for, **570**
Locked property, **653**
losing focus, UIClient event, **612**
lowercase, changing selected text to, **36, 94**

M

MainMenuName parameter, **624**
making the Properties dialog appear, **793**
male gender, **769**
manual loading of a grammar file, **328, 330**
maximum
 number of characters allowed, **80, 199**
 size of UIServer string value properties, **546**
MaxText event, **80, 199**
member variable, creating for a class, **34, 92, 638**
menu
 creating custom, **523**
 custom, specifying location, **557**
 items, Custom Interface, **535**
 options, Custom Interface, **535**
 styles, setting, **545**
menu group name parameter, **570**
menu item caption, maximum number of characters allowed, **546**
menu items
 constants, **547**
 parameter, **574**
 position, **574**
menu option
 check mark indicator, **534**
 constants, **557**
 Enabled indicator, **534**
 path of help file, **534**
 specifying ID numbers, **534**
 Text, **534**

- Visible indicator, **534**
- MenuName parameter, **570, 574, 577, 587, 603**
- MergeRecoPhrases method, **725**
- methods
 - About, **63, 171**
 - AboutBox, **791**
 - Add, **253**
 - AddApplicationByName, **566**
 - AddApplicationByWindow, **568**
 - AddPhrase, **234**
 - AppendMenuItem, **570**
 - BeginSpeechRecognized, **337**
 - Cancel, **792**
 - Command, **63, 171, 660**
 - Correct, **663, 713**
 - DeleteMenuItem, **574**
 - DeleteText, **666**
 - DictLite control, **384**
 - DoProperties, **793**
 - Drag, **63, 171**
 - Error Correction control, **460**
 - ExecuteCommand, **64**
 - Exist, **255, 355**
 - GetAlternate, **668**
 - GetFlags, **719**
 - GetMenuItemInfo, **577**
 - GetNumberValue, **580**
 - GetStringValue, **582**
 - GetText, **671**
 - GetWavData, **721**
 - GetWordInfo, **673, 723**
 - Grammar control, **327**
 - GrammarLite control, **394**
 - Initialize, **585**
 - InsertMenuItem, **587**
 - LoadFromSource, **328, 330**
 - LoadRTF, **174**
 - LoadTextFile, **176**
 - MergeRecoPhrases, **725**
 - Move, **63, 171**
 - Pause, **794, 799**
 - Phrase object, **275**
 - PhraseColl collection, **252**
 - PhraseCollGroup object, **354**
 - Phrases control, **233**
 - PhrasesLite control, **402**
 - PutText, **678**
 - Refresh, **63, 171**
 - RefreshingUIText, **236**
 - RefreshUIText, **331**
 - Remove, **257**
 - RemoveAll, **259**
 - RemoveApplicationByName, **591**
 - RemoveApplicationByWindow, **593**
 - Resume, **795, 799**
 - SaveTextFile, **186**
 - SelPrint, **188**
 - SetBookMark, **727**
 - SetClientCallback, **597**
 - SetClientCallbackFlags, **597**
 - SetContext, **729**
 - SetFocus, **63, 171**
 - SetLanguageByID, **599**
 - SetLanguageByString, **601**
 - SetMenuItemInfo, **603**
 - SetNumberValue, **606**
 - SetSelection, **680**
 - SetStringValue, **609**
 - ShowTrainDialog, **334**
 - ShowWhatsThis, **63, 171**
 - Speak, **766, 796**
 - SpeechRecognized, **339**
 - SplitOutLeftWord, **731**
 - supported in RichEdit control, **171**
 - supported in TextBox control, **63**
 - TrainingRequired, **342**
 - User Interface control, **565**
 - Virtual Voices, **790**
 - VVDictation control, **712**
 - VVDictationMgr control, **659**
 - Z-Order, **63, 171**
- microphone

button, **497, 520**
 button disabled, why?, **629**
 index constants
 vvUIMICINDEX_MICSTATE, **538**
 vvUIMICINDEX_WAITSTATE, **538**
 state constants
 UIMSF_DISABLED, **552**
 UIMSF_ERROR, **552**
 UIMSF_OFF, **552**
 UIMSF_ON, **552**
 UIMSF_SLEEP, **552**
 wait state constants
 UIMSF_ADDWAIT, **553**
 UIMSF_CLEARWAIT, **553**
 UIMSF_REMOVEWAIT, **553**
 minimized, event to change view, **621**
 misrecognized words, **663, 713**
 ModeGuid property, **769**
 modifying
 custom menu items, **535**
 custom menu options, **534, 535**
 state of UIServer component, **609**
 UIServer components, **538**
 mouse, EventButtonPressed, **615**
 moving the cursor, **36, 37, 94, 95**

N

Name property, **271**
 nDataSize parameter, **582**
 neutral gender, **769**
 new phrase, adding, **253**
 nIndex parameter, **582, 606, 609**
 nIndexLong parameter, **580**
 notifying
 client application
 Pause, **806**
 Reset, **807**
 Resume, **808**
 WordPosition, **811**
 container application
 event, **802**

 StartSpeaking, **809**
 StopSpeaking, **810**
 UIServer about application programs, **566**
 number of external lists in a group, **346**
 number of phrases in a collection, **246**

O

object
 Phrase events, **275**
 Phrase methods, **275**
 PhraseCollGroup methods, **354**
 PhraseCollGroup properties, **345**
 object hierarchy for VVPhrases, **203**
 objects
 hierarchy
 VVRichEdit control, **85**
 VVTextBox control, **27**
 information, accessing, **580**
 Phrase
 properties, **260**
 removing all from a collection, **259**
 VVPhraseColl, **214**
 VVPhrases
 hierarchy, **218**
 overview
 Virtual Voices, **745**

P

parameters
 ApplicationMenuName, **625**
 ApplicationName, **566, 591, 612**
 ApplicationTitle, **624**
 bActive, **612**
 bShow, **77, 196**
 ciComponent, **582, 606, 609, 615, 617**
 CmdID, **70, 191**
 DictationOn, **73, 193**
 dwFlags, **597**
 dwMenuItemId, **619**
 dwValueData, **606**

- fByPosition, **577, 588, 604**
- HelpMenuName, **625**
- hresult, **77, 196**
- hwndApplication, **568, 593, 612**
- hwndTarget, **615, 620**
- hwndWindow, **624**
- identifying commands, CmdID, **70, 191**
- IHelpID, **77, 196**
- LangID, **509**
- LangStr, **601**
- MainMenuName, **624**
- MenuName, **570, 574, 577, 587, 603**
- nDataSize, **582**
- nIndexLong, **580**
- pdwDockFlags, **621**
- pdwValueData, **580**
- phwndWindow, **621**
- pIMenuInfo, **578, 588, 604**
- pResult, **612, 615, 617, 620, 621, 625**
- pstrDescription, **76, 195**
- riid, **595**
- strCommand, **70, 191, 461, 464, 466**
- uItem, **574, 577, 587, 603**
- uIMenuGroup, **570, 574, 577, 603**
- ValueData, **582, 609**
- VtViewType, **624**
- wLangID, **599**
- Paste, **750**
- pasting text from the clipboard, **37, 95**
- path of help file, **534**
- Pause
 - event, **806**
 - example of using, **799**
 - method, **794**
- pausing playback, **794**
- pdwDockFlags parameter, **621**
- pdwValueData parameter, **580**
- phrase formatting flags
 - dictation, **741**
- Phrase object
 - events, **275**
 - methods, **275**
 - properties, **260**
- phrase objects
 - creating new, **234**
 - programmer-assigned name for, **271**
- phrase, characters permitted in, **277**
- PhraseColl Collection
 - methods, **252**
 - properties, **245**
- PhraseReco event, **739**
- PhraseRecognized event, **385, 395, 408**
- phrases
 - adding, **212, 234**
 - number in a collection, **246**
- Phrases control
 - creating an instance of, **205**
 - events, **239**
 - getting started, **205**
 - hierarchy, **203**
 - introduction, **203**
 - methods, **233**
 - properties, **221**
 - questions, **277**
 - summary, **20**
- Phrases property, **214, 231**
- PhrasesLite control
 - events, **407**
 - introduction, **361**
 - methods, **402**
 - properties, **399**
 - using, **374**
- phwndWindow parameter, **621**
- pIMenuInfo parameter, **578, 588, 604**
- Pitch property, **771**
- PitchFluctuation property, **782**
- placing the cursor at the beginning of the
 - next word, **36, 95**
- playback audio, **721**
- playback, pausing, **794**
- pResult parameter, **612, 615, 617, 620, 621, 625**
- preventing docking UIServer, **541**

- previous word, moving the cursor to, **37, 95**
- ProcessingMacro property, **710**
- programming
 - interfaces, Virtual Voices control, **755**
 - notes, Visual Basic, **813**
- programming notes
 - Virtual Voices, **813**
- properties
 - ActionDesc, **261**
 - ActorName, **760**
 - Age, **761**
 - AllowProperties, **762**
 - Alternates, **296**
 - Annotations, **298**
 - AutoConnect, **222, 303**
 - AutoDictation, **42, 46, 100, 104, 106**
 - AutoDictationWindow, **642, 702**
 - AutoLoad, **305**
 - AutoUI, **223, 307**
 - BackColor, **763**
 - Breathiness, **780**
 - BulletIndentation, **109**
 - Caption, **534**
 - Checked, **534**
 - Clipping, **764**
 - Commands, **49, 111**
 - CommandsEnabled, **52, 114**
 - Count, **246, 346**
 - CursorIndex, **645**
 - DefaultExpression, **766**
 - defining maximum size for string values, **546**
 - Description, **263**
 - DictationOn, **54, 116, 647, 704**
 - DictLite control, **381**
 - Enabled, **225, 248, 265, 309, 349, 382, 390, 400, 403, 405, 534**
 - Engine, **56, 118, 227, 311, 649, 706**
 - Error Correction control, **431**
 - ExePathName, **534**
 - ExpandMacros, **651, 708**
 - Expression, **767**
 - ExternLists, **314**
 - Gender, **769**
 - Grammar control, **295**
 - GrammarLite control, **389**
 - GrammarSource, **317, 320, 392**
 - HeadSize, **781**
 - ID, **267, 534**
 - Item, **250, 352**
 - ItemData, **269**
 - LanguageUI, **58, 562**
 - Layout, **229**
 - Lite controls, **381**
 - Locked, **653**
 - ModeGuid, **769**
 - Name, **271**
 - Phrase object, **260**
 - PhraseColl Collection, **245**
 - PhraseCollGroup object, **345**
 - Phrases, **214, 231**
 - Phrases control, **221**
 - PhrasesLite control, **399**
 - Pitch, **771**
 - PitchFluctuation, **782**
 - ProcessingMacro, **710**
 - RightMargin, **127**
 - Roughness, **783**
 - Rules, **322**
 - SelAlignment, **129**
 - SelBold, **131**
 - SelBullet, **133**
 - SelCharOffset, **135**
 - SelColor, **137**
 - SelFontName, **139**
 - SelFontSize, **141**
 - SelHangingIndent, **143**
 - SelIndent, **145**
 - SelIRightIndent, **153**
 - SelItalic, **147**
 - SelLength, **149**
 - SelProtected, **151, 157**

SelRTF, 155
SelStrikeThru, 159
SelTabCount, 122, 161
SelTabs, 163
SelText, 165
SelUnderline, 167
ShowDictationIcon, 61
ShowMenu, 772
SourceType, 324
SpeakText, 773
Speed, 774
Text, 273
TextRTF, 169
UppercaseOn, 657
UseFace, 775
User Interface control, 561
UseWave, 776
Virtual Voices, 759
Visibility, 786
Visible, 534
VVDictation control, 701
VVDictationMgr control, 641
VVRichEdit control, 99
VVTextBox control, 41
Properties dialog, controlling when appears, 793
properties page, Virtual Voices, 750
pstrDescription parameter, 76, 195
PutText event, 687
PutText method, 678

Q

querying
 states of components in UI server, 554, 555
 UIServer components, 538
questions, Virtual Voices, 835

R

recognition
 commands, controlling, 52, 114

 TrainingRequired event, 243
recognized speech input, 739
recognizing
 speech, 241
 text, 273
rectangular window, 764
reference
 speech engine, 311
refreshing text, 236
RefreshUIText method, 236, 331
Remove method, 257
RemoveAll method, 259
RemoveApplicationByName method, 591
RemoveApplicationByWindow method, 593
RemoveClientConstants method, 550
removing
 all objects from a collection, 259
 custom menu items, 574
 object from a collection, 257
 phrase objects from a collection, 257
 program from list of programs, 591, 593
reporting errors, 76, 195
Reset event, 807
Resume
 event, 808
 example of using, 799
 method, 795
resuming after a pause, 795
retrieve formatting flags, 719
retruning
 external lists, 352
returning from Speak method, 766
RichEdit control
 creating instance of, 87
 events, 190
 introduction, 85
 methods, 171
 object hierarchy, 85
 properties, 99
 questions, 201
 summary, 20

RightMargin
property, 127
riid parameter, 595
Roughness property, 783
Rules property, 322

S

Sample Voice button, 751
SAMPLES directory, 755
SaveTextFile method, 186
SDK

architecture, 25
component installation, 19
list of controls, 19

SDK ActiveX
introduction, 19

SelAlignment
property, 129

SelBold
property, 131

SelBullet
property, 133

SelCharOffset
property, 135

SelColor
property, 137

SelFontName
property, 139

SelFontSize
property, 141

SelHangingIndent
property, 143

SelIndent
property, 145

SelItalic
property, 147

Sellength
property, 149

SelPrint method, 188

SelProtected
property, 151, 157

SelRightIndent
property, 153

SelRTF
property, 155

SelStrikeThru
property, 159

SelTabCount
property, 122, 161

SelTabs
property, 163

SelText
property, 165

SelUnderline
property, 167

sErrorID parameter, 76, 195

server, UI, interacting with, 513

SetBookMark method, 727

SetClientCallback method, 597

SetClientCallbackFlags method, 597

SetContext method, 729

SetLanguageByID method, 599

SetLanguageByString method, 601

SetMenuItemInfo method, 603

SetNumberValue method, 606

SetSelection method, 680

SetStringValue method, 609

setting

baseline frequency of text-to-speech voice,
771

maximum number of characters, 80, 199

menu item characteristics, 603

menu styles, 545

path for GrammarSource, 324

path to a grammar file, 317, 320

refer to engine object, 56, 118, 649, 706

state of dictation mode, 54, 116, 647, 704

text-to-speech voice speed, 774

UI characteristics, 520

UIServer properties, 538

voice smoothness or roughness, 783

ShowDictationIcon property, 61

- showing the error correction dialog, **37, 95**
- ShowMenu property, **772**
- ShowTrainDialog method, **334**
- single phrase, turning off recognition, **265**
- size of speaker's head, controlling, **781**
- smooth voice, **783**
- snapshot of the control, **229**
- SourceType property, **324**
- Speak
 - method, **796**
 - option, **749**
 - UseWave property, **776**
- SpeakText property, **773**
- specifying
 - ID number of a menu option, **534**
 - location of custom menu item, **557**
 - UIServer components for query or modification, **538**
- speech
 - capturing, **35, 93, 639, 697**
 - recognizing, **241**
- speech engine
 - automatic connection to, **222**
 - reference, **311**
- speech recognition
 - TrainingRequired event, **243**
 - turning on and off, **290**
- SpeechRecognized
 - event handling, **212**
 - events, **241**
- SpeechRecognized method, **339**
- Speed property, **774**
- SplitOutLeftWord method, **731**
- sprite, definition of, **787**
- StartSpeaking event, **796, 809**
- state of component, modifying, **606**
- state of dictation mode, setting, **54, 116, 647, 704**
- state of dictation, display, **61**
- state of UIServer, **624**
- StopSpeaking event, **796, 810**
- storing
 - additional data with a phrase, **269**
 - data, **269**
 - list of command phrases, **212**
- strCommand parameter, **70, 191, 461, 464, 466**
- strHelp parameter, **77, 196**
- string values, defining maximum size, **546**
- strSource parameter, **77, 196**
- structure
 - User Interface control, **535**
- structures, UIMenuItemInfo, **534**
- sublanguage ID, **509**
- summary
 - Detective control, **24**
 - Dictation control, **23**
 - DictationMgr control, **23**
 - Engine control, **21**
 - Error CorrectionTextBox control, **22**
 - Grammar control, **21**
 - Lite control, **21**
 - Phrases control, **20**
 - RichEdit control, **20**
 - TextBox control, **20**
 - User Interface control, **22**
 - Virtual Voices control, **23**
 - VVDictation, **697**
 - VVDictationMgr, **640**
- support
 - Drag-Drop-n-Go, **211**
- synchronize events, **727**
- T**
- taskbar
 - component ID, **554**
 - event to change view, **621**
- text
 - changing to lowercase, **36, 94**
 - changing to uppercase, **37, 95**
 - command to capitalize, **36, 94**
 - copying to the clipboard, **36, 94**
 - correcting selected, **36, 94**

- correction, 38, 97
- cutting to the clipboard, 36, 94
- deleting dictated, 37, 95
- disabling Caps Lock, 37, 95
- enabling uppercase, 37, 95
- moving the cursor, 36, 94
- pasting from the clipboard, 37, 95
- recognizing, 273
- selecting at the cursor, 37, 95
- Text property, 273
- TextBox
 - language for specific client, 58
- TextBox control
 - creating instance of, 29
 - events, 69
 - introduction, 27
 - methods, 63
 - object hierarchy, 27
 - properties, 41
 - questions, 83
 - recognizing commands, 52, 114
 - summary, 20
- TextRTF
 - property, 169
- text-to-speech
 - Engine attributes
 - Breathiness, 780
 - description, 779
 - HeadSize, 781
 - PitchFluctuation, 782
 - Roughness, 783
 - setting
 - baseline frequency, 771
 - voice speed, 774
 - SpeakText property, 773
- toolbar buttons, setting state of, 554
- TrainingRequired event, 243, 342
- TrainingRequired method, 342
- turning off
 - Caps Lock, 37, 95
 - command recognition, 248

- external lists, 349
- recognition of command phrases, 214
- speech recognition, 290
- turning on
 - Caps Lock, 37, 95
 - command recognition, 248
 - external lists, 349
 - recognition of command phrases, 214
 - speech recognition, 290
- TVIEWTYPE, 555

U

- UIAPIRC_ERROR_ALREADYINITIALIZED constant, 559
- UIAPIRC_ERROR_FAILED constant, 559
- UIAPIRC_ERROR_INVALIDCLIENT constant, 559
- UIAPIRC_ERROR_INVALIDPARAM constant, 559
- UIAPIRC_ERROR_NOSERVER constant, 559
- UIAPIRC_ERROR_NOTCURRENTCLIENT constant, 559
- UIAPIRC_ERROR_OUTOFMEMORY constant, 559
- UIAPIRC_ERROR_SERVERBUSY constant, 559
- UIAPIRC_OK constant, 559
- UIClient
 - creating an instance of the control, 499
 - defining events received from UIServer, 543
 - informing about actions in events, 556
 - initializing, 506
 - noting change of focus, 612
 - possible events
 - vvUIEVENT_ACTIVEAPP_CHANGED, 543
 - vvUIEVENT_ALL, 543
 - vvUIEVENT_BUTTON_PRESSED, 543
 - vvUIEVENT_COMPONENT_UPDATED, 543

- `vvUIEVENT_MENUITEM_SELECTED`, 543
 - `vvUIEVENT_NONE`, 543
 - `vvUIEVENT_VIEW_QUERYFLAGS`, 544
 - `vvUIEVENT_VIEW_QUERYMENUINFO`, 544
- return code, 559
- shutting down, specifying UI Server response, 550
- UIEVENTRC enumerated type, 556
- UIEVENTRC_NOTPROCESSED constant, 556
- UIEVENTRC_PROCESSED constant, 556
- UIMenuGroup enumeration, 557
- UIMenuItemInfo structure, 534, 535
- UIMENUTYPE, 558
- UIMFG_DYNAMIC_APPLICATION constant, 557
- UIMFG_DYNAMIC_HELP constant, 557
- UIMFG_DYNAMIC_MAIN constant, 557
- UIMFG_STATIC_HELP constant, 557
- UIMFT_CLIENT constant, 558
- UIMFT_EXECUTE constant, 558
- UIMSF_ADDWAIT constant, 553
- UIMSF_CLEARWAIT constant, 553
- UIMSF_DISABLED constant, 552
- UIMSF_ERROR constant, 552
- UIMSF_OFF constant, 552
- UIMSF_ON constant, 552
- UIMSF_REMOVEWAIT constant, 553
- UIMSF_SLEEP constant, 552
- UIServer
 - button-clicking event, 615
 - causing to appear, 585
 - changing component characteristics, 617
 - components, maximum size for string value properties, 546
 - connecting automatically, 223
 - current appearance, 555
 - current state, 624
 - docking constants
 - `vvDVSF_ADJUST_ORIGIN`, 542
 - `vvDVSF_DEFAULT`, 542
 - `vvDVSF_NORMAL_BACKGROUND`, 542
 - `vvDVSF_TRANSPARENT_BACKGROUND`, 542
 - docking style constants, 542
 - docking view mode, 540
 - getting
 - numeric information about, 580
 - properties, 538
 - string information about, 582
 - handling menu options, 558
 - interacting with, 513
 - language for specific client, 562
 - microphone
 - object, 552
 - modifying
 - components, 554
 - state of component, 606, 609
 - notifying about applications, 566
 - object IDs
 - `ciComponent`, 580, 582, 609
 - preventing docking, 541
 - removing program from list, 591
 - setting properties, 538
 - specifying
 - language to use, 599, 601
 - response when UI Client shuts down, 550
 - string value sizes, 546
 - user
 - changing view of, 621
 - menu option selection event, 619
 - request to view menu, 624
- `uItem` parameter, 574, 577, 587, 603
- UIVIEW_AGENT constant, 555
- UIVIEW_DOCKED constant, 555
- UIVIEW_SYSTRAY constant, 555
- UIVIEW_TASKBAR constant, 555
- uppercase, changing selected word to, 37, 95
- UppercaseOn property, 657
- UseFace property, 775
- user

- interface
 - getting and setting, **520**
 - request to view UIServer menu, **624**
 - starts speaking, event, **240**
- User Interface control
 - getting started, **499**
 - questions, **629**
 - summary, **22**
- UseWave property, **776**
- using
 - DictLite, **363**
 - GrammarLite, **368**
 - PhrasesLite, **374**
- using window handles, **568**
- uUIMenuGroup group, **587**
- uUIMenuGroup parameter, **570, 574, 577, 603**

V

- ValueData parameter, **582, 609**
- ViaVoice
 - component installation, **19**
 - Error Correction control methods, **460**
 - Error Correction Control properties, **431**
 - Grammar Control methods, **327**
 - Grammar Control properties, **295**
 - SDK architecture, **25**
 - User Interface class, **534**
 - User Interface constants, **537**
 - User Interface enumerations, **551**
 - User Interface events, **611**
 - User Interface introduction, **497**
 - User Interface methods, **565**
 - User Interface properties, **561**
 - User Interface structure, **535**
 - Virtual Voices events, **799**
 - Virtual Voices face customization notes, **814**
 - Virtual Voices methods, **790**
 - Virtual Voices programming notes, **813**
 - Virtual Voices properties, **759**
- ViaVoice ActiveX
 - controls included, **19**
 - introduction, **19**
- ViaVoice Outloud engine attributes, **779**
- Virtual Voices
 - context menu, **749**
 - control, **755**
 - introduction, **757**
 - keyboard control for Visual Basic, **813**
 - overview, **745**
- Virtual Voices control
 - getting started, **745**
 - summary, **23**
- Visibility property, **786**
- Visible property, **534**
- Visual Basic programming notes, **813**
- voice
 - setting speed, **774**
 - smoothness or roughness, setting, **783**
 - text-to-speech, setting baseline frequency, **771**
- voices, adult, elderly, and child, **769**
- VtViewType parameter, **624**
- VUMeter event, **387, 397, 410**
- VVCFGram control
 - getting started, **281**
 - introduction, **279**
 - using external lists, **291**
- VVDictation control
 - getting started, **693**
- VVDictationMgr control
 - getting started, **633**
- vvDVAF_ALLOW_DOCK constant, **540**
- vvDVAF_ALLOW_TOPMOST_DOCK constant, **540**
- vvDVAF_DEFAULT constant, **540**
- vvDVAF_NEVER_DOCK constant, **541**
- vvDVAF_STAY_DOCK_TO_PREVIOUS constant, **541**
- vvDVSF_ADJUST_WIDTH constant, **542**
- vvDVSF_ADJUST_ORIGIN constant, **542**
- vvDVSF_DEFAULT constant, **542**

- vvDVSF_NORMAL_BACKGROUND
 - constant, 542
- vvDVSF_TRANSPARENT_BACKGROUND
 - D constant, 542
- VVECWin control
 - getting started, 417
 - introduction, 415
- vvMAX_MENU_OPERATION constant, 546
- vvMAX_MENU_STRING constant, 546
- vvMAX_USERINFO_DESC_LEN, 546
- vvMAX_USERINFO_DESC_LEN
 - constant, 546
- vvMAX_USERINFO_ID_LEN constant, 546
- vvMAX_WORDHISTORY_TEXT
 - constant, 546
- VVPhraseColl Collection
 - object, 214
- VVPhraseCollGroup object
 - methods, 354
 - properties, 345
- VVPhrases control
 - creating instance of, 205
 - custom designer, 216
 - Drag-Drop-n-Go, 211
 - object hierarchy, 203, 218
- vvTBCapitalizeThis command phrase, 36, 94
- vvTBCopy command phrase, 36, 94
- vvTBCorrectionThis command phrase, 36, 94
- vvTBCut command phrase, 36, 94
- vvTBDeleteThis command phrase, 36, 94
- vvTBHideEC command phrase, 36, 94
- vvTBLowercaseThis command phrase, 36, 94
- vvTBMoveBeginning command phrase, 36, 94
- vvTBMoveEnd command phrase, 36, 94
- vvTBNextWord command phrase, 36, 95
- vvTBPasteThis command phrase, 37, 95
- vvTBPreviousWord command phrase, 37, 95
- vvTBScratchThat command phrase, 37, 95
- vvTBSelectText command phrase, 95
- vvTBSelectThis command phrase, 37, 95
- vvTBShowEC command phrase, 37, 95
- vvTBUppercaseOff command phrase, 37, 95
- vvTBUppercaseOn command phrase, 37, 95
- vvTBUppercaseThis command phrase, 37, 95
- VVTextBox control
 - creating instance of, 31, 89
- VVUICNST.h Include file, 504
- vvUIDocking constants, 540
- vvUIDockingStyle constants, 542
- vvUIEVENT_ACTIVEAPP_CHANGED
 - constant, 543
- vvUIEVENT_ALL constant, 543
- vvUIEVENT_BUTTON_PRESSED
 - constant, 543
- vvUIEVENT_COMPONENT_UPDATED
 - constant, 543
- vvUIEVENT_MENUITEM_SELECTED
 - constant, 543
- vvUIEVENT_NONE constant, 543
- vvUIEVENT_VIEW_QUERYFLAGS
 - constant, 544
- vvUIEVENT_VIEW_QUERYMENUINFO
 - constant, 544
- vvUIEventCallbackFlags constants, 543
- vvUIExtendedMenuFlags, 545
- vvUIMaxConstants, 546
- VVUIMenuInfo class, 534
- vvUIMenuItemConstants, 547
- vvUIMICINDEX_MICSTATE constant, 538
- vvUIMICINDEX_WAITSTATE constant, 538
- vvUIRCF_CLOSE constant, 550
- vvUIRCF_CLOSE_IF_LAST_CLIENT
 - constant, 550
- vvUIRCF_CLOSE_IF_LAST_CLIENT_DELAY constant, 550
- vvUIRCF_DEFAULT constant, 550

vvUIRCF_NO_CLOSE constant, 550
vvUIRemoveClientConstants, 550
VVUITYPE.h, Include file, 504
vvUIINFOINDEX_ENROLL_DESCRIPTION constant, 539
vvUIINFOINDEX_ENROLLID constant, 538
vvUIINFOINDEX_TASK_DESCRIPTOR constant, 539
vvUIINFOINDEX_TASKID constant, 538
vvUIINFOINDEX_USER_DESCRIPTOR constant, 538
vvUIINFOINDEX_USERID constant, 538
vvUIUserInfoIndex (COMPID_USERINFORMATION), 538
vvUIVOLINDEX_VOLLEVEL constant, 539
vvUIVolumendex (COMPID_USERINFORMATION), 539
vvUIWHINDEX_TAGGEDTEXT constant, 539
vvUIWordHistoryIndex (COMPID_USERINFORMATION), 539
vvVIAVOICE_IDMENU_STOP_READING, 549
vvVIAVOICEUI_IDMENU_BEGIN_DICTATION constant, 547
vvVIAVOICEUI_IDMENU_BEGIN_READING constant, 547
vvVIAVOICEUI_IDMENU_CORRECT_ERROR constant, 547
vvVIAVOICEUI_IDMENU_EXIT constant, 548
vvVIAVOICEUI_IDMENU_MICROPHONE constant, 548
vvVIAVOICEUI_IDMENU_STOP_DICTATION constant, 548
vvVIAVOICEUI_IDMENU_WHAT_CAN_I_SAY constant, 549

W

wave audio, using, 776
window borderless, rectangular, 764
window handle, 568, 593
Wizard, Class, 34, 92, 638
wLangID parameter, 599
WordPosition event, 811
words per minute speed, 774