



Welcome to the Microsoft Speech SDK, version 5.1

The Microsoft® Speech SDK 5.1 is the developer kit for the Microsoft® Windows environment. Tools, information, and sample engines and applications are provided to help you integrate and optimize your speech recognition and speech synthesis engines with the new Microsoft Speech API 5 (SAPI 5). The Speech SDK also includes updated releases of the Microsoft advanced speech recognition engine and Microsoft concatenated speech synthesis engine.

End-User License Agreement

Please read and understand the End-User License Agreement before using the Microsoft Speech SDK.

Redistribution Code Rights

Please read and understand the Redistributable Code Rights before building applications or engines. This document outlines which files may be redistributed with our own products. It also describes limitations for modifying the Speech SDK 5.1 files and samples.

System Requirements

This section lists the software required including supported operating systems and the compiler environment. Hardware requirements are also listed and include recommended computer speeds, available RAM and audio equipment.

Developer Support

This section provides information on the developer support choices available to you. For additional support options, see support phone numbers and options.

Getting Started for First-Time Users

Getting Started introduces the Microsoft® Speech SDK to first-time users and explains its contents, system requirements, and features.

Release Notes

Release Notes describes new information at the time of release, documents known issues, and identifies any new redistribution file updates. Release notes may be found on the installation disk or, if not using a disk, at the installation source.

Microsoft Speech Technologies Web Site

This site provides product news updates, technical articles, and links to useful resources.

Is the documentation helpful?

We'd like to know. Please send any comments and suggestions you have to: sapi5@microsoft.com.



Getting Started for First-Time Users

The following topics introduce the Microsoft Speech SDK (SDK) to first-time users and explain its contents and features:

- [What is the SDK?](#)
- [What can I do with the SDK?](#)
- [What is covered in the SDK documentation?](#)
- [Programmer's Guide](#)
- [What is not covered by the SDK?](#)
- [How can I start using the SDK?](#)
- [Removing the SDK](#)
- [What special things do I need to know?](#)

What is the SDK?

Microsoft Speech SDK is a software development kit for building speech engines and applications for Microsoft Windows. Designed primarily for the desktop speech developer, the SDK contains the Microsoft® Win32®-compatible speech application programming interface (SAPI), the Microsoft continuous speech recognition engine and Microsoft concatenated speech synthesis (or text-to-speech) engine, a collection of speech-oriented development tools for compiling source code and executing commands, sample application and tutorials that demonstrate the use of Speech with other engine technologies, sample speech recognition and speech synthesis engines for testing with speech-enabled applications, and documentation on the most important SDK features.

What can I do with the SDK?

You can use the SDK components and redistributable

SAPI/engine run-time to build applications that incorporate speech recognition and speech synthesis.

Automation Support

SAPI 5.1 supports OLE automation. That means languages other than C/C++ may now use SAPI for application development. The languages themselves need to support OLE automation. Common languages which may be used includes Visual Basic, C#, and JScript. See [Automation Interfaces and Objects](#) for additional information. Overviews for automation and understanding the API suite for SAPI is found at [Automation Overview](#). This is also a good starting point for programmers new to OLE automation programming.

Speech Components and Services

Included in the Speech API architecture is a collection of speech components for directly managing the audio, training wizard, events, grammar compiler, resources, speech recognition manager, and TTS manager for low-level control and greater flexibility. The Speech API also enables support and manages shared recognition events for running multiple speech-enabled applications.

SDK Tools

The tools in the Tools directory assist with the verification and testing of SAPI development. This directory contains source code and project for compliance testing and may be modified to fit your needs.

SDK Samples

The Microsoft Speech SDK includes samples that can be used as a reference for creating speech-enabled applications. The compiled samples and demonstration applications are available on the **Start->Programs->Microsoft Speech**

SDK 5.1 menu. The binary and source files, projects, are available in the Samples folder of the Microsoft Speech SDK 5.1 folder. A description of each sample, installation, and set of usage instructions is provided.

Coexistence and Third Party Support

Microsoft Speech API 5.1 has been designed to coexist on the same device with prior versions of the Microsoft Speech API (versions 3.0, 4.0, 4.0a, and 5.0). Microsoft is also working with many of the top speech recognition engine vendors on providing SAPI 5 support. Visit the [Third Party Products](#) page for more the latest list on SAPI 5-compatible engines.

For more information on setup, see the [Microsoft Speech SDK Setup 5.1](#).

What is covered in the SDK Documentation?

The Microsoft Speech SDK documentation provides information for both the experienced speech developer and the beginner. It is located in the **Start->Programs->Microsoft Speech SDK 5.1** menu.

Programmer's Guide

The Programmer's Guide provides information on the following Microsoft Speech API topics:

C/C++

- [Application level interfaces](#)
- [Engine level interfaces](#)
- [Structures](#)
- [Enumerations](#)
- [Helper functions](#)

Automation

- [Interfaces and Objects](#)
- [Enumerations](#)

SDK Samples, Tools, and Tutorials

This section includes descriptions and references for the samples, tools, and tutorials for the SAPI 5 SDK.

Engine Compliance Testing Reference

The Testing Reference describes the compliance testing requirements for engine vendors porting their speech engine to SAPI 5.

White Papers

The White Papers include technical background articles on the technology. They also include sample code that addresses more specific programming solutions.

What is not covered by the Speech SDK?

The Microsoft Speech SDK is not an enduser application, GUI, or voice-user interface (VUI) development environment with menus, buttons, toolbars. It is a development kit which allows programmers to write applications incorporating speech into them. Tools are provided in the SDK which may be run from the MS-DOS® command line (e.g., gc.exe) or with executable applications. The Microsoft Speech SDK assumes knowledge of programming for C, C++, or a language which supports OLE automation such as Visual Basic, or C#. SAPI has a strong reliance on COM. Although direct experience with COM or COM programming is not required, understanding COM principles will make programming and application design easier.

How can I start using the SDK?

The organization of the Speech SDK documentation is similar to other traditional Microsoft SDKs. The Finding Information section of the Microsoft Speech SDK documentation contains important information on how to use the documentation's Help Viewer, including use of the toolbar buttons and full text search, and finding a Help topic, and much more.

Visit the [Microsoft® Speech.NET Technologies](#) home page frequently. Here you can find the latest news and updates to the SDK and the Microsoft speech engines.

If for some reason you cannot locate a particular type of documentation in the help system, please e-mail sapi5@microsoft.com to fill a request.

Removing the SDK

If you want to remove the SAPI SDK from the computer, use Add/Remove Programs properties from Control Panel (Start->Settings->Control Panel). It is not advised to delete individual files. However, only one version of the SDK may be installed at a time. Attempts to install a newer version of SAPI with a previous edition already loaded will prompt the installation package to remove SAPI first. After removing SAPI in this manner, the installer may be run again and the new version will be loaded.

What special things do I need to know?

If you are developing an application that intends to use the Microsoft speech setup files, your Setup.exe needs to install the Microsoft Windows Installer if it is not already present. Please go to [Windows Installer 1.5](#) download page, or [Platform SDK Start Page](#) to download the Windows Installer SDK and search for "Windows Installer".



System Requirements

Operating Systems

Supported operating systems are:

- Windows XP Professional or Home editions; all language versions.
- Windows.NET Server editions; all language versions.
- Microsoft Windows 2000 Professional Workstation or Server; all language versions.
- Microsoft Windows Millennium edition.
- Microsoft Windows 98 all editions.
- Microsoft Windows ® NT Workstation or Server 4.0, service pack 6a, English, Japanese, or Simplified Chinese edition.
- Windows 95 or earlier is not supported.

Software Requirements

- Microsoft Internet Explorer 5.0 or later version. Users of Windows NT 4 with any version of the service packs require Microsoft Internet Explorer 5.5 or later. Download the latest version of [Microsoft Internet Explorer](#).
- Microsoft Visual C++ 6.0, service pack 3 or later version is needed to run the SAPI 5 SDK samples. In general, any 32-bit C compiler will work for writing SAPI applications.
- Microsoft Visual Basic is needed to write applications incorporating SAPI automation, or for compiling the Visual Basic sample code. Since SAPI supports COM automation, other languages and compilers may be used with SAPI automation provided it supports OLE automation. Microsoft Visual Studio 7, also called Visual

Studio.NET, is needed to compile the C# examples.

- Platform SDK is generally not needed although some samples and functionality may require it. See the specific samples for confirmation. If required, see [Microsoft Platform SDK](#) for loading information.

Hardware Requirements

- A PentiumII\PentiumII-equivalent or later processor at 233 MHz with 128 megabytes (MB) of RAM is recommended.
- SAPI 5 can now take advantage of a computer and operating system that supports multiple processors, including all those mentioned above. Additionally, you can use SAPI 5 in a distributed application environment.
- A microphone or some other sound input device to receive the sound is required for speech recognition. In general, the microphone should be a high quality device with noise filters built in. The speech recognition rate is directly related to the quality of the input. The recognition rate will be significantly lower or perhaps even unacceptable with a poor microphone.
- Not all sound cards or sound devices are supported by SAPI 5, even if the operating system supports them otherwise.
- The following table outlines the RAM usage:

Component	Minimum RAM	Recommended RAM
TTS Engine	14.5 MB	32 MB
SR Command and Control	16 MB	32 MB
SR Dictation	25.5 MB	128 MB

SR Both 26.5 MB 128 MB

- The following table outlines the disk usage:

File Name	Approximate File Size	Setup Merge Names
Sapi.dll and Sapisvr.exe	0.5 MB	Sp5.msm
Sapi.cpl	36 KB	Sp5Intl.msm
SR Engine	1.7 MB	Sp5Sr.msm
Command and Control Datafiles	13.4 MB	Sp5CCInt.msm
TTS Engine and voices	7.8 MB	Sp5TTInt.msm
Files common to both Microsoft SAPI 5.1 TTS and SR.	92 KB	SpCommon.Msm
Language-specific SAPI 5.1 inverse text normalization (ITN) components.	108 KB	Sp5itn.Msm

For more information on setup, see the [Microsoft Speech SDK Setup 5.1](#).

Microsoft Speech Software Development Kit, Version 5.1

Microsoft Speech Software Development Kit, Version 5.1

END-USER LICENSE AGREEMENT FOR MICROSOFT SOFTWARE
IMPORTANT-READ CAREFULLY: This Microsoft End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and Microsoft Corporation for the Microsoft software product identified above, which includes DEVICE software (including SAPI 5.1, Microsoft continuous speech recognition engine and Microsoft concatenative speech synthesis engine), and may include associated media, printed materials, and "online" or electronic documentation ("SOFTWARE PRODUCT"). The SOFTWARE PRODUCT also includes any updates and supplements to the original SOFTWARE PRODUCT provided to you by Microsoft. Any software that may be provided along with the SOFTWARE PRODUCT that is associated with a separate end-user license agreement is licensed to you under the terms of that license agreement. By installing, copying, downloading, accessing or otherwise using the SOFTWARE PRODUCT, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, do not install or use the SOFTWARE PRODUCT.

SOFTWARE PRODUCT LICENSE

The SOFTWARE PRODUCT is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The SOFTWARE PRODUCT is licensed, not sold.

1. **GRANT OF LICENSE.** This EULA grants you the following rights:

- **SOFTWARE PRODUCT.** You may install copies of the SOFTWARE PRODUCT on up to ten (10) digital electronic devices, including computers, workstations, terminals, handheld PCs, pagers, "smart phones," or other digital electronic devices (each a "DEVICE") to design, develop, and test software programs that use the Microsoft Speech Application Programming Interface ("SAPI 5.1") and run on one or more Microsoft Windows operating system products that support SAPI 5.1 ("Windows Platforms"), provided that you are the only individual using the SOFTWARE PRODUCT on each such DEVICE. If you are a single entity, you may designate one individual within your organization to have the right to use the SOFTWARE PRODUCT in the manner described herein.

- **Sample Code.** Solely with respect to those portions of the SOFTWARE PRODUCT identified as sample code ("Sample Code"), Microsoft also grants you the right to modify the source code version of the Sample Code for the sole purposes of designing, developing, and testing software programs that use SAPI 5.1 (each, "a SAPI Application") and to reproduce and distribute the Sample Code along with any modifications thereof, in object code form only, provided that you comply with the Distribution Requirements described below. For purposes of this section, "modifications" shall mean enhancements to the functionality of the Sample Code.

- **Redistributable Code.** Portions of the SOFTWARE PRODUCT are designated as "Redistributable Code" file located in REDISTRIB.CHM. Your distribution rights associated with each file of the Redistributable Code are subject to the distribution requirements described below.

- **Distribution Requirements.** You may copy and redistribute the Sample Code and/or Redistributable Code (collectively "REDISTRIBUTABLE COMPONENTS") as described above, provided that (a) you distribute the REDISTRIBUTABLE COMPONENTS only in conjunction with, and as a part of, your SAPI Application; (b) your SAPI Application adds significant and primary functionality to the REDISTRIBUTABLE COMPONENTS; (c) the REDISTRIBUTABLE COMPONENTS only operate in conjunction with the Windows Platforms; (d) you do not permit further redistribution of the REDISTRIBUTABLE COMPONENTS by your end-user customers; (e) you do not use Microsoft's name, logo, or trademarks to market your SAPI Application; (f) you include a valid copyright notice on your SAPI Application; (g) you include the entire text located in REDISTRIB.CHM in your SAPI Application End User License Agreement; and (h) you agree to indemnify, hold harmless, and defend Microsoft from and against any claims or lawsuits, including attorneys' fees, that arise or result from the use or distribution of your SAPI Application. Contact Microsoft for the applicable royalties due and other licensing terms for all other uses and/or distribution of the REDISTRIBUTABLE COMPONENTS.

- **Reservation of Rights.** All rights not expressly granted are reserved by Microsoft.

2. DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS.

- **Limitations on Reverse Engineering, Decompilation, and Disassembly.** You may not reverse engineer, decompile, or disassemble the SOFTWARE PRODUCT, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.

- **Separation of Components.** The SOFTWARE PRODUCT is licensed as a single product. Its component parts may not be separated for use on more than one DEVICE.
- **Trademarks.** This EULA does not grant you any rights in connection with any trademarks or service marks of Microsoft.
- **Rental.** You may not rent, lease, or lend the SOFTWARE PRODUCT.
- **Support Services.** No technical support will be provided for the SOFTWARE PRODUCT.
- **Termination.** Without prejudice to any other rights, Microsoft may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy all copies of the SOFTWARE PRODUCT and all of its component parts.

3. **COPYRIGHT.**

All title and copyrights in and to the SOFTWARE PRODUCT (including but not limited to any images, photographs, animations, video, audio, music, text, and "applets" incorporated into the SOFTWARE PRODUCT), the accompanying printed materials, and any copies of the SOFTWARE PRODUCT are owned by Microsoft or its suppliers. All title and intellectual property rights in and to the content that may be accessed through use of the SOFTWARE PRODUCT is the property of the respective content owner and may be protected by applicable copyright or other intellectual property laws and treaties. This EULA grants you no rights to use such content. If this SOFTWARE PRODUCT contains documentation that is provided only in electronic form, you may print one copy of such electronic documentation. You may not copy the printed materials accompanying the SOFTWARE PRODUCT.

4. DUAL-MEDIA SOFTWARE.

You may receive the SOFTWARE PRODUCT in more than one medium. Regardless of the type or size of medium you receive, you may use only one medium that is appropriate for your single DEVICE. You may not install, copy or use the other medium on another DEVICE. You may not loan, rent, lease, or otherwise transfer the other medium to another user, except as part of the permanent transfer (as provided above) of the SOFTWARE PRODUCT.

5. U.S. GOVERNMENT RESTRICTED RIGHTS.

All SOFTWARE PRODUCTS provided to the U.S. Government pursuant to solicitations issued on or after December 1, 1995 is provided with the commercial license rights and restrictions described elsewhere herein. All SOFTWARE PRODUCTS provided to the U.S. Government pursuant to solicitations issued prior to December 1, 1995 is provided with "Restricted Rights" as provided for in FAR, 48 CFR 52.227-14 (JUNE 1987) or DFAR, 48 CFR 252.227-7013 (OCT 1988), as applicable. The reseller is responsible for ensuring that the SOFTWARE PRODUCT is marked with the "Restricted Rights Notice" or "Restricted Rights Legend," as required. All rights not expressly granted are reserved.

6. EXPORT RESTRICTIONS.

You acknowledge that the SOFTWARE PRODUCT is of U.S. origin. You agree to comply with all applicable international and national laws that apply to the SOFTWARE PRODUCT, including the U.S. Export Administration Regulations, as well as end-user, end-use and country/region destination restrictions issued by U.S. and other governments. For additional information on exporting Microsoft products, see <http://www.microsoft.com/exporting/>.

MISCELLANEOUS.

If you acquired this SOFTWARE PRODUCT in the United States, this EULA is governed by the laws of the State of Washington. If you acquired this SOFTWARE PRODUCT in Canada, unless expressly prohibited by local law, this EULA is governed by the laws in force in the Province of Ontario, Canada; and, in respect of any dispute which may arise hereunder, you consent to the jurisdiction of the federal and provincial courts sitting in Toronto, Ontario. If this SOFTWARE PRODUCT was acquired outside the United States, then local law may apply.

Should you have any questions concerning this EULA, or if you desire to contact Microsoft for any reason, please contact the Microsoft subsidiary serving your country/region, or e-mail:

sapi5@microsoft.com

NO WARRANTIES. MICROSOFT EXPRESSLY DISCLAIMS ANY WARRANTY FOR THE SOFTWARE PRODUCT. THE SOFTWARE PRODUCT AND ANY RELATED DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OR MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE PRODUCT REMAINS WITH YOU.

LIMITATION OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT

SERVICES, EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES AND JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

EXCLUSION DE GARANTIE. MICROSOFT EXCLUT EXPRESSÉMENT TOUTE GARANTIE RELATIVE AU PRODUIT LOGICIEL. LE PRODUIT LOGICIEL ET LA DOCUMENTATION Y AFFÉRENTE SONT FOURNIS "EN L'ÉTAT", SANS GARANTIE D'AUCUNE SORTE, EXPRESSE OU IMPLICITE, NOTAMMENT SANS AUCUNE GARANTIE IMPLICITE DE QUALITÉ, D'ADÉQUATION À UN USAGE PARTICULIER OU D'ABSENCE DE CONTREFAÇON. VOUS ASSUMEZ L'ENSEMBLE DES RISQUES DÉCOULANT DE L'UTILISATION OU DES PERFORMANCES DU PRODUIT LOGICIEL. LE PRÉSENT ARTICLE EST SANS PRÉJUDICE DE LA GARANTIE LÉGALE CONTRE LES VICES CACHÉS DONT VOUS POURRIEZ BÉNÉFICIER, LE CAS ÉCHÉANT.

LIMITATION DE RESPONSABILITÉ. DANS TOUTE LA MESURE PERMISE PAR LA RÉGLEMENTATION EN VIGUEUR, MICROSOFT OU SES FOURNISSEURS NE POURRONT EN AUCUN CAS ÊTRE TENUS POUR RESPONSABLES DE TOUT DOMMAGE, DE QUELQUE NATURE QUE CE SOIT, (NOTAMMENT ET DE MANIÈRE NON LIMITATIVE, TOUTE PERTE DE BÉNÉFICES, INTERRUPTION D'ACTIVITÉ, PERTE D'INFORMATIONS COMMERCIALES OU TOUTE AUTRE PERTE PÉCUNIAIRE) RÉSULTANT DE L'UTILISATION OU DE L'IMPOSSIBILITÉ D'UTILISER LE PRODUIT LOGICIEL OU DE LA FOURNITURE OU DU DÉFAUT DE FOURNITURE DES SERVICES D'ASSISTANCE, MÊME SI MICROSOFT A ÉTÉ PRÉVENU DE L'ÉVENTUALITÉ DE TELS DOMMAGES. CERTAINS PAYS ET CERTAINES JURIDICTIONS N'AUTORISENT PAS LES EXCLUSIONS OU LIMITATIONS DE RESPONSABILITÉ, DE SORTE QUE LA LIMITATION CI-DESSUS PEUT NE PAS VOUS ÊTRE APPLICABLE.



About the SDK

The Microsoft Speech SDK is designed to work with the industry-leading Speech Application Programming Interface (SAPI) and Microsoft continuous speech recognition engine and Microsoft concatenated speech synthesis engine (or text-to-speech). Microsoft SDK includes tools, samples, and documentation for building speech applications.

Microsoft Visual Basic Support

A set of COM-supported speech Automation interfaces is included in this release of Microsoft Speech SDK. That means languages other than C/C++ may now use SAPI for application development. The languages themselves need to support OLE automation. Common languages which may be used includes Visual Basic, C#, and JScript. See Automation Interfaces and Objects for additional information.

This section includes the following topics:

- [Legal Information](#)
- [Redistributable Code Rights](#)
- [Who Should Use This SDK](#)
- [How to Read Newsgroups](#)
- [Developer Support](#)
- [Platform SDK Requirements](#)



Legal Information for Microsoft Speech SDK

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. Companies, names, and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, unless expressly permitted by Microsoft Corporation.

Microsoft may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property rights.

© 1995-2000 Microsoft Corporation. All rights reserved.

Microsoft Speech Software Development Kit, Version 5.1

Redistributable Code

The Redistributable Code is the property of Microsoft Corporation and its suppliers and is protected by copyright law and international treaty provisions. You are authorized to make and use copies of the Redistributable Code either as part of the application in which you received the Redistributable Code, or in conjunction with the application for which its use is intended. Except as expressly provided in the foregoing sentence, you are not authorized to reproduce and distribute the Redistributable Code. Microsoft reserves all rights not expressly granted. You may not reverse engineer, decompile, or disassemble the Redistributable Code, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.

THE REDISTRIBUTABLE CODE IS PROVIDED TO YOU "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE. YOU ASSUME THE ENTIRE RISK AS TO THE ACCURACY AND THE USE OF THE REDISTRIBUTABLE CODE. MICROSOFT SHALL NOT BE LIABLE FOR ANY DAMAGES WHATSOEVER ARISING OUT OF THE USE OF OR INABILITY TO USE THE REDISTRIBUTABLE CODE, EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Redistributable Code

Redistributable Code is identified as the following files and all of the files can be found at the following location:

- [Licensing Microsoft Speech Technology](#)



Who Should Use This SDK

To get the most out of this SDK, you should be familiar with the following:

- C/C++ programming concepts.
- Visual Basic programming concepts. SAPI 5.1 supports OLE automation so that any language capable of accessing automation objects may be used. Visual Basic is a widespread and popular application that supports OLE automation. As a result many of SAPI SDK's samples and API references follow Visual Basic syntax. There is, however, no requirement to use Visual Basic for OLE automation.
- Component Object Model (COM). Developers should understand COM programming concepts, obtaining pointers to interfaces, and calling methods.
- All developers should understand the Win32 application programming interface (API).

For more information about Windows programming, see the documentation included in the Microsoft Win32 Software Development Kit.

For more information about C/C++ programming, see the documentation for the Microsoft® Visual C++™ development system for Windows.



Microsoft Speech Recognition Newsgroups

How to Read Newsgroups

You can use any newsreader software to access the Microsoft newsgroups. Microsoft Outlook Express is installed as part of Microsoft Internet Explorer for your convenience. Visit the [Microsoft Internet Explorer Home page](#) for information on how to download and install this product.

With Outlook Express installed, after you click a newsgroup link, you will be prompted for configuration information. When prompted for News Server, specify **newsvr**. You do not need to enter an account name or password; make sure that the option This Server Requires Me To Log On is not checked on the Server tab of the News Reader properties window.

If you are using an NNTP newsreader (sometimes called a news client) other than Outlook Express, be sure to configure it to read the Microsoft newsgroups. You can access the Microsoft news server at the same address above with **newsvr**.

Before posting to the newsgroups, please review the [Microsoft Newsgroup Rules of Conduct](#).

Available Newsgroups

[Microsoft Newsgroups](#) lists available newsgroups from Microsoft.



Developer Support

[Microsoft Support Center](#)

The Microsoft Support Center site maintains a wealth of resources to help you get the most from your product. Here you can search the entire Knowledge Base, view all the troubleshooting wizards, and access all the downloadable files. You can also view the extensive online glossary of computer terms, and find phone numbers and support options for all Microsoft products.

Speech newsgroups

For assistance with specific problems, check first with the following newsgroup. For advice on configuring a newsreader, see [How to Read Newsgroups](#).

- microsoft.public.speech_tech.sdk

Microsoft Speech.NET Technologies Feedback

If you have feedback on the Microsoft Speech API (SAPI) or other Microsoft Speech-related questions, please send an e-mail message to the following address. This e-mail address is monitored regularly, but questions will receive only limited response. Please use the newsgroup forum above for most questions and inquiries.

- sapi5@microsoft.com

Known Issues

Information on known issues in SAPI 5 can be found in the Knowledge Base articles in the [Microsoft Speech.NET Technologies](#) support home page.



Microsoft Platform SDK

The Microsoft Platform SDK (PSDK) is not a requirement in general for using SAPI. However, at least one SDK sample does require it. Other manufactures products may also require it for their products. Check with the specific SDK sample or manufacturer's product for complete details.

SAPI 5.1 SDK Samples Requiring PSDK

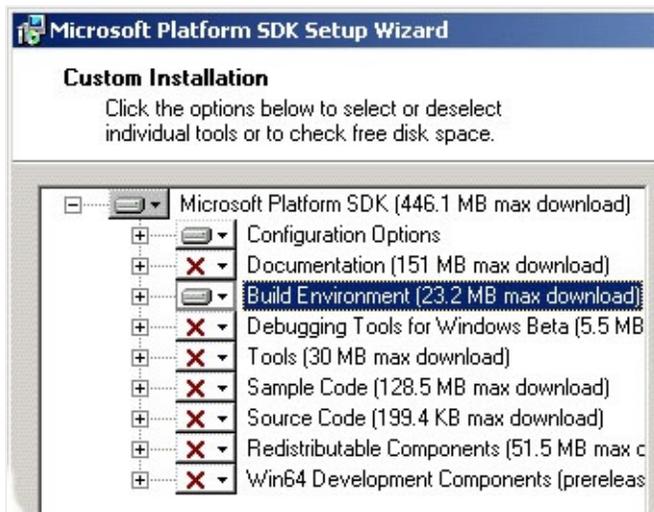
- [Simple Telephony](#)
- [Sample Speech Recognition Engine](#)

Download PSDK

If the PSDK is needed, it may downloaded from the [Microsoft Platform SDK](#) site.

Requirements

Platform SDK (PSDK) April 2000 or later edition. Compiling SDK projects requires components of the PSDK. Within Microsoft Visual C++ 6.0, the PSDK include directories must be listed before the Visual C++. Use the Directories tab to change the order in the Tools->Options menu. Move PSDK directories above all Visual C++ directories, if needed.



To save disk space, you can load a minimal configuration. This includes enabling only the following two options:

1. Configuration Options
2. Build Environment

These options may require 13 MB on the system drive and another 80 MB on any other drive. No other options are needed.



SAPI 5 Introduction

This section provides a SAPI 5 introduction. The following topic is available:

- [SAPI5 Overview](#)



SAPI 5 Overview

The SAPI application programming interface (API) dramatically reduces the code overhead required for an application to use speech recognition and text-to-speech, making speech technology more accessible and robust for a wide range of applications.

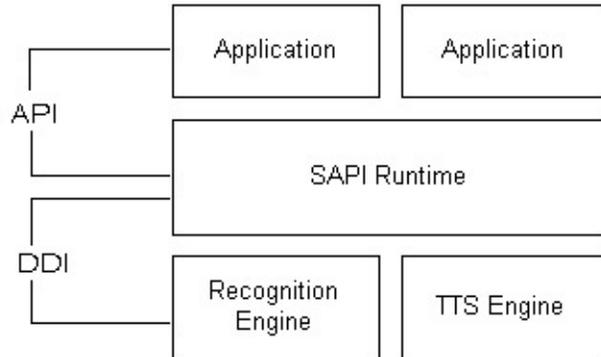
This section covers the following topics:

- [API Overview](#)
- [API for Text-to-Speech](#)
- [API for Speech Recognition](#)

API Overview

The SAPI API provides a high-level interface between an application and speech engines. SAPI implements all the low-level details needed to control and manage the real-time operations of various speech engines.

The two basic types of SAPI engines are text-to-speech (TTS) systems and speech recognizers. TTS systems synthesize text strings and files into spoken audio using synthetic voices. Speech recognizers convert human spoken audio into readable text strings and files.



API for Text-to-Speech

Applications can control text-to-speech (TTS) using the [ISpVoice](#) Component Object Model (COM) interface. Once an application has created an ISpVoice object (see [Text-to-Speech Tutorial](#)), the application only needs to call [ISpVoice::Speak](#) to generate speech output from some text data. In addition, the ISpVoice interface also provides several methods for changing voice and synthesis properties such as speaking rate [ISpVoice::SetRate](#), output volume [ISpVoice::SetVolume](#) and changing the current speaking voice [ISpVoice::SetVoice](#)

Special SAPI controls can also be inserted along with the input text to change real-time synthesis properties like voice, pitch, word emphasis, speaking rate and volume. This synthesis markup [sapi.xsd](#), using standard XML format, is a simple but powerful way to customize the TTS speech, independent of the specific engine or voice currently in use.

The [ISpVoice::Speak](#) method can operate either synchronously (return only when completely finished speaking) or asynchronously (return immediately and speak as a background process). When speaking asynchronously (SPF_ASYNC), real-time status information such as speaking state and current text location can be polled using [ISpVoice::GetStatus](#). Also while speaking asynchronously, new text can be spoken by either immediately interrupting the current output (SPF_PURGEBEFORESPEAK), or by automatically appending the new text to the end of the current output.

In addition to the ISpVoice interface, SAPI also provides many utility COM interfaces for the more advanced TTS applications.

Events

SAPI communicates with applications by sending events using standard callback mechanisms (Window Message, callback proc or Win32 Event). For TTS, events are mostly used for

synchronizing to the output speech. Applications can sync to real-time actions as they occur such as word boundaries, phoneme or viseme (mouth animation) boundaries or application custom bookmarks. Applications can initialize and handle these real-time events using [ISpNotifySource](#), [ISpNotifySink](#), [ISpNotifyTranslator](#), [ISpEventSink](#), [ISpEventSource](#), and [ISpNotifyCallback](#).

Lexicons

Applications can provide custom word pronunciations for speech synthesis engines using methods provided by [ISpContainerLexicon](#), [ISpLexicon](#) and [ISpPhoneConverter](#).

Resources

Finding and selecting SAPI speech data such as voice files and pronunciation lexicons can be handled by the following COM interfaces: [ISpDataKey](#), [ISpRegDataKey](#), [ISpObjectTokenInit](#), [ISpObjectTokenCategory](#), [ISpObjectToken](#), [IEnumSpObjectTokens](#), [ISpObjectWithToken](#), [ISpResourceManager](#) and [ISpTask](#).

Audio

Finally, there's an interface for customizing the audio output to some special destination such as telephony and custom hardware ([ISpAudio](#), [ISpMMSysAudio](#), [ISpStream](#), [ISpStreamFormat](#), [ISpStreamFormatConverter](#)).

☐ [Back to top](#)

API for Speech Recognition

Just as `ISpVoice` is the main interface for speech synthesis, [ISpRecoContext](#) is the main interface for speech recognition. Like the `ISpVoice`, it is an `ISpEventSource`, which means that it is the speech application's vehicle for receiving notifications for the requested speech recognition events.

An application has the choice of two different types of speech recognition engines ([ISpRecognizer](#)). A shared recognizer that could possibly be shared with other speech recognition applications is recommended for most speech applications. To create an `ISpRecoContext` for a shared `ISpRecognizer`, an application need only call COM's `CoCreateInstance` on the component `CLSID_SpSharedRecoContext`. In this case, SAPI will set up the audio input stream, setting it to SAPI's default audio input stream. For large server applications that would run alone on a system, and for which performance is key, an `InProc` speech recognition engine is more appropriate. In order to create an `ISpRecoContext` for an `InProc` `ISpRecognizer`, the application must first call `CoCreateInstance` on the component `CLSID_SpInprocRecoInstance` to create its own `InProc` `ISpRecognizer`. Then the application must make a call to [ISpRecognizer::SetInput](#) (see also [ISpObjectToken](#)) in order to set up the audio input. Finally, the application can call [ISpRecognizer::CreateRecoContext](#) to obtain an `ISpRecoContext`.

The next step is to set up notifications for events the application is interested in. As the `ISpRecognizer` is also an [ISpEventSource](#), which in turn is an [ISpNotifySource](#), the application can call one of the `ISpNotifySource` methods from its `ISpRecoContext` to indicate where the events for that `ISpRecoContext` should be reported. Then it should call [ISpEventSource::SetInterest](#) to indicate which events it needs to be notified of. The most important event is the `SPEI_RECOGNITION`, which indicates that the `ISpRecognizer` has recognized some speech for this `ISpRecoContext`. See [SPEVENTENUM](#) for details on the other

available speech recognition events.

Finally, a speech application must create, load, and activate an [ISpRecoGrammar](#), which essentially indicates what type of utterances to recognize, i.e., dictation or a command and control grammar. First, the application creates an ISpRecoGrammar using [ISpRecoContext::CreateGrammar](#). Then, the application loads the appropriate grammar, either by calling [ISpRecoGrammar::LoadDictation](#) for dictation or one of the [ISpRecoGrammar::LoadCmdxxx](#) methods for command and control. Finally, in order to activate these grammars so that recognition can start, the application calls [ISpRecoGrammar::SetDictationState](#) for dictation or [ISpRecoGrammar::SetRuleState](#) or [ISpRecoGrammar::SetRuleIdState](#) for command and control.

When recognitions come back to the application by means of the requested notification mechanism, the *IParam* member of the [SPEVENT](#) structure will be an [ISpRecoResult](#) by which the application can determine what was recognized and for which ISpRecoGrammar of the ISpRecoContext.

An ISpRecognizer, whether shared or InProc, can have multiple ISpRecoContexts associated with it, and each one can be notified in its own way of events pertaining to it. An ISpRecoContext can have multiple ISpRecoGrammars created from it, each one for recognizing different types of utterances.

□ [Back to top](#)



Application-Level Interfaces

This section describes the interfaces and methods for incorporating speech into applications. They are intended for use at the API or application level. Some managers or interfaces may have entries also in [Engine-Level Interfaces](#) section. However, entries listed here apply only to the application level.

- [Audio interfaces](#)
- [Eventing interfaces](#)
- [Grammar Compiler interfaces](#)
- [Lexicon interfaces](#)
- [Resource interfaces](#)
- [Speech Recognition interfaces](#)
- [Text-to-Speech interfaces](#)



Audio interfaces

This section provides SAPI 5 audio interfaces.

Audio inherits from the standard COM IStream interface. See the MSDN documentation for a complete discussion of IStream and associated methods. However, since the audio devices represent hardware, ::Clone may be not be used and will return E_NOTIMPL.

- [ISpAudio](#)
- [ISpMMSysAudio](#)
- [ISpStream](#)
- [ISpStreamFormat](#)
- [ISpStreamFormatConverter](#)

The following interface does not inherit from IStream:

- [ISpTranscript](#)

Development Helpers

Helper Enumerations, Functions and Classes	Description
SPSTREAMFORMAT	SAPI supported stream formats.
CSpEvent	Class for decoding event structures.
CSpDynamicString	Class for managing dynamically sized WCHAR strings.
SpBindToFile	Function converts the specified stream format into a wave format structure.
CSpStreamFormat	Class for managing SAPI supported stream formats and WAVEFORMATEX structures.



ISpAudio

Objects implementing this interface are real-time audio streams, such as those connected to a live microphone or telephone line. ISpAudio methods support control of real-time audio streams. IStream Read and Write methods transfer data to or from an object.

When to Implement

This interface should be implemented when the audio input or output source is not a standard windows Multimedia device. It is expected to supply an infinite amount of data and hence its state should not change externally to SAPI. For the majority of users, it will not be necessary to implement an object providing this interface. An example of where this might be needed would be to provide a telephony audio device or to perform echo cancellation of audio output on the input.

Telephone application programming interface (TAPI) provides a mechanism to treat a telephony device as a Windows multimedia device allowing the use of the SAPI provided multimedia audio objects.

If this implements a real time audio input or output object and runs in a desktop or graphical environment, support may be needed for volume (see [SPDUI_AudioVolume](#)) and audio properties (see [SPDUI_AudioProperties](#)) UI. The preferred method for SAPI to implement the UI is to have the object inherit from [ISpTokenUI](#). This will enable applications (including Speech properties in Control Panel) to display the UI in a simple and consistent manner.

In order to prevent multiple TTS voices or engines from speaking simultaneously, SAPI serializes output to objects which implement the ISpAudio interface. To disable serialization of outputs to an ISpAudio object, place an attribute called "NoSerializeAccess" in the Attributes folder of its object token.

Implemented By

- [SpMMAudioIn](#)
- [SpMMAudioOut](#)

Methods in Vtable Order

Value	Description
ISpStreamFormat interface	Inherits from ISpStreamFormat and all those methods are accessible from an ISpAudio object.
SetState	Sets the state of the audio device.
SetFormat	Sets the format of the audio device.
GetStatus	Passes back the status of the audio device.
SetBufferInfo	Sets the audio stream buffer information.
GetBufferInfo	Passes back the audio stream buffer information.
GetDefaultFormat	Passes back the default audio format.
EventHandle	Returns a Win32 event handle that applications can use to wait for status changes in the I/O stream.
GetVolumeLevel	Passes back the current volume level.
SetVolumeLevel	Sets the current volume level.
GetBufferNotifySize	Retrieves the audio stream buffer size information.
SetBufferNotifySize	Sets the audio stream buffer size information.

Development Helpers

Helper Enumerations, Functions and Classes	Description

SPSTREAMFORMAT

SAPI supported stream formats.

CSpStreamFormat

Class for managing SAPI supported stream formats and WAVEFORMATEX structures.



ISpAudio::SetState

ISpAudio::SetState sets the state of the audio device.

```
HRESULT SetState(  
    SPAUDIOSTATE    NewState,  
    ULONGLONG        ullReserved  
);
```

Parameters

NewState

[in] The flag of type [SPAUDIOSTATE](#) for the new state of the audio device.

ullReserved

[in] Reserved, do not use. This value must be zero.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>ullReserved</i> is not zero or <i>NewState</i> is not one of the allowed values.
SPERR_DEVICE_BUSY	Hardware device is in use by another thread or process.
SPERR_UNSUPPORTED_FORMAT	Current format set by ISpAudio::SetFormat is not supported by the hardware device.

Remarks

When transitioning from the SPAS_CLOSED state to any other state, the caller should be ready to handle various error conditions, specifically, SPERR_UNSUPPORTED_FORMAT and SPERR_DEVICE_BUSY. Many multi-media devices do not correctly report their capabilities for handling different audio formats and fail only when an attempt is made to open the device.

Also, in many older systems, audio output devices can be opened only by a single process. In all current versions of Windows, only a single process can open an audio input device. Therefore, SPERR_DEVICE_BUSY will return if an attempt is made to open a device that is being used by a different process or thread.

On some older sound cards, recording and playback are not possible simultaneously or only possible at the same frequency. An application making use of the input and output audio should be aware of this and in particular attempt to gracefully degrade from higher quality frequencies to the same frequency for both if the sound card makes this necessary.

In general, applications need not change the state of the audio device directly. With the shared recognizer in particular, this will often cause unexpected results. SAPI will automatically manage the state of the audio device based on the state of all the grammars, recognition contexts and the recognizer instance.



ISpAudio::SetFormat

ISpAudio::SetFormat sets the format of the audio device.

```
HRESULT SetFormat(  
    REFGUID          rguidFmtId,  
    const WAVEFORMATEX *pWaveFormatEx  
);
```

Parameters

rguidFmtId

[in] The REFGUID for the format to set. Typically this will be SPDFID_WaveFormatEx. This is required for the SAPI multimedia objects.

pWaveFormatEx

[in] Address of the [WAVEFORMATEX](#) structure containing the wave file format information.

Return values

Value	Description
S_OK	Function completed successfully. See note about supported formats.
E_INVALIDARG	<i>pWaveFormatEx</i> is invalid or bad.
SPERR_DEVICE_BUSY	Device is not in the SPAS_CLOSED state.
SPERR_UNINITIALIZED	Audio stream not initialized.
SPERR_UNSUPPORTED_FORMAT	Specified format is not supported.
FAILED(hr)	Appropriate error message.

Remarks

This method can be called only when the audio device is in the [SPAS_CLOSED](#) state. Note that successfully setting the format on an audio device does not necessarily mean that the format is supported. An attempt must be made to place the device into a non-closed state (SPAS_STOP, SPAS_PAUSE or SPAS_RUN) to be sure that the device can handle the format.

The format can be retrieved by calling the [ISpStreamFormat::GetFormat](#) method.

The helper class [CSpStreamFormat](#) and the [SPSTREAMFORMAT](#) enumeration can be used to avoid the possibility of typos or mistakes when filling in the [WAVEFORMATEX](#) structure.



ISpAudio::GetStatus

ISpAudio::GetStatus passes back the status of the audio device.

This method determines whether the device is running, stopped, closed, or paused. It also determines the size of any buffered data.

```
HRESULT GetStatus(  
    SPAUDIOSTATUS *pStatus  
);
```

Parameters

pStatus

[out] Pointer to an [SPAUDIOSTATUS](#) buffer to be filled with the status details.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pStatus</i> is invalid.



ISpAudio::SetBufferInfo

ISpAudio::SetBufferInfo sets the audio stream buffer information.

```
HRESULT SetBufferInfo(  
    const SPAUDIOBUFFERINFO *pBuffInfo  
);
```

Parameters

pBuffInfo

[in] Pointer to the [SPAUDIOBUFFERINFO](#) buffer providing the requested settings.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_UNINITIALIZED	Audio stream not initialized.
E_INVALIDARG	<i>pBuffInfo</i> is invalid or the parameters do not meet the criteria described above.
SPERR_DEVICE_BUSY	Audio device is not in the SPAS_CLOSED state.
FAILED(hr)	Appropriate error message.

Remarks

This method can be called only when the audio device is in the [SPAS_CLOSED](#) state. The [SPAUDIOBUFFERINFO](#) members must conform to the following restrictions:

SPAudioBufferInfo.ulMsMinNotification cannot be larger than one quarter the size of SPAudioBufferInfo.ulMsBufferSize and

must not be zero.

SPAudioBufferInfo.ulMsEventBias cannot be larger than
SPAudioBufferInfo.ulMsBufferSize.

SPAudioBufferInfo.ulMsBufferSize must be greater than or
equal to 200 milliseconds.



ISpAudio::GetBufferInfo

ISpAudio::GetBufferInfo passes back the audio stream buffer information.

```
HRESULT GetBufferInfo(  
    SPAUDIOBUFFERINFO *pBuffInfo  
);
```

Parameters

pBuffInfo

[out] Pointer to the SPAUDIOBUFFERINFO buffer.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pBuffInfo</i> is invalid.



ISpAudio::GetDefaultFormat

ISpAudio::GetDefaultFormat passes back the default audio format.

```
HRESULT GetDefaultFormat(  
    GUID          *pFormatId,  
    WAVEFORMATEX **ppCoMemWaveFormatEx  
);
```

Parameters

pFormatId

[out] Pointer to the GUID of the default format.

ppCoMemWaveFormatEx

[out] Address of a pointer to the [WAVEFORMATEX](#) structure that receives the wave file format information. SAPI allocates the memory for the [WAVEFORMATEX](#) data structure using CoTaskMemAlloc, but it is the caller's responsibility to call CoTaskMemFree on the returned [WAVEFORMATEX](#) pointer.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_UNINITIALIZED	Stream is uninitialized.
E_POINTER	<i>pFormatId</i> is invalid.

Remarks

Other formats may be supported by the audio device; however, this format is guaranteed to work. Older sound cards can potentially fail when attempting to use this format if they are not fully duplex or do not support full duplex operation at

different frequencies. An application should attempt to degrade gracefully when this occurs.



ISpAudio::EventHandle

ISpAudio::EventHandle returns a Win32 event handle that applications can use to wait for status changes in the I/O stream.

```
HANDLE EventHandle( void );
```

Parameters

None

Return values

Value	Description
HANDLE	Returns valid event handle.

Remarks

The handle may use one of the various Win32 wait functions, such as `WaitForSingleObject` or `WaitForMultipleObjects`.

For read streams, set the event when there is data available to read and reset it whenever there is no available data. For write streams, set the event when all of the data has been written to the device, and reset it at any time when there is still data available to be played.

The caller should not close the returned handle, nor should the caller ever use the event handle after calling `Release()` on the audio object. The audio device will close the handle on the final release of the object.



ISpAudio::GetVolumeLevel

ISpAudio::GetVolumeLevel passes back the current volume level.

The volume level is on a linear scale from zero to 10000.

```
HRESULT GetVolumeLevel(  
    ULONG    *pLevel  
);
```

Parameters

pLevel

[out] Pointer to the returned volume level.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_UNINITIALIZED	Audio interface is not initialized.
SPERR_DEVICE_NOT_SUPPORTED	The device is not valid or does not support volumes.
E_POINTER	<i>pLevel</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Remarks

For input devices with a boost control for the microphone, SAPI will split the volume range into two to allow automated use of the boost. The boost will be off from zero to 4999 and on from 5000 to 10,000. In each range, the full volume range of the device will be used independently. This can lead to discontinuity in the input energy level for a constant volume sound source.

On some sound cards, the boost is applied to the input volume, but on others, the boost is applied to the playback volume resulting in the two ranges behaving identically because the input level is unaffected.

Microphone wizards determining the best volume input level should take into consideration the potential discontinuity and ensure that the algorithm used to adjust the input volume level can handle the various possible forms of discontinuity at 5,000

For devices without a boost, there is no discontinuity at 5000.



ISpAudio::SetVolumeLevel

ISpAudio::SetVolumeLevel sets the current volume level. It is on a linear scale from zero to 10000.

```
HRESULT SetVolumeLevel(  
    ULONG    Level  
);
```

Parameters

Level

[in] The new volume level.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Level is greater than 10,000.
SPERR_DEVICE_NOT_SUPPORTED	The device is not valid or does not support volumes.
FAILED(hr)	Appropriate error message.

Remarks

For input devices with a boost control for the microphone, SAPI will split the volume range into two to allow automated use of the boost. The boost will be off from zero to 4,999 and on from 5,000 to 10,000. In each range, the full volume range of the device will be used independently. This can lead to discontinuity in the input energy level for a constant volume sound source. On some sound cards, the boost is applied to the input volume, but on others, the boost is applied to the playback volume resulting in the two ranges performing identically because the

input level is unaffected.

Microphone wizards determining the best volume input level should take into consideration the potential discontinuity and ensure that the algorithm used to adjust the input volume level can handle the various possible forms of discontinuity at 5,000

For devices without a boost, there is no discontinuity at 5,000.



ISpAudio::GetBufferNotifySize

ISpAudio::GetBufferNotifySize retrieves the audio stream buffer size information. This information is used to determine when the event returned by [ISpAudio::EventHandle](#) is set or reset.

```
HRESULT GetBufferNotifySize(  
    ULONG    *pcbSize  
);
```

Parameters

pcbSize

[out] Address of the size information, specified in bytes, that is associated with the audio stream buffer.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Invalid pointer.

Remarks

For read streams, the event is set if the audio buffered is greater than or equal to the value set in *pcbSize*, otherwise the event information is reset.

For write streams, the event is set if the audio buffered is less than the value set in *pcbSize*, otherwise the event information is reset.



ISpAudio::SetBufferNotifySize

ISpAudio::SetBufferNotifySize sets the audio stream buffer size information. This information is used to determine when the event returned by [ISpAudio::EventHandle](#) is set or reset.

```
HRESULT SetBufferNotifySize(  
    ULONG    cbSize  
);
```

Parameters

cbSize

[in] The size, specified in bytes, of the information associated with the audio stream buffer.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.

Remarks

For read streams, the event is set if the audio buffered is greater than or equal to the value set in *cbSize*, otherwise the event information is reset.

For write streams, the event is set if the audio buffered is less than the value set in *cbSize*, otherwise the event information is reset.



ISpMMSysAudio

This is the interface to the audio implementation for the standard Windows multimedia layer (wave in and wave out). Audio objects created through an object token do not allow [ISpMMSysAudio::SetDeviceId](#) to work because the token specifies which audio device ID to use. If an application wants to associate an audio object with a specific multimedia wave in or wave out device ID, it should use CoCreateInstance with CLSID_SpMMAudioOut or CLSID_SpMMAudioIn and then use the [ISpMMSysAudio::SetDeviceId](#) method to select the device. In normal application development, this will not be necessary for two reasons:

- Desktop applications will generally use the shared recognizer instance which automatically uses the default audio device. This cannot be changed.
- For the InProc recognizer instance, tokens exist for the quick creation of the correct [SpMMAudioIn](#) or [SpMMAudioOut](#) for all of the multimedia devices on the system.

For input devices, SAPI will attempt to automatically identify the microphone line on the input device. On non-English versions of Windows and on a small number of English systems, it may not be possible for SAPI to automatically detect the correct microphone line. In this case, no error will be detected or returned. In such cases, if speech input is not correctly detected, the user must set the microphone input line directly using Speech properties in Control Panel-->Speech Recognition tab-->Audio Input Settings-->Properties. In particular, it may be necessary to adjust the microphone input line used on devices with multiple microphone inputs such as the SoundBlaster Live Platinum because the default input line may not be the input preferred by the user.

Implemented By

- [SpMMAudioIn](#)
- [SpMMAudioOut](#)

Methods in Vtable Order

ISpMMSysAudio Methods	Description
ISpStreamFormat interface	Inherits from ISpStreamFormat and all methods are accessible from an ISpMMSysAudio object.
ISpAudio interface	Inherits from ISpAudio and all methods are accessible from an ISpMMSysAudio object.
GetDeviceId	Passes back the multimedia device ID being used by the audio object.
SetDeviceId	Sets the multimedia device ID.
GetMMHandle	Passes back a multimedia audio stream handle.
GetLineId	Retrieves the line identifier associated with the multimedia device.
SetLineId	Sets the line identifier associated with the multimedia device.

IStream functions as inherited from ISpAudio

Please see the Microsoft[®] [Platform Software Development Kit](#) (PSDK) for a complete description of the IStream interface.

IStream Methods	Description
Read	Reads data from the multimedia audio device.
Write	Writes data to the multimedia audio device.
Seek	Retrieves only the current device

	position because multimedia devices represent hardware.
SetSize	Not used because multimedia devices represent hardware.
CopyTo	Copies a specified number of bytes from the current seek pointer in the stream to the current seek pointer in another stream.
Commit	Updates device state and commit buffered data. SAPI will automatically manage device state and buffered data, so that the developer is not expected to call this method.
Revert	Returns E_NOTIMPL because multimedia devices represent hardware.
LockRegion	Returns E_NOTIMPL because multimedia devices represent hardware.
UnlockRegion	Returns E_NOTIMPL because multimedia devices represent hardware.
Stat	Retrieves the current device position.
Clone	Returns E_NOTIMPL because multimedia devices represent hardware.



ISpMMSysAudio::Read

ISpMMSysAudio::Read reads the data from the multimedia audio device.

The audio device should not be directly manipulated when performing speech recognition (see [ISpRecognizer](#)).

```
HRESULT Read(  
    void      *pv,  
    ULONG     cb,  
    ULONG     *pcbRead  
);
```

Parameters

pv

[in] Pointer to the buffer into which the stream data is read. If an error occurs, this value is NULL.

cb

[in] Specifies the number of bytes of data to attempt to read from the audio device.

pcbRead

[out] Pointer to a ULONG variable that receives the actual number of bytes read from the stream object. If set to NULL, no byte value is passed back.

Return values

Value	Description
S_OK	Function completed successfully.

SPERR_AUDIO_BUFFER_OVERFLOW	SAPI's internal audio buffer has filled, and the device has been closed. See <i>Remarks section</i> .
SPERR_AUDIO_BUFFER_UNDERFLOW	The multimedia object has not received audio data from the device quickly enough, and the device has been closed. See <i>Remarks section</i> .
SPERR_AUDIO_STOPPED	Multimedia device state has been set to stopped.
E_OUTOFMEMORY	Exceeded available memory
E_POINTER	At least one of <i>pcbRead</i> or <i>pv</i> are invalid or bad.
STG_E_ACCESSDENIED	Multimedia device is read-only and no bytes will be read. Error will occur when reading from an output device.
FAILED (hr)	Appropriate error message.

Remarks about audio buffer overflows and underflows

SAPI automatically stores data in a buffer before it is read from the device. Buffering the audio data ensures that applications and SAPI-compliant speech recognition engines will not lose real-time audio data.

An errant application or speech recognition engine that does not call *Read* often enough could frequently fill the audio buffer. To ensure that large amounts of system memory are not filled, SAPI limits the buffer size to 30 times the average bytes per

second ([WAVEFORMATEX->nAvgBytesPerSec](#))- approximately 30 seconds. If the audio buffer is filled, SAPI will automatically set the device state to SPAS_CLOSED (see [SPAUDIOSTATE](#)) and return a buffer overflow error (i.e., SPERR_AUDIO_BUFFER_OVERFLOW) when `ISpMMSysAudio::Read` is called.

An errant multimedia device (and/or driver) that does not return audio data quickly enough could greatly reduce the speed of a speech application or SR engine. To prevent the degradation of application or SR engine performance, SAPI requires that the multimedia device return data at least once every five seconds. If the audio is not returned before five seconds, SAPI will automatically set the device state to SPAS_CLOSED (see [SPAUDIOSTATE](#)) and return a buffer underflow error (i.e., SPERR_AUDIO_BUFFER_UNDERFLOW) when `ISpMMSysAudio::Read` is called.

Applications should manually reopen the audio device (see [ISpAudio::SetState](#)) to prevent losing input data that could impact the user.

For SR engines, SAPI automatically attempts to restart the multimedia device after the SR engine exits [ISpSREngine::RecognizeStream](#).



ISpMMSysAudio::Write

ISpMMSysAudio::Write writes data to the multimedia audio device.

The audio device should not be directly manipulated when performing speech recognition (see [ISpRecognizer](#)).

```
HRESULT Write(  
    const void *pv,  
    ULONG      cb,  
    ULONG      *pcbWritten  
);
```

Parameters

pv

[in] Pointer to the buffer containing the data that is to be written to the audio device. A valid pointer must be provided for this parameter even when *cb* is zero.

cb

[in] The number of bytes of data to attempt to write to the audio device. This value may be zero.

pcbWritten

[out] Pointer to a ULONG variable where this method writes the actual number of bytes written to the audio device. If set to NULL, no byte value is passed back.

Return values

Value	Description
S_OK	Function completed successfully.

SPERR_AUDIO_STOPPED	Multimedia device has been stopped.
E_OUTOFMEMORY	Exceeded available memory.
E_POINTER	At least one of <i>pcbWritten</i> or <i>pv</i> are invalid or bad.
STG_E_ACCESSDENIED	Multimedia device is write-only and no bytes will be written. Error will occur when writing to an input device.
FAILED (hr)	Appropriate error message.



ISpMMSysAudio::GetDeviceId

ISpMMSysAudio::GetDeviceId passes back the multimedia device ID being used by the audio object.

```
HRESULT GetDeviceId(  
    UINT *puDeviceId  
);
```

Parameters

puDeviceId

[out] Pointer receiving the device ID.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>puDeviceId</i> is a bad pointer.

Remarks

The default device ID for SpMMSysAudio objects that are created using CoCreateInstance is the WAVE_MAPPER. For audio objects created using an object token, the ID will always be a specific wave in or wave out device ID.

Example

The following code snippet illustrates the use of `ISpMMSysAudio::GetDeviceId` using `CoCreateInstance`.

```
HRESULT hr = S_OK;

// create the multimedia input object
hr = cpMMSysAudio.CoCreateInstance(CLSID_SpMMAudioIn);
// Check hr

// get the default device id
UINT uiDeviceId;
hr = cpMMSysAudio->GetDeviceId(&uiDeviceId);
// Check hr

// uiDeviceId == WAVE_MAPPER
```

The following code snippet illustrates the use of `ISpMMSysAudio::GetDeviceId` using an [ISpObjectToken](#)

```
HRESULT hr = S_OK;

// get the current multimedia object's object token
hr = cpMMSysAudio.QueryInterface(&cpObjectWithToken);
// Check hr

// Find the preferred multimedia object token
hr = SpFindBestToken(SPCAT_AUDIOIN, L"Technology=MMSys",
// Check hr

// set the current multimedia object to the preferred mu.
hr = cpObjectWithToken->SetObjectToken(cpObjectToken);
// Check hr

// get the device id for the object
UINT uiDeviceId;
hr = cpMMSysAudio->GetDeviceId(&uiDeviceId);
// Check hr

// uiDeviceId != WAVE_MAPPER
```



ISpMMSysAudio::SetDeviceId

ISpMMSysAudio::SetDeviceId sets the multimedia device ID.

```
HRESULT SetDeviceId(  
    UINT    uDeviceId  
);
```

Parameters

uDeviceId

[in] The device ID of the device to set.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_DEVICE_BUSY	Object is not in the SPAS_CLOSED state.
SPERR_ALREADY_INITIALIZED	Object was created using an object token.
E_INVALIDARG	<i>uDeviceId</i> is invalid. It is not set to WAVE_MAPPER or device does not exist.

Remarks

This method works only on audio objects that were not created using an object token, and only when the object is in the SPAS_CLOSED state. This method should not be used in normal application development. SAPI provides tokens for all the available sound devices in a computer and these can be used to create an initialized SpMMSysAudio object. This method is available for non-standard multimedia audio devices. See the [Simple Telephony](#) sample for an example of when this method is

useful.

Example

The following code snippet illustrates the use of `ISpMMSysAudio::SetDeviceId`.

```
HRESULT hr = S_OK;

// create the multimedia output object
hr = cpMMSysAudio.CoCreateInstance(CLSID_SpMMAudioOut);
// Check hr

// set the output device to an alternate multimedia device
hr = cpMMSysAudio->SetDeviceId(ALTERNATE_MM_DEVICE);
// Check hr
```



ISpMMSysAudio::GetMMHandle

ISpMMSysAudio::GetMMHandle passes back a multimedia audio device stream handle.

```
HRESULT GetMMHandle(  
    void    **pHandle  
);
```

Parameters

pHandle

The wave in or wave out device handle.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pHandle</i> is invalid.
SPERR_UNINITIALIZED	Audio object is in the SPAS_CLOSED state.

Remarks

The audio object must not be in the SPAS_CLOSED state or this call will fail because the multimedia device will not have been opened yet. The caller must not close the passed back handle. The caller must not use the handle either after changing the state of the audio object to SPAS_CLOSED or after releasing the object.



ISpMMSysAudio::GetLineId

ISpMMSysAudio::GetLineId retrieves the current line identifier associated with the multimedia device. Mixer lines are not supported for output devices.

```
HRESULT GetLineId(  
    UINT    *puLineId  
);
```

Parameters

puLineId

[out] Address of the structure that receives the line identifier information.

Return values

Value	Description
S_OK	Function completed successfully.
E_NOTIMPL	Not implemented for output devices.
E_POINTER	<i>puLineId</i> is invalid.
SPERR_NOT_FOUND	The audio device must have been created from a token.
FAILED(hr)	Appropriate error message.

Remarks

For more information on the uses of device lines, please see the Win32 multimedia mixer API (e.g., mixerOpen, mixerGetId, mixerGetLineInfo, etc.)

For input devices, SAPI will attempt to automatically identify the microphone line on the input device. On non-English versions of Windows and on a small number of English systems, it may not

be possible for SAPI to automatically detect the correct microphone line. In this case, no error will be detected or returned. In such cases, if speech input is not correctly detected, the user must set the microphone input line directly using Control Panel-->Speech properties-->Speech Recognition tab-->Audio Settings-->Properties. In particular, this may be necessary to adjust the microphone input line used on devices with multiple microphone inputs such as the SoundBlaster Live Platinum as the automatically chosen line may not be the input preferred by the user.



ISpMMSysAudio::SetLineId

ISpMMSysAudio::SetLineId sets the line identifier associated with the multimedia device. Mixer lines are not supported for output devices.

```
HRESULT SetLineId(  
    UINT    uLineId  
);
```

Parameters

uLineId

[in] Value specifying the line identifier information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	The specified <i>uLineId</i> is not supported on the current device.
SPERR_NOT_FOUND	The audio device must have been created from a token.
E_NOTIMPL	Not implemented for output devices.
FAILED(hr)	Appropriate error message.

Remarks

For more information on the uses of device lines, please see the Win32 multimedia mixer API (e.g., mixerOpen, mixerGetId, mixerGetLineInfo, etc.)

For input devices, SAPI attempts to automatically identify the microphone line on the input device. On non-English versions of Windows and on a few English systems, it may not be possible

for SAPI to automatically detect the correct microphone line. In this case, no error will be detected or returned. If speech input is not correctly detected, the user must set the microphone input line directly using Control Panel-->Speech properties-->Speech Recognition tab-->Audio Settings-->Properties. It may be necessary to adjust the microphone input line used on devices with multiple microphone inputs such as the SoundBlaster Live Platinum. In these cases, the automatically chosen line may not be the input preferred by the user.



ISpStream

This interface provides two distinct functions:

- The application developer can wrap an existing stream up by providing both an IStream and its format so that the underlying [ISpStreamFormat](#) can provide this data to SAPI when required. The new ISpStream object can be used as an input for SAPI wherever SAPI requires an [ISpStreamFormat](#).
- ISpStream creates an object from a file suitable for SAPI usage using [BindToFile](#). The helper function [SPBindToFile](#) may also be used to simplify this process even further.

Implemented By

- [SpStream](#)

Methods in Vtable Order

ISpStream Methods	Description
SetBaseStream	Sets the base address of the audio stream.
GetBaseStream	Retrieves the base address of the audio stream.
BindToFile	Binds the audio stream to the file that it identifies.
Close	Closes the audio stream.

Development Helpers

Helper Enumerations, Functions and Classes	Description
SPSTREAMFORMAT	SAPI supported stream formats
CSpStreamFormat	Class for managing SAPI supported

stream formats and WAVEFORMATEX
structures



ISpStream::SetBaseStream

ISpStream::SetBaseStream initializes the ISpStream object with the format of the IStream and an object to encapsulate.

```
HRESULT SetBaseStream(  
    IStream          *pStream,  
    REFGUID         rguidFormat,  
    const WAVEFORMATEX *pWaveFormatEx  
);
```

Parameters

pStream

Address of the IStream containing the base stream data.

rguidFormat

The data format identifier associated with the stream.

pWaveFormatEx

Address of the [WAVEFORMATEX](#) structure that contains the wave file format information. If guidFormatId is SPDFID_WaveFormatEx, this must point to a valid [WAVEFORMATEX](#) structure. For other formats, it should be NULL.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
SPERR_ALREADY_INITIALIZED	The object has already been initialized.

FAILED (hr)

Appropriate error message.

Remarks

The helper class [CSpStreamFormat](#) and the [SPSTREAMFORMAT](#) enumeration can be used to avoid the possibility of typos or mistakes when filling in the [WAVEFORMATEX](#) structure.



ISpStream::GetBaseStream

ISpStream::GetBaseStream retrieves the encapsulated IStream object for an instance of the ISpStream object.

```
HRESULT GetBaseStream(  
    IStream    **ppStream  
);
```

Parameters

ppStream

Address of a pointer to the encapsulated IStream that contains an audio stream or text stream.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	The <i>ppStream</i> pointer is invalid or bad.
SPERR_STREAM_CLOSED	The stream is closed or unavailable.
S_FALSE	The ISpStream instance has not been initialized with an IStream.



ISpStream::BindToFile

ISpStream::BindToFile binds the input stream to the file that it identifies.

```
HRESULT BindToFile(  
    const WCHAR          *pszFileName,  
    SPFILEMODE          eMode,  
    const GUID           *pguidFormatId,  
    const WAVEFORMATEX *pWaveFormatEx,  
    ULONGLONG            ullEventInterest  
);
```

Parameters

pszFileName

Address of a null-terminated string containing the file name of the file to bind the stream to.

eMode

Flag of the type [SPFILEMODE](#) to define the file opening mode. When opening an audio wave file, *eMode* must be `SPFM_OPEN_READONLY` or `SPFM_CREATE_ALWAYS`, otherwise the call will fail.

pguidFormatId

The data format identifier associated with the stream. This can be NULL and the format will be determined from the supplied wave file, if the file has the wav extension. If it does not, the file is assumed to be a text file.

pWaveFormatEx

Address of the [WAVEFORMATEX](#) structure that contains the wave file format information. If *guidFormatId* is

SPDFID_WaveFormatEx, this must point to a valid [WAVEFORMATEX](#) structure. For other formats, it should be NULL.

ullEventInterest

Flags of type [SPEVENTENUM](#) for the format converter to watch.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	At least one of the following was encountered. <i>pszFileName</i> or <i>pguidFormatId</i> is invalid or bad; <i>eMode</i> exceeds SPFM_CREATE_ALWAYS; an operation could not be completed.
E_OUTOFMEMORY	Exceeded available memory.
STG_E_FILENOTFOUND	File <i>pszFileName</i> does not exist.
SPERR_ALREADY_INITIALIZED	The object has already been initialized.
FAILED (hr)	Appropriate error message.

Remarks

In speech recognition, `::BindToFile` supports only wave audio files. It passes SAPI an audio file to pass to the engine. In text-to-speech, `::BindToFile` supports both audio and text files. See [ISpVoice::SpeakStream](#) for more information.

The helper class [CSpStreamFormat](#) and the [SPSTREAMFORMAT](#) enumeration can be used to avoid the possibility of typos or mistakes when filling in the [WAVEFORMATEX](#) structure.

Example

The following code snippet illustrates the use of `ISpStream::BindToFile` for creating a writable wave file

```
HRESULT hr = S_OK;

// create the stream object
hr = cpSpStream.CoCreateInstance(CLSID_SpStream);
// Check hr

// create a stream format helper for 22khzm 16-bit, mono
CSpStreamFormat Fmt(SPSF_22kHz16BitMono, &hr);
// Check hr

// create the new stream and its corresponding file on the file system
// NOTE: Specify the file format when creating the file
hr = cpSpStream->BindToFile(WAVE_FILENAME, SPFM_CREATE_ALWAYS);
// Check hr

// write some data to the stream
hr = cpSpStream->Write(WAVE_DATA_CHUNK, sizeof(WAVE_DATA_CHUNK));
// Check hr
```

The following code snippet illustrates the use of `ISpStream::BindToFile` for creating a read-only wave file

```
HRESULT hr = S_OK;

// create the stream object
hr = cpSpStream.CoCreateInstance(CLSID_SpStream);
// Check hr

// create a new stream, by opening a wave file from the file system
// NOTE: Since an existing file is being read, SAPI will not create the file
hr = cpSpStream->BindToFile(WAVE_FILENAME, SPFM_OPEN_EXISTING);
// Check hr

// read some data from the stream
hr = cpSpStream->Read(&bData, sizeof(WAVE_DATA_CHUNK), &cbData);
// Check hr
```



ISpStream::Close

ISpStream::Close closes the audio stream and validates the close operation.

```
HRESULT Close ( void );
```

Parameters

None.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_STREAM_CLOSED	The stream is closed or unavailable.
FAILED (hr)	Appropriate error message.

Remarks

Releasing the ISpStream object will automatically call this method. However, any errors encountered will not be sent as notifications to the application. Hence the application can explicitly call this method first to check that no errors occurred during the close operation.



ISpStreamFormat

ISpStreamFormat inherits from IStream.

This is the minimum extra interface required by SAPI in addition to the IStream interface. Using this interface, SAPI can query the stream to determine the format of the stream data. Almost all SAPI functions requiring or returning a stream will require or return an ISpStreamFormat.

When to Implement

This interface should be implemented when implementing a stream from scratch. If an IStream already exists, SAPI can use it with the provided [ISpStream](#) object. In normal usage, this interface does not need to be implemented.

Implemented By

- [SpMMAudioIn](#)
- [SpMMAudioOut](#)
- [SpStreamFormatConverter](#)
- [SpStream](#)

Methods in Vtable Order

ISpStreamFormat Methods	Description
GetFormat	Passes back the cached format of the stream.

Development Helpers

Helper Enumerations, Functions and Classes	Description
--	-------------

SPSTREAMFORMAT

SAPI supported stream formats

CSpStreamFormat

Class for managing SAPI supported stream formats and WAVEFORMATEX structures



ISpStreamFormat::GetFormat

ISpStreamFormat::GetFormat passes back the cached format of the stream.

SAPI uses this data to determine how to handle the stream data present in the underlying IStream.

```
HRESULT GetFormat(  
    GUID *pguidFormatId,  
    WAVEFORMATEX **ppCoMemWaveFormatEx  
);
```

Parameters

pguidFormatId

Address of a pointer to GUID data object that receives the format of the stream being used. This is typically either SPDFID_Text or SPDFID_WaveFormatEx.

ppCoMemWaveFormatEx

Address of a pointer to a [WAVEFORMATEX](#) data structure that receives the wave file format information. This is only applicable when the return GUID is SPDFID_WaveFormatEx. SAPI allocates the memory for the [WAVEFORMATEX](#) data structure using CoTaskMemAlloc, but it is the caller's responsibility to call CoTaskMemFree on the returned [WAVEFORMATEX](#) pointer.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Pointer is locating a memory block that is NULL or either too small or is

	not writable.
SPERR_UNINITIALIZED	The object has not been properly initialized.
SPERR_STREAM_CLOSED	The stream is closed or unavailable.



ISpStreamFormatConverter

ISpStreamFormatConverter is the primary interface implemented by the SAPI audio data format converter. SAPI uses the format converter to compensate for differences between supported SR and TTS engine formats, and the I/O formats requested by the application. Typically applications and engines do not use this object directly. The format converter is a wrapper object that encapsulates the specified base stream. It performs conversion on the fly during read/write operations. The Windows ACM (Audio Compression Manager) layer performs the conversion.

Several methods are included in addition to the [ISpStreamFormat](#) interface to allow data conversion.

Implemented By

- [SpStreamFormatConverter](#)

Remarks

SAPI utilizes the host system's installed audio codecs to perform the conversion. SAPI currently supports 1-stage and 2-stage stream conversions, but does not support 3-or-more-stage conversions.

An example of a 1-stage stream format conversion is the conversion of a PCM format to another PCM format (e.g., 8kHz 16-bit Stereo PCM [[SPSF_8kHz16BitStereo](#)] -> 44kHz 8-bit Mono [[SPSF_44kHz8BitMono](#)]). This requires only one codec (e.g., "Microsoft PCM Converter").

An example of a 2-stage stream conversion is the conversion of a compressed format to a PCM format (e.g., TrueSpeech 8kHz 1-Bit Mono [[SPSF_TrueSpeech_8kHz1BitMono](#)] -> 8kHz 8-bit Mono PCM [[SPSF_8kHz8BitMono](#)] -> 44kHz 16-bit Stereo [[SPSF_44kHz16BitStereo](#)]). This requires two codecs (e.g., "DSP

Group TrueSpeech(TM) Audio" and "Microsoft PCM Converter"). Note that one of the formats must be a PCM format.

An example of an unsupported 3-stage stream conversion is the conversion of a compressed format to another compressed format (e.g., TrueSpeech 8kHz 1-Bit Mono [[SPSF_TrueSpeech_8kHz1BitMono](#)] -> 8kHz 8-bit Mono PCM [[SPSF_8kHz8BitMono](#)] -> 8kHz 8-bit Stereo PCM [[SPSF_8kHz8BitStereo](#)] -> ALaw 8kHz Stereo [[SPSF_CCITT_ALaw_8kHzStereo](#)]). This would require three codecs (e.g., "DSP Group TrueSpeech(TM) Audio", "Microsoft PCM Converter", and "Microsoft CCITT G.771 Audio"). Note that SAPI is capable of converting between two compressed non-PCM formats if a single codec can do the entire conversion.

Methods in Vtable Order

ISpStreamFormatConverter Methods	Description
ISpStreamFormat interface	Inherits from ISpStreamFormat and all those methods are accessible from an ISpStreamFormatConverter object.
SetBaseStream	Sets audio stream to be wrapped by the format converter.
GetBaseStream	Gets the base audio stream that is being wrapped.
SetFormat	Sets the conversion (output) format.
ResetSeekPosition	Resets the format converter's stream seek position to the start of the stream.
ScaleConvertedToBaseOffset	Maps an offset in the converted stream into an

	offset in the base stream.
ScaleBaseToConvertedOffset	Maps an offset in the base stream into an offset in the converted stream.

Development Helpers

Helper Enumerations, Functions and Classes	Description
SPSTREAMFORMAT	SAPI supported stream formats
CSpStreamFormat	Class for managing SAPI supported stream formats and WAVEFORMATEX structures



ISpStreamFormatConverter::SetBaseStre

ISpStreamFormatConverter::SetBaseStream sets an audio stream to be wrapped by the format converter. The format converter is a stream object that encapsulates the base stream and performs format conversion on the fly during read/write operations.

```
HRESULT SetBaseStream(  
    ISpStreamFormat *pStream,  
    BOOL fSetFormatToBaseStreamFormat,  
    BOOL fWriteToBaseStream  
);
```

Parameters

pStream

[in] The stream to be wrapped. If NULL, the current base stream is released and any associated resources are released.

fSetFormatToBaseStreamFormat

[in] Flag specifies that the converter's stream format will be set to the same format as the base stream (set up as a pass through).

If *pStream* == NULL and this is set to TRUE, the format converter's stream format is reset to be undefined.

fWriteToBaseStream

[in] If TRUE, data will be written to the base stream. If FALSE, data will be read from the base stream. The format converter can only be in one I/O mode or the other at a time.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pStream</i> was invalid.



ISpStreamFormatConverter::GetBaseStre

ISpStreamFormatConverter::GetBaseStream gets the base audio stream that is being wrapped.

```
HRESULT GetBaseStream(  
    ISpStreamFormat **ppStream  
);
```

Parameters

ppStream

[out] This parameter can be NULL to simply test if there is an associated base stream.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	No base stream is present.
E_POINTER	Pointer is bad or invalid.



ISpStreamFormatConverter::SetFormat

ISpStreamFormatConverter::SetFormat sets the output format of the converter.

The [ISpStreamFormat::GetFormat](#) method returns the format of the output (converted) stream.

```
HRESULT SetFormat(  
    REFGUID                rguidFormatIdOfConvertedStream,  
    const WAVEFORMATEX    *pWaveFormatExOfConvertedStream  
);
```

Parameters

rguidFormatIdOfConvertedStream

[in] Address of the data format identifier associated with the requested output stream. Can be GUID_NULL or SPDFID_WaveFormatEx.

pWaveFormatExOfConvertedStream

[in] Address of the [WAVEFORMATEX](#) structure containing the wave file format information of the converted stream. Must be NULL with GUID_NULL. Must be a valid [WAVEFORMATEX](#) with SPDFID_WaveFormatEx.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One of the following was encountered: <i>rguidFormatIdOfConvertedStream</i> is neither GUID_NULL or SPDFID_WaveFormatEx;

pWaveFormatExOfConvertedStream
is not valid for the supplied REFGUID.

Remarks

The helper class [CSpStreamFormat](#) and the [SPSTREAMFORMAT](#) enumeration can be used to avoid the possibility of typos or mistakes when filling in the [WAVEFORMATEX](#) structure.



ISpStreamFormatConverter::ResetSeekPo

ISpStreamFormatConverter::ResetSeekPosition resets the format converter's stream seek position to the start of the stream. This method changes the seek position of the base stream to zero.

```
HRESULT ResetSeekPosition( void );
```

Parameters

None.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_UNINITIALIZED	Current stream base is uninitialized.



ISpStreamFormatConverter::ScaleConver

ISpStreamFormatConverter::ScaleConvertedToBaseOffset

maps a stream offset from the converted stream to the equivalent offset in the base stream.

```
HRESULT ScaleConvertedToBaseOffset(  
    ULONGLONG    ullOffsetConvertedStream,  
    ULONGLONG    *pullOffsetBaseStream  
);
```

Parameters

ullOffsetConvertedStream

The offset of the output (converted) stream.

pullOffsetBaseStream

The equivalent offset in the base (unconverted) stream.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pullOffsetBaseStream</i> is invalid.
SPERR_UNINITIALIZED	The base stream has not been initialized.

Remarks

When performing a mapping with a compressed format, it is possible to introduce small rounding errors, since the content of the audio is not used to perform the conversion.



ISpStreamFormatConverter::ScaleBaseTo

ISpStreamFormatConverter::ScaleBaseToConvertedOffset

converts an offset in the base stream into the equivalent offset in the converted stream. This method is primarily used internally to map event offsets.

```
HRESULT ScaleBaseToConvertedOffset(  
    ULONGLONG    ullOffsetBaseStream,  
    ULONGLONG    *pullOffsetConvertedStream  
);
```

Parameters

ullOffsetBaseStream

The current offset in the base (unconverted) stream.

pullOffsetConvertedStream

The new offset in the output (converted) stream.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pullOffsetConvertedStream</i> is bad or invalid.
SPERR_UNINITIALIZED	The base stream is not initialized.
E_INVALIDARG	<i>ullOffsetBaseStream</i> is less than the initial seek position of the current stream. <i>*pullOffsetConvertedStream</i> is set to 0xFFFFFFFFFFFFFFFF.

Remarks

When performing a mapping with a compressed format, it is possible to introduce small rounding errors, since the content of the audio is not used to perform the conversion.



ISpTranscript

A transcript is a text string associated with a piece of audio data. The SAPI SpStream object supports the *ISpTranscript* interface for wav audio files.

Associated Class IDs

The following class IDs (CLSID) may be used with this interface. A complete CLSID listing for all interfaces is in the [Class IDs](#) section.

CLSID_SpStream

Methods in Vtable Order

ISpTranscript Methods	Description
GetTranscript	Gets the current transcript.
AppendTranscript	Adds the current text to the transcript.

Development Helpers

Helper Enumerations, Functions and Classes	Description
CSpDynamicString	Class for managing dynamically sized Unicode strings



ISpTranscript::GetTranscript

ISpTranscript::GetTranscript gets the current transcript. The string returned will be allocated by CoTaskMemAlloc and applications implementing this method must call CoTaskMemFree() to free memory associated with this string.

```
HRESULT GetTranscript(  
    WCHAR    **ppszTranscript  
);
```

Parameters

ppszTranscript

[out, string] A pointer to the null-terminated transcription string.

Return values

Value	Description
S_OK	Function completed successfully. <i>ppszTranscript</i> contains a CoTaskMemAllocated string.
E_OUTOFMEMORY	Exceeded available memory.
SPERR_UNINITIALIZED	Object has not been initialized.
E_POINTER	<i>ppszTranscript</i> is bad or invalid.
S_FALSE	No transcript is present and <i>ppszTranscript</i> will be NULL.
FAILED (hr)	Appropriate error message.

Example

The following code snippet illustrates the use of `ISpTranscript::GetTranscript`.

```
HRESULT hr = S_OK;

// Bind a stream to an existing wavefile
hr = SPBindToFile( FILENAME, SPFM_READ_ONLY, &cpStream);
// Check hr

hr = cpStream.QueryInterface(&cpTranscript);
// Check hr

PWCHAR pwszTranscript;
hr = cpTranscript->GetTranscript(&pwszTranscript);
// Check hr
```



ISpTranscript::AppendTranscript

ISpTranscript::AppendTranscript adds the current text to the transcript.

```
HRESULT AppendTranscript(  
    const WCHAR    *pszTranscript  
);
```

Parameters

pszTranscript

[in, string] The text of the transcript. If *pszTranscript* is NULL, the current transcript is deleted. Otherwise, the text is appended to the current transcript.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszTranscript</i> is bad or invalid.
E_OUTOFMEMORY	Exceeded available memory.
FAILED (hr)	Appropriate error message.

Example

The following code snippet illustrates the use of `ISpTranscript::AppendTranscript`.

```
HRESULT hr = S_OK;

// Bind a stream to an existing wavefile
hr = SPBindToFile( FILENAME, SPFM_CREATE_ALWAYS, &cpStream);
// Check hr

hr = cpStream.QueryInterface(&cpTranscript);
// Check hr

hr = cpTranscript->AppendTranscript(L"this is a test");
// Check hr
```



Eventing interfaces

This section provides SAPI 5 event information.

- [ISpNotifySource](#)
- [ISpNotifySink](#)
- [ISpNotifyTranslator](#)
- [ISpEventSink](#)
- [ISpEventSource](#)
- [ISpNotifyCallback](#)



ISpNotifySource

In both speech synthesis and speech recognition, applications receive notifications when words have been spoken or when phrases have been recognized. SAPI components that generate notifications implement an ISpNotifySource.

The ISpNotifySource and [ISpNotifySink](#) interfaces alone only provide a mechanism for a notification but no information on the events that caused the notification. With an ISpEventSource object, an application can retrieve information about the events that caused the notification.

Applications will not typically use the free-threaded [ISpNotifySink](#) mechanism for receiving SAPI event notifications. They will use one of the simplified methods of either a window message, callback or Win32 event.

Note that both variations of callbacks as well as the window message notification require a window message pump to run on the thread that initialized the notification source. Callback will only be called as the result of window message processing, and will always be called on the same thread that initialized the notify source. However, using Win32 events for SAPI event notification does not require a window message pump.

Implemented By

- [SpRecoContext](#)
- [SpSharedRecoContext](#)
- [SpVoice](#)
- [SpMMAudioIn](#)
- [SpMMAudioOut](#)
- [SpRecPlayAudio](#)
- [SpStreamFormatConverter](#)

Methods in Vtable Order

ISpNotifySource Methods	Description
<u>SetNotifySink</u>	Sets up the instance to make free-threaded calls through ISpNotifySink::Notify. This method can also be used to unregister an existing notification.
<u>SetNotifyWindowMessage</u>	Sets a window handle to receive notifications as window messages.
<u>SetNotifyCallbackFunction</u>	Sets a callback function to receive notifications.
<u>SetNotifyCallbackInterface</u>	Sets an object derived from <u>ISpTask</u> to receive notifications.
<u>SetNotifyWin32Event</u>	Sets up a Win32 event object to be used by this instance for notifications.
<u>WaitForNotifyEvent</u>	A blocking call which waits for a notification.
<u>GetNotifyEventHandle</u>	Retrieves Win32 event handle associated with this notify source.



ISpNotifySource::SetNotifySink

ISpNotifySource::SetNotifySink sets up the instance to make free-threaded notification calls through [ISpNotifySink::Notify](#).

```
HRESULT SetNotifySink(  
    ISpNotifySink *pNotifySink  
);
```

Parameters

pNotifySink

[in] Pointer to the notification interface. If *pNotifySink* is NULL, any current notification mechanism (notify sink, window message, callback, or Win32 event) is removed.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Interface pointer is invalid.
FAILED (hr)	Appropriate error message.

Remarks

If *pNotifySink* is NULL, any notification mechanism currently associated with this notify source is removed.

Because free-threaded notifications can occur on any thread, at any point during execution, they are extremely prone to deadlocks and re-entrance problems. See the documentation for [ISpNotifySink](#) for more details. Most applications will find one of the other notification mechanisms much easier to use.



ISpNotifySource::SetNotifyWindowMessage

ISpNotifySource::SetNotifyWindowMessage sets up the instance to send window messages to a specified window.

```
HRESULT SetNotifyWindowMessage(  
    HWND      hWnd,  
    UINT      Msg,  
    WPARAM    wParam,  
    LPARAM    lParam  
);
```

Parameters

hWnd

[in] Handle to the window whose message handler function will receive SAPI notifications.

Msg

[in] Message number which will be passed into the message handler function of the window *hWnd*.

wParam

[in] *wParam* that will be passed into the message handler function of the window *hWnd*.

lParam

[in] *lParam* that will be passed into the message handler function of the window *hWnd*.

Return values

Value	Description
S_OK	Function completed successfully.

E_INVALIDARG	<i>hWnd</i> is an invalid window handle.
FAILED (hr)	Appropriate error message.



ISpNotifySource::SetNotifyCallbackFunction

ISpNotifySource::SetNotifyCallbackFunction sets up this instance to send notifications using a standard C-style callback function.

```
HRESULT SetNotifyCallbackFunction(  
    SPNOTIFYCALLBACK    *pfnCallback,  
    WPARAM               wParam,  
    LPARAM               lParam  
);
```

Parameters

pfnCallback

[in] The notification callback function to be used.

wParam

[in] Constant WPARAM value that will be passed to the *pfnCallback* function when it is called.

lParam

[in] Constant LPARAM value that will be passed to the *pfnCallback* function when it is called.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Callback function is invalid.
FAILED (hr)	Appropriate error message.

Remarks

It is the responsibility of the client code to control the lifetime of a notification. To remove an installed notify callback, call `ISpEventSource::SetNotifySink (NULL)`. The final release of an object that supports `ISpEventSource` will automatically remove an installed notify callback.

The SAPI implementation uses a hidden window to call back the client on the same thread that was used to initialize the event source. Notification callbacks are the result of processing a window message. When this notification mechanism is used:

1. The `SPNOTIFYCALLBACK` method will always be called on the thread that initialized the event source or notify translator object.
2. The thread must have a window message pump.

The `SPNOTIFYCALLBACK` function is declared as follows:

```
typedef void __stdcall SPNOTIFYCALLBACK(WPARAM wParam,  
LPARAM lParam);
```



ISpNotifySource::SetNotifyCallbackInterface

ISpNotifySource::SetNotifyCallbackInterface sets up this instance to call the virtual method [ISpNotifyCallback::NotifyCallback](#) for notifications.

```
HRESULT SetNotifyCallbackInterface(  
    ISpNotifyCallback *pSpCallback,  
    WPARAM          wParam,  
    LPARAM          lParam  
);
```

Parameters

pSpCallback

[in] A pointer to an application-defined implementation of the [ISpNotifyCallback](#) interface.

wParam

[in] Constant WPARAM value that will be passed to the [ISpNotifyCallback::NotifyCallback](#) method when it is called.

lParam

[in] Constant LPARAM value that will be passed to the [ISpNotifyCallback::NotifyCallback](#) method when it is called.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pSpCallback</i> is invalid.
FAILED (hr)	Appropriate error message.

Remarks

The application will be called back on the same thread that calls this method. The callback will be called as a result of window message processing, so the thread must have a message pump. For more details, see the documentation for [ISpNotifyCallback](#).



ISpNotifySource::SetNotifyWin32Event

ISpNotifySource::SetNotifyWin32Event sets up a Win32 event object to be used by this instance for notifications. The event handle can be retrieved through [GetNotifyEventHandle](#).

```
HRESULT SetNotifyWin32Event ( void );
```

Parameters

None

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Remarks

For an explanation of Win32 event objects, see the Win32 Platform SDK documentation. Once an event object has been initialized for this instance, use either [WaitForNotifyEvent](#) or use [GetNotifyEventHandle](#) and wait using one of the various Win32 synchronization functions. Note that Win32 event objects and SAPI events are different objects.



ISpNotifySource::WaitForNotifyEvent

ISpNotifySource::WaitForNotifyEvent is a blocking call that waits on a Win32 event handle for a SAPI notification.

```
HRESULT WaitForNotifyEvent(  
    DWORD    dwMilliseconds  
);
```

Parameters

dwMilliseconds

[in] Number of milliseconds for the timeout on a blocking call. If set to INFINITE, there is no timeout.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	The operation timed-out.
SPERR_ALREADY_INITIALIZED	This event source has been initialized to use a notification mechanism other than a Win32 event. It was unable to re-initialize the notification.
FAILED (hr)	Appropriate error message.

Remarks

A blocking call returns when a SAPI notification has fired, a timeout has passed, or the initialized Win32 event object has signaled. Calling this method will automatically initialize the event source to use an event handle if no other notification mechanism has been initialized.



ISpNotifySource::GetNotifyEventHandle

ISpNotifySource::GetNotifyEventHandle retrieves the Win32 event object handle. This event can be used in any of the Win32 WaitForxxx methods.

```
HANDLE GetNotifyEventHandle ( void );
```

Parameters

None

Return values

Value	Description
Win32 event handle	Initialized by SetNotifyWin32Event on this instance.
INVALID_HANDLE_VALUE	Interface not initialized.

Remarks

Do not close the returned handle, as it is owned by the event source object. Calling this method will automatically initialize the event source to use an event handle if no other notification mechanism has been initialized.



ISpNotifySink

In both speech synthesis and speech recognition, applications receive notifications when words have been spoken or when phrases have been recognized. SAPI components that generate notifications implement an ISpNotifySource.

The [ISpNotifySource](#) and ISpNotifySink interfaces alone only provide a mechanism for a notification but no information on the events that caused the notification. With an ISpEventSource object, an application can retrieve information about the events that caused the notification. An ISpEventSource also provides the mechanism to filter and queue events. By default, an application (really an ISpNotifySink) receives no notifications from ISpEventSource until SetInterests has been called to specify on which events to notify or queue.

When an application is notified of an event that is not queued, an application will take measures based on which event sink is receiving the notification. From context an application might know exactly what it needs to do, or it may need to interact with the components which sent the notifications. If an application is notified of an event that is queued, the application will call ISpEventSource::GetEvents to retrieve the actual events that caused a notification.

When to Implement

Implement this interface only if the application can take advantage of the slightly reduced latency of a free-threaded notification. Most applications should use one of the simplified notification mechanisms in [ISpEventSource](#) instead of implementing this interface directly. Free-threaded notifications are difficult to implement without deadlocking the system. Furthermore, other notification mechanisms require less code on the part of the developer. The application needs to handle free-threaded notifications and implement the ISpNotifySink interface when an ISpNotifySink object is to be notified.

Because free-threaded notifications can occur on any thread, at any point during execution, they are extremely prone to deadlocks and re-entrance problems. The only interface that can be called on an event source object is [ISpEventSource::GetEvents](#). If the ISpNotifySink interface is implemented directly, the code should only use some mechanism to signal another thread to process the notification (for example, a Win32 event or an I/O completion port), and, if needed, call [ISpEventSource::GetEvents](#). Do not call any other methods.

Methods in Vtable Order

ISpNotifySink Methods	Description
Notify	Called by an ISpNotifySource object to notify the sink when the state of the notify source has changed.



ISpNotifySink::Notify

ISpNotifySink::Notify is called by an [ISpNotifySource](#) object to notify the sink when the state of the notify source has changed.

```
HRESULT Notify ( void );
```

Parameters

None

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Object failed notification.

Remarks

The only interface that can be called on an event source object is [ISpEventSource::GetEvents](#). The Notify() method should only signal another thread to process the notification (for example, a Win32 event or an I/O completion port), and if needed, call [ISpEventSource::GetEvents](#). Do not call any other methods.



ISpNotifyTranslator

Once the SpNotifyTranslator object has been initialized, the [ISpNotifySource::SetNotifySink](#) method can be called, passing the translator object interface as the parameter. The translator will then convert the call to ISpNotifySink::Notify into the appropriate notification.

Because [ISpNotifySource](#) supports most of the functionality of this interface, application writers will not normally use either this interface or the [SpNotifyTranslator](#) object. In fact, SAPI uses this object to implement the various methods of [ISpNotifySource](#). The method [InitWin32Event](#) supports the ability to initialize the translator with a specific event object, and so could be used for that purpose.

Implemented By

- [SpNotifyTranslator](#)

Methods in Vtable Order

ISpNotifyTranslator Methods	Description
ISpNotifySink interface	Inherits from ISpNotifySink and those methods are accessible from an ISpNotifyTranslator object.
InitWindowMessage	Sets up the instance to send window messages to a specified window.
InitCallback	Sets up this instance to send notifications using a standard C-style callback function.
InitSpNotifyCallback	Enables an object derived from ISpNotifyCallback to receive notifications.
InitWin32Event	Sets up a Win32 event object to be

	used by this instance.
<u>Wait</u>	A blocking call that returns when a SAPI notification has fired and the associated Win32 event object has been signaled or a timeout has passed.
<u>GetEventHandle</u>	Returns the Win32 event handle associated with the translator.



ISpNotifyTranslator::InitWindowMessage

ISpNotifyTranslator::InitWindowMessage sets up the instance to send window messages to a specified window.

```
HRESULT InitWindowMessage(  
    HWND      hWnd,  
    UINT      Msg,  
    WPARAM    wParam,  
    LPARAM    lParam  
);
```

Parameters

hWnd

[in] Handle to the window whose message handler function will receive SAPI notifications.

Msg

[in] Message number which will be passed into the message handler function of the window *hWnd*.

wParam

[in] *wParam* that will be passed into the message handler function of the window *hWnd*.

lParam

[in] *lParam* that will be passed into the message handler function of the window *hWnd*

Return values

Value	Description

S_OK	Function completed successfully.
SPERR_ALREADY_INITIALIZED	SpTranslator object already initialized.
E_INVALIDARG	<i>hWnd</i> is invalid or bad.
FAILED(hr)	Appropriate error message.



ISpNotifyTranslator::InitCallback

ISpNotifyTranslator::InitCallback sets up this instance to send notifications using a standard C-style callback function.

```
HRESULT InitCallback(  
    SPNOTIFYCALLBACK    *pfnCallback,  
    WPARAM               wParam,  
    LPARAM               lParam  
);
```

Parameters

pfnCallback

[in] The notification callback function to be used.

wParam

[in] Constant WPARAM value that will be passed to the *pfnCallback* function when it is called.

lParam

[in] Constant LPARAM value that will be passed to the *pfnCallback* function when it is called.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_ALREADY_INITIALIZED	SpTranslator object is already initialized.
E_INVALIDARG	<i>pfnCallback</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Remarks

The translator implementation uses a hidden window to call back the client on the same thread that was used to initialize the event source. Notify callbacks are the result of processing a window message. When this notification mechanism is used:

1. The SPNOTIFYCALLBACK method will always be called on the thread that initialized the event source or notify translator object.
2. The thread must have a window message pump.

The SPNOTIFYCALLBACK function is declared as follows:

```
typedef void __stdcall SPNOTIFYCALLBACK(WPARAM wParam,  
LPARAM lParam);
```



ISpNotifyTranslator::InitSpNotifyCallback

ISpNotifyTranslator::InitSpNotifyCallback sets up this instance to call the virtual method [ISpNotifyCallback::NotifyCallback](#) for notifications.

```
HRESULT InitSpNotifyCallback(  
    ISpNotifyCallback *pSpCallback,  
    WPARAM          wParam,  
    LPARAM          lParam  
);
```

Parameters

pSpCallback

[in] A pointer to an application-defined implementation of the [ISpNotifyCallback](#) interface.

wParam

[in] Constant WPARAM value that will be passed to the [ISpNotifyCallback::NotifyCallback](#) method when it is called.

lParam

[in] Constant LPARAM value that will be passed to the [ISpNotifyCallback::NotifyCallback](#) method when it is called.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_ALREADY_INITIALIZED	Translator object is already initialized.
E_INVALIDARG	<i>pSpNotifyCallback</i> is invalid or bad.

FAILED(hr)

Appropriate error message.

Remarks

The application will be called back on the same thread that calls this method. The callback will be called as a result of window message processing, so the thread must have a message pump. For more details, see the documentation for [ISpNotifyCallback](#).



ISpNotifyTranslator::InitWin32Event

ISpNotifyTranslator::InitWin32Event sets up a Win32 event object to be used by this instance.

```
HRESULT InitWin32Event(  
    HANDLE    hEvent,  
    BOOL      fCloseHandleOnRelease  
);
```

Parameters

hEvent

Handle of an existing Win32 event object for the application to use with ISpNotifyTranslator. If this parameter is NULL, a new event will be created.

fCloseHandleOnRelease

Specifies whether the *hEvent* handle should be closed when the object is released. If *hEvent* is NULL, this parameter is ignored and the handle will always be closed upon release of the object.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_ALREADY_INITIALIZED	Interface is already initialized.
FAILED(hr)	Appropriate error message.

Remarks

For an explanation of Win32 event objects, see the Win32 Platform SDK documentation. The translator will call the Win32

method `::SetEvent()` whenever the translator's `Notify()` method is called.

Initialize an event object for this instance, and then use either the `WaitForNotifyEvent` or `GetNotifyEventHandle` method. Win32 event objects and SAPI events are different. This method is similar to [ISpNotifySource::SetNotifyWin32Event](#).



ISpNotifyTranslator::Wait

ISpNotifyTranslator::Wait is a blocking call that returns when a SAPI notification has fired and the associated Win32 event object has been signaled or a timeout has passed. This method is applicable only with objects using Win32 events.

```
HRESULT Wait(  
    DWORD    dwMilliseconds  
);
```

Parameters

dwMilliseconds

[in] Number of milliseconds for the timeout on a blocking call. If set to INFINITE, there is no timeout.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	The event was not set and the call was timed out.
SPERR_UNINITIALIZED	InitWin32Event did not return successfully or has not been called.



ISpNotifyTranslator::GetEventHandle

ISpNotifyTranslator::GetEventHandle returns the Win32 event handle associated with the translator.

Returns the Win32 event object handle initialized by `InitWin32Event` on this `ISpNotifyTranslator` instance. This method is applicable only with objects using Win32 events.

The handle is not a duplicated handle and should not be closed by the caller.

```
HANDLE GetEventHandle ( void );
```

Parameters

None

Return values

Value	Description
<i>handle</i>	The handle to the event
INVALID_HANDLE_VALUE	Translator has not been properly initialized by calling <code>InitWin32Event()</code> .



ISpEventSink

This interface allows event sources to send events directly to an event sink through a free-threaded call.

Associated Class IDs

The following class IDs (CLSID) may be used with this interface. A complete CLSID listing for all interfaces is in the [Class IDs](#) section.

CLSID_SpStreamFormatConverter

CLSID_SpStream

Methods in Vtable Order

ISpEventSink Methods	Description
AddEvents	Adds events directly to an event sink.
GetEventInterest	Passes back the event interest for the voice.



ISpEventSink::AddEvents

ISpEventSink::AddEvents adds events directly to an event sink.

```
HRESULT AddEvents(  
    const SPEVENT *pEventArray,  
    ULONG ulCount  
);
```

Parameters

pEventArray

Pointer to an array of [SPEVENT](#) event structures.

ulCount

Number of event structures being passed in.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pEventArray</i> is bad or invalid
FAILED(hr)	Appropriate error message.



ISpEventSink::GetEventInterest

ISpEventSink::GetEventInterest passes back the event interest for the voice.

```
HRESULT GetEventInterest(  
    ULONGLONG    *pullEventInterest  
);
```

Parameters

pullEventInterest

[out] Set of flags of type [SPEVENTENUM](#) defining the event interest.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Pointer bad or invalid.
FAILED(hr)	Appropriate error message.



ISpEventSource

The ISpEventSource inherits from the [ISpNotifySource](#) interface. Using the methods on [ISpNotifySource](#) an application can specify the mechanism by which they receive notifications. Applications can configure which events should trigger notifications and which events retrieve queued events.

An ISpEventSource provides the mechanism to filter and queue events. By default, an application (really an [ISpNotifySink](#)) receives no notifications from the SpVoice object, until SetInterest has been called to specify on which events to notify or queue. For the SpRecoContext object, the default event interest is set to queue only recognition events.

When an application is notified of an event that is not queued, it will proceed based on which event sink receives the notification. From context, an application might know exactly what it needs to do, or it may need to interact with the components that sent the notifications. If an application is notified of a queued event, the application will call ISpEventSource::GetEvents to retrieve the actual events that caused a notification.

Implemented By

- [SpRecoContext](#)
- [SpSharedRecoContext](#)
- [SpVoice](#)
- [SpMMAudioIn](#)
- [SpMMAudioOut](#)
- [SpRecPlayAudio](#)
- [SpStreamFormatConverter](#)

Methods in Vtable Order

ISpEventSource Methods	Description
<u>ISpNotifySource</u> inherited methods	All methods of ISpNotifySource are accessible from this interface
<u>SetInterest</u>	Sets the type of events the client is interested in.
<u>GetEvents</u>	Retrieves and removes the queued events.
<u>GetInfo</u>	Retrieves information about the event queue.



ISpEventSource::SetInterest

ISpEventSource::SetInterest sets the type of events the client is interested in.

Sets the type of events which will invoke a notification and become queued.

```
HRESULT SetInterest(  
    ULONGLONG    ullEventInterest,  
    ULONGLONG    ullQueuedInterest  
);
```

Parameters

ullEventInterest

[in] Event ID flags indicating which events should invoke a notification to the event sink that this event source uses. Must be of type [SPEVENTENUM](#).

ullQueuedInterest

[in] Event ID flags indicating which events should be queued. The event flags set here must also be set in *dwEventInterest*. Must be of type [SPEVENTENUM](#).

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Invalid flags passed in one or more fields.
FAILED(hr)	Appropriate error message.

Remarks

If `SetInterest` is never called, the SR engine defaults to `SPEI_RECOGNITION` as the only event and queued interest. A TTS engine defaults to zero for both event and queued interest. With either engine, no events will be passed through if both parameters are set to zero.

Note that the [SPFEI\(\)](#) macro will be used to convert an event enumeration into the appropriate flags to pass to this method. For example, to receive the `SPEI_RECOGNITION` and `SPEI_HYPOTHESIS` events, call this function as follows:

```
ULONGLONG ullMyEvents = SPFEI(SPEI_RECOGNITION) | SPFEI(SPEI_HYPOTHESIS);  
hr = pEventSource->SetInterest(ullMyEvents, ullMyEvents)
```

Events specified in *ullEventInterest* must be a superset of those specified in the *ullQueuedInterest*. Therefore it is possible to have notifications of events but not actually queue them. This can be useful for polling the `GetStatus` method, especially for TTS.



ISpEventSource::GetEvents

ISpEventSource::GetEvents retrieves and removes the queued events. Clients will want to use the helper class [CSpEvent](#) to retrieve and manipulate these events.

```
HRESULT GetEvents(  
    ULONG      ulCount,  
    SPEVENT   *pEventArray,  
    ULONG      *pulFetched  
);
```

Parameters

ulCount

[in] Maximum number of events that SPEVENT structures can return.

pEventArray

[out] Pointer to array of SPEVENT structures. Each returned event is written to one of these SPEVENT structures.

pulFetched

[out] Pointer to the number of events returned. If *ulCount* is one, this parameter is not required.

The events are then removed from the queue. The events not returned are left for a future call to GetEvents. It is possible that by the time an application calls GetEvents, another thread has processed the events and there are no events to be returned. This may be the result of subsequent Notify calls.

Return values

Value	Description
S_OK	Function completed successfully and all requested events were returned.
S_FALSE	Success, but less than the requested amount of events were returned.
E_POINTER	<i>pEventArray</i> is invalid.
FAILED(hr)	Appropriate error message.



ISpEventSource::GetInfo

ISpEventSource::GetInfo retrieves information about the event queue.

```
HRESULT GetInfo(  
    SPEVENTSOURCEINFO *pInfo  
);
```

Parameters

pInfo

[out] Pointer to an [SPEVENTSOURCEINFO](#) structure about the event.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pInfo</i> is invalid.
FAILED(hr)	Appropriate error message.



ISpNotifyCallback

This is not a COM interface. This is a C++ virtual interface that can be implemented by a SAPI client application to receive notifications. Since it is not a COM interface, the application does not need to implement QueryInterface, AddRef, or Release. It is the responsibility of the client code to control the lifetime of an ISpNotifyCallback-style notification. To remove an installed notify callback, call ISpEventSource::SetNotifySink(NULL). The final release of an object that supports ISpEventSource will automatically remove an installed notify callback.

The SAPI implementation uses a hidden window to call the client back on the same thread that was used to initialize the event source. Notification callbacks are the result of processing a window message. This means that when the notification mechanism is used:

1. The NoitifyCallback method will always be called on the thread that initialized the event source or notify translator object.
2. The thread must have a window message pump.

Methods in Vtable Order

ISpNotifySource Methods	Description
NotifyCallback	Client implemented method is called by an object that supports ISpEventSource when an event occurs.



ISpNotifyCallback::NotifyCallback

ISpNotifyCallback::NotifyCallback is implemented by the client of the SpEventSource object. When this method is called, the *wParam* and *lParam* parameters are set to the values specified by the client when it called [ISpNotifySource::SetNotifyCallbackInterface](#). The client should examine the appropriate event source object for events.

```
HRESULT NotifyCallback(  
    WPARAM    wParam,  
    LPARAM    lParam  
);
```

Parameters

wParam

[in] *wParam* parameter is specified by the client when it called the ISpEventSource::SetNotifyCallback interface.

lParam

[in] *lParam* parameter is specified by the client when it called the ISpEventSource::SetNotifyCallback interface.

Return values

Clients should return S_OK. Other values may be used for debugging purposes, but will be ignored by SpEventSource objects.



Grammar Compiler Interfaces (API-level)

Many speech recognition applications are built on voice commands, or command and control (C and C). For example, users playing Solitaire using a graphical application, may want to add C and C. This enables the user to speak "new game" or "play the ace of spades" into their computer microphone instead of using menu options or keyboard accelerators. Microsoft Office XP has a C and C mode, which enables user to speak voice commands mapped to virtually every menu command and many parts of the user interface (e.g., "File New", "Tools Options", "Outlook Today", etc.).

Applications can use SAPI 5's C and C features to implement functionality similar to a voice-command enabled Solitaire, Office XP, and other innovative applications. The C and C features of SAPI 5 are implemented as context-free grammars (CFGs). A CFG is a structure that defines a specific set of words, and the combinations of these words that can be used. In basic terms, a CFG defines the sentences that are valid, and in SAPI 5, defines the sentences that are valid for recognition by a speech recognition (SR) engine.

The CFG format in SAPI 5 defines the structure of grammars and grammar rules using Extensible Markup Language (XML). The CFG/Grammar compiler transforms the XML tags defining the grammar elements into a binary format used by SAPI 5-compliant SR engines. This compiling process can be performed either before or during application run time.

The Speech SDK includes a grammar compiler, which can be used to author text grammars, compile text grammars into the SAPI 5 binary format, and perform basic testing before integration into an application. Also see the [SDK Sample: Grammar Compiler](#).

SAPI 5 also enables applications to create CFG structures programmatically using the [ISpGrammarBuilder](#) interface, which

is inherited by [ISpRecoGrammar](#). The application can use the ISpGrammarBuilder API to dynamically update an already loaded SAPI 5 XML grammar, create an in-memory SAPI 5 grammar, and/or save an in-memory SAPI 5 grammar to a memory stream (e.g., for saving grammars to the hard disk).

The following section covers:

- [Text Grammar Format](#): SAPI 5-defined XML grammar format for defining a CFG with plain text.
- [ISpGrammarBuilder](#): SAPI 5 API for programmatically creating, editing, or saving in-memory and binary CFGs.

Applications that do not need to modify a grammar at run time, or applications that want to increase performance of their CFG-based application should load the compiled binary form statically (not dynamically). If loading the backend grammar compiler at application run time, note that SAPI must allow for modification and validation of complicated state/transition graphs.



Text Grammar Format

The context-free grammar (CFG) format in SAPI 5 defines the structure of grammars and grammar rules using the Extensible Markup Language (XML) tagging language. The CFG compiler transforms the XML tags defining the grammar elements into a binary format used by speech engines. This compiling process can be performed either before or during application run time. Speech recognition engines use CFGs to constrain the user's words to words that it will recognize.

Text Grammar Format Section Topics:

Name	Description
Text Grammar Format Overview	Describes grammar terminology and basic structure.
Grammar Rules and State Graphs	Describes rule state graphs with examples.
Designing Grammar Rules	General and advanced guidelines for command & control grammars.
Grammar Format Tags	Describes the entire set of XML Grammar tags with examples.
Grammar Format Tags: Special Characters	Describes tags which use special characters.
SAPI Grammar Example: Solitaire	Sample XML grammar covering basic grammar structure.



Text Grammar Format Overview

The Extensible Markup Language (XML) format inside a **GRAMMAR** XML element (block), is an "expert-only-readable" declaration of a grammar that a speech application uses to accomplish the following:

- Improve recognition accuracy by restricting and indicating to an engine what words it should expect.
- Improve maintainability of textual grammars, by providing constructs for reusable text components (internal and external rule references), phrase lists, and string and numeric identifiers.
- Improve translation of recognized speech into application actions. This is made easier by providing "semantic tags," (property name, and value associations) to words/phrases declared inside the grammar.

A **GRAMMAR** XML element (block) appears in a XML source code file. The XML source is compiled into a binary grammar format and is the format used by SAPI during application run time.

The following section covers:

- [Extensible Markup Language \(XML\)](#)
- [Attributes](#)
- [Contents](#)
- [Comments](#)
- [How SAPI utilizes XML information](#)
- [Frequently used definitions](#)
- [Non-empty concatenated recognition contents](#)

Extensible Markup Language

The textual grammar format is an application of the XML. Every XML element consists of a start tag (<SOME_TAG>) and an end tag (</SOME_TAG>) with a case-insensitive tag name and contents between these tags. The start tag and the end tag are the same if the element is empty. For example, the tag (<SOME_TAG/>). For more information on the use of XML grammars, please see the [Grammar XML Schema](#) section. Additionally, more information about XML and the XML specification is available at: <http://www.w3.org/TR/REC-xml>.

For example, all grammars contain the opening tag <GRAMMAR> as follows:

```
<GRAMMAR>
```

```
... grammar content
```

```
</GRAMMAR>
```

Note that the contents of the grammar is contained between an opening tag and a trailing, closing tag.

[Back to top](#)

Attributes

Attributes of an XML element appear inside the start tag. Each attribute is in the form of a name followed by an equal sign followed by a string which must be surrounded by either single or double quotation marks. An attribute of a given name may only appear once in a start tag.

In summary, the literal string cannot contain either < or ', if the string is surrounded by single quotation marks. It may not contain ", if the string is surrounded by double quotation marks. Furthermore, use all ampersand (&) characters only in an entity reference such as & and >. When a literal string is parsed, the resulting replacement text will resolve all entity references such as > into its corresponding text, such as >. In this specification, only the resulting replacement text needs to be defined for attribute value strings. More information about XML and the XML specification is available at: <http://www.w3.org/TR/REC-xml>.

For example, the grammar author can specify the language (id) of the grammar as follows.

```
<GRAMMAR LANGID="409">  
... grammar content  
</GRAMMAR>
```

The grammar element (<GRAMMAR>) has an attribute, called *LANGID* which must be a numeric value. The grammar author specifies the language attribute by placing the attribute inside the brackets of the opening tag, and enclosing the attribute value (e.g. 409) in quotation marks.

[Back to top](#)

Contents

The contents of an element consists of text or subelements. Formal definitions of valid contents in this specification are provided as regular and "multi-set" expressions. The pseudo-element name "Text" indicates untagged text. With these definitions, the XML specification defines the exact file syntax details.

For example, the grammar author can place either text or subelements inside a phrase tag as follows.

```
<PHRASE>  
  hello  
</PHRASE>
```

```
<PHRASE>  
  <OPT>world</OPT>  
</PHRASE>
```

The grammar author should review the [SAPI 5 Grammar XML Schema](#) to determine the type of content support in each tag (e.g. text and sub-elements, only text, only sub-elements, etc.).

[Back to top](#)

Comments

The SAPI 5 XML parser treats HTML comment tags as unknown XML tag elements. The engine should provide support for comments and other unknown XML elements.

It is recommended that grammar authors place comments in their XML files (e.g. *mygrammar.xml*), similar to commenting source code, since the XML parser will safely parse the comments without affecting the grammar itself. Similarly, there is increase in size of the binary form of the grammar (e.g. *mygrammar.cfg*) since the SAPI 5 grammar compiler strips out the comments.

An example of a comment in an XML grammar is as follows.

```
<!-- the 'travel' rule is the main voice command for our :
<RULE ID="RID_Travel" TOPLEVEL="ACTIVE">
  <PHRASE>travel from</PHRASE>

  <!-- include location grammar component, so we can char
  <RULEREf REFID="RID_Location" PROPID="PID_FromDestinat.
  <PHRASE>to</PHRASE>

  <!-- include location grammar component, so we can char
  <RULEREf REFID="RID_Location" PROPID="PID_ToDestinatio
</RULE>
```

Note that the comment blocks always begin with <!-- and end with -->.

[Back to top](#)

How SAPI utilizes XML information

SAPI uses XML content in the following two methods.

1. The SAPI context-free grammar compiler, compiles the XML grammar into a binary grammar format. The compiled binary grammar is loaded into the SAPI run-time environment from a file, memory, or object (.DLL) resource.
2. The speech recognition (SR) engine queries the run-time environment for available grammar information.

[☐ Back to top](#)

Frequently used definitions

Untagged text declaring a sequence of words that the recognition engine will recognize. Tentatively this text is only the not-necessarily-phonetic representation of words used for reading words whose pronunciation is unknown to the user (for example, for Japanese, kana, not kanji); this form will be called the spelling form. In further definitions in this section, *Text* will be referenced as though it were a pseudo-element.

☐ [Back to top](#)

Non-empty concatenated recognition contents

The contents of a number of XML elements in this specification such as, the **P** element, contain a sequence of grammar constructs which are concatenated together (one grammar construct after another). These grammar elements must be recognized in order for the contents defined to be recognized.

The contents must be one of the following (and not both):

Text and any number of **L**, **P**, **O**, or **RULEREF** elements in any order with at least one **L**, **P**, or **RULEREF**.

For more information on the use of XML grammars, please see the [Grammar XML Schema](#) section.

[□ Back to top](#)



Grammar Rules and State Graphs

Grammar rules are elements that SAPI 5-compliant speech recognition (SR) engines use to restrict the possible word or sentence choices during the SR process. SR engines employ grammar rules to control the elements of sentence construction using the predetermined list of recognized word or phrase choices. This list of recognized words or phrase choices contained in the grammar rules forms the basis of the SR engine vocabulary.

The phrase or sentence uses each grammar rule element to determine the recognition path. For example, examine the phrase describing travel plans, "I would like to drive from Seattle to New York," and note that there are elements that determine the resulting information. In this example, a person is planning to drive to New York from Seattle. This is a very simple illustration of what could be a very complex problem.

Determining the same travel plans without limiting the method, direction, and travel destination would result in an infinite number of travel options.

The resulting information can be determined by restricting the available choices for a given sentence. Using this method, the resulting information can be composed only from certain choices, thus eliminating the possibility of an infinite number of travel plan combinations.

I would like to drive from Seattle to New York.

	[Method]				
/	\				
Fly	<u>Drive</u>				
	[Direction]				
/	\				
<u>From</u>	To				

```

      | | |
    [City] | | |
    Seattle | | |
    New York | | |
    Los Angeles | | |
    Albuquerque | | |
      | | |
    [Direction] |
      / \ |
    To From |
      |
      [City]
      Seattle
      New York
      Los Angeles
      Albuquerque

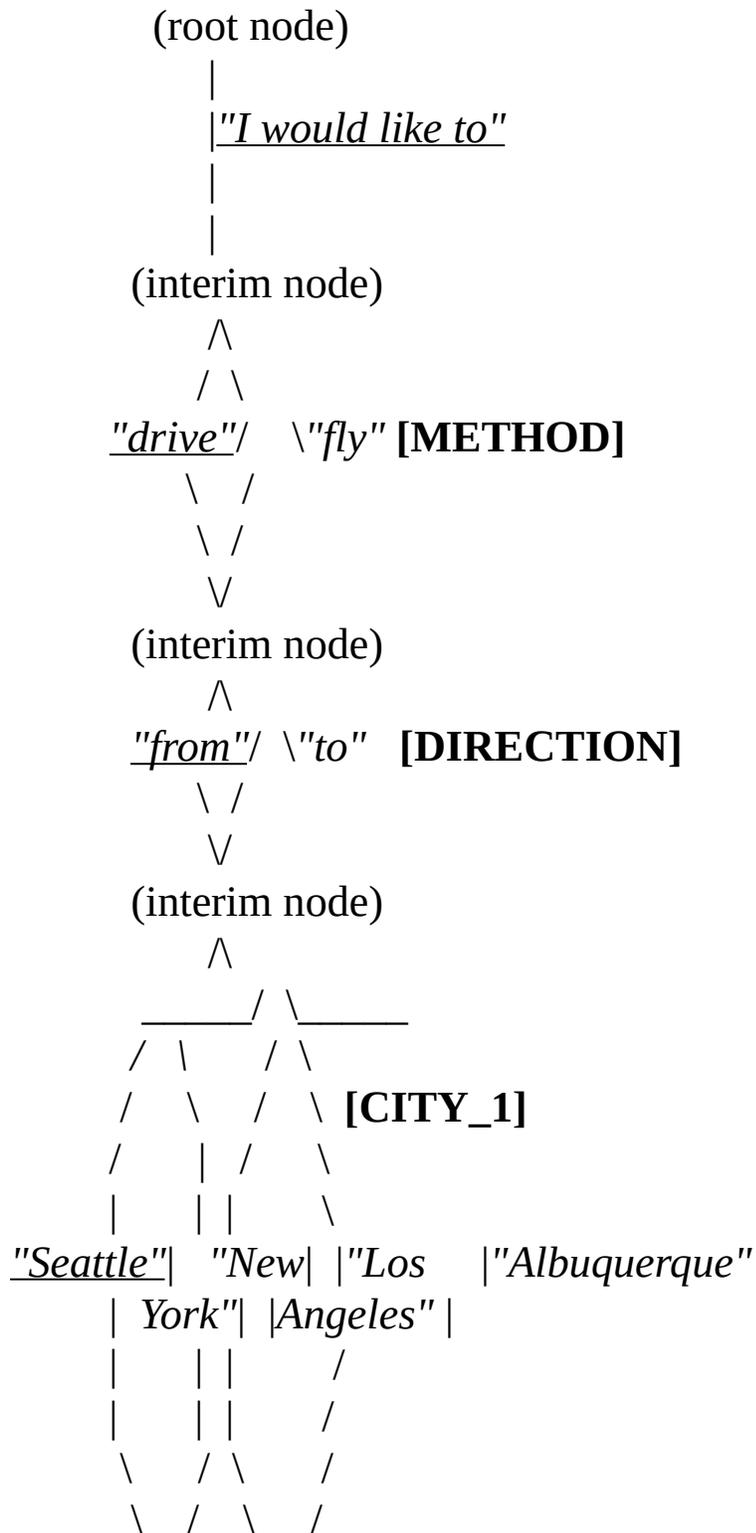
```

The elements of interest in the example phrase are as follows:

- Method of travel (fly or drive), specifically "drive"
- Travel direction (from or to), specifically "from"
- The city of origin for the travel plan (from), specifically "Seattle"
- Travel direction compliment (from or to), specifically "to"
- The city of destination for the travel plan (to), specifically "New York"

The information can also be displayed as a graph of states and arcs, where each arc can have text (or semantic tags/properties) attached. The valid phrases are the unique paths through the graph, starting at the root and ending at a terminal state. Each state is denoted by the term (root node, interim node, and null)

for the terminal node. The spoken text is denoted by words surrounded by quotation marks. The semantic property names are denoted by bold, block quoted words.



information by restricting the input choice to a limited set of possibilities. Otherwise, obtaining the travel plan information from the same sample phrase, "I would like to travel from Seattle to New York," would be considerably more ambiguous.

The complexity of parsing the same sentence increases exponentially without using a defined set of choices. Imagine the possible number of combinations in a sentence that is not restricted to a finite list of combinations. For example, examine the possible choice combinations by moving the mouse over the following sentence.

To display the available choice selections in the example phrase, move the mouse over the underlined text below:

"I want to—(unknown travel method)
—(unknown travel direction)—(unknown city)
—(unknown travel direction) (unknown city)." The amount of predictable information is significantly reduced without the ability to constrain the available choices within a sentence.

The semantic structure (using name/value pairs) is:

**[METHOD="drive"], [DIRECTION="from"],
[CITY_1="Seattle"], [DIRECTION="to"], [CITY_2="New
York"]**

By parsing the semantic structure, the application can easily and accurately analyze the content of the original phrase, without parsing or analyzing individual words. The application developer can then write application logic to perform specific actions based on the previously mentioned semantic names, and specialize the action based on the values of each semantic property. The grammar author can add to or delete from the lists of words, without breaking the application logic.

Grammar rules apply to the following:

TOPLEVEL versus non-TOPLEVEL

A grammar tagged as TOPLEVEL can be in an active or

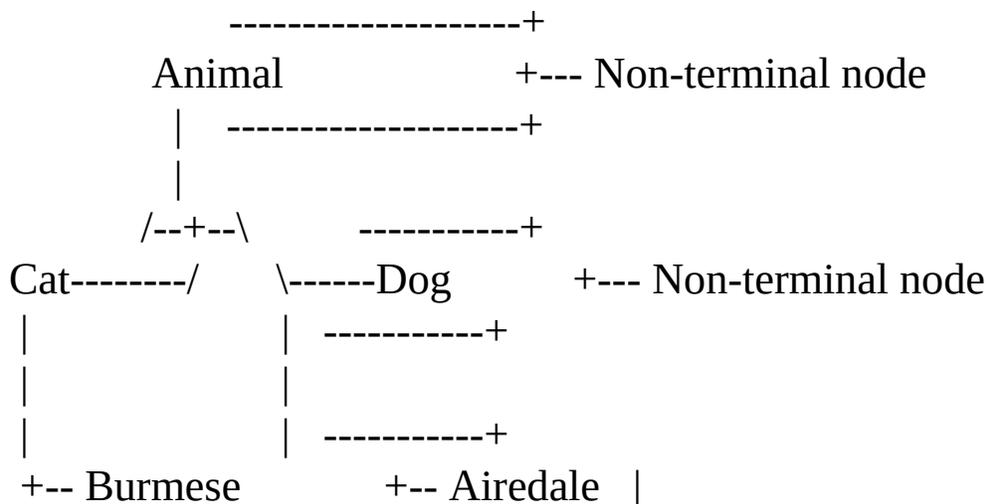
inactive state. The rules that import a grammar can override the activation state of a rule. This conditional state can be configured dynamically at run time. If an inactive grammar is included in another grammar or grammar rule, ignore the inactive state. When a rule is activated, an SR engine will accept only speech satisfying at least one of the active rules contained in the loaded grammar. If a rule is not marked TOPLEVEL, then it is a component rule, and not directly accessible (i.e., the user can only speak TOPLEVEL rules for valid recognition).

Non-terminal

A grammar node is considered to be non-terminal if it is the beginning of a choice selection or a group of choice selections. For example, the grammar node Dog is non-terminal when the subsequent choice selections are types of dogs. This type of grammar node is defined as non-terminal because of its choice selections.

Terminal

A grammar node is considered to be terminal if it's the only word in the recognized vocabulary which can be spoken. Using the Dog example above, terminal grammar nodes are the type of dogs.



```

+-- Himalayan      +-- Poodle   +--- Terminal nodes
+-- Persian        +-- Schnauzer |
+-- Siamese        +-- Whippet  |
                   +-----+

```

The text format grammar XML tags follow block scope methods that are similar to HTML tags. That is, each tag has an opening tag and a corresponding closing tag. There is more information about XML syntax in the [Grammar XML Schema](#) section.

XML tag syntax	Contents
<sometag NAME="some_name" VAL="some_value">	Start of "sometag" tag scope which includes the name and value information.
</sometag>	End of the "sometag" scope.

[Back to top](#)



Designing Grammar Rules

Speech applications often use context-free grammars (CFG) to parse the recognizer output and in some instances, to act as the recognizer's language model. Speech recognition engines use CFGs to constrain the user's words to words that it will recognize. If the CFG is augmented with semantic information (property names and property values as explained below), a SAPI component converts the recognized word string into a name/value-meaning representation. The application then uses the meaning representation to control its part of the conversation with the user.

The following section covers:

- [Semantic Properties or Tags](#)
- [Separation of Dynamic and Static Content](#)
- [Use Dynamic Rules for Language Flexibility](#)
- [Retrieving Semantic Tags or Properties from Recognition Results](#)
- [Using Semantic Properties, Hypotheses, and "Property Pushing"](#)

meeting with Amy Anderson," would be as follows:

request:meeting

participants:AmyAnd

[Back to top](#)

Separation of dynamic and static content

Applications should separate dynamic rule content from static rule content to implement good grammar design and to improve initial SAPI grammar compiler performance. For example, using the above grammar that uses a list of names, the application could create a separate rule (isolated in its own grammar) that contained only the names. The list of names, based on an address book or past user data, can be updated at run time. The static grammar would then contain a rule reference (e.g., RULEREF) to the dynamic content. When the application starts up, it can quickly load the static content, without loading the SAPI grammar compiler, to prevent delay in the startup sequence. Then, the application could load the dynamic content, which requires SAPI to initialize the backend grammar compiler.

☐ [Back to top](#)

Use dynamic rules for language flexibility

Suppose an application needs to support a phrase such as "send new e-mail to NAME." The phrase "send new e-mail to" is static, and known by the application at design time, well before run time. The application could use the following static XML grammar to support these phrases.

```
<GRAMMAR LANGID="409"><!-- american english grammar -->
  <RULE NAME="E-MAIL" TOPLEVEL="INACTIVE"><!-- inactive by (
    <PHRASE>send new e-mail to</P>
    <RULEREFF NAME="ADDRESS_BOOK" PROPNAME="NAME"/><!-- add
  </RULE>
  <RULE NAME="ADDRESS_BOOK" DYNAMIC="TRUE">
    <PHRASE>placeholder</PHRASE><!-- we'll stick placeholder
  </RULE>
</GRAMMAR>
```

The source code to manipulate the dynamic rule, "ADDRESS_BOOK" follows:

```
HRESULT hr = S_OK;

// create a new grammar object
hr = cpRecoContext->CreateGrammar(GRAM_ID, &cpRecoGrammar
// Check hr

// deactivate the grammar to prevent premature recognition
hr = cpRecoGrammar->SetGrammarState(SPGS_DISABLED);
// Check hr

// load the email grammar dynamically, so changes can be
hr = cpRecoGrammar->LoadCmdFromFile(L"email.xml", SPLO_D
// Check hr

SPSTATEHANDLE hRule;

// first retrieve the dynamic rule ADDRESS_BOOK
hr = cpRecoGrammar->GetRule(L"ADDRESS_BOOK", NULL, SPRAF_
// Check hr
```

```

// clear the placeholder text, and everything else in the
hr = cpRecoGrammar->ClearRule(hRule);
// Check hr

// add the real address book (e.g. "Frank Lee", "self",
// Note that ISpRecoGrammar inherits from ISpGrammarBuild
// so application gets the grammar compiler and ::AddWord

hr = cpRecoGrammar->AddWordTransition(hRule, NULL, L"Frank Lee");
// Check hr
hr = cpRecoGrammar->AddWordTransition(hRule, NULL, L"self");
// Check hr
hr = cpRecoGrammar->AddWordTransition(hRule, NULL, L"SAP");
// Check hr
// ... add rest of address book

// commit the grammar changes, which updates the grammar
// and notifies the SR Engine about the rule change (
hr = cpRecoGrammar->Commit(NULL);
// Check hr

// activate the grammar since "construction" is finished
// and ready for receiving recognitions
hr = cpRecoGrammar->SetGrammarState(SPGS_ENABLED);
// Check hr

```

[Back to top](#)

Retrieving semantic tags or properties from recognition results

Note the XML grammar used a semantic property tag, NAME, in the static grammar. The property will enable the application to retrieve the dynamic phrase very easily at run time. Whenever recognition is received with rule name, "E-MAIL," search the property tree (see SPPHRASE.pProperties) for the property named "NAME." Then call ISpRecoResult::GetPhrase with (SPPHRASEPROPERTY)pNameProp.ulFirstElement and (SPPHRASEPROPERTY)pNameProp.ulFirstElement, and the application can retrieve the exact text that the user spoke into the dynamic rule (e.g., user says "send new e-mail to Frank Lee," and you retrieve "Frank Lee," user says "send new e-mail to self," and you retrieve "self," etc.).

```
// activate the e-mail rule to begin receiving recognition
hr = cpRecoGrammar->SetRuleState(L"EMAIL", NULL, SPRS_ACTIVE);
// Check hr

PWCHAR pwszEmailName = NULL;

// default event interest is recognition, so wait for recognition
// NOTE: this could be placed in a loop to process multiple
hr = cpRecoContext->WaitForNotifyEvent(MY_REASONABLE_TIMEOUT);
// Check hr

// event notification fired
if (S_OK == hr) {

    CSpEvent spEvent;
    // if event retrieved and it is a recognition
    if (S_OK == spEvent.GetFrom(cpRecoContext) && SPEI_RULE) {

        // get the recognition result
        CComPtr<ISpRecoResult> cpRecoResult = spEvent.RecoResult();

        if (cpRecoResult) {
            SPPHRASE* pPhrase = NULL;
        }
    }
}
```


Using semantic properties, hypotheses, and "property pushing"

SAPI supports a feature called "semantic property pushing" which enables applications to detect the semantic property structure more accurately at recognition time. "Property pushing" is done by SAPI at compile time, whereby the compiler moves semantic properties to the last terminal node within a rule that remains unambiguous.

For example, the phrases "a b c d" and "a b e f g" both have prefixes of "a b". The compiler will automatically split the phrases into three separate phrases, "a b", "c d", and "e f g", where the first phrase is the common prefix to both recognizable phrases.

The purpose of this feature is to enable applications that place properties on the phrases to detect which branch is being hypothesized as soon as the first unambiguous (non-common) portion of the phrase is spoken. When the user speaks "a b" it is not clear if the user will say "a b c d" or "a b e f g". If the user then says "e", the application can obviously eliminate the "a b c d" option. If the grammar author attached properties to the end of both phrases, the semantic property would be returned as soon as the user spoke the first unambiguous portion of the text (e.g., "c" or "e").

Note that the compiler will report an error ("Ambiguous Semantic Property") if multiple properties are pushed to the same node and two phrases are not unique. For example, the following grammar will fail with "ambiguous semantic property" because both phrases are the same and the compiler cannot determine which property to assign to phrases.

```
<RULE NAME="AmbiguousProperty" TOPLEVEL="ACTIVE">
  <L>
    <P PROPID="42">this is a test</P>
    <P PROPID="3">different sentence</P>
```

```
<P PROPID="75">this is a test</P>  
</L>  
</RULE>
```

The first and third phrases are the same. Note that these results are by design and are meant to prevent creating grammars that have multiple phrases with conflicting semantic properties.

There are a number of scenarios where property pushing can be helpful for an application.

One possibility is an application that wants to detect failures more intelligently. When a false recognition occurs, the application can detect the last semantic property returned and display an error message relevant to the attempted voice command.

Another scenario might be that of high-performance applications that wish to increase responsiveness of the user interface when a long voice command is spoken. The application can wait for the first unambiguous semantic property to be received (using hypothesis) and then fire the response action without waiting for the voice command to complete. This has the added benefit of allowing users to speak partial voice commands (e.g., instead of "go to website w w w Microsoft com" the user can say the slightly shorter "go to website w w w Microsoft"). The drawback is that the application must guard against performing critical, unrecoverable actions before completing the phrase (e.g., "delete hard drive" might fire after only "delete" if there are no other "delete" commands). Careful application design should enable the application to appear quicker and easier to use, without sacrificing robustness. By performing user studies, the application designer can decide which commands are capable of short circuiting and which are more critical.

[□ Back to top](#)



Grammar Format Tags

The SAPI text grammar format is composed of XML tags, which can be structured to define the phrases that the speech recognition engine recognizes. The formal XML schema for the text grammar format is defined in a separate document, called [XML Schema: Grammar](#). The following document explains each tag in more detail, including sample source code, sample XML grammar snippets, and relevant application scenarios.

The XML tags descriptions are organized by XML element, where each element description contains information for relevant attributes.

XML Tags: Elements

<DEFINE>

Summary: The **DEFINE** tag is used for declaring a set of strings.

XML Attributes:

None

XML Parent Elements:

GRAMMAR: The container for the entire XML grammar.

XML Child Elements:

ID (1 or more required): The **DEFINE** tag can contain one or more **ID** elements, each of which defines one string identifier.

Detailed Description:

None

XML Grammar Sample(s):

```

<GRAMMAR>
  <DEFINE>
    <ID NAME="TheNumberFive" VAL="5"/>
  </DEFINE>

  <!-- Note that the ID takes a number, which is actually
  <RULE ID="TheNumberFive" TOPLEVEL="ACTT
    <P>five</P>
  </RULE>
</GRAMMAR>

```

Programmatic Equivalent:

See the [ID](#) tag.

[Back to top](#)

<DICTATION>

Summary: The **DICTATION** tag is used in rules or phrases that

XML Attributes:

MAX (optional, type=VT_I4, default=MIN): Specifies the maximum number of words that can be recognized.

The application must specify a MAX value that is greater than or equal to the MIN value. The application can specify a pseudo-infinity value by specifying INF as the MAX. The pseudo-infinity value is used for dictation words.

An application that needs free-form dictation, such as an email should use a large MAX. Alternatively, an application that needs to recognize a person's name may want a small MAX, such as 5 words.

MIN (optional, type=VT_I4, default=1): Specifies the minimum number of words that must be recognized.

If the grammar author specifies the MIN value, and the value does not meet the minimum, the rule will fail to be recognized.
A Scenario where it may make sense to set a value greater than 1 is an application that is asking for a first and last name.
PROPID (optional, type=VT_I4): Specifies the semantic property ID.
PROPNAME (optional): Specifies the semantic property name.

XML Parent Elements:

LIST, L: List of phrases which can be recognized.

PHRASE, P: Phrase that must be recognized for the content.

OPT, O: Optional phrase that may be recognized.

RULE: Rule that contains phrases or text to be recognized.

XML Element Children:

None.

Detailed Description:

The **DICTATION** tag is designed for applications that need to integrate voice control and dictation support into a CFG. For example, an application may allow the user to speak free-form dictation into a command line, such as *our family's budget* where "our family's budget" is a phrase. The application may also create a CFG which supports a set of phrases, and also includes a single **DICTATION** tag in case of a phrase. For example, a CFG may include a set of address book entries, and if the user speaks another name, then the application can perform the validation of the dictated result. Note that the SR engine does not suffer by mixing dictation and CFG phrases together, and a CFG is generally preferred for applications that use similar words.

The grammar author can also use a special character, asterisk, in an XML tag. See [XML Grammar Format: Special Dictation](#). By using semantic properties, the application can easily recognize

was dictated by the speaker. To specify a semantic pr the grammar author should specify the **PROPID** and SAPI run time will automatically set the semantic tag allowing the application to search for the specific sen properties hierarchy (see [SPPHRASEPROPERTY](#).ul words are recognized by the SR engine (e.g. **DICTA** run time will generate multiple semantic properties, c all of the properties will have the same numeric ID ar

If the speech recognition engine supports multiple dictatio general, legal, medical, etc.), the **DICTION** tag in topic that was selected when [ISpRecoGrammar::Loa](#) topic was not explicitly selected, then the default SR will be loaded. Currently, it is not possible to load mu inside of a single command & control grammars. Apj grammar objects to implement the latter scenario.

If there is ambiguity between a dictation phrase and a CFG recognition engine will typically choose the CFG phr dictation prevents dictation from automatically consu

The speech recognition engine must support dictation insi to load and activate successfully. The application can supports the **DICTION** tag by retrieving the SR e [ISpRecognizer::GetRecognizer](#)), and then checking f engine attribute "*DictationInCFG*" (see [ISpObjectTo](#) The engine can specify support for the **DICTION** CFG phrase (attribute value="*Anywhere*"), or only at value="*Trailing*").

XML Grammar Sample(s):

```
<GRAMMAR>
  <!-- basic command to create a self-note for the user
  <RULE ID="SelfNote" TOPLEVEL="ACTIVE">
    <P>note to self</P>
    <DICTION MAX="INF"/>
```

```

</RULE>

<!-- command to query a name from an address book
<RULE ID="QueryName" TOPLEVEL="ACTIVE":
    <P>list first names of all persons with last name
    <!-- Store only one word for the last name, more
    <DICTATION MAX="1">
</RULE>

<!-- command to handle first and last names with sen
<!-- By using semantic properties, the application can
    the text returned, except for the text associated w
    tags' semantic properties "PID_FirstName" and
<RULE ID="SubmitName" TOPLEVEL="ACTIVE"
    <P>
        my first name is
        <!-- Note the implicit maximum is only one
        <DICTATION PROPID="PID_FirstName"
        and my last name is
        <!-- Note the implicit maximum is two wor
        <DICTATION PROPID="PID_LastName"
    </P>
</RULE>
</GRAMMAR>

```

Programmatic Equivalent:

To programmatically create a dictation transition (i.e. **DIC**) can use the [ISpGrammarBuilder::AddRuleTransition](#) called **SPRULETRANS_DICTATION**. For example, a command called "SendMail" which recognizes the cc

```
SPSTATEHANDLE hsSendMail;
```

```

// Create new top-level rule called "SendMail"
hr = cpRecoGrammar->GetRule(L"SendMail", [
    SPRAF_TopLevel | SPRAF_
    &hsSendMail]);

// Check hr

// Create an interim state before the dictation tra
SPSTATEHANDLE hsBeforeDictation;
hr = cpRecoGrammar->CreateNewState(hsSend
// Check hr

// Add the command words "send mail to"
hr = cpRecoGrammar->AddWordTransition(hsS
    L"send mail to", L" ", SPWT_LEXIC.
// Check hr

// Add trailing dictation transition
hr = cpRecoGrammar->AddRuleTransition(hsB
    SPRULETRANS_DIC
// Check hr

// save/commit changes
hr = cpRecoGrammar->Commit(NULL);
// Check hr

```

Note that the previous sample code only supports one dict more than one word, the code would need to build m states, each of which begins at the previous dictation a series of consecutive single-word dictation transitio

[□ Back to top](#)

<GRAMMAR>

Summary: The **GRAMMAR** tag is the outermost container for

XML Attributes:

LANGID (optional, type=numeric): The language identifier. The identifier will be compared against the supported languages of the Speech Recognition engine. If the language is not supported, the grammar load call will fail (e.g. `SR_SpeakGrammar`). It is recommended that all XML grammars include the `LANGID` attribute where the SR engine tries to load a grammar with a language ID that is not to confusing words.

SAPI supports fuzzy language ID matching, in that it will report that it supports the major portion of the Language ID which means the SR engine will try to load and recognize a major portion of the language ID.

LEXDELIMITER (optional): The **LEXDELIMITER** attribute specifies the sequence of characters used for lexicon entries specified in the grammar.

Grammar authors are able to specify the lexicon information as a sequence of characters. The sequence of characters is specified by the **LEXDELIMITER** attribute.

LEXDELIMITERDisplayFormLEXDELIMITER

The default delimiter is the backslash character "\".

See also [PHRASE](#).

WORDTYPE (optional): The **WORDTYPE** attribute specifies the word type in the grammar.

The default value is "**LEXICAL**".

The value must be "**LEXICAL**".

XML Parent Elements:

None

XML Child Elements:

DEFINE (optional): Specifies the constant definitions for the grammar.

RULE (1 or more required): Specifies the rules, including the grammar.

Detailed Description:

Every XML grammar must have the container tag, **GRAM**

XML Grammar Sample(s):

```
<!-- Language ID = British English -->
<GRAMMAR LANGID="413" LEXDELIMITER="|" W
  <RULE NAME="HelloWorld" TOPLEVEL="ACTI
    <!-- when the user says the following pronuncia
      <P>|Hiya|Hello|h eh l ow;</P>
    </RULE>
  </GRAMMAR>
```

Programmatic Equivalent:

To programmatically set the language ID of a new gramm call [ISpGrammarBuilder::ResetGrammar](#).

The application developer does not need to change the **LE** interface can be used to modify the lexicon.

[Back to top](#)

<ID>

Summary: The **ID** tag is used for declaring a string identifier f values.

XML Attributes:

NAME (required): The **NAME** attribute defines the strin with the constant value.

VAL (required, type=VT_UI4,VT_I4,VT_R4,VT_R8): value that will be associated with the string

XML Parent Elements:

DEFINE: The container for the constant definitions.

XML Child Elements:

None

Detailed Description:

The **ID** tag should be used by grammar author to make the maintain. The grammar author can use string identifier; the use of the identifier (e.g. RID_FileNew, PVAL_MA compiler stores the identifiers in the binary format, and typically much larger than numeric identifiers. Also, th can use a simple numeric comparison to handle rule an rather than performing a more complex string comparis

XML Grammar Sample(s):

```
<GRAMMAR>
  <DEFINE>
    <ID NAME="RuleId_A" VAL="1"/>
    <ID NAME="PropId_B" VAL="2"/>
    <ID NAME="PropVal_AB" VAL="3"/>
  </DEFINE>

  <!-- Note that Rule ID, Phrase PROPID and VAL tak
  <RULE ID="RuleId_A" TOPLEVEL="ACTIVE">
    <P PROPID="PropId_B" VAL="PropVal_AB">
  </RULE>
</GRAMMAR>
```

Programmatic Equivalent:

The Grammar Compiler that ships in the Microsoft Speech argument to generate a C-style header (see "-h"), whic

constant definitions for all of the **IDs** defined in the XML developer can include the header file and easily use them in the application logic, without needing to redefine and recompile. The XML Grammar Sample above would create the following

```
#define RuleId_A 1
#define PropId_B 2
#define PropVal_AB 3
```

[Back to top](#)

<LIST>, <L>

Summary: The **LIST** tag is used for specifying a list of phrase

XML Attributes:

PROPID (optional, type=VT_I4): The numeric identifier for the semantic properties in the child elements (e.g. phrases).
PROPNAME (optional): The string identifier that will be used for the semantic properties in the child elements (e.g. phrases).

XML Parent Elements:

LIST, L: List of phrases or rules which can be recognized.
PHRASE, P: Phrase that must be recognized for the content.
OPT, O: Optional phrase causing the rule reference to be optional.
RULE: Rule that contains phrases or text to be recognized.

XML Child Elements:

RULEREFF: Import, or reference, another rules contents.
PHRASE, P: Specifies text or leaf nodes.
LIST, L: Specifies a list of phrases or transitions for recognition.
TEXTBUFFER: Specifies a reference to the run-time application text-buffer.
WILDCARD: Specifies a garbage word; one or more non-recognized words.

DICTATION: Specifies a piece of text recognized by the

Detailed Description:

The **LIST** tag is a quick and efficient way to support lists of creating separate rules for each piece of text, the **L** where its children are the phrase, rule reference, or ot The grammar author can use the shorthand version of the l The **LIST** tag is more of a virtual tag, since it does not aff hierarchy (**LIST** children are not child properties). W author to specify a string or numeric identifier, the id to pass on to the child element as a default property is

XML Grammar Sample(s):

```
<GRAMMAR>
  <!-- Note that rule is not top-level and is only used as
  <RULE NAME="Numbers">
    <!-- The list tag includes a semantic property Id,
    is inherited by all child phrase elements -->
    <LIST PROPID="PID_Value">
      <!-- If the user says "one" then the semanti
      be the name/value pair "PID_Value"/"
      <P VAL="1">one</P>
      <P VAL="2">two</P>
      <P VAL="3">three</P>
      <P VAL="4">four</P>
      <P VAL="5">five</P>
    </LIST>
  </RULE>

  <!-- The rule contains a list of various types of transit
  <RULE NAME="Sampler" TOPLEVEL="ACTIVE'
  <!-- the list property specifies a default property
```

which will be overridden by specific list children.

```

<LIST PROPNAME="TYPE_NUMBER">
  <P VAL="1">one</P>
  <P VAL="2">two</P>
  <P VAL="3">three</P>
  <P PROPNAME="TYPE_STRING" VALS
  <P PROPNAME="TYPE_NONE">five</P>
  <RULEREF NAME="Numbers" PROPNAME="TYPE_NUMBER">
  <TEXTBUFFER PROPNAME="TYPE_NUMBER">
  <DICTATION PROPNAME="TYPE_DICTATION">
</LIST>
</RULE>
</GRAMMAR>

```

Programmatic Equivalent:

To programmatically create a list, or a set of sibling/parallel elements, you need to create a start state, then create multiple transitions. For example, the following sample code shows how to create a list containing the words "one", "two", and "three".

```

SPSTATEHANDLE hsList;
// Create new top-level rule called "List"
hr = cpRecoGrammar->GetRule(L"List", NULL,
                           SPRAF_TopLevel | SPRAF_Recursive,
                           &hsList);
// Check hr

// Add the word "one" to the list
hr = cpRecoGrammar->AddWordTransition(hsList,
                                       L"one", L" ",
                                       SPWT_LEXICAL, NULL, NULL);
// Check hr

```

```

// Add the word "two" to the list
hr = cpRecoGrammar->AddWordTransition(hsL
    L"two", L" ",
    SPWT_LEXICAL, NULL, NULL);
// Check hr

// Add the word "three" to the list
hr = cpRecoGrammar->AddWordTransition(hsL
    L"three", L" ",
    SPWT_LEXICAL, NULL, NULL);
// Check hr

// save/commit changes
hr = cpRecoGrammar->Commit(NULL);
// Check hr

```

The application developer can use similar code to create dictation, or text buffer transitions. To change the `::AddWordTransition` call to `::AddRuleTrans`

[Back to top](#)

<OPT>, <O>

Summary: The **OPT** tag is used for specifying optional text in

XML Attributes:

DISP (optional): Specifies the display form of the phrase

MAX (optional, type=VT_I4, default=MIN): Specifies the number of times the phrase can repeat and still be successfully recognized

MIN (optional, type=VT_I4, default=1): Specifies the number of times the phrase must repeat and still be successfully recognized

PRON (optional): Specifies the pronunciation to be used

for the text.

PROPID (optional, type=VT_I4): Specifies the numeric tag's semantic property.

PROPNAME (optional): Specifies the string identifier to semantic property.

VAL (optional, type=VT_I4): Specifies the semantic property.

VALSTR (optional): Specifies the semantic property's string value.

WEIGHT (type=VT_UI4,VT_I4,VT_R4,VT_R8, default=1): Specifies the weight of the sibling transition or phrase.

XML Parent Elements:

RULEREF: Import, or reference, another rules contents

PHRASE, P: Specifies text or leaf nodes.

OPT, O: Optional phrase causing the rule reference to be optional.

LIST, L: Specifies a list of phrases or transitions for recognition.

TEXTBUFFER: Specifies a reference to the run-time application text-buffer.

WILDCARD: Specifies a garbage word; one or more non-dictation words.

DICTATION: Specifies a piece of text recognized by the dictation engine.

XML Child Elements:

RULEREF: Import, or reference, another rules contents

PHRASE, P: Specifies text or leaf nodes.

OPT, O: Optional phrase causing the rule reference to be optional.

LIST, L: Specifies a list of phrases or transitions for recognition.

TEXTBUFFER: Specifies a reference to the run-time application text-buffer.

WILDCARD: Specifies a garbage word; one or more non-dictation words.

DICTATION: Specifies a piece of text recognized by the dictation engine.

Detailed Description:


```

<!-- Speaking "one two three" will return three differ
semantic properties, with different names, and d
values -->
<!-- Speaking "one three" will return two different ur
semantic properties, with different names, and d
values -->
<!-- Speaking "one two" will return two different uni
semantic properties, with different names, and d
values -->
<!-- Speaking "one" will return two different unique
semantic properties, with different names, and d
values -->
<!-- Note that the number of semantic properties retu
variable, and that the application should be desi
handle all of the variations -->
<RULE NAME="UseProps" TOPLEVEL="ACTIVE"
  <!-- named property, without value -->
  <P PROPNAME="NOVALUE">one</P>

  <!-- named property, with numeric value -->
  <O PROPNAME="NUMBER" VAL="2">two<

  <!-- named property, with string value -->
  <O PROPNAME="STRING" VALSTR="three"
</RULE>

<!-- Create a rule for optional command prefix -->
<!-- Note that entire rule reference is optional. In cas
there are properties associated with the rule refe
semantic property tree may change -->
<!-- the rule supports the phrases "play cards", "pleas
"please play cards" -->
<RULE NAME="PlayCard" TOPLEVEL="ACTIVE

```

```
<O><RULEREF NAME="PLEASE"/></O>
<P>play cards</P>
</RULE>
```

```
<!-- The first word "pretty" is optional, while the second is not -->
<RULE NAME="PLEASE">
    <O>pretty</O>
    <P>please</P>
</RULE>
</GRAMMAR>
```

Programmatic Equivalent:

To add an optional phrase to a rule, SAPI provides an API [ISpGrammarBuilder::AddWordTransition](#). The application uses the optional structure as follows:

```
SPSTATEHANDLE hsHelloWorld;
// Create new top-level rule called "HelloWorld"
hr = cpRecoGrammar->GetRule(L"HelloWorld"
    SPRAF_TopLevel | SPRAF_Optional
    &hsHelloWorld);
// Check hr

// create an interim state
SPSTATEHANDLE hInterim;
hr = cpRecoGrammar->CreateNewState(hsHelloWorld,
// Check hr

// Add the command word "hello" which terminates the
// interim state
hr = cpRecoGrammar->AddWordTransition(hsHelloWorld,
    L"hello", NULL,
    SPWT_LEXICAL, NULL, NULL);
```

```

// Check hr

// Add the optional command word "world"
hr = cpRecoGrammar->AddWordTransition(hIn
    L"hello", NULL,
    SPWT_LEXICAL, NULL, NULL);
// Check hr

// Add the epsilon transition, which means no w
hr = cpRecoGrammar->AddWordTransition(hIn
    NULL, NULL,
    SPWT_LEXICAL, NULL, NULL);
// Check hr

// save/commit changes
hr = cpRecoGrammar->Commit(NULL);
// Check hr

```

[Back to top](#)

<PHRASE>, <P>

Summary: The **PHRASE** tag and the **OPT** tags are the sole m recognized by the speech recognition engine.

XML Attributes:

DISP (optional): Specifies the display form of the phrase

MAX (optional, type=VT_I4, default=MIN): Specifies t can repeat the phrase and still be successfully recogn

MIN (optional, type=VT_I4, default=1): Specifies the m must repeat the phrase and still be successfully recog

PRON (optional): Specifies the pronunciation to be used for the text.

PROPID (optional, type=VT_I4): Specifies the numeric tag's semantic property.

PROPNAME (optional): Specifies the string identifier to semantic property.

VAL (optional, type=VT_I4): Specifies the semantic property's value.

VALSTR (optional): Specifies the semantic property's string value.

WEIGHT (type=VT_UI4,VT_I4,VT_R4,VT_R8, default=1): Specifies the weight that the user will speak the contents of the PHRASE sibling transition or phrase.

XML Parent Elements:

RULEREF: Import, or reference, another rules contents

PHRASE, P: Specifies text or leaf nodes.

OPT, O: Optional phrase causing the rule reference to be optional.

LIST, L: Specifies a list of phrases or transitions for recognition.

TEXTBUFFER: Specifies a reference to the run-time application's text-buffer.

WILDCARD: Specifies a garbage word; one or more non-recognized words.

DICTATION: Specifies a piece of text recognized by the user.

XML Child Elements:

RULEREF: Import, or reference, another rules contents

PHRASE, P: Specifies text or leaf nodes.

OPT, O: Optional phrase causing the rule reference to be optional.

LIST, L: Specifies a list of phrases or transitions for recognition.

TEXTBUFFER: Specifies a reference to the run-time application's text-buffer.

WILDCARD: Specifies a garbage word; one or more non-recognized words.

DICTATION: Specifies a piece of text recognized by the user.

Detailed Description:

The **PHRASE** tag along with the **OPT** tag are the only tags that can be used to define a phrase.

contain recognizable text. Except for grammars that c
 references, every grammar must have at least one **PH**
 The grammar author can use the shorthand version of the l
 The grammar author can also specify custom word pronun
 text by using the **PRON** and **DISP** attributes. For exa
 might contain application or domain specific text, wh
 pronunciation. The author can specify the pronunciat
PHRASE tag to avoid the need for updating the user
 lexicon (especially if the pronunciation is command s
 The grammar author can also use special shorthand charac
 content section of the **PHRASE** tag (e.g. dictation, w
 the [XML Special Characters](#).

XML Grammar Sample(s):

```
<GRAMMAR>
  <!-- Create a simple "hello world" rule -->
  <RULE NAME="HelloWorld" TOPLEVEL="ACTIV"
    <P>hello world</P>
  </RULE>

  <!-- Create a more advanced "hello world" rule that c
    display form. When the user says "hello world"
    text will be "Hiya there!" -->
  <RULE NAME="HelloWorld_Dispatch" TOPLEVEL="ACTIV"
    <P DISP="Hiya there!">hello world</P>
  </RULE>

  <!-- Create a rule that changes the pronunciation and
    form of the phrase. When the user says "eh" the
    text will be "I don't understand?". Note the user
    say "huh". The pronunciation for "what" is speci
    phrase tag and is not changed for the user or app
    lexicon, or even other instances of "what" in the
```

```
<RULE NAME="Question_Pron" TOPLEVEL="AC
  <P DISP="I don't understand" PRON="eh">wh:
</RULE>
```

```
<!-- Create a rule demonstrating repetition -->
<!-- the rule will only be recognized if the user says '
  diddle" -->
```

```
<RULE NAME="NurseryRhyme" TOPLEVEL="AC
  <P>hey</P>
  <P MIN="2" MAX="2">diddle</P>
</RULE>
```

```
<!-- Create a list with variable phrase weights -->
<!-- If the user says similar phrases, the recognizer w
  the weights to pick a match -->
```

```
<RULE NAME="UseWeights" TOPLEVEL="ACTI
  <LIST>
    <!-- Note the higher likelihood that the use
      expected to say "recognizer speech" --
    <P WEIGHT=".95">recognize speech</P>
    <P WEIGHT=".05">wreck a nice beach</P>
  </LIST>
</RULE>
```

```
<!-- Create a phrase with an attached semantic proper
<!-- Speaking "one two three" will return three differ
  semantic properties, with different names, and d
  values -->
```

```
<RULE NAME="UseProps" TOPLEVEL="ACTIVE
  <!-- named property, without value -->
  <P PROPNAME="NOVALUE">one</P>
```

```
<!-- named property, with numeric value -->
```

```

    <P PROPNAME="NUMBER" VAL="2">two</P>
    <!-- named property, with string value -->
    <P PROPNAME="STRING" VALSTR="three">three</P>
</RULE>
</GRAMMAR>

```

Programmatic Equivalent:

To add a phrase to a rule, SAPI provides an API called [ISpGrammarBuilder::AddWordTransition](#). The application uses the sentences as follows:

```

SPSTATEHANDLE hsHelloWorld;
// Create new top-level rule called "HelloWorld"
hr = cpRecoGrammar->GetRule(L"HelloWorld"
    SPRAF_TopLevel | SPRAF_
    &hsHelloWorld);
// Check hr

// Add the command words "hello world"
// Note that the lexical delimiter is " ", a space character
// By using a space delimiter, the entire phrase
// in one method call
hr = cpRecoGrammar->AddWordTransition(hsHelloWorld,
    L"hello world", L" ",
    SPWT_LEXICAL, NULL, NULL);
// Check hr

// Add the command words "hiya there"
// Note that the lexical delimiter is "|", a pipe character
// By using a pipe delimiter, the entire phrase
// in one method call
hr = cpRecoGrammar->AddWordTransition(hsHelloWorld,

```

```
L"hiya|there", L"|",
SPWT_LEXICAL, NULL, NULL);
// Check hr

// save/commit changes
hr = cpRecoGrammar->Commit(NULL);
// Check hr
```

[Back to top](#)

<RESOURCE>

Summary: The **RESOURCE** tag is used by grammar authors to provide additional data on rules (e.g. for use by a CFG Interpreter, or an SRG engine to access the resources).

XML Attributes:

NAME: specifies the name of the resource to attach to the rule.

XML Parent Elements:

[RULE](#): The rule that contains the resource reference.

XML Child Elements:

[CDATA] (required): The resource value is specified by a CDATA element.

For example,

```
<![CDATA[This is a test string]]>
```

The **RESOURCE** tag contains the CDATA element, which provides the resource value.

Detailed Description:

The **RESOURCE** tag is a facility allowing the grammar author to provide additional information [attached to rules] to a CFG Interpreter (e.g. [ISpCFGInterpreter](#) and [ISpCFGInterpreterSite::GetResource](#)).

speech recognition engine that is aware of the resource (see [ISpSREngineSite::GetResource](#)).

XML Grammar Sample(s):

```
<GRAMMAR>
  <!-- Note resource value can be any string -->
  <RULE ID="RID_TestResource" TOPLEVEL="AC"
    <RESOURCE NAME="AResource">
      <![CDATA[AResource's Value: String]]>
    </RESOURCE>
    <P>test an embedded resource</P>
  </RULE>
</GRAMMAR>
```

Programmatic Equivalent:

To add a resource to a rule, SAPI provides an API called [ISpGrammarBuilder::AddResource](#). The application adds the aforementioned resource (see XML Grammar Sample code):

```
SPSTATEHANDLE hsTestResource;
// Create new top-level rule called "TestResource"
hr = cpRecoGrammar->GetRule(NULL, RID_TopLevel,
    SPRAF_TopLevel | SPRAF_IncludeResource,
    &hsTestResource);
// Check hr

// Add the command words "test an embedded resource"
hr = cpRecoGrammar->AddWordTransition(hsTestResource,
    L"test an embedded resource", L"",
    SPWT_LEXICAL, NULL, NULL);
// Check hr
```

```

// Add the resource named "AResource"
hr = cpRecoGrammar->AddResource(hsTestRes
                                L"AResource",
                                L"AResource's Value: String");
// Check hr

// save/commit changes
hr = cpRecoGrammar->Commit(NULL);
// Check hr

```

Then, the SR-Engine can retrieve the resource value of the rule updates or CFG-recognition by making

```

// set hRule to handle with resource

hr = cpSREngineSite->GetResource(hRule
                                L"AResource",
                                &pwszResValue);
if (S_OK == hr)
{
    // pwszResValue contains the value
    // perform value-sensitive processing

    // release value memory
    ::CoTaskMemFree(pwszResValue);
}

```

[Back to top](#)

<RULE>

Summary: The **RULE** tag is the core tag for defining which c

recognition. Every grammar must have at least one top-level rule and every rule must have at least one rule reference or recursive reference.

XML Attributes:

DYNAMIC (optional, default is FALSE): Specifies whether a rule can be modified at run time. By default, an application cannot modify a rule in an XML grammar. To modify a rule, the rule must be marked with the **DYNAMIC** attribute. The grammar must be loaded with the dynamic flag (see [SPLOADOPTIONS](#)). Dynamic rules cannot be marked with **EXPORT**.

EXPORT (optional, default is FALSE): Specifies whether a rule is exported from a grammar to reference it. For example, a grammar author must mark rules with **EXPORT="TRUE"**. Exported rules cannot be modified at run time.

ID (required, type=VT_I4): Specifies the numeric identifier of the rule. The **NAME** must be specified, or both. The identifier is used to identify the rule namespace, which is the entire grammar (see [TOPLEVEL](#)).

INTERPRETER (optional, default is FALSE): Specifies whether a rule is interpreted by a CFG interpreter (see [ISpCFGInterpreter](#)) when it is referenced. A rule might contain semantic properties or text that is evaluated at run time (e.g. replace value of the semantic property with the system's current date and time).

NAME (required): Specifies the string identifier of the rule. The **ID** must be specified, or both. The identifier is used to identify the rule namespace, which is the entire grammar (see [TOPLEVEL](#)).

TOPLEVEL (optional): Specifies that the rule is directly referenced by a top-level rule structure. If the **TOPLEVEL** tag is not specified, then the rule is not a top-level rule structure unless it is referenced by another top-level rule structure. Top-level rule structures (see [RULEREFF](#)) do not need to specify **TOPLEVEL**. When a grammar author specifies a rule as **TOPLEVEL**, the rule is enabled by default. If the rule is enabled by default (e.g. **TOPLEVEL="ACTIVE"**), then when the application sets the set of rules (e.g. [ISpRecoGrammar::SetRuleState\(NUL\)](#)), then the rule will be activated. If a rule is specified as

TOPLEVEL="INACTIVE", then it will only be active (see [ISpRecoGrammar::SetRuleState](#) and [ISpRecoGrammar::SetRuleIdState](#)).

XML Parent Elements:

[GRAMMAR](#): The container for the entire XML grammar

XML Child Elements:

[RULEREF](#): Import, or reference, another rules contents

[PHRASE, P](#): Specifies text or leaf nodes.

[LIST, L](#): Specifies a list of phrases for recognition.

[OPT, O](#): Specifies an optional piece of text that can be spoken

[TEXTBUFFER](#): Specifies a reference to the run-time application text-buffer.

[WILDCARD](#): Specifies a garbage word; one or more non-words

[DICTATION](#): Specifies a piece of text recognized by the speech recognizer

[RESOURCE](#): Specifies a labeled piece of arbitrary string accessed by a special SR engine, or a CFG interpreter

Detailed Description:

The **RULE** tag is the core of the XML grammar text format. A CFG is to define a specific set of words and phrases spoken by the user and recognized by the speech recognizer. Rules can be written by the grammar author in a way that is reusable, textually maintainable, and conducive to application that is based on semantic properties or actions (not on words). Each rule must contain at least one piece of text, or a rule reference (has the same requirements). Effectively, every rule with a piece of text (i.e. leaf or terminal node).

The rule can be identified by either a numeric identifier (**ID**) or a **(NAME)**. The grammar author can use the **DEFINE** tag to use numeric identifiers for numeric values. By using the constant **TOPLEVEL="INACTIVE"**, then it will only be active (see [ISpRecoGrammar::SetRuleState](#) and [ISpRecoGrammar::SetRuleIdState](#)).

returned as soon as the user spoke the first unambiguous (e.g. "c" or "e"). [See Semantic Properties, Hypotheses](#)

XML Grammar Sample(s):

```
<GRAMMAR>
  <DEFINE>
    <ID NAME="RID_Hello" VAL="1"/>
    <ID NAME="RID_World" VAL="2"/>
    <ID NAME="RID_AddNumbers" VAL="3"/>
    <ID NAME="RID_Numbers" VAL="4"/>
    <ID NAME="RID_Numbers_Exportable" VAL="5"/>
    <ID NAME="RID_Names" VAL="6"/>
  </DEFINE>
  <!-- create a simple top-level rule that uses a constant -->
  <RULE ID="RID_Hello" TOPLEVEL="ACTIVE">
    <P>hello</P>
  </RULE>

  <!-- Create a simple top-level rule that is inactive by default -->
  <RULE NAME="Hiya" TOPLEVEL="INACTIVE">
    <P>hiya</P>
  </RULE>

  <!-- Create a rule, which a CFG-interpreter can re-interpret -->
  <RULE NAME="InterpretedRule" TOPLEVEL="ACTIVE">
    <P PROPNAME="TODAY">what is today's date?</P>
  </RULE>

  <!-- Create a simple top-level rule that references another rule -->
  <RULE ID="RID_AddNumbers" TOPLEVEL="ACTIVE">
    <P>add</P>
    <RULEREF REFID="RID_Numbers"/>
  </RULE>
</GRAMMAR>
```

```
<P>to</P>
<RULEREFF REFID="RID_Numbers"/>
</RULE>
```

<!-- Note that rule is not top-level and is only used as

```
<RULE ID="RID_Numbers">
  <LIST PROPID="PID_Value">
    <P VAL="1">one</P>
    <P VAL="2">two</P>
    <P VAL="3">three</P>
    <P VAL="4">four</P>
    <P VAL="5">five</P>
  </LIST>
</RULE>
```

<!-- mark the rule as dynamic so the application can use it
at runtime -->

```
<RULE ID="RID_Names" DYNAMIC="TRUE">
  <LIST>
    <P>bob</P>
    <P>jane</P>
    <P>kate</P>
    <P>tom</P>
  </LIST>
</RULE>
```

<!-- Mark the rule as exportable, so other external gra

```
<RULE ID="RID_Numbers_Exportable" EXPORT=
  <LIST PROPID="PID_Value">
    <P VAL="6">six</P>
    <P VAL="7">seven</P>
    <P VAL="8">eight</P>
    <P VAL="9">nine</P>
```

```

        <P VAL="10">ten</P>
    </LIST>
</RULE>
</GRAMMAR>

```

Programmatic Equivalent:

Application developers can programmatically add rules to [ISpGrammarBuilder](#) interface inherited by [ISpRecoC](#) shows how to add a rule to a grammar. To choose the [ISpGrammarBuilder::GetRule](#) method and [SPCFGR](#)

```

SPSTATEHANDLE hHelloWorld;
// Create new rule called "HelloWorld"
// Note that the second parameter is the ID, which can
// Note also that the rule is marked as top-level and as
hr = cpRecoGrammar->GetRule(L"SpeakNumber", 1,
                           TRUE, &hHelloWorld);

// Check hr

// add the text "hello world"
hr = cpRecoGrammar->AddWordTransition(hHelloWorld,
                                     L" ", SPWT_LEXICAL, 1, 1);

// Check hr

// save the grammar changes
hr = cpRecoGrammar->Commit(NULL);
// Check hr

```

The following sample code shows how to modify a rule in the code will update the list of names rule shown in the section. By updating the names rule, all rules that refer to it will automatically be able to recognize the updated names

```

SPSTATEHANDLE hNames;

// Get a handle to the existing rule
// Note the use of the constant identifier RID_Names,
// XML sample. See the ID tag for information on
hr = cpRecoGrammar->GetRule(NULL, RID_Names
// Check hr

// clear the rule to update the entire list
hr = cpRecoGrammar->ClearRule(hNames);
// Check hr

// add name "sally"
hr = cpRecoGrammar->AddWordTransition(hNames
                                     SPWT_LEXICAL, NULL, 1
// Check hr

// add name "jim"
hr = cpRecoGrammar->AddWordTransition(hNames
                                     SPWT_LEXICAL, NULL, 1
// Check hr
// add name "diane"
hr = cpRecoGrammar->AddWordTransition(hNames
                                     SPWT_LEXICAL, NULL, 1
// Check hr

// save grammar changes
hr = cpRecoGrammar->Commit(NULL);
// Check hr

```

[Back to top](#)

<RULEREF>

Summary: The **RULEREF** tag is used for importing rules from a grammar. The **RULEREF** tag is especially useful for off-the-shelf rules and grammars.

XML Attributes:

NAME (required): Specifies the string identifier of the rule. The **REFID** or the **REFID** must be specified. If both are specified, they must refer to the same rule.

OBJECT (optional): Specifies the programmatic identifier of the object which contains the compiled grammar (see [ISpCFGInterpreter::InitGrammar](#)).

PROPID (optional, type=VT_I4): Specifies the numeric property attached to the rule reference.

PROPNAME (optional): Specifies the string identifier of the property attached to the rule reference.

REFID (required, type=VT_I4): Specifies the numeric identifier of the rule. The **NAME** or the **REFID** must be specified. If both are specified, they must refer to the same rule.

URL (optional): Specifies the uniform resource locator (URL) of the grammar. The URL can be prefixed by "**http://**", "**file://**", or no prefix. The URL can reference either a compiled grammar or an uncompiled XML grammar (e.g. *.xml) which will be compiled.

VAL (optional): Specifies the numeric value that will be a property attached to the rule reference.

VALSTR (optional): Specifies the string value that will be a property attached to the rule reference.

WEIGHT (optional, type=VT_UI4,VT_I4,VT_R4,VT_REAL): Specifies the probability of the contents of the rule (which is referred to by the user).

XML Parent Elements:

[LIST, L](#): List of phrases or rules which can be recognized

PHRASE, P: Phrase that must be recognized for the context
OPT, O: Optional phrase causing the rule reference to be optional
RULE: Rule that contains phrases or text to be recognized

XML Child Elements:

None

Detailed Description:

The **RULEREf** tag is provided to grammar authors to allow structuring semantic properties into a hierarchy.

Grammar reusability is provided by allowing rules to refer to an independent software vendor (ISV) could develop supported mathematic operations and easy to speak natural their grammars via either a web site (URL, http), a C++ compiled grammar. Grammar authors who want to reuse need to add a **RULEREf** tag into their grammar which file or resource location. Similarly, grammar authors can build complex commands by reusing the basic rule components into their grammars (e.g. spelling, number build complex commands by reusing the basic rule components).

Structured, hierarchical semantic properties are built on top of the semantic properties specified inside of a rule are specified in order of declaration in the recognized transition path) that are in rules referenced by another rule are child properties of the rule that made the reference. For example, examine the following XML:

```
<RULE NAME="A" TOPLEVEL="ACTIVE">
  <P PROPNAME="ROOT">
    <RULEREf NAME="B" PROPNAME="CHILD">
  </P>
</RULE>
<RULE NAME="B">
  <P PROPNAME="CHILD">hello</P>
  <P PROPNAME="LEAF">world</P>
```

</RULE>

The grammar contains two rules, one top-level rule w
The top-level rule contains two semantic properties, c
(e.g. "ROOT"), and the other attached to the rule refe
"ROOT_SIBLING"). The second rule also contains t
attached to a phrase tag (e.g. "CHILD), and the other
(e.g. "LEAF"). If the recognized phrase is "hello wor
structure is as follows:

SPPHRASE->pProperties.pszName == "ROOT"

SPPHRASE->pProperties->pNextSibling.pszNa

SPPHRASE->pProperties->pFirstChild.pszNam

SPPHRASE->pProperties->pFirstChild->pNext

Note that no matter how many phrases or semantic p
single **RULE**, all of the properties are siblings. Child
created by using rule references. See also the [Whitep
Retrieving Semantic Properties.](#)

XML Grammar Sample(s):

```
<GRAMMAR>
```

```
<DEFINE>
```

```
<ID NAME="RID_Numbers" VAL="1"/>
```

```
<ID NAME="RID_AddNumbers" VAL="2"/>
```

```
<ID NAME="PID_Value" VAL="1"/>
```

```
</DEFINE>
```

```
<!-- create a simple rule that reuses the local numbers
```

```
<RULE ID="RID_AddNumbers" TOPLEVEL="AC'
```

```
<P>add</P>
```

```
<!-- the first operand will be a number from the
```

```
<!-- the application can retrieve the child proper  
which has a value of 1-5 -->
```

```
<RULEREF REFID="RID_Numbers" PROPNA/
```

```
<P>to</P>
```

```
<!-- the second operand will be a number from t
```

```

        <!-- the application can retrieve the child proper
              which has a value of 1-5 -->
        <RULEREFF REFID="RID_Numbers" PROPNAME="
</RULE>

<!-- Note that rule is not top-level and is only used as
<RULE ID="RID_Numbers">
    <LIST PROPID="PID_Value">
        <P VAL="1">one</P>
        <P VAL="2">two</P>
        <P VAL="3">three</P>
        <P VAL="4">four</P>
        <P VAL="5">five</P>
    </LIST>
</RULE>

<RULE NAME="SearchWeb" TOPLEVEL="ACTIVE"
    <P>search web for site named</P>
    <!-- Reference a fictitious rule located on the web
          list of SR-friendly web site names -->
    <RULEREFF NAME="SiteNames" URL="http://
</RULE>

<RULE NAME="SearchAddressBook" TOPLEVEL="ACTIVE"
    <P>find address of</P>
    <!-- Reference a fictitious rule located in a registry
          a dynamic list of Exchange server address list
    <RULEREFF NAME="FullNames" OBJECT="Exchange
</RULE>
</GRAMMAR>

```

Programmatic Equivalent:

Application developers can programmatically import rules

Rule Name = "URL:" + FILENAME + "\\ " RULENAME
For example, to import a rule called "Numbers" from the file

```
SPSTATEHANDLE hSpeakNumber;
SPSTATEHANDLE hsBeforeImport;
SPSTATEHANDLE hsRuleImport;
// Create new rule called "SpeakNumber"
hr = cpRecoGrammar->GetRule(L"SpeakNumber", 1, 1);
// Check hr

// Create new state for the beginning text
hr = cpRecoGrammar->CreateNewState(hSpeakNumber, 1, 1);
// Check hr

// add the beginning text "speak the number"
hr = cpRecoGrammar->AddWordTransition(hSpeakNumber, L"speak the number", SPWT_LEXICAL, 1, 1);
// Check hr

// Import the rule "Numbers" from A.cfg
hr = cpRecoGrammar->GetRule(L"URL:file://A.cfg\\Numbers", 1, 1);
// Check hr

// reference the "Numbers" rule after the beginning text
hr = cpRecoGrammar->AddRuleTransition(hsBeforeImport, hSpeakNumber, hr);
// Check hr

hr = cpRecoGrammar->Commit(NULL);
// Check hr
```

[□ Back to top](#)

<TEXTBUFFER>

Summary: The **TEXTBUFFER** tag is used for applications need a text box or text selection with a voice command.

XML Attributes:

- PROPID (optional, type=VT_I4):** Specifies the semantic property
- PROPNAME (optional):** Specifies the semantic property name
- WEIGHT (optional, type=VT_UI4,VT_I4,VT_R4,VT_BOOL):** Specifies the probability of the **TEXTBUFFER**-based phrase

XML Parent Elements:

- LIST, L:** List of phrases which can be recognized.
- PHRASE, P:** Phrase that must be recognized for the content
- OPT, O:** Optional phrase that may be recognized.
- RULE:** Rule that contains phrases or text to be recognized

XML Child Elements:

None

Detailed Description:

The **TEXTBUFFER** tag is useful for applications that have a text buffer and want to allow the user to speak portions of the text in the buffer. An example is likely the text selection user interface. The **TEXTBUFFER** tag defines a buffer of text, and allows the user to select any contiguous portion of the buffer. For example, when the text is "a b c d e", the user can select "a b c" and "c d e", but not "b e" since it is not a contiguous portion of the text buffer.

The **TEXTBUFFER** tag allows the grammar author to define a dynamic text buffer which will be set and maintained by the application. For example, the grammar might contain the command "select a b c", which, when using the previous text sample, would allow the user to say "select a b c", "select "c d e", but not "select b e". The grammar author's efforts on building commands to operate on the text buffer are simplified by the **TEXTBUFFER** tag.

application developer need only focus on maintaining [ISpRecoGrammar::SetWordSequenceData](#) and [ISpRe](#) responding to the **TEXTBUFFER**-based commands.

The **TEXTBUFFER** has three main components, the context text subsets in the buffer, and the active selection. This is a string of text characters, which is double-NULL terminated for using a double-NULL to allow for multiple exclusives to be active (e.g. each subset is a paragraph). The recognizer does not recognize phrases which span the exclusive subsets (e.g. NULL character). The third component is the active selection, a portion of the buffer that should be recognizable (e.g. update the selection to include on the text visible on the screen or the text selected by the user). Note that any portion of the buffer not included in the **TEXTBUFFER**'s active selection is not available.

The **TEXTBUFFER** tag is shared across all of the commands of the grammar object. For applications that need to support multiple text buffers, the application has three options. If the text buffers are unique but do not need to be active simultaneously, the application can use the selection feature (of the **TEXTBUFFER**) to switch between buffers. If the buffers are unique, but the buffers need to be active simultaneously, the application can use the single-NULL terminated subsets. If the application has multiple text buffers, requires the buffers to be active simultaneously, and uses different commands for each buffer, the application can use a single grammar object for each buffer.

The application should use semantic properties (see attributes) to quickly and easily parse the **TEXTBUFFER**-related commands. The SAPI will automatically set the semantic property's parameter element range to match the elements taken from the **TEXTBUFFER**.

The speech recognition engine must support text-buffers in the grammar to load and activate successfully. The application must ensure an engine supports the **TEXTBUFFER** tag by retrieving the **TEXTBUFFER** token (see [ISpRecognizer::GetRecognizer](#)), and then


```

// Check hr

// Add text-buffer transition
hr = cpRecoGrammar->AddRuleTransition(hsB
                                     SPRULETRANS_TEX
// Check hr

// save/commit changes
hr = cpRecoGrammar->Commit(NULL);
// Check hr

// ... perform other processing/setup

// Setup text-buffer

// Place the contents of text buffer into pwszCoM
//   the length of the text in cch
SPTXTSELECTIONINFO tsi;
tsi.ulStartActiveOffset = 0;
tsi.cchActiveChars = cch;
tsi.ulStartSelection = 0;
tsi.cchSelection = cch;
pwszCoMem2 = (WCHAR *)CoTaskMemAlloc
if (pwszCoMem2)
{
    // SetWordSequenceData requires double N
    memcpy(pwszCoMem2, pwszCoMem, size
    pwszCoMem2[cch] = L'\0';
    pwszCoMem2[cch+1] = L'\0';

    // set the text buffer data
    hr = cpRecoGrammar->SetWordSequenceI
// Check hr

```

```
// set the text selection information independent of the
hr = cpRecoGrammar->SetTextSelection(8);
// Check hr
CoTaskMemFree(pwszCoMem2);
}
CoTaskMemFree(pwszCoMem);

// the SR engine is now capable of recognizing t
```

[Back to top](#)

<WILDCARD>

Summary: The **WILDCARD** tag is used in rules or phrases that provide flexibility for the speaker's phrasing.

XML Attributes:

None

XML Parent Elements:

LIST, L: List of phrases which can be recognized.

PHRASE, P: Phrase that must be recognized for the context.

OPT, O: Optional phrase that may be recognized.

RULE: Rule that contains phrases or text to be recognized.

XML Element Children:

None.

Detailed Description:

The **WILDCARD** tag is designed for applications that want to provide

some phrases without failing due to irrelevant, or ign... example, an application may have a command with th... Many users may trivially modify the phrase by saying "save the document", "save this document", etc.. With phrases would all fail to be recognized due to the ext... author can add a wildcard, or garbage field, which wi... words, and allow the application to successfully hand... In the aforementioned case, the grammar would need "document".

The **WILDCARD** is different from **DICTATION** in that... recognized garbage words, even though they were re... application and grammar author should not place wil... affect the intended user action (e.g. "cancel save" is n... save".

The grammar author can also use a special character, ellip... XML tag. See [XML Grammar Format: Special Wildc](#)

The speech recognition engine must support wildcards ins... to load and activate successfully. The application can... supports the **WILDCARD** tag by retrieving the SR e... [ISpRecognizer::GetRecognizer](#)), and then checking f... engine attribute "WildcardInCFG" (see [ISpObjectTo](#)... The engine can specify support for the **WILDCARD**... CFG phrase (attribute value="Anywhere"), or only at... value="Trailing").

XML Grammar Sample(s):

```
<GRAMMAR>
```

```
<!-- basic command to play the queen of hearts -->
```

```
<RULE ID="PlayCard" TOPLEVEL="ACTIVE">
```

```
<P>play <WILDCARD/> queen of hearts</P>
```

```
</RULE>
```

```
<!-- basic command to play the queen of hearts, using
```

```

<RULE ID="PlayCard_Ellipsis" TOPLEVEL="ACT
  <P>play ... queen of hearts</P>
</RULE>
</GRAMMAR>

```

Programmatic Equivalent:

To programmatically create a wildcard transition in a CFC can use the [ISpGrammarBuilder::AddRuleTransition](#) called **SPRULETRANS_WILDCARD**. For example command called "PlayCard" which recognizes the co

```

SPSTATEHANDLE hsPlayCard;
// Create new top-level rule called "PlayCard"
hr = cpRecoGrammar->GetRule(L"PlayCard", T
    SPRAF_TopLevel | SPRAF_
    &hsPlayCard);
// Check hr

// Create an interim state before the wildcard tra
SPSTATEHANDLE hsBeforeWildcard;
hr = cpRecoGrammar->CreateNewState(hsPlayt
// Check hr

// Add the command word "play"
hr = cpRecoGrammar->AddWordTransition(hsS
    L"play", L" ", SPWT_LEXICAL, NU
// Check hr

// Create an interim state after the wildcard trans
SPSTATEHANDLE hsAfterWildcard;
hr = cpRecoGrammar->CreateNewState(hsPlayt
// Check hr

```

```

// Add interim wildcard transition
hr = cpRecoGrammar->AddRuleTransition(hsB
                                     SPRULETRANS_WIL

// Check hr

// Add the command words "queen of hearts"
hr = cpRecoGrammar->AddWordTransition(hsA
                                     L"queen of hearts", L" ", SPWT_LEX
// Check hr

// save/commit changes
hr = cpRecoGrammar->Commit(NULL);
// Check hr

```

The previous sample code will support any of the following
 "play *the* queen of hearts"
 "play *a* queen of hearts"
 "play *the left* queen of hearts"
 etc.

Note that the italicized words will be recognized by the sp but will not be returned to the application. The application logic sensitive inside of a wildcard, s returned.

[Back to top](#)



Grammar Format Tags: Special Characters

The following Grammar Format tags have associated special characters which can be used as shorthand to modify words within a rule or used to modify the rule itself. These tags allow greater flexibility of a rule by accepting a wider range of words for a given position.

The following tags are available:

[Optional word \(OPT,O\): ?](#)

[Wildcard \(WILDCARD\): ...](#)

[Dictation \(DICTATION\): *](#)

[Confidence increase \(No associated XML tag\): +](#)

[Confidence decrease \(No associated XML tag\): -](#)

Also see the [CoffeeS6 Tutorial](#) for more about embedded dictation and grammar modifiers.

Tags

Optional word: ?

The question before a word marks that word as optional. The word may or may not be used in the position and have no effect on possible rule activation. Using optional words allows for a more natural speaking manner. For example,

```
<P>?Please play the card<P>
```

The equivalent complete form of the phrase would be:

```
<P><0>Please</0>play the card<P>
```

This rule permits the user to eliminate the word "please" and still fire a valid rule. If it were required, the user would have to say "please" in order for the speech recognition (SR) engine to recognize the rule.

[Back to top](#)

Wildcard: ...

An ellipsis is used to accept words not critical to the rule's intent and allow any word or words to be spoken for that position. Unlike other words in a rule that are explicitly listed, the speaker may use any word or words in this position. The SR engine will attempt to recognize the words, and if successful, will accept them as valid elements for activating the rule. However, the words are not returned in the subsequent phrase list. Regardless of the number of words spoken, the phrase element will contain only one representation for all the words. Using wildcards allows for a more natural speaking manner and is meant for words that are unimportant to the intent of the rule. Wildcards extend the optional (question mark tag) by not requiring an explicit list of all the words. For example,

```
<P>I would like a hamburger<P>  
<P>... hamburger<P>
```

The equivalent complete form of the phrase would be:

```
<P><WILDCARD/>hamburger<P>
```

Both phrases attempt the same action, that of ordering a hamburger. In the first instance, the user would have to use the exact syntax stated. Even optional words would have to be listed and the rule could be cumbersome. The second instance the user can say virtually anything preceding the word "hamburger."

However, listing other words is not redundant to using just the wildcard. Explicitly listing alternative phrases increases the confidence of the statement. It also increases the SR engine's recognition process because more is known about the possible spoken content. "Please get me a hamburger," is a much more common phrase, and hence is easier to recognize than a non sequitur word allowed by a wildcard. As an example, "aardvark hamburger," is a valid statement according to the rules above, but would be harder for the SR engine to recognize.

[Back to top](#)

Dictation: *

The dictation asterisk allows any word or words to be spoken for that position and each word is returned by the SR engine in the phrase element list. Unlike the wildcard, the word or words are considered important to the rule. Using the dictation asterisk, a user can say any word without the engine expecting or anticipating its context. For example, using the following rule, users can speak their first name:

```
<P>My first name is *<P>
```

The equivalent complete form of the previous phrase would be:

```
<P>My first name is <DICTATION/><P>
```

If additional words are needed, use the plus sign after dictation to indicate multiple words (up to a pseudo-infinite number, 255):

```
<P>My full name and address is *+<P>
```

The equivalent complete form of the previous phrase

would be:

<P>My full name and address is <DICTATION MAX="INF"/><

[□ Back to top](#)

Confidence increase: +

Confidence decrease: -

One of these two signs placed in front of words respectively increases or decreases the required confidence for a successful recognition. Increasing the required confidence means that the SR engine will have to be much more certain that the word it recognizes really is the expected word. For example, if the user is responding to an important question such as "Reform hard disk?," increasing the required confidence is additional confirmation a "yes" really is "yes." To be certain, the rule would be noted as "+yes".

Likewise, the minus sign decreases the required confidence for the word. This de-emphasizes words. Although the word is required for the rule, it is not important to verify that the user actually said it.

For example, in the following rule the user makes a request. However, while the command is required, it is more important that "name" is properly recognized rather than "enter."

<P>-enter +name<P>

NOTE: There is no XML Grammar tag equivalent for the confidence special characters, "-" and "+".

The exact numerical measurement of low confidence, normal confidence, and high confidence as mapped to the recognition process is defined by the SR engine

vendor. See also the [SR Properties White Paper](#) for more information on manipulating the confidence threshold by the SR engine.

[Back to top](#)



SAPI Grammar Example: Solitaire

Grammar rules define sentence contents and phrase elements. Each grammar and grammar element determines the speech recognition (SR) engine's ability to effectively construct phrase elements. Phrases and sub-expressions are commonly represented by a separate rule and combined into larger phrases and sentences with higher level rules. For more information, see the [Grammar rules](#) section.

The card game called Solitaire, uses semantic objects such as cards, suits, and ranks, and semantic actions, such as "move *SomeCard* to *AnotherCard*", and "new game." The following example illustrates how to implement a grammar for a game of solitaire, which supports the previously mentioned semantic objects and actions. The example also specifies the exact voice command phrases (in American English) that must be spoken in order to play the game.

```
<!-- The grammar tag surrounds the entire CFG description
      Specify the language of the grammar as
      English-American ('409') -->
<GRAMMAR LANGID="409">
  <!-- Specify a set of easy-to-read strings to
        represent specific values. Similar to
        constants or #define in Visual Basic or
        C/++ programming languages -->
  <DEFINE>
    <ID NAME="FROM" VAL="1"/>
    <ID NAME="TO" VAL="2"/>
    <ID NAME="SUIT" VAL="3"/>
    <ID NAME="COLOR" VAL="4"/>
    <ID NAME="RANK" VAL="5"/>
    <ID NAME="ColorRed" VAL="11101"/>
    <ID NAME="ColorBlack" VAL="10011"/>
  </DEFINE>

  <!-- Define a top-level rule for the new game
        command, called 'newgame' -->
  <!-- Make the rule 'active', by default, so
```

```

        the rule is available as soon as speech
        is activated in the application -->
<RULE NAME="newgame" TOPLEVEL="ACTIVE">
  <!-- Require high confidence for the word,
        game, to avoid accidental recognition
        of this important rule -->
  <!-- Make the last word, please, optional,
        only require low-confidence to
        make the command phrasing more
        flexible -->
  <P>new +game</P><O>-please</O>
</RULE>

<!-- Define another active top-level rule,
        called 'playcard' which enables the user
        to use voice commanding to play cards -->
<!-- Define the 'playcard' rule as exportable, so
        we can create other solitaire or card game
        grammars which can re-use the 'playcard'
        rule functionality. -->
<RULE NAME="playcard" TOPLEVEL="ACTIVE" EXPORT="1">
  <O>please</O>
  <P>play the</P>
  <!-- Allow for extraneous garbage words
        from the user. The user could say
        "play the little ace of spades" without
        breaking the voice command -->
  <O>...</O>
  <!-- Use a rule reference to a card grammar
        which is defined elsewhere in the
        overall solitaire grammar. Using
        rule references is similar to
        reusable components in an object-
        oriented programming language or
        component model -->
  <RULEREf NAME="card"/>
  <O>please</O>
</RULE>

<!-- Define another top-level voice command for
        moving one card to another location -->
<!-- Note that phrase structure allows for two

```

types of *move*-commands by making *to_card* section optional. The grammar supports both 'move *from_card* to *to_card*' and simply 'move *from_card*'. The application can select the *from_card* when the latter voice command is recognized, or the application can perform the full move with the former voice command. -->

```
<RULE NAME="movecard" TOPLEVEL="ACTIVE">
  <O>please</O>
  <P>
    <L>
      <P>move</P>
      <P>put</P>
    </L>
    <P>the</P>
  </P>

  <!-- Use a semantic tag/property, called 'FROM'
        which represents the from_card, and
        will contain the phrase structure
        recognized in the rule reference. By
        using semantic properties, the application
        can abstract away the exact phrase text
        and build application logic based on
        the action (i.e., Action=move
        FromCard=(Rank=Ace, Suit=Hearts) -->
  <RULEREF PROPNAME="from" PROPID="FROM" NAME="card"/>
  <O>
    <L>
      <P>on</P>
      <P>to</P>
    </L>
    <P>the</P>
    <!-- Use another semantic property for the
          ToCard information -->
    <RULEREF PROPNAME="to" PROPID="TO" NAME="card"/>
  </O>
  <O>please</O>
</RULE>
```

<!-- Create a reusable *card* grammar, which contains

the structure of a *card's* descriptor (e.g., "red ace", "ace of hearts", or "heart").
Note: It is *not* a top-level rule, since it is only used by other top-level rules and is not directly recognizable -->

```
<RULE NAME="card">
```

```
<!-- Use a phrase list to allow the descriptor to be one of three forms, including a color and rank, or a rank and suit, or only a suit. -->
```

```
<!-- The application can decode the card by analyzing the semantic property structure. For example, the color and rank form will include a property called 'color' with a value either ColorRed or ColorBlack (note that these values are actually numeric defines).-->
```

```
<L>
```

```
<P><!-- color and rank form -->
```

```
<L PROPNAME="color" PROPID="COLOR">
```

```
<P VAL="ColorRed">red</P>
```

```
<P VAL="ColorBlack">black</P>
```

```
</L>
```

```
<RULEREf NAME="rank"/>
```

```
</P>
```

```
<P><!-- rank and suit form -->
```

```
<RULEREf NAME="rank"/>
```

```
<0>
```

```
<P>of</P>
```

```
<L PROPNAME="suit" PROPID="SUIT">
```

```
<P VAL="0">clubs</P>
```

```
<P VAL="1">hearts</P>
```

```
<P VAL="2">diamonds</P>
```

```
<P VAL="3">spades</P>
```

```
</L>
```

```
</0>
```

```
</P>
```

```
<!-- suit only form -->
```

```

    <L PROPNAME="suit" PROPID="SUIT">
      <P VAL="0">club</P>
      <P VAL="1">heart</P>
      <P VAL="2">diamond</P>
      <P VAL="3">spade</P>
    </L>
  </L>
</RULE>
<!-- Create a reusable grammar component, called
      'rank' which represents the numeric rank
      of the various cards. Note that each card
      has an associated value.
      The application can use the semantic property
      called 'rank' (specified by PROPNAME), and
      the value (specified by VAL) to abstract
      itself from the phrasing, and use only the
      numeric rank values. Note that the words
      'king' and 'emperor' both refer to the
      value 13. The grammar author can change
      or update the text without breaking the
      application's semantically-dependent
      logic -->
<RULE NAME="rank">
  <!-- Specify the property name/id in the LIST
        tag, which will be inherited by all
        of the list tag's child phrase tags.
        Specifying the name/id in the LIST tag avoids
        having to specify it multiple times, once
        for each P tag -->
  <L PROPNAME="rank" PROPID="RANK">
    <P VAL="1">ace</P>
    <P VAL="2">two</P>
    <P VAL="3">three</P>
    <P VAL="4">four</P>
    <P VAL="5">five</P>
    <P VAL="6">six</P>
    <P VAL="7">seven</P>
    <P VAL="8">eight</P>
    <P VAL="9">nine</P>
    <P VAL="10">ten</P>
    <P VAL="11">jack</P>
    <P VAL="12">queen</P>
  </L>
</RULE>

```

```
    <P VAL="13">king</P>
    <P VAL="12">lady</P>
    <P VAL="13">emperor</P>
  </L>
</RULE>
<!-- End of Grammar definition -->
</GRAMMAR>
```

[Back to top](#)



ISpGrammarBuilder

This interface details the SAPI context-free grammar (CFG) backend compiler. These methods can be used to programmatically construct and modify grammars.

When To Use

Applications should use the ISpGrammarBuilder interface to change and save dynamically loaded grammars (see [ISpRecoGrammar](#)) and to change and save previously compiled binary grammars (see [ISpGramCompBackend](#)).

Methods in Vtable Order

ISpGrammarBuilder Methods	Description
<u>ResetGrammar</u>	Clears all grammar rules (un-defines them) and resets the grammar's language to NewLanguage.
<u>GetRule</u>	Retrieves a grammar rule's initial state information (and defines the rule if requested).
<u>ClearRule</u>	Removes all of the grammar rule information except for the rule's initial state handle.
<u>CreateNewState</u>	Creates a new state in the same grammar rule as <i>hState</i> .
<u>AddWordTransition</u>	Adds a word or a sequence of words to the grammar.
<u>AddRuleTransition</u>	Adds a rule (reference) transition from one grammar rule to another.
<u>AddResource</u>	Adds a resource (name and string value) to the grammar rule specified in <i>hRuleState</i> .
<u>Commit</u>	Performs consistency checks of the grammar structure, creates the serialized format, saves the grammar structure, or reloads the grammar structure to the stream provided by <i>SetSaveObjects</i> , or reloads it into the SR engine.

Additionally, a sample code is provided to demonstrate ISpGrammarBuilder.

- [Example application of ISpGrammarBuilder](#)



Example application of ISpGrammarBuilder

The code example below illustrates an implementation of a travel grammar, using the [ISpGrammarBuilder](#) interface.

An approximation of the XML form is included for each of the following three grammar authoring approaches.

```
// HRESULT checking code omitted for brevity

SPSTATEHANDLE hStateTravel;
// create (if rule does not already exist) top-level Rule
hr = pGrammarBuilder->GetRule(L"Travel", 0, SPRAF_TopLevel);

{ // Approach 1: list all possible phrases
  // This is the most intuitive approach, and it does not require
  //   because the grammar builder will merge shared states.
  // Internally, SAPI may break the transitions into
  //   there are common roots (e.g. "fly to Seattle" and "drive to Seattle").
  // There is only one root state, hStateTravel, and the transitions
  //   transitions between the root state and the NULL state.

  /* XML Approximation:
    <RULE NAME="Travel" TOPLEVEL="ACTIVE">
      <PHRASE>fly to Seattle</PHRASE>
      <PHRASE>fly to New York</PHRASE>
      <PHRASE>fly to Washington DC</PHRASE>
      <PHRASE>drive to Seattle</PHRASE>
      <PHRASE>drive to New York</PHRASE>
      <PHRASE>drive to Washington DC</PHRASE>
    </RULE>
  */

  // create set of peer phrases, each containing complete phrase
  // Note: the word delimiter is set as " ", so that the grammar can handle
  //   multiple words (e.g. "fly to Seattle" is a single phrase).
  hr = pGrammarBuilder->AddWordTransition(hStateTravel, L"fly", L" ");
  hr = pGrammarBuilder->AddWordTransition(hStateTravel, L"to", L" ");
  hr = pGrammarBuilder->AddWordTransition(hStateTravel, L"Seattle", L" ");
  hr = pGrammarBuilder->AddWordTransition(hStateTravel, L"New York", L" ");
  hr = pGrammarBuilder->AddWordTransition(hStateTravel, L"Washington DC", L" ");
  hr = pGrammarBuilder->AddWordTransition(hStateTravel, L"drive", L" ");
  hr = pGrammarBuilder->AddWordTransition(hStateTravel, L"to", L" ");
  hr = pGrammarBuilder->AddWordTransition(hStateTravel, L"Seattle", L" ");
  hr = pGrammarBuilder->AddWordTransition(hStateTravel, L"New York", L" ");
  hr = pGrammarBuilder->AddWordTransition(hStateTravel, L"Washington DC", L" ");
}
```

```

    hr = pGrammarBuilder->AddWordTransition(hStateTravel,
    hr = pGrammarBuilder->AddWordTransition(hStateTravel,
    hr = pGrammarBuilder->AddWordTransition(hStateTravel,
}

{ // Approach 2: construct the directed-graph using inte
  // This approach gives you more control of the gramma
  //   easier to implement when you have some combinat.
  // Using this approach, there is one root state (hSta
  //   (hStateTravel_Second), and the final terminal NU
  //   unique transitions between the root state and the
  //   three more unique transitions between the interi
  // Note that graph includes only 2-transition paths.
  //   only the first transition or the second transiti
  //   phrase as is "Seattle", but "fly to Seattle" is

  /* XML Approximation:
    <RULE NAME="Travel" TOPLEVEL="ACTIVE">
      <LIST>
        <PHRASE>fly to</PHRASE>
        <PHRASE>drive to</PHRASE>
        <PHRASE>take train to</PHRASE>
      </LIST>
      <LIST>
        <PHRASE>Seattle</PHRASE>
        <PHRASE>New York</PHRASE>
        <PHRASE>Washington DC</PHRASE>
      </LIST>
    </RULE>
  */

  SPSTATEHANDLE hStateTravel_Second;
  // create a new transition which starts at the root s
  hr = pGrammarBuilder->CreateNewState(hStateTravel, &h

  // attach the first part of the phrase to to first tr
  hr = pGrammarBuilder->AddWordTransition(hStateTravel,
  hr = pGrammarBuilder->AddWordTransition(hStateTravel,
  hr = pGrammarBuilder->AddWordTransition(hStateTravel,

  // attach the second and final part of the phrase to

```

```

    hr = pGrammarBuilder->AddWordTransition(hStateTravel_);
    hr = pGrammarBuilder->AddWordTransition(hStateTravel_);
    hr = pGrammarBuilder->AddWordTransition(hStateTravel_);
}

{ // Approach 3: using sub rules
  // This approach let you structure the grammars and i
  // since it allows for reusable component rules (s
  // Note that forward-declarations are allowed, since
  // until the XML is compiled or the GrammarBuilder
  // The main difference between Approach 2 and Approach
  // are combined into one top-level rule. This faci
  // in other rules (e.g. create a second rule calle
  // "where is" with the "Dest" rule, allowing the u
  // requiring the grammar author/designer to place
  // of the grammar leading to grammar maintenance p

  /* XML Approximation:
    <RULE NAME="Travel" TOPLEVEL="ACTIVE">
      <RULEREF NAME="Method"/>
      <RULEREF NAME="Dest"/>
    </RULE>
    <RULE NAME="Method">
      <LIST>
        <PHRASE>fly to</PHRASE>
        <PHRASE>drive to</PHRASE>
        <PHRASE>take train to</PHRASE>
      </LIST>
    </RULE>
    <RULE NAME="Dest" DYNAMIC="TRUE">
      <LIST>
        <PHRASE>Seattle</PHRASE>
        <PHRASE>New York</PHRASE>
        <PHRASE>Washington DC</PHRASE>
      </LIST>
    </RULE>
  */

  SPSTATEHANDLE hStateMethod;
  SPSTATEHANDLE hStateDest;
  // Note the two new rules ("Method" & "Dest") are NOT

```

```

//      reused by other top-level rules, and are not by
hr = pGrammarBuilder->GetRule(L"Method", 0, 0, TRUE, &
// Marking the "Dest" rules as Dynamic allows the prog
//      update only the "Dest" rule after the initial :
//      destinations depending on user history, preferen
hr = pGrammarBuilder->GetRule(L"Dest", 0, SPRAF_Dynam

SPSTATEHANDLE hStateTravel_Second;
// Create an interim state (same as Approach 2)...
hr = pGrammarBuilder->CreateNewState(hStateTravel, &h
// ... then attach rules to the transitions from Root
hr = pGrammarBuilder->AddRuleTransition(hStateTravel,
hr = pGrammarBuilder->AddRuleTransition(hStateTravel_

// Add the set of sibling transitions for travel "met
hr = pGrammarBuilder->AddWordTransition(hStateMethod,
hr = pGrammarBuilder->AddWordTransition(hStateMethod,
hr = pGrammarBuilder->AddWordTransition(hStateMethod,

// Add the set of sibling transitions for travel "des
hr = pGrammarBuilder->AddWordTransition(hStateDest, NI
hr = pGrammarBuilder->AddWordTransition(hStateDest, NI
hr = pGrammarBuilder->AddWordTransition(hStateDest, NI
}

// Must Commit before the grammar changes before using tl
// Note: grammar changes are only given to the engine at
hr = pGrammarBuilder->Commit(0);

```



ISpGrammarBuilder::ResetGrammar

ISpGrammarBuilder::ResetGrammar clears all grammar rules (un-defines them) and resets the grammar's language to *NewLanguage*. The state handles for this grammar are no longer valid after this point.

```
HRESULT ResetGrammar(  
    LANGID NewLanguage  
);
```

Parameters

NewLanguage

[in] Language identifier associated with the grammar rule.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.

Example

The following code snippet illustrates the use of ResetGrammar.

```
HRESULT hr = S_OK;

// ResetGrammar when no rules
hr = pGrammarBuilder->ResetGrammar(409);
// Check hr

// Set language to default user language
hr = pGrammarBuilder->ResetGrammar(SpGetUserDefaultUILang);
// Check hr

// Set language to non-english
hr = pGrammarBuilder->ResetGrammar(MAKELANGID(LANG_CHINESE, SUBLANG_CHINESE_SIMPLIFIED));
// Check hr

hr = pGrammarBuilder->ResetGrammar(MAKELANGID(LANG_JAPANESE, SUBLANG_JAPANESE_ROMAJI));
// Check hr
```



ISpGrammarBuilder::GetRule

ISpGrammarBuilder::GetRule retrieves grammar rule's initial state.

```
HRESULT GetRule(  
    const WCHAR    *pszRuleName,  
    DWORD          dwRuleId,  
    DWORD          dwAttributes,  
    BOOL           fCreateIfNotExist,  
    SPSTATEHANDLE *phInitialState  
);
```

Parameters

pszRuleName

[in] Address of the null-terminated string containing the grammar rule name. If NULL, no search is made for the name.

dwRuleId

[in] Grammar rule identifier. If zero, no search is made for the rule ID.

dwAttributes

[in] Grammar rule attributes for the new rule created. Ignored if the rule already exists. Must be of type [SPCFGRULEATTRIBUTES](#). Values may be combined to allow for multiple attributes.

fCreateIfNotExist

[in] Boolean indicating that the grammar rule is to be created if one does not currently exist. TRUE allows the creation; FALSE does not.

phInitialState

[out] The initial state of the rule. May be NULL.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_RULE_NOT_FOUND	No rule matching the specified criteria can be found and a new rule is not created.
SPERR_RULE_NAME_ID_CONFLICT	One of the name and ID matches an existing rule but the other does not match the same rule.
E_INVALIDARG	At least one parameter is invalid. Also returned when both <i>pszRuleName</i> and <i>dwRuleId</i> are NULL.
E_OUTOFMEMORY	Not enough memory to complete operation.

Remarks

Either the rule name or ID must be provided (the other unused parameter can either be NULL or zero). If both a grammar rule name and identifier are provided, they both must match in order for this call to succeed. If the grammar rule does not already exist and *fCreateIfNotExists* is true, the grammar rule is defined. Otherwise this call will return an error.

Example

The following code snippet illustrates the use of GetRule.

```
HRESULT hr = S_OK;
SPSTATEHANDLE hState;

//=====
// Create a rule with name and ID
hr = pGrammarBuilder->GetRule(L"rule1", 1, SPRAF_Dynamic
//Check return value

//=====
// Create a rule with name only
hr = pGrammarBuilder->GetRule(L"rule", 0, SPRAF_Dynamic,
//Check return value

//=====
// Create a rule with ID only
hr = pGrammarBuilder->GetRule(NULL, 2, SPRAF_Dynamic, TRI
//Check return value

//=====
// Get an existing rule by ID
hr = pGrammarBuilder->GetRule(L"rule1", 1, SPRAF_Dynamic
//Check return value
hr = pGrammarBuilder->GetRule(NULL, 1, SPRAF_Dynamic, FA
//Check return value

//=====
// Get an existing rule by name
hr = pGrammarBuilder->GetRule(L"rule1", 0, SPRAF_Dynamic
//Check return value

//=====
// Get rule references to other grammars
// Compose the name of the rule as follows
// Please note the double back-slash before the rule name.
// OBJECT --> pszRuleName = L"SAPI5OBJECT:MyApp.ClassId\\\\"
```

```
// URL --> pszRuleName = L"URL:http://myserver.com\\\\\\RuleN:  
    hr = pBackend->GetRule(pszRuleName, 0 , SPRAF_Import, TRI  
    //Check return value  
  
// phTarget contains a valid rule handle that can be used to
```



ISpGrammarBuilder::ClearRule

ISpGrammarBuilder::ClearRule removes all of the grammar rule information except for the rule's initial state handle.

```
HRESULT ClearRule(  
    SPSTATEHANDLE    hState  
);
```

Parameters

hState

[in] Handle to the any of the states in the grammar rule to be cleared. Only the rule's initial state handle is still valid.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Value specified in <i>hState</i> is not valid.

Example

The following code snippet illustrates the use of ClearRule.

```
HRESULT hr = S_OK;

SPSTATEHANDLE hInit;
SPSTATEHANDLE hState;
hr = pGrammarBuilder->GetRule(L"rule1", 1, 0, TRUE, &hIn.

// ClearRule using hInitState
hr = pGrammarBuilder->CreateNewState(hInit, &hState);
hr = pGrammarBuilder->AddWordTransition(hInit, hState, L
hr = pGrammarBuilder->ClearRule(hInit);
// Check hr

hr = pGrammarBuilder->AddWordTransition(hInit, hState, L
// E_INVALIDARG because hState in no longer valid

// ClearRule using hState != hInit
hr = pGrammarBuilder->CreateNewState(hInit, &hState);
hr = pGrammarBuilder->AddWordTransition(hInit, hState, L
hr = pGrammarBuilder->ClearRule(hState);
// Check hr

hr = pGrammarBuilder->AddWordTransition(hInit, hState, L
// E_INVALIDARG because hState in no longer valid
```



ISpGrammarBuilder::CreateNewState

ISpGrammarBuilder::CreateNewState creates a new state in the same grammar rule as *hState*.

```
HRESULT CreateNewState(  
    SPSTATEHANDLE    hState,  
    SPSTATEHANDLE    *phState  
);
```

Parameters

hState

[in] Handle to any existing state in the grammar rule.

phState

[out] Address of the state handle for a new state in the same grammar rule.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	The <i>hState</i> is not a valid state handle.
E_POINTER	The <i>phState</i> pointer is invalid.
E_OUTOFMEMORY	Exceeded available memory.

Example

The following code snippet illustrates the use of CreateNewState.

```
HRESULT hr = S_OK;

SPSTATEHANDLE hInit;
hr = pGrammarBuilder->GetRule(L"rule1", 1, 0, TRUE, &hIn.

// CreateNewState using the hInitState
SPSTATEHANDLE hState;
hr = pGrammarBuilder->CreateNewState(hInit, &hState);
// Check hr

// CreateNewState using hState != hInit
SPSTATEHANDLE hState2;
hr = pGrammarBuilder->CreateNewState(hState, &hState2);
// Check hr
```



ISpGrammarBuilder::AddWordTransition

ISpGrammarBuilder::AddWordTransition adds a word or a sequence of words to the grammar.

```
HRESULT AddWordTransition(  
    SPSTATEHANDLE          hFromState,  
    SPSTATEHANDLE          hToState,  
    const WCHAR            *psz,  
    const WCHAR            *pszSeperators,  
    SPGRAMMARWORDTYPE      eWordType,  
    float                  weight,  
    const SPPROPERTYINFO   *pPropInfo  
);
```

Parameters

hFromState

[in] Handle of the state from which the arc (or sequence of arcs in the case of multiple words) should originate.

hToState

[in] Handle of the state where the arc (or sequence of arcs) should terminate. If NULL, the final arc will be to the (implicit) terminal node of this grammar rule.

psz

[in] Address of a null-terminated string containing the word or words to be added. If *psz* is NULL, an epsilon arc will be added.

pszSeperators

[in] Address of a null-terminated string containing the transition word separation characters.

psz points to a single word if *pszSeperators* is NULL, or else *pszSeperators* specifies the valid separator characters. This parameter may not contain a forward slash ("/") as that is used for the complex word format.

eWordType

[in] The [SPGRAMMARWORDTYPE](#) enumeration that specifies the word type. Currently, only SPWT_LEXICAL is supported.

Weight

[in] Value specifying the arc's relative weight in case there are multiple arcs originating from *hFromState*.

pPropInfo

[in] The [SPPROPERTYINFO](#) structure containing property name and value information that is associated with this arc or sequence of arcs.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	At least one of <i>psz</i> , <i>pszSeparators</i> , or <i>pPropInfo</i> is invalid or bad; <i>eWordType</i> is a value other than SPWT_LEXICAL; <i>flWeight</i> is less than 0.0; a slash ("/") is used as a separators.
SPERR_WORDFORMAT_ERROR	Invalid word format.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.



ISpGrammarBuilder::AddRuleTransition

ISpGrammarBuilder::AddRuleTransition adds a rule (reference) transition from one grammar rule to another.

```
HRESULT AddRuleTransition(  
    SPSTATEHANDLE          hFromState,  
    SPSTATEHANDLE          hToState,  
    SPSTATEHANDLE          hRule,  
    float                  Weight,  
    const SPPROPERTYINFO *pPropInfo  
);
```

Parameters

hFromState

[in] Handle of the state from which the arc should originate.

hToState

[in] Handle of the state where the arc should terminate. If NULL, the final arc will be to the (implicit) terminal node of this grammar rule.

hRule

[in] Handle of any state of the rule to be called with this transition. Get the *hRule* using the [ISpGrammarBuilder::GetRule\(\)](#) call. To refer to a rule in another grammar, and "import" that rule by calling [ISpGrammarBuilder::GetRule](#)(... , [SPRAF_Import](#), TRUE */*fCreatIfNotExist*/*, ...).

hRule can also be one of the following special transition handles:

Transition handle	Description
-------------------	-------------

SPRULETRANS_WILDCARD	<WILDCARD> transition
SPRULETRANS_DICTATION	<DICTATION> single word from dictation
SPRULETRANS_TEXTBUFFER	<TEXTBUFFER> transition

Weight

[in] Value specifying the arc's relative weight in case there are multiple arcs originating from *hFromState*.

pPropInfo

[in] The [SPPROPERTYINFO](#) structure containing property name and value information that is associated with this arc.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	At least one parameter is invalid.
E_OUTOFMEMORY	Not enough memory to complete operation.
FAILED(hr)	Appropriate error message.



ISpGrammarBuilder::AddResource

ISpGrammarBuilder::AddResource adds a resource (name and string value) to the grammar rule specified in *hRuleState*. The resource can be queried by a rule interpreter using [ISpCFGInterpreterSite::GetResourceValue\(\)](#).

```
HRESULT AddResource(  
    SPSTATEHANDLE    hRuleState,  
    const WCHAR      *pszResourceName,  
    const WCHAR      *pszResourceValue  
);
```

Parameters

hRuleState

[in] Handle of a state in the rule to which the resource is to be added.

pszResourceName

[in] Address of a null-terminated string specifying the resource name.

pszResourceValue

[in] Address of a null-terminated string specifying the resource value.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	At least one of the parameters is invalid.

SPERR_DUPLICATE_RESOURCE_NAME	The resource already exists.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.

Example

The following code snippet illustrates the use of AddResource.

```
HRESULT hr = S_OK;

SPSTATEHANDLE hInit;
hr = pGrammarBuilder->GetRule(L"rule1", 1, 0, TRUE, &hIn.

SPSTATEHANDLE hState;
hr = pGrammarBuilder->CreateNewState(hInit, &hState);
// Check hr

// AddResource using the hInitState
hr = pGrammarBuilder->AddResource(hInit, L"ResName1", L"
// Check hr

// AddResource using hState != hInit
hr = pGrammarBuilder->AddResource(hState, L"ResName2", L
// Check hr
```



ISpGrammarBuilder::Commit

ISpGrammarBuilder::Commit performs consistency checks of the grammar structure, creates the serialized format, saves the grammar structure, or reloads the grammar structure.

The grammar structure may be saved it to the stream provided by *SetSaveObjects*, or reloaded into the SR engine. Commit must be called before any changes to the grammar can take effect.

```
HRESULT Commit(  
    DWORD    dwReserved  
);
```

Parameters

dwReserved
Reserved. Must be zero.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>dwReserved</i> is not zero.
SPERR_UNINITIALIZED	Stream not initialized. Call <i>SetSaveObjects</i> before <i>Commit</i> .
SPERR_NO_RULES	A grammar must have at least one rule and one word.
SPERR_NO_TERMINATING_RULE_PATH	At least one rule is not empty but has no

	terminating path (path of transitions from the initial state to a NULL state).
SPERR_CIRCULAR_RULE_REF	At least one rule has left recursion (a direct or indirect rule reference to itself originated from the initial state).
SPERR_STATE_WITH_NO_ARCS	At least one rule has a node with no outgoing transitions.
SPERR_EXPORT_DYNAMIC_RULE	Dynamic rules or rules referencing dynamic rules (directly or indirectly) cannot be exported.



Lexicon interfaces

The following section covers:

- [Lexicon Interfaces Overview](#)
- [ISpContainerLexicon](#)
- [ISpLexicon](#)
- [ISpPhoneConverter](#)



Lexicon Interfaces Overview

The [ISpLexicon](#) interface provides a uniform way for applications and engines to access the user lexicon, application lexicon, and engine private lexicons.

The following topics are covered in this section:

- [ISpLexicon information for application developers](#)
- [ISpLexicon information for engine developers](#)

ISpLexicon information for application developers

[SpLexicon](#) is the SAPI standard lexicon object which implements the [ISpLexicon](#) interface. It contains the user lexicon and all application lexicons registered in the system when the SpLexicon object was CoCreated. Engines can add their private lexicons through [ISpContainerLexicon](#) interface. However, if an application uses ISpContainerLexicon to add a lexicon to an instance of [SpLexicon](#), the engine will not use the lexicon.

You can get pronunciations from both the user and application lexicons:

```
hr = cpLexicon->GetPronunciation(... eLEXTYPE_USER ...);  
hr = cpLexicon->GetPronunciation(... eLEXTYPE_APP ...);  
hr = cpLexicon->GetPronunciation(... eLEXTYPE_USER | eLEXT
```

You can also add or remove an application lexicon. To do so, let your COM object implement the [ISpLexicon](#) and [ISpObjectWithToken](#) interfaces, and register it as illustrated in the following example:

```
hr = SpCreateNewTokenEx(SPCAT_APPLEXICONS, pszLangIndependent  
// hr = cpDataKeyAttribs->SetStringValue(name1, value1); //  
// hr = cpDataKeyAttribs->SetStringValue(name2, value2); //
```

```
// ...
```

You can also use the SAPI-provided CLSID_SpUnCompressedLexicon to implement your application lexicon as follows (the CLSID_SpCompressedLexicon is intended for engine vendors):

```
hr = SpCreateNewTokenEx(SPCAT_APPLEXICONS, pszLangIndependent,
// hr = cpDataKeyAttribs->SetStringValue(name1, value1); //
// hr = cpDataKeyAttribs->SetStringValue(name2, value2); //
// ...

hr = SpCreateObjectFromToken(cpToken, &cpAppLexicon);

cpAppLexicon->AddPronunciation(...);
cpAppLexicon->AddPronunciation(...);
cpAppLexicon.Release(); // the CLSID_SpUnCompressedLexicon
```

To remove an application lexicon from the system:

1. Locate your lexicon with the [SpFindBestToken](#)(..., &cpToken) function.
2. Call cpToken->Remove (NULL) to remove the lexicon from the system.

When an application lexicon is added to the system, it is shared by all applications. Any [SpLexicon](#) object created afterward will automatically load the application lexicon. Application lexicons will override engine private lexicons.

[Back to top](#)

ISpLexicon information for engine developers

[ISpLexicon](#) provides a uniform format to access user, application, and engine private lexicons. CLSID_SpLexicon implements [ISpContainerLexicon](#), which is derived from ISpLexicon and has one more method, AddLexicon.

```
ComPtr<ISpContainerLexicon> cpLexicon;
```

```
// load user lexicon and all application lexicons register  
hr = cpLexicon.CoCreateInstance(CLSID_SpLexicon);  
  
// create your private lexicons implementing ISpLexicon, e  
hr = cpLexicon->AddLexicon(pMyLex1, eLEXTYPE_PRIVATE1);  
hr = cpLexicon->AddLexicon(pMyLex2, eLEXTYPE_PRIVATE2);  
...
```

Engines can access lexicons in the following manner:

```
hr = cpLexicon->GetPronunciations(... eLEXTYPE_USER ...);  
hr = cpLexicon->GetPronunciations(... eLEXTYPE_APP ...);  
hr = cpLexicon->GetPronunciations(... eLEXTYPE_USER | eLEXTYPE_APP ...);
```

[GetPronunciations](#) will return a [SPWORDPRONUNCIATIONLIST](#) structure consisting of pronunciations found in all the specified lexicons.

The lexicon pronunciation information is returned from `GetPronunciations` in the following order:

1. Pronunciation from user lexicon (could have multiple pronunciations)
2. Pronunciation from application lexicon(s) (could have multiple pronunciations)
3. Pronunciation from the added lexicons in the same order the lexicons were added.

The expected order of priority is the same as what is returned from `GetPronunciations` in the above list.

The [ISpLexicon](#) interface can be used to add or remove words from the user lexicon. However, engine developers will not typically use `ISpLexicon::AddPronunciation` and `ISpLexicon::RemovePronunciation` to add or remove words from user lexicons.

When private lexicons are implemented through the `ISpLexicon` interface, SAPI will only call `GetPronunciations` and `GetWords` on

the private lexicon. Implementing the AddPronunciation or RemovePronunciation methods to populate lexicons can modify private lexicons.

If you cache the pronunciations from user or application lexicons, you need to call GetGeneration periodically to determine if the pronunciation has been modified. For optimum efficiency, an engine should maintain synchronization with the user and application lexicons. The SAPI 5 compliance test can verify an engine's ability to detect changes in the user or application lexicons. For more information, please see the [Compliance Tests White Paper](#).

When the call to GetGeneration returns a larger generation number than the previous call, the engine should call GetGenerationChange or GetWords to update the cache.

Private lexicons can be added to the ISpContainerLexicon interface, or engine developers can elect to create their own method of implementing a private lexicon. However, to ensure consistent performance among all applications, engines should always use the pronunciations from the user and application lexicons.

[Back to top](#)



ISpContainerLexicon

The container lexicon object automatically loads the user lexicon and all available application lexicons when created. This allows an application and the engine to quickly access all the additional lexicon information present on the system.

ISpContainerLexicon inherits from [ISpLexicon](#).

Implemented By

- [SpLexicon](#)

Methods in Vtable Order

ISpContainerLexicon Methods	Description
AddLexicon	Adds a lexicon and its type to the lexicon stack.



ISpContainerLexicon::AddLexicon

ISpContainerLexicon::AddLexicon adds a lexicon and its type to the lexicon stack. Mainly used by engines to add private lexicons to their instance of the container lexicon for consistency of lexicon access.

```
HRESULT AddLexicon(  
    ISpLexicon *pAddLexicon,  
    DWORD dwFlags  
);
```

Parameters

pAddLexicon

[in] Pointer to the lexicon to be added.

dwFlags

[in] flags of type [SPLEXICONTYPE](#) indicating the lexicon type. Should use exactly one of the types from eLEXTYPE_PRIVATE1 through eLEXTYPE_PRIVATE20.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Either <i>dwFlag</i> is invalid or bad, or the lexicon could not be added.
SPERR_ALREADY_INITIALIZED	Attempted to add either the user or application.
E_POINTER	<i>pAddLexicon</i> is invalid or bad.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.

Remarks

For an application to create a new application lexicon, calling `AddLexicon` for the new lexicon on the application's instance of the container lexicon will not update the engine's instance of the container lexicon. The correct way to update an instance of the container lexicon is to release it and recreate the object. At this point it will re-enumerate all available application lexicons. To guarantee an update of the engine's instance of the container lexicon, the engine must be released and recreated, at which point it will recreate its instance of the container lexicon.



ISpLexicon

The ISpLexicon interface is used to access the lexicons, which contain information about words that can be recognized or spoken. For more information, please see the [ISpLexicon Overview](#).

Implemented By

- [SpLexicon](#)
- [SpCompressedLexicon](#)
- [SpUncompressedLexicon](#)

Methods in Vtable Order

ISpLexicon Methods	Description
GetPronunciations	Gets pronunciations and parts of speech for a word.
AddPronunciation	Adds pronunciation and parts of speech of a word to the user lexicon.
RemovePronunciation	Removes a word from the user lexicon.
GetGeneration	Passes back the generation ID for a word.
GetGenerationChange	Passes back a list of words which have changed between the current and a specified generation.
GetWords	Gets a list of all words in the lexicon.



ISpLexicon::GetPronunciations

ISpLexicon::GetPronunciations gets pronunciations and parts of speech for a word.

```
HRESULT GetPronunciations(  
    const WCHAR                *pszWord,  
    LANGID                     LangID,  
    DWORD                       dwFlags,  
    SPWORDPRONUNCIATIONLIST *pWordPronunciationList  
);
```

Parameters

pszWord

[in] Pointer to a null-terminated text string as a search keyword. Length must be equal to less than [SP_MAX_WORD_LENGTH](#).

LangID

[in] The language ID of the word. May be zero to indicate that the word can be of any LANGID.

dwFlags

[in] Bitwise flags of type [SPLEXICONTYPE](#) indicating that the lexicons searched for this word.

pWordPronunciationList

[in, out] Pointer to [SPWORDPRONUNCIATIONLIST](#) structure in which the pronunciations and parts of speech are returned.

Return values

Value	Description

S_OK	Function complete successfully.
SP_WORD_EXISTS_WITHOUT_PRONUNCIATION	The word exists but does not have a pronunciation.
E_POINTER	<i>pWordPronunciation</i> is not a valid write pointer.
E_INVALIDARG	At least one of the parameters is invalid.
E_OUTOFMEMORY	Exceeded available memory.
SPERR_UNINITIALIZED	The interface has not been initialized.
SPERR_NOT_IN_LEX	Word is not found in the lexicon.
FAILED(hr)	Appropriate error message.

Example

The following example is a code fragment demonstrating the use of `GetPronunciations`.

```

SPWORDPRONUNCIATIONLIST spwordpronlist;
memset(&spwordpronlist, 0, sizeof(spwordpronlist));

hr = pISpLexicon->GetPronunciations(L"resume", 409, &spwordpronlist);
//test for results
if( !SUCCEEDED(hr)) return;

for (
    SPWORDPRONUNCIATION pwordpron = spwordpronlist->pFirstWordPron;
    wordpron != NULL;
    wordpron = pwordpron->pNextWordPron
)
{

```

```
        DoSomethingWith(pwordpron->ePartOfSpeech, pwordp  
    }  
  
    //free all the buffers  
    CoTaskMemFree(spwordpronlist.pvBuffer);
```



ISpLexicon::AddPronunciation

ISpLexicon::AddPronunciation adds word pronunciations and parts of speech (POS) to the user lexicon. .

```
HRESULT AddPronunciation(  
    const WCHAR          *pszWord,  
    LANGID               LangID,  
    SPPARTOFSPEECH       ePartOfSpeech,  
    const SPPHONEID      *pszPronunciation  
);
```

Parameters

pszWord

[in] The word to add.

LangID

[in] The language ID of the word. The speech user default will be used if LANGID is omitted. Length must be equal to or less than SP_MAX_WORD_LENGTH.

ePartOfSpeech

[in] The part of speech of type SPPARTOFSPEECH.

pszPronunciation

[in] Null-terminated pronunciation of the word in the NUM phone set. Multiple pronunciations may be added for a single word. The length must be equal to or less than SP_MAX_PRON_LENGTH. *pszPronunciation* may be NULL.

Return values

Value	Description

S_OK	Function completed successfully.
E_INVALIDARG	At least one of the parameters is not valid.
SP_ALREADY_IN_LEX	The same pronunciation of the word already exists in the user lexicon.
SPERR_APPLEX_READ_ONLY	Cannot add a word to application lexicon.
SPERR_UNINITIALIZED	The interface has not been initialized.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.

Remarks

See the documentation on [ISpPhoneConverter](#) for more information on phone sets.

SAPI will not modify the word if spelling, pronunciation, and POS are the same as an existing entry in the user lexicon. A word can be added without pronunciation by passing in NULL as the *pszPronunciation*

Example

The following is an example of AddPronunciation.

```

HRESULT hr;
CComPtr<ISpLexicon> cpLexicon;
hr = cpLexicon.CoCreateInstance(CLSID_SpLexicon);

// 0x409 for English
LANGID langidUS = MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US);
CComPtr cpPhoneConv;
SPPHONEID wszId[SP_MAX_PRON_LENGTH];
if(SUCCEEDED(hr))
{
    hr = SpCreatePhoneConverter(langidUS, NULL, NULL, &cpPhoneConv);
}

```

```
}
if(SUCCEEDED(hr))
{
    hr = cpPhoneConv->PhoneToId(L"r eh d", wszId);
}
if(SUCCEEDED(hr))
{
    hr = cpLexicon->AddPronunciation(L"red", langidUS, SI
}
}
```



ISpLexicon::RemovePronunciation

ISpLexicon::RemovePronunciation removes a word and all its pronunciations from a user lexicon.

```
HRESULT RemovePronunciation(  
    const WCHAR      *pszWord,  
    LANGID           LangID,  
    SPPARTOFSPEECH   ePartOfSpeech,  
    void             *pvReserved  
);
```

Parameters

pszWord

[in] The word to remove.

LangID

[in] The language ID of the word. The speech user default will be used if LangID is omitted.

ePartOfSpeech

[in] The part of speech of type [SPPARTOFSPEECH](#).

pvReserved

[in] Reserved variable. This is required to be NULL.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One of the parameters is not valid.
E_OUTOFMEMORY	Exceeded available memory.

SPERR_NOT_IN_LEX	Word is not found in the lexicon.
SPERR_APPLEX_READ_ONLY	Cannot remove a word from application lexicon.
SPERR_UNINITIALIZED	Interface not initialized.
FAILED(hr)	Appropriate error message.



ISpLexicon::GetGeneration

ISpLexicon::GetGeneration passes back the generation ID for a word.

Passes back the current generation ID of the user lexicon. It is used to detect the changes in the user lexicon because each change in the user lexicon (add/remove a word or install/uninstall an application lexicon) will increment the generation ID.

```
HRESULT GetGeneration(  
    DWORD    *pdwGeneration  
);
```

Parameters

pdwGeneration

The generation ID. This is a relative count of how many times the custom lexicons have changed.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pdwGeneration</i> is not a valid write pointer.
SPERR_UNINITIALIZED	Interface is not initialized.
FAILED(hr)	Appropriate error message.

Remarks

[ISpLexicon::GetGenerationChange](#) and `ISpLexicon::GetGeneration` can be used when an application wants to determine what it has been done to the lexicon over a given period of time. That is, it can back out of changes it has

made due to a user cancel. To do this before it begins modifying the lexicon, the application would call `ISpLexicon::GetGeneration` and store the generation ID. Later, when the application wants to see what words in the lexicon it has modified, it would call `ISpLexicon::GetGenerationChanges` with the stored ID. This can only be done for small changes, as `SPERR_LEX_VERY_OUT_OF_SYNC` will be returned once sufficient changes have been made.



ISpLexicon::GetGenerationChange

ISpLexicon::GetGenerationChange passes back a list of words which have changed between the current and a specified generation.

```
HRESULT GetGenerationChange(  
    DWORD          dwFlags,  
    DWORD          *pdwGeneration,  
    SPWORDLIST    *pWordList  
);
```

Parameters

dwFlags

[in] The lexicon category of type [SPLEXICONTYPE](#). Currently it must be zero for the [SpLexicon](#) (container lexicon) object, and must be the correct flag for the type of [SpUnCompressedLexicon](#) object (either `eLEXTYPE_USER` or `eLEXTYPE_APP`).

pdwGeneration

[in, out] The generation ID of client when passed in. The current generation ID is passed back on successful completion of the call.

pWordList

[in, out] The buffer containing the word list and its related information. This must be initialized (memset to zero) before first use. If *pWordList* is successfully returned, `CoTaskMemFree` must be used to free the list (`pWordList->pvBuffer`) when no longer needed.

Return values

Value	Description
S_OK	Function completed successfully.
SP_LEX_NOTHING_TO_SYNC	Nothing changed since the passed in generation ID.
SPERR_LEX_VERY_OUT_OF_SYNC	There are too many changes since the passed in generation ID, so that a change history is not available. It could also be returned after installation/uninstallation of an application lexicon. Use ISpLexicon::GetWords if GetGenerationChange returns SPERR_LEX_VERY_OUT_OF_SYNC to regenerate an entire list of words based on the current generation.
E_POINTER	<i>pdwGeneration</i> or <i>pWordList</i> is not a valid write pointer.
E_INVALIDARG	<i>dwFlags</i> is invalid.
SPERR_UNINITIALIZED	Interface has not been initialized.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.

Remarks

An application can determine what has been done to a lexicon over a given period of time using [ISpLexicon::GetGenerationChange](#) and [ISpLexicon::GetGeneration](#). That is, it can back out of changes it has made due to a user cancel. To do this, before it starts modifying the lexicon, the application would call [ISpLexicon::GetGeneration](#) and store the generation ID. Later, when the application wants to see what words in the lexicon it

has modified, it would call `ISpLexicon::GetGenerationChanges` with the stored ID. This can only be done for small changes because, past a certain point, `SPERR_LEX_VERY_OUT_OF_SYNC` will be returned and the change history will not be available from the original generation.

Example

The following is an example of `GetGenerationChange`.

```
for (;;)
{
    hr = pISpLexicon->GetGenerationChange(eLEXTYPE_USER,

    // If, for example, a new application lexicon was ad
    // to rebuild from scratch.
    if (hr == SPERR_LEX_VERY_OUT_OF_SYNC)
    {
        Rebuild(); // Call GetWords
    }
    else if (FAILED(hr))
    {
        DealWithOtherErrors();
    }
    else
    {
        // Loop thru the changed words, and their new pro
        for (SPWORD *pword = spwordlist.pFirstWord;
            pword != NULL;
            pword = pword->pNextWord)
        {
            for (SPWORDPRON pwordpron = pword->pFirstWord;
                pwordpron != NULL;
                pwordpron = pwordpron->pNextWordPron)
            {
                if(pword->eWordType == eWORDTYPE_ADDED)
                {
                    AddPronunciationToEngineDataStructure(
                        pword->pszWord,
                        pwordpron->ePartOfSpeech,
                        pwordpron->pszPronIPA);
                }
            }
        }
    }
}
```

```
    }  
    else // pword->eWordType == eWORDTYPE_DEI  
    {  
        RemovePronunciationFromEngineDataStri  
        pword->pszWord,  
        pwordpron->ePartOfSpeech,  
        pwordpron->pszPronIPA);  
    }  
} } } } }  
}
```



ISpLexicon::GetWords

ISpLexicon::GetWords gets a list of all words in the lexicon.

```
HRESULT GetWords(  
    DWORD          dwFlags,  
    DWORD          *pdwGeneration,  
    DWORD          *pdwCookie,  
    SPWORDLIST    *pWordList  
);
```

Parameters

dwFlags

[in] Bitwise flags of type [SPLEXICONTYPE](#) from which words are to be retrieved.

pdwGeneration

[out] The current generation ID of the custom lexicon.

pdwCookie

[in, out] Cookie passed back by this call. It should subsequently be passed back in to get more data. If the call returns S_FALSE, data is remaining and GetWords should be called again. The initial value of the cookie passed in must be zero or *pdwCookie* will be a NULL pointer. NULL *pdwCookie* indicates the method should return all words contained in the lexicon at once. If it cannot, SP_LEX_REQUIRES_COOKIE is returned instead.

pWordList

[in, out] The buffer containing the word list and its related information. If *pWordList* is successfully returned, CoTaskMemFree must be used to free the list (pWordList-

>pvBuffer) when no longer needed.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	Additional words are left in the lexicon(s) to process.
SPERR_LEX_REQUIRES_COOKIE	A complete list of words cannot be returned at once from the container lexicon. <i>pdwCookie</i> must not be NULL.
E_POINTER	At least one of <i>pdwGeneration</i> , <i>pdwCookie</i> , <i>pWordList</i> is not valid. Alternatively, the block of memory is too small or is not writable.
E_INVALIDARG	At least one of the parameters is not valid.
E_OUTOFMEMORY	Exceeded available memory.
SPERR_UNINITIALIZED	Interface not initialized.
FAILED(hr)	Appropriate error message.

Remarks

This method is called repeatedly with the cookie (set to zero before the first time) until S_OK is returned. S_FALSE is returned indicating additional information is left. Optionally, the cookie pointer passed in may be NULL, which specifies the application wants all of the words at once. However, the lexicon is not required to support this and may return the error SP_LEX_REQUIRES_COOKIE. The [SpLexicon](#) object (container lexicon) requires a cookie currently.

Example

The following is an example of using GetWords.

```
SPWORDLIST spwordlist;
memset(&spwordlist, 0, sizeof(spwordlist));
dwCookie = 0;

while (SUCCEEDED(hr = pISpLexicon->GetWords(eLEXTYPE_USEI
{
    for (SPWORD *pword = spwordlist.pFirstWord;
        pword != NULL;
        pword = pword->pNextWord)
    {
        for (SPWORDPRONUNCIATION *pwordpron = pword->pFi
            pwordpron != NULL;
            pwordpron = pwordpron->pNextWordPronunciatio
        {
            DoSomethingWith(pwordpron->ePartOfSpeech, pwr
        }
    }

    if (hr == S_OK)
        break; // nothing more to retrieve
}

//free all the buffers
CoTaskMemFree(spwordlist.pvBuffer);
```



ISpPhoneConverter

The ISpPhoneConverter interface enables the client to convert from the SAPI character phoneset to the Id phoneset.

When to Use

Call methods of the ISpPhoneConverter interface to convert between character and NUM phonesets.

Implemented By

- [SpPhoneConverter](#)

Methods in Vtable Order

ISpPhoneConverter Methods	Description
ISpObjectWithToken interface	Inherits from ISpObjectWithToken and those methods are accessible from an ISpPhoneConverter object.
PhoneTold	Converts a character phoneme string to an ID code string.
IdToPhone	Converts a null-terminated ID code array to the SAPI character format.



ISpPhoneConverter::PhoneToId

ISpPhoneConverter::PhoneToId converts a character phoneme string to an ID code string.

The [English](#) and [Chinese](#) phoneme sets require the phonemes to be space separated. The [Japanese](#) phoneme set requires the phoneme character form to be continuous. See the individual entries for more details about the character sets.

```
HRESULT PhoneToId(  
    const WCHAR *pszPhone,  
    SPPHONEID *pId  
);
```

Parameters

pszPhone

[in] Address of a null-terminated string that contains the phoneme string information.

pId

[out] Address of the SPPHONEID array that receives the phoneme identifiers. On return the array will be a null-terminated list of SPPHONEIDs.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid, or phoneme not found.
SPERR_UNINITIALIZED	Interface not initialized.
E_FAIL	<i>pId</i> is invalid or bad.
E_FAILED(hr)	Appropriate error message.



ISpPhoneConverter::IdToPhone

ISpPhoneConverter::IdToPhone converts a null-terminated ID code array to the SAPI character format.

The [English](#) and [Chinese](#) phoneme character sets require the phonemes to be space separated. The [Japanese](#) phoneme set requires the phoneme character form to be continuous. See the individual entries for more details about the character sets.

```
HRESULT IdToPhone(  
    const SPPHONEID *pId,  
    WCHAR *pszPhone  
);
```

Parameters

pId

[in] Address of the null-terminated array of SPPHONEIDs that contains the phoneme identifiers.

pszPhone

[out] Address of a null-terminated string that receives the phoneme string information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid. Alternatively, <i>pId</i> exceeds SP_MAX_PRON_LENGTH .
E_POINTER	<i>pszPhone</i> or <i>pId</i> is invalid or bad.
SPERR_UNINITIALIZED	Interface not initialized.
E_FAIL	Member exceeds available size.

FAILED(hr)

Appropriate error message.



Resource interfaces

The following section covers:

- [Object Tokens Overview](#)
- [ISpDataKey](#)
- [ISpRegDataKey](#)
- [ISpObjectTokenInit](#)
- [ISpObjectTokenCategory](#)
- [ISpObjectToken](#)
- [IEnumSpObjectTokens](#)
- [ISpObjectWithToken](#)
- [ISpResourceManager](#)
- [ISpTask](#)



Object Tokens Overview

This document is a high level summary of Object Tokens, Categories and Registries used in SAPI. It is intended to assist developers of speech-enabled applications in understanding the concepts of tokens and categories and using them.

Token

A token is an object representing a resource. It provides an application an easy mechanism with which to inspect the various attributes of a resource without instantiating it. SAPI stores information about tokens in the registry. A token is represented in the registry by a key, and the key's underlying keys and values. For example, MSMary is a token, it represents the Microsoft Mary Voice, and its TokenId is

HKEY_LOCAL_MACHINE\Software\Microsoft\Speech\Voices'

Generally, a token contains a language-independent name, a CLSID used to instantiate the object from token, and a set of attributes. It may also contain a CLSID for certain types of user interfaces (UIs), and a set of files from which SAPI returns the paths to all the associated files for the token.

Categories

An Object Token Category is a class of tokens. It is represented in the registry by a key containing one or more token keys under it. Categories contain a single key called Tokens, and the keys for the tokens that belong to that category under it, or keys for token enumerators. See [Object Tokens and Registry Settings](#) for detailed descriptions of token enumerators. For example, Voice is a Category, and it contains the Microsoft Mary, Microsoft Sam, Microsoft Mike voices. Its CategoryId is

HKEY_LOCAL_MACHINE\Software\Microsoft\Speech\Voices
SAPI categories that are located under

HKEY_LOCAL_MACHINE\Software\Microsoft\Speech are
Voices, Recognizers, AppLexicons, AudioInput, AudioOutput and
PhoneConverter. Another category, Recoprofiles, is located
under **HKEY_CURRENT_USER\Software\Microsoft\Speech**.

Using Tokens and Categories

To enumerate tokens, the application can use either the helper function `SpEnumTokens` or call `ISpObjectTokenCategory::EnumTokens`. Following is an example of a call to `EnumTokens`.

```
CComPtr<ISpObjectTokenCategory> cpCategory;  
CComPtr<IEnumSpObjectTokens> cpEnum;  
  
HRESULT hr = cpCategory.CoCreateInstance(CLSID_SpObjectToken  
//check hr  
hr = cpCategory->SetId(SPCAT_VOICES, false);  
//check hr  
hr = cpCategory->EnumTokens(SPCAT_VOICES, L"Gender=Female",  
//check hr
```

This sample code requests all female voices. Adult voices will be listed at the beginning of the enumerator. Using the helper function, the code is equal to the following.

```
CComPtr<IEnumSpObjectTokens> cpEnum;  
  
hr = SpEnumTokens(SPCAT_VOICES, L"Gender=Female", L"Age=Adult");  
  
After getting enumerator, use methods in IEnumSpObjectTokens  
  
CComPtr<ISpObjectToken> cpToken;  
  
hr = cpEnum->Next(1, &cpToken;, NULL);
```

Other helper functions that can simplify the steps, for example: `SpGetDefaultTokenFromCategoryId`, `SpFindBestToken`. Please see the [Helper Functions](#) document for more detail descriptions.

Engine developers also need to associate files with tokens and be able to create new tokens. A token can query for all the files under the Files key using `ISpObjectToken::GetStorageFileName`. SAPI does not store full paths, for example:

%1c%\Microsoft\Speech\Files\MSASR\SP_63EB435D95104977BDB68E3I

Use `ISpObjectToken::RemoveStorageFileName` to remove the files.

Keys under a token can be inspected. Following is a sample code to add a Special attribute to the default Voices token:

```
CComPtr<ISpObjectToken> cpToken;
CComPtr<ISpDataKey> cpKey;

hr = SpGetDefaultTokenFromCategoryId(SPCAT_VOICES, &cpToken;
//check hr
hr = cpToken->OpenKey(L"Attributes", &cpKey;);
//check hr
hr = cpKey->SetStringValue(L"Special", L"fun");
//check hr
WCHAR *psz = NULL;
hr = cpKey->GetStringValue(L"Special", &psz;);
//check hr
::CoTaskMemFree(psz);
```

To create a key under default voice token, call `CreateKey` from the Token:

```
CComPtr<ISpObjectToken> cpToken;
CComPtr<ISpDataKey> cpKey;

hr = SpGetDefaultTokenFromCategoryId(SPCAT_VOICES, &cpToken;
//check hr
hr = cpToken->CreateKey(L"CreatedKey", &cpKey;);
//check hr
hr = cpKey->SetStringValue(L"Attri", L"data");
//check hr
```

Detailed information can be found in [Object Tokens and Registry Settings](#) and individual API documents.



ISpDataKey

The ISpDataKey interface provides a mechanism for storing and retrieving string and other data. ISpDataKey is used in conjunction with object tokens, which implement [ISpObjectToken](#), which inherits from ISpDataKey. For example, data can be stored in an object token representing a recognizer or TTS engine using this interface.

Implemented By

- [SpObjectToken](#) object. This is the standard class used for all existing SAPI object tokens. The data for each object token is stored in the registry.
- [SpDataKey](#) object. This class stores the data associated with the data key in the registry.
- Applications or engines can implement this interface directly if they wish to provide a custom data key implementation.

Methods in Vtable Order

ISpDataKey Methods	Description
SetData	Sets the binary data for a token.
GetData	Retrieves the binary data for a token.
SetStringValue	Sets the string value information for a specified token.
GetStringValue	Retrieves the string value information from a specified token.
SetDWORD	Sets the value information for a specified token.
GetDWORD	Retrieves the value information from a specified token.
OpenKey	Opens a specified token subkey.

<u>CreateKey</u>	Creates a new token subkey.
<u>DeleteKey</u>	Deletes a specified token key and all its descendants.
<u>DeleteValue</u>	Deletes a named value from the specified token.
<u>EnumKeys</u>	Enumerates the subkeys of the specified token.
<u>EnumValues</u>	Enumerates the values of the specified token.



ISpDataKey::SetData

ISpDataKey::SetData sets the binary data for a token.

```
HRESULT SetData(  
    const WCHAR    *pszValueName,  
    ULONG          cbData,  
    const BYTE     *pData  
);
```

Parameters

pszValueName

[in] Address of a null-terminated string that contains the registry key value name.

cbData

[in] Size of the *pData* parameter.

pData

[out] Pointer to the buffer containing the information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Either <i>pszValueName</i> or <i>pData</i> is an invalid or bad pointer.
FAILED(hr)	Appropriate error message.



ISpDataKey::GetData

ISpDataKey::GetData retrieves the binary data for a token.

```
HRESULT GetData(  
    const WCHAR    *pszValueName,  
    ULONG          *pcbData,  
    BYTE           *pData  
);
```

Parameters

pszValueName

Address of a null-terminated string containing the name of the registry key from which to retrieve the registry key value.

pcbData

[in] Size of the *pData* parameter.

pData

[out] Pointer to the buffer receiving the information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszValueName</i> is invalid or bad.
E_POINTER	Either <i>pcbData</i> or <i>pData</i> is an invalid or bad pointer.
SPERR_NOT_FOUND	Token key not found.
FAILED(hr)	Appropriate error message.



ISpDataKey::SetStringValue

ISpDataKey::SetStringValue sets the string value information for a specified token.

```
HRESULT SetStringValue(  
    const WCHAR    *pszValueName,  
    const WCHAR    *pszValue  
);
```

Parameters

pszValueName

Address of the null-terminated string specifying the name of the string value. If NULL, the default value of the token is used.

pszValue

Address of a null-terminated string that contains the string value to be set for the specified key.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Either <i>pszValueName</i> or <i>pszValue</i> is invalid or bad.
FAILED(hr)	Appropriate error message.



ISpDataKey::GetStringValue

ISpDataKey::GetStringValue retrieves the string value information from a specified token.

```
HRESULT GetStringValue(  
    const WCHAR    *pszValueName,  
    WCHAR          **ppszValue  
);
```

Parameters

pszValueName

Address of a null-terminated string that specifies the name of the registry key. If NULL, the default value of the token is read.

ppszValue

Address of a pointer to a null-terminated string that receives the string value for the specified key.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszValueName</i> is invalid or bad.
E_POINTER	<i>ppszValue</i> is invalid or bad.
SPERR_NOT_FOUND	Registry file not found.
FAILED(hr)	Appropriate error message.



ISpDataKey::SetDWORD

ISpDataKey::SetDWORD sets the value information for a specified token.

```
HRESULT SetDWORD(  
    const WCHAR    *pszKeyName,  
    DWORD          dwValue  
);
```

Parameters

pszKeyName

Address of a null-terminated string that contains the attribute name.

dwValue

The data buffer containing the attribute key value.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszKeyName</i> is invalid or bad.
FAILED(hr)	Appropriate error message.



ISpDataKey::GetDWORD

ISpDataKey::GetDWORD retrieves the value information from a specified token.

```
HRESULT GetDWORD(  
    const WCHAR    *pszKeyName,  
    DWORD          *pdwValue  
);
```

Parameters

pszKeyName

[in] Address of a null-terminated string containing the token name.

pdwValue

[out] Address of the destination data buffer receiving the token key value.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszKeyName</i> is invalid or bad.
E_POINTER	<i>pdwValue</i> is invalid or bad.
SPERR_NOT_FOUND	Registry key not found.
FAILED(hr)	Appropriate error message.



ISpDataKey::OpenKey

ISpDataKey::OpenKey opens a specified token subkey. Passes back a new object that supports ISpDataKey for the specified subkey.

```
HRESULT OpenKey(  
    const WCHAR *pszSubKeyName,  
    ISpDataKey **ppSubKey  
);
```

Parameters

pszSubKeyName

Address of a null-terminated string specifying the name of the key to open.

ppSubKey

Address of a pointer to an ISpDataKey interface.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszSubKeyName</i> is invalid or bad.
E_POINTER	<i>ppSubKey</i> is invalid or bad.
SPERR_NOT_FOUND	Registry key not found.
FAILED(hr)	Appropriate error message.



ISpDataKey::CreateKey

ISpDataKey::CreateKey creates a new token subkey. Returns a new object which supports ISpDataKey for the specified subkey. If the key already exists, the function will open the existing key instead of overwriting it.

```
HRESULT CreateKey(  
    const WCHAR    *pszSubKeyName,  
    ISpDataKey    **ppSubKey  
);
```

Parameters

pszSubKeyName

Address of a null-terminated string specifying the name of the key to create.

ppSubKey

Address of a pointer to an ISpDataKey interface.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Either <i>pszSubKeyName</i> or <i>ppKey</i> is invalid or bad.
FAILED(hr)	Appropriate error message.



ISpDataKey::DeleteKey

ISpDataKey::DeleteKey deletes a specified token key and all its descendants.

```
HRESULT DeleteKey(  
    const WCHAR    *pszSubKeyName  
);
```

Parameters

pszSubKeyName

Address of a null-terminated string specifying the name of the key or subkey to delete.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszSubKeyName</i> is invalid or bad.
SPERR_NOT_FOUND	Token key not found.
FAILED(hr)	Appropriate error message.



ISpDataKey::DeleteValue

ISpDataKey::DeleteValue deletes a named value from the specified token.

```
HRESULT DeleteValue(  
    const WCHAR    *pszValueName  
);
```

Parameters

pszValueName

Address of a null-terminated string specifying the value name to be deleted.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszValueName</i> is invalid or bad.
SPERR_NOT_FOUND	Registry key not found.
FAILED(hr)	Appropriate error message.



ISpDataKey::EnumKeys

ISpDataKey::EnumKeys enumerates the subkeys of the specified token.

```
HRESULT EnumKeys(  
    ULONG      Index,  
    WCHAR     **ppszSubKeyName  
);
```

Parameters

Index

[in] Value indicating which token in the enumeration sequence to locate.

ppszSubKeyName

[out] Address of a pointer to a null-terminated string that receives the enumerated key name. This must be freed with `CoMemTaskFree()` when no longer required.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>ppszSubKeyName</i> is invalid or bad.
SPERR_NOT_FOUND	Registry key not found.
E_OUTOFMEMORY	Not enough memory to allocate string.
SPERR_NO_MORE_ITEMS	No items could be accessed.
FAILED(hr)	Appropriate error message.



ISpDataKey::EnumValues

ISpDataKey::EnumValues enumerates the values of the specified token.

```
HRESULT EnumValues(  
    ULONG      Index,  
    WCHAR     **ppszValueName  
);
```

Parameters

Index

[in] Value indicating which token in the enumeration sequence to locate.

ppszValueName

Address of a pointer to a null-terminated string that receives the enumerated registry key values. This must be freed with CoMemTaskFree() when no longer required.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>ppszValueName</i> is invalid or bad.
SPERR_NOT_FOUND	Registry key not found.
E_OUTOFMEMORY	Not enough memory to allocate string.
SPERR_NO_MORE_ITEMS	No items could be accessed.
FAILED(hr)	Appropriate error message.



ISpRegDataKey

This interface is used to create a new data key using a specific key in the registry for storage. The ISpRegDataKey inherits from [ISpDataKey](#).

The ISpRegDataKey inherits from [ISpDataKey](#).

Applications will not normally need to use or implement this interface.

Implemented By

- [SpDataKey](#). This class stores all the data for a data key in the registry.

Methods in Vtable Order

ISpRegDataKey Methods	Description
SetKey	Sets the hive registry key (HKEY) to use for subsequent token operations.



ISpRegDataKey::SetKey

ISpRegDataKey::SetKey sets the hive registry key (HKEY) to use for subsequent token operations.

```
HRESULT SetKey(  
    HKEY    hkey,  
    BOOL    fReadOnly  
);
```

Parameters

hkey

[in] The registry key to use.

fReadOnly

[in] Boolean flag setting the keys to read/write status. If TRUE, the registry is read only; FALSE sets it to read and write.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_ALREADY_INITIALIZED	Interface is already initialized.

Example

The following code snippet adds, tests and deletes a superfluous key from the speech registry.

```
HRESULT hr;
```

```
ComPtr<ISpRegDataKey> cpSpRegDataKey;  
ComPtr<ISpDataKey> cpSpCreatedDataKey;
```

```
CComPtr<ISpDataKey> cpSpDataKey;
CComPtr<ISpObjectTokenCategory> cpSpCategory;
HKEY hkey;

//create a bogus key under Voices
hr = g_Unicode.RegCreateKeyEx(HKEY_LOCAL_MACHINE, L"SOFTWARE\
//Check error

hr = cpSpRegDataKey.CoCreateInstance(CLSID_SpDataKey);
//Check error

hr = cpSpRegDataKey->SetKey(hkey, false);
//Check error

hkey = NULL;
//Do not need to do RegCloseKey on this hkey, the handle get:

hr = cpSpRegDataKey->QueryInterface(&cpSpCreatedDataKey);
//Check error

//delete this bogus key
hr = SpGetCategoryFromId(SPCAT_VOICES, &cpSpCategory);
//Check error

hr = cpSpCategory->GetDataKey(SPDKL_LocalMachine, &cpSpDataKey);
//Check error

hr = cpSpDataKey->DeleteKey(L"bogus");
//Check error
```



ISpObjectTokenInit

This interface inherits from [ISpObjectToken](#).

Associated Class IDs

The following class IDs (CLSID) may be used with this interface. A complete CLSID listing for all interfaces is in the [Class IDs](#) section.

CLSID_SpObjectToken

Methods in Vtable Order

ISpObjectTokenInit Methods	Description
InitFromDataKey	Initializes a token to use a specified datakey.



ISpObjectTokenInit::InitFromDataKey

ISpObjectTokenInit::SetObjectToken initializes a token to use a specified datakey.

```
HRESULT InitFromDataKey(  
    const WCHAR    *pszCategoryId,  
    const WCHAR    *pszTokenId,  
    ISpDataKey    *pDataKey  
);
```

Parameters

pszCategoryId

[in] The null-terminated string name of the categoryId from which to create the token.

pszTokenId

[in] The null-terminated string name of the TokenId.

pDataKey

[in] Address of an ISpDataKey interface that specifies the system registry key from which to create the token.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	At least one of the parameters is invalid or bad.
SPERR_ALREADY_INITIALIZED	Token is already initialized.
SPERR_TOKEN_DELETED	Key has been deleted.
E_OUTOFMEMORY	Exceeded available memory.

Remarks

Dynamic token enumerators can use this to create tokens under their token enumerator's token. Once created, this enables [ISpDataKey::CreateKey](#) to make a new data key, create a new object token, and then use `InitFromDataKey`.



ISpObjectTokenCategory

Each object token category represents a collection of similar tokens, such as voices, recognizers, and audio input devices. Categories may be created or manipulated with helper functions or methods from this interface.

Each category has a Category ID that is unique and identifies only one type of object token. The set of available categories are listed in [Token Category IDs](#). Category IDs are always null-terminated strings.

An application can create an [SpObjectTokenCategory](#) object, which implements this interface. Then it calls [SetId](#) in order to set the Category ID that this object is using. The application can then enumerate the object tokens associated with this category using [EnumTokens](#). Applications can also locate and change the default object token for a category with the methods [SetDefaultTokenId](#) and [GetDefaultTokenId](#).

ISpObjectTokenCategory inherits from [ISpDataKey](#).

Implemented By

- [SpObjectTokenCategory](#) object. This is the standard class used for categories in SAPI. The category and the list of associated tokens are stored in the registry.

How Created

- Applications will normally create object token categories by directly creating the SpObjectTokenCategory class.
- If the application is using the category to find an associated token that matches certain attributes, it is often easier to use the helper functions [SpEnumTokens](#) or [SpFindBestToken](#).

Methods in Vtable Order

ISpObjectToken Methods	Description
<u>SetId</u>	Sets the category ID.
<u>GetId</u>	Retrieves the token ID.
<u>GetDataKey</u>	Gets the data key associated with a specific location.
<u>EnumTokens</u>	Enumerates the tokens for the category.
<u>SetDefaultTokenId</u>	Sets a specific TokenId as the default for the category.
<u>GetDefaultTokenId</u>	Retrieves the default TokenId for the category.



ISpObjectTokenCategory::SetId

ISpObjectTokenCategory::SetId sets the category ID.

This method may be called only once. If called more than once, SPERR_ALREADY_INITIALIZED will return.

```
HRESULT SetId(  
    const WCHAR *pszCategoryId,  
    BOOL fCreateIfNotExist  
);
```

Parameters

pszCategoryId

[in] The null-terminated string name of category to set. SAPI-defined categories are listed in [Token Category IDs](#).

fCreateIfNotExist

[in] Indicates creating the category if one is not already present. TRUE creates the entry. FALSE does not.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_ALREADY_INITIALIZED	Category interface is already initialized.
E_INVALIDARG	<i>pszCategoryId</i> is invalid or bad.
FAILED(hr)	Appropriate error message.



ISpObjectTokenCategory::GetId

ISpObjectTokenCategory::GetId retrieves the category ID.

```
HRESULT GetId(  
    WCHAR    **ppszCoMemCategoryId  
);
```

Parameters

ppszCoMemCategoryId

[in] The null-terminated string name of the current category. *ppszCoMemCategoryId* must be freed with `CoMemTaskFree` when no longer required.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_UNINITIALIZED	Category interface is not initialized.
E_POINTER	<i>ppszCoMemCategoryId</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Example

The following code snippet retrieves `CategoryId` for `SPCAT_VOICES`.

```
HRESULT hr;
```

```
CCoPtr<ISpObjectTokenCategory> cpSpCategory;  
CSpCoTaskMemPtr<WCHAR> cpwszOldID;
```

```
hr = SpGetCategoryFromId(SPCAT_VOICES, &cpSpCategory);  
//Check return code
```

```
hr = cpSpCategory->GetId(&cpwszOldID);  
//Check return code
```

```
CoMemTaskFree(cpwszOldID);
```



ISpObjectTokenCategory::GetDataKey

ISpObjectTokenCategory::GetDataKey gets the data key associated with a specific location.

```
HRESULT GetDataKey(  
    SPDATAKEYLOCATION    spdk1,  
    ISpDataKey          **ppDataKey  
);
```

Parameters

spdk1

[in] The registry's top-level node to be searched.

ppDataKey

[out] The data key interface associated with the location *spdk1*.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_UNINITIALIZED	Data key interface is not initialized.
E_POINTER	<i>ppDataKey</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Example

The following code snippet retrieves the data key associated with the local computer registry for SPCAT_VOICES.

```
HRESULT hr;
```

```
CComPtr<ISpObjectTokenCategory> cpSpCategory;  
CComPtr<ISpDataKey> cpSpDataKey;
```

```
hr = SpGetCategoryFromId(SPCAT_VOICES, &cpSpCategory);  
//Check return code
```

```
hr = cpSpCategory->GetDataKey(SPDKL_LocalMachine, &cpSpData  
//Check return code
```



ISpObjectTokenCategory::EnumTokens

ISpObjectTokenCategory::EnumTokens enumerates the tokens for the category by attempting to match specified requirements. Attributes, enumerations, and searches are discussed in [Object Tokens and Registry Settings White Paper](#).

```
HRESULT EnumTokens(  
    const WCHAR          *pszReqAttribs,  
    const WCHAR          *pszOptAttribs,  
    IEnumSpObjectTokens **ppEnum  
);
```

Parameters

pszReqAttribs

[in] The null terminated string of required attributes for the token.

pszOptAttribs

[in] The null terminated string of optional attributes for the token. The order in which the tokens are listed in *ppEnum* is based on the order they match *pszOptAttribs*.

ppEnum

[out] The enumerated list of tokens found.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_UNINITIALIZED	Data key interface is not initialized.
E_POINTER	At least one of the parameters is

	invalid or bad.
FAILED(hr)	Appropriate error message.

Example

The following code snippet demonstrates getting a complete enumerated token list. Since no specific requirement is given (*pszReqAttribs* and *pszOptAttribs* are NULL), all values are returned for SPCAT_VOICES.

```
CComPtr<ISpObjectTokenCategory> cpSpCategory;  
CComPtr<IEnumSpObjectTokens> cpSpEnumTokens;  
HRESULT hr;  
  
hr = SpGetCategoryFromId(SPCAT_VOICES, &cpSpCategory  
//Check hr  
  
hr = cpSpCategory->EnumTokens(NULL, NULL, &cpSpEnumT  
//Check hr
```



ISpObjectTokenCategory::SetDefaultTokenId

ISpObjectTokenCategory::SetDefaultTokenId sets a specific token ID as the default for the category.

```
HRESULT SetDefaultTokenId(  
    const WCHAR *pszTokenId  
);
```

Parameters

pszTokenId

[in] The null-terminated string name of the token ID to be used as the default.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_UNINITIALIZED	Data key interface is not initialized.
E_INVALIDARG	<i>pszTokenId</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Remarks

The defaults are stored either directly in the category by setting the DefaultTokenID value in the category data key, or indirectly by the DefaultTokenIDLocation. Default tokens are discussed in [Object Tokens and Registry Settings White Paper](#).



ISpObjectTokenCategory::GetDefaultTokenId

ISpObjectTokenCategory::GetDefaultTokenId retrieves the default token ID for the category.

```
HRESULT GetDefaultTokenId(  
    const WCHAR    **ppszCoMemTokenId  
);
```

Parameters

ppszCoMemTokenId

[in] The null-terminated string name of the token ID used as the default. Must be released with CoMemTaskFree () when no longer needed.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_UNINITIALIZED	Data key interface is not initialized.
E_POINTER	<i>ppszCoMemTokenId</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Remarks

There is a hierarchy for returning a default token ID value. A default token has an attribute marked as DefaultTokenID. When a token is not explicitly marked as such, SAPI attempts to return the default from the user profile. If none exists there, SAPI returns a specially named token called DefaultdefaultTokenID for the category ID. Default tokens are discussed in [Object Tokens and Registry Settings White Paper](#).



ISpObjectToken

The ISpObjectToken interface handles object token entries.

An object token is an object representing a resource that is available on a computer, such as a voice, recognizer, or an audio input device. A token provides an application a simple way to inspect the various attributes of a resource without having to instantiate it. The Vendor of a Recognizer, and Gender of a Voice are examples of attributes of resources. An application can enumerate the various tokens that exist on the computer by using the [SpEnumTokens](#) helper function, or by using the [ISpObjectTokenCategory::EnumTokens](#) method to enumerate the tokens of a particular category. Applications can find the best token that matches certain attributes by using the [SpFindBestToken](#) function.

Conceptually, a token contains the following information:

- An identifier that uniquely identifies the object token.
- The language-independent name is the name that should be displayed wherever the name of the token is displayed. The implementer of the token may also choose to provide a set of language-dependent names in several languages.
- The CLSID used to instantiate the object from the token.
- A set of Attributes, which are the set of queryable values in a token. SAPI provides a mechanism to query for tokens whose attributes match certain values.

A token may also contain the following:

- If a token has user interfaces (UIs), such as the properties of a Recognizer or a wizard to customize a Voice to display, the token will also contain the CLSID for the COM object used to instantiate each type of UI.

- The set of Files from which SAPI returns the paths to all the associated files for the token.

Attributes are null-terminated strings forming a series of key-pair entries. This is usually in the form of definition relationships. For example, a token may be defined as:

```
"vendor=microsoft;language=409;someflag"
```

In this instance:

- "vendor=microsoft" means a string exists under TokenID\attributes with name *vendor* and value "microsoft";
- "language=409" means a string exists under TokenID\attributes with name *language* and value "409" (representing US English);
- "someflag" means a string exists under TokenID\attributes with name *someflag* but has no additional information. Sometimes the presence or absence of the attribute name itself is indicative.

Implemented By

- [SpObjectToken](#) object. This is the standard class used for all existing SAPI object tokens. The data for each object token is stored in the registry.
- Applications or engines can implement this interface directly to provide a custom object token implementation. For instance, this could be used to avoid storing object token data in the registry, or to provide the ability for object tokens to be downloaded from a server. In this case [IEnumSpObjectTokens](#) would also need to be implemented so that the new object tokens to be enumerated.

How Created

- Applications will normally create object tokens from an [object token enumerator](#) or helper function, rather than by directly creating them.
- Various API methods also return an object token referring to a specific type of resource, e.g.,
[ISpRecognizer::GetRecognizer](#) returns the object token associated with the current recognition engine;
[ISpRecognizer::GetRecoProfile](#) returns the object token referring to the current recognition profile; and
[ISpVoice::GetVoice](#) returns the object token referring to the current TTS engine.

ISpObjectToken inherits from [ISpDataKey](#).

Object tokens are discussed in more detail in the [Object Tokens and Registry Settings White Paper](#).

Methods in Vtable Order

ISpObjectToken Methods	Description
SetId	Sets the category identifier for object token.
GetId	Retrieves the object identifier for an object token.
GetCategory	Retrieves the category for a specified token if one is available.
CreateInstance	Creates an instance of an object.
GetStorageFileName	Retrieves the object token file name.
RemoveStorageFileName	Removes the object token file name.

<u>Remove</u>	Removes an object token.
<u>IsUISupported</u>	Determines if the UI associated with the object is supported.
<u>DisplayUI</u>	Displays the UI associated with the object.
<u>MatchesAttributes</u>	Determines if an object token supports a specified attribute.



ISpObjectToken::SetId

ISpObjectToken::SetId sets the CategoryId for object token.

This may be called only once.

```
HRESULT SetId(  
    const WCHAR *pszCategoryId,  
    const WCHAR *pszTokenId,  
    BOOL fCreateIfNotExist  
);
```

Parameters

pszCategoryId

[in] The null-terminated string name of category to set.

pszTokenId

[in] The null-terminated string name of token to set.

fCreateIfNotExist

[in] A Boolean indicating the object is to be created if not currently existing. TRUE allows the creation; FALSE does not.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_ALREADY_INITIALIZED	Category interface is already initialized.
SPERR_TOKEN_DELETED	Key has been deleted.
E_INVALIDARG	Either <i>pszCategoryId</i> and/or <i>pszTokenId</i> is invalid or bad.

FAILED(hr)

Appropriate error message.

Notes

CategoryIds appear in the fully qualified form as:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Recognizers

The only acceptable HKEYs are:

HKEY_CLASSES_ROOT,

HKEY_CURRENT_USER,

HKEY_LOCAL_MACHINE,

HKEY_CURRENT_CONFIG



ISpObjectToken::GetId

ISpObjectToken::GetId retrieves the object identifier for an object token. This identifier can be used later to recreate a token instance.

```
HRESULT GetId(  
    WCHAR    **ppszCoMemTokenId  
);
```

Parameters

ppszCoMemTokenId

Address of a pointer to a null-terminated string that receives the identifier for the token object. The caller must call `CoTaskMemFree()` to free the string pointer.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppszCoMemTokenId</i> is invalid or bad.
E_OUTOFMEMORY	Exceeded available memory.
SPERR_UNINITIALIZED	TokenId interface is not initialized.
FAILED(hr)	Appropriate error message.



ISpObjectToken::GetCategory

ISpObjectToken::GetCategory retrieves the category for a specified token if one is available.

```
HRESULT GetCategory(  
    ISpObjectTokenCategory **ppTokenCategory  
);
```

Parameters

ppTokenCategory

[out] The category interface for the token. *ppTokenCategory* must be freed when no longer required.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppTokenCategory</i> is invalid or bad.
SPERR_UNINITIALIZED	Token does not have a category.
FAILED(hr)	Appropriate error message.



ISpObjectToken::CreateInstance

ISpObjectToken::CreateInstance creates an instance of an object.

```
HRESULT CreateInstance(  
    IUnknown *pUnkOuter,  
    DWORD dwClsContext,  
    REFIID riid,  
    void **ppvObject  
);
```

Parameters

pUnkOuter

[in] If the object is being created as part of an aggregate, this is a pointer to the controlling IUnknown interface of the aggregate. Otherwise, *pUnkOuter* must be NULL.

dwClsContext

[in] Context in which the code that manages the newly created object will run. It should be one of the following values:

- CLSCTX_INPROC_SERVER
- CLSCTX_INPROC_HANDLER
- CLSCTX_LOCAL_SERVER
- CLSCTX_REMOTE_SERVER

riid

[in] Reference to the identifier of the interface used to communicate with the newly created object. If *pUnkOuter* is NULL, this parameter is frequently the IID of the initializing interface; if *pUnkOuter* is non-NULL, *riid* must be

IID_Unknown.

ppvObject

[out, iid_is(riid)] Address of pointer variable that receives the interface pointer requested in *riid*. Upon successful return, *ppvObject* contains the requested interface pointer. If the object does not support the interface specified in *riid*, the implementation must set *ppvObject* to NULL.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppvObject</i> is invalid or bad.
E_INVALIDARG	<i>pUnkOuter</i> is invalid or bad.
SPERR_UNINITIALIZED	Either the data key or the token delegator interface is not initialized.
SPERR_TOKEN_DELETED	Key has been deleted.
FAILED(hr)	Appropriate error message.

Remarks

This method is used to create the underlying object that the object token represents. This method looks at the CLSID value stored in the object token and creates a COM object from this CLSID.

For example, when this method is called on an object token from the [audio input category](#), an audio object that implements *ISpStreamFormat* will be created and returned.

This method is not used to create speech recognition or text-to-speech engines. Instead, an *SpRecognizer* or *SpVoice* object is created and the engine is then created by passing an object token to the [ISpRecognizer::SetRecognizer](#) or [ISpVoice::SetVoice](#) methods.

Example

The following code snippet creates an InProc server instance.

```
HRESULT hr;
```

```
CComPtr<ISpObjectToken> cpSpObjectToken;
```

```
CComPtr<ISpObjectWithToken> cpSpObjectWithToken;
```

```
hr = SpGetDefaultTokenFromCategoryId(SPCAT_VOICES, &cpSpObjectToken);
```

```
//Check return value
```

```
hr = cpSpObjectToken->CreateInstance( NULL, CLSCTX_INPROC_SERVER, IID_ISpObjectWithToken, (void**)&cpSpObjectWithToken);
```

```
//Check return value
```



ISpObjectToken::GetStorageFileName

ISpObjectToken::GetStorageFileName retrieves the object token file name from the registry.

```
HRESULT GetStorageFileName(  
    REFCLSID      clsidCaller,  
    const WCHAR  *pszValueName,  
    const WCHAR  *pszFileNameSpecifier,  
    ULONG        nFolder,  
    WCHAR        **ppszFilePath  
);
```

Parameters

clsidCaller

[in] Globally unique identifier (GUID) of the calling object. The registry is searched for an entry key name of *clsidCaller*, and then a corresponding "Files" subkey. If the registry entry is not present, one is created.

pszValueName

[in] The name of the attribute file for the registry entry of *clsidCaller*. This attribute stores the location of the resource file.

pszFileNameSpecifier

[in] The specifier that is either NULL or a path/file name for storage file.

If this starts with "X:\" or "\\\" it is assumed to be a full path.

Otherwise it is assumed to be relative to special folders given in the *nFolder* parameter.

If it ends with a '\\', or is NULL a unique file name will be created. The file name will be something like: "SP_7454901D23334AAF87707147726EC235.dat". "SP_" and ".dat" are the default prefix name and file extension name. The numbers in between are generated guid number to make sure the file name is unique.

If the name contains a %d the %d is replaced by a number to give a unique file name. The default file extension is .dat, the user can specify anything else.

Intermediate directories are created.

If a relative file is being used the value stored in the registry includes the *nFolder* value as %nFolder% before the rest of the path.

nFolder

[in] A CSIDL value that identifies the folder whose path is to be retrieved. The user can force the creation of a folder by combining the folder's CSIDL with CSIDL_FLAG_CREATE. If *pszFileNameSpecifier* is NULL or "\\", *nFolder* must have a specified CSIDL folder combined with CSIDL_FLAG_CREATE if the user wants to force to create the file.

ppszFilePath

[out] Address of a pointer to the null-terminated string that receives the file path information. Must be freed when no longer required.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppszFilePath</i> is invalid or bad.
E_OUTOFMEMORY	Exceeded available memory.

S_FALSE	A new file was created.
E_INVALIDARG	<i>pszValueName</i> is invalid or bad.
SPERR_UNINITIALIZED	Either the data key or the token delegate interface is uninitialized.
SPERR_TOKEN_DELETED	Key has been deleted.
FAILED(hr)	Appropriate error message.

Example

The following code snippet creates and removes a token object for a test file.

```

HRESULT hr;
GUID guid0;
GUID guid1;

CComPtr<ISpObjectToken> cpSpObjectToken;
CSpCoTaskMemPtr<WCHAR> cpFileName;
CSpCoTaskMemPtr<WCHAR> cpFileName2;

hr = SpGetDefaultTokenFromCategoryId(SPCAT_VOICES, &cpSpObjectToken);
//Check return value

ZeroStruct(guid0);

hr = CoCreateGuid(&guid1);
//Check return value

hr = cpSpObjectToken->GetStorageFileName( guid0, L"TestFile"
//The created file will have default format, and will be stored in the default location
//serves as a common repository for application-specific data
//Check return value

hr = cpSpObjectToken->Remove(&guid0);
//Check return value

hr = cpSpObjectToken->GetStorageFileName(guid1, L"TestFile2"
//The created file will be stored under C:\Program Files, and will have a unique name
//Check return value

```

```
hr = cpSpObjectToken->Remove(&guid1);  
//Check return value
```



ISpObjectToken::RemoveStorageFileName

ISpObjectToken::RemoveStorageFileName removes the object token file name.

```
HRESULT RemoveStorageFileName(  
    REFCLSID      clsidCaller,  
    const WCHAR  *pszKeyName,  
    BOOL          fDeleteFile  
);
```

Parameters

clsidCaller

[in] Globally unique identifier (GUID) of the calling object.

pszKeyName

[in] Address of a null-terminated string containing the registry key name.

fDeleteFile

[in] Value specifying if the file should be deleted. TRUE deletes the file afterward; FALSE does not.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszKeyName</i> is invalid or bad.
SPERR_UNINITIALIZED	Either the data key or token delegate interface is not initialized.
SPERR_TOKEN_DELETED	Key has been deleted.
FAILED(hr)	Appropriate error message.

Example

The following code snippet creates a test file, removes it and manually deletes it. It may also have been deleted automatically by setting *fDeleteFile* to TRUE.

```
HRESULT hr;
GUID guid0;

CComPtr<ISpObjectToken> cpSpObjectToken;
CComPtr<ISpObjectWithToken> cpSpObjectWithToken;
CSpCoTaskMemPtr<WCHAR> cpFileName;

hr = SpGetDefaultTokenFromCategoryId(SPCAT_VOICES, &cpSpObje
//Check return value

ZeroStruct(guid0);
// Create subkeys and value item to be deleted
hr = cpSpObjectToken->GetStorageFileName(guid0, L"test file"

if (SUCCEEDED(hr))
{
    hr = cpSpObjectToken->RemoveStorageFileName(guid0, L"test
    //Check return value

    cpFileName.Clear();
}
```



ISpObjectToken::Remove

ISpObjectToken::Remove removes an object token.

```
HRESULT Remove(  
    const CLSID *pclsidCaller  
);
```

Parameters

pclsidCaller

[in] Address of the identifier associated with the object token to remove. If *pclsidCaller* is NULL, the entire token is removed; otherwise, only the specified section is removed.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pclsidCaller</i> is invalid or bad.
SPERR_UNINITIALIZED	The token ID interface is uninitialized.
SPERR_TOKEN_DELETED	Key has been deleted.
FAILED(hr)	Appropriate error message.

Example

The following code snippet creates and removes a token object for a test file.

```
HRESULT hr;  
GUID guid0;  
  
CComPtr<ISpObjectToken> cpSpObjectToken;  
CSpCoTaskMemPtr<WCHAR> cpFileName;
```

```
hr = SpGetDefaultTokenFromCategoryId(SPCAT_VOICES, &cpSpObjectToken);  
//Check return value
```

```
ZeroStruct(guid0);  
hr = cpSpObjectToken->GetStorageFileName( guid0, L"TestFile"  
//Check return value
```

```
hr = cpSpObjectToken->Remove(&guid0);  
//Check return value
```



ISpObjectToken::IsUISupported

ISpObjectToken::IsUISupported determines if the user interface (UI) associated with the object is supported.

Ultimately, `ISpObjectToken::IsUISupported` is similar to creating an [ISpTokenUI](#) object and calling [ISpTokenUI::IsUISupported](#).

```
[local] HRESULT IsUISupported(  
    const WCHAR *pszTypeOfUI,  
    void *pvExtraData,  
    ULONG cbExtraData,  
    IUnknown *punkObject,  
    BOOL *pfSupported  
);
```

Parameters

pszTypeOfUI

[in] Address of the null-terminated string containing the UI type that is being queried. Must be a [SPDUI_XXX](#) type.

pvExtraData

[in] Pointer to additional information needed for the object. The [ISpTokenUI](#) object implementer dictates the format and usage of the data provided. See Remarks section.

cbExtraData

[in] Size, in bytes, of the *ExtraData*. The [ISpTokenUI](#) object implementer dictates the format and usage of the data provided.

punkObject

[in] Address of the `IUnknown` interface pointer. See Remarks

section.

pfSupported

[out] Address of a variable that receives the value indicating support for the interface. This value is set to TRUE when this interface is supported, and FALSE when it is not. If this value is TRUE, but the return code is S_FALSE, the UI type (guidTypeOfUI) is supported, but not with the current parameters or run-time environment. Check with the implementer of the UI object to verify run-time requirements.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	The UI is supported but not with the current run-time environment or parameters.
E_INVALIDARG	One of the parameters is invalid or bad.
SPERR_UNINITIALIZED	Either the data key or token delegate interface is not initialized.
SPERR_TOKEN_DELETED	Key has been deleted.
FAILED(hr)	Appropriate error message.

Remarks

pvExtraData and *punkObject* Parameters: When asking an [ISpObjectToken](#) to display a particular piece of UI, the UI object may require extra functionality that only it understands.

Common implementation practice for accessing this functionality is to QueryInterface off of a known [IUnknown](#) interface. The caller of [ISpTokenUI::IsUISupported](#) can set the *punkObject* parameter with the necessary [IUnknown](#) interface. For example, to display a Speech Recognition Training UI (see

[SPDUI_UserTraining](#)) requires a specific SR engine.

Example

The following code snippet illustrates the use of `ISpObjectToken::IsUISupported` using [SPGUID_EngineProperties](#).

```
HRESULT hr = S_OK;

// get the default text-to-speech engine object token
hr = SpGetDefaultTokenFromCategoryId(SPCAT_VOICES, &cpObj
// Check hr

// create the engine object based on the object token
hr = SpCreateObjectFromToken(cpObjectToken, &cpVoice);
// Check hr

// create a data key for the voice's UI objects
hr = cpObjectToken->OpenKey(L"UI", &cpUIDataKey);
// Check hr

// create a data key for the specific Engine Properties
hr = cpUIDataKey->OpenKey(SPDUI_EngineProperties, &cpEng
// Check hr

// get the GUID for the voice's engine properties UI
hr = cpEngPropsDataKey->GetStringValue(L"CLSID", &pwszEng
// Check hr

// convert GUID string to pure GUID
hr = CLSIDFromString(pwszEngPropsCLSID, &clsidEngProps);
// Check hr

// check if the default voice object has UI for Properties
hr = cpObjectToken->IsUISupported(&clsidEngProps, NULL,
// Check hr

// if fSupported == TRUE, then default voice object has
```



ISpObjectToken::DisplayUI

ISpObjectToken::DisplayUI displays the user interface (UI) associated with the object.

```
[local] HRESULT DisplayUI(  
    HWND          hwndParent,  
    const WCHAR   *pszTitle,  
    const WCHAR   pszTypeOfUI,  
    void          *pvExtraData,  
    ULONG         cbExtraData,  
    IUnknown      *punkObject  
);
```

Parameters

hwndParent

[in] Specifies the handle of the parent window.

pszTitle

[in] Address of a null-terminated string containing the window title. Set this value to NULL to indicate that the [ISpTokenUI](#) object should use its default window title.

pszTypeOfUI

[in] Address of the null-terminated string containing the UI type that is being queried. Must be a [SPDUI_xxx](#) type.

pvExtraData

[in] Pointer to additional information needed for the object. The [ISpTokenUI](#) object implementer dictates the format and usage of the data provided.

cbExtraData

[in] Size, in bytes, of the *ExtraData*. See Remarks section.

punkObject

[in] Address of the IUnknown interface pointer. See Remarks section.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	The UI is supported but not with the current run-time environment or parameters.
E_INVALIDARG	One of the parameters is invalid or bad.
SPERR_UNINITIALIZED	Either the data key or token delegate interface is not initialized.
SPERR_TOKEN_DELETED	Key has been deleted.
FAILED(hr)	Appropriate error message.

Remarks

pvExtraData and *punkObject* Parameters: When requesting an [ISpObjectToken](#) to display a particular piece of UI, the UI object may require extra functionality. Common implementation practice for accessing this functionality is to QueryInterface from a known [IUnknown](#) interface. The caller of [ISpTokenUI::DisplayUI](#) can set the *punkObject* parameter with the necessary [IUnknown](#) interface. For example, asking to display Speech Recognition Training UI (see [SPDUI_UserTraining](#)) requires the use of a specific SR engine.

The best practice for using [ISpObjectToken::DisplayUI](#) is to call [ISpObjectToken::IsUISupported](#) with a specific UI type before calling [DisplayUI](#). Ultimately, [ISpObjectToken::DisplayUI](#) is similar to creating an [ISpTokenUI](#) object and calling

[ISpTokenUI::DisplayUI](#)

The call to DisplayUI is synchronous and the call will not return until the UI has been closed.

Example

The following code snippet illustrates the use of ISpObjectToken::DisplayUI using [SPGUID_EngineProperties](#).

```
HRESULT hr = S_OK;

// get the default text-to-speech engine object token
hr = SpGetDefaultTokenFromCategoryId(SPCAT_VOICES, &cpOb
// Check hr

// create the engine object based on the object token
hr = SpCreateObjectFromToken(cpObjectToken, &cpVoice);
// Check hr

// create a data key for the voice's UI objects
hr = cpObjectToken->OpenKey(L"UI", &cpUIDataKey);
// Check hr

// create a data key for the specific Engine Properties I
hr = cpUIDataKey->OpenKey(SPDUI_EngineProperties, &cpEngl
// Check hr

// get the GUID for the voice's engine properties UI
hr = cpEngPropsDataKey->GetStringValue(L"CLSID", &pwszEn
// Check hr

// convert GUID string to pure GUID
hr = CLSIDFromString(pwszEngPropsCLSID, &clsidEngProps);
// Check hr

// check if the default voice object has UI for Properti
hr = cpObjectToken->DisplayUI(MY_HWND, MY_APP_VOICE_PROPI
// Check hr
```



ISpObjectToken::MatchesAttributes

ISpObjectToken::MatchesAttributes determines if an object token supports a specified attribute.

```
HRESULT MatchesAttributes(  
    const WCHAR    *pszAttributes,  
    BOOL           *pfMatches  
);
```

Parameters

pszAttributes

[in] Address of the null-terminated string specifying the object token attribute to match.

pfMatches

[out] Address of a variable that receives the value indicating a match of the object token attribute specified in *pszAttributes*. This value is set to TRUE when the object token attribute matches, and FALSE when it does not match.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
SPERR_UNINITIALIZED	The object has not been properly initialized.
FAILED(hr)	Appropriate error message.



IEnumSpObjectTokens

The IEnumSpObjectTokens interface is used to enumerate speech object tokens.

Associated Class IDs

The following class IDs (CLSID) may be used with this interface. A complete CLSID listing for all interfaces is in the [Class IDs](#) section.

CLSID_SpMMAudioEnum

Methods in Vtable Order

IEnumSpObjectTokens Methods	Description
Next	Retrieves the next object token in the enumeration sequence.
Skip	Skips a specified number of object tokens in the enumeration sequence.
Reset	Resets the enumeration sequence to the beginning.
Clone	Creates a new enumerator object with the same items.
Item	Locates a specific token in the enumeration.
GetCount	Retrieves the number of object tokens contained in the enumeration sequence.



IEnumSpObjectTokens::Next

IEnumSpObjectTokens::Next retrieves the next object token in the enumeration sequence.

If there are fewer than the requested number of elements left in the sequence, the remaining elements are retrieved.

```
HRESULT Next(  
    ULONG                celt,  
    ISpObjectToken    **pelt,  
    ULONG                *pceltFetched  
);
```

Parameters

celt

[in] The number of object tokens to retrieve.

pelt

[out] Address of an array that receives ISpObjectToken pointers. If an error value is returned, no entries in the array are valid.

pceltFetched

[out] Address of a variable that receives the number of ISpObjectToken pointers actually copied to the array. This parameter cannot be NULL if *celt* is greater than one. If this parameter is NULL, *celt* must be one.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	One of the following errors was

	encountered: <i>pelt</i> is bad or invalid, <i>pceltFetched</i> is bad or invalid, <i>pceltFetched</i> is bad and <i>celt</i> is greater than one.
E_INVALIDARG	<i>celt</i> is zero.
SPERR_UNINITIALIZED	Attribute parser interface is not initialized.
S_FALSE	<i>celt</i> is greater than the number of objects available.
FAILED (hr)	Appropriate error message.



IEnumSpObjectTokens::Skip

IEnumSpObjectTokens::Skip skips a specified number of object tokens in the enumeration sequence.

```
HRESULT Skip(  
    ULONG    celt  
);
```

Parameters

celt

[in] Number of object tokens to skip in the enumeration sequence.

Return values

Value	Description
S_OK	Number of elements skipped was <i>celt</i> .
S_FALSE	Number of elements skipped was less than <i>celt</i> .
SPERR_UNINITIALIZED	Attribute parser interface is not initialized.
FAILED (hr)	Appropriate error message.



IEnumSpObjectTokens::Reset

IEnumSpObjectTokens::Reset resets the enumeration sequence to the beginning.

```
HRESULT Reset ( void );
```

Parameters

None

Return values

Value	Description
S_OK	Method completed successfully.
SPERR_UNINITIALIZED	Attribute parser interface is not initialized.



IEnumSpObjectTokens::Clone

IEnumSpObjectTokens::Clone creates a new enumerator object with the same items.

Returns a new enumerator object with the same items but an independent index. The items in the clone are not guaranteed to be in the same order as the original enumerator.

```
HRESULT Clone(  
    IEnumSpObjectTokens    **ppEnum  
);
```

Parameters

ppEnum

[out] Address of the IEnumSpObjectTokens pointer variable that receives the interface pointer to the cloned enumerator. Using Clone, it is possible to record a particular point in the enumeration sequence and then return to that point at a later time. The enumerator returned is of the same interface type as the one being cloned.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_UNINITIALIZED	Attribute parser interface is not initialized.
FAILED (hr)	Appropriate error message.



IEnumSpObjectTokens::Item

IEnumSpObjectTokens::Item locates a specific token in the enumeration.

```
HRESULT Item(  
    ULONG          Index,  
    ISpObjectToken **ppToken  
);
```

Parameters

Index

[in] Value indicating which token in the enumeration sequence to locate.

ppToken

[out] Address of an ISpObjectToken interface pointer.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_NO_MORE_ITEMS	<i>Index</i> is greater than the amount of items available.
E_POINTER	<i>ppToken</i> is bad or invalid.
SPERR_UNINITIALIZED	Attribute parser interface is not initialized.
FAILED (hr)	Appropriate error message.



IEnumSpObjectTokens::GetCount

IEnumSpObjectTokens::GetCount retrieves the number of object tokens contained in the enumeration sequence.

```
HRESULT GetCount(  
    ULONG    *pulCount  
);
```

Parameters

pulCount

[out] The number of object token items contained in the enumeration sequence.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pulCount</i> is bad or invalid.
SPERR_UNINITIALIZED	Attribute parser interface is not initialized.
FAILED (hr)	Appropriate error message.



ISpObjectWithToken

Any object associated with an object token implements the ISpObjectWithToken interface. When the method [ISpObjectToken::CreateInstance](#) is called on the object token, SAPI creates the associated object. If this object implements ISpObjectWithToken, the [SetObjectToken](#) method will be called, passing in a reference to the object token.

After SAPI calls ISpObjectWithToken::SetObjectToken, a token object is created and its data may be accessed.

Applications do not need to implement this interface, but engine, lexicon, or custom audio developers may implement it in order to access their object token data.

Implemented By

- SpMMAudioEnum
- SpRecPlayAudio
- SpUnCompressedLexicon
- SpCompressedLexicon

Methods in Vtable Order

ISpObjectWithToken Methods	Description
SetObjectToken	Binds the instance object to the specified token.
GetObjectToken	Retrieves an object token.



ISpObjectWithToken::SetObjectToken

ISpObjectWithToken::SetObjectToken binds the instance object to the specified token.

```
HRESULT SetObjectToken(  
    ISpObjectToken *pToken  
);
```

Parameters

pToken

[in] The token interface pointer this instance object is to combine with.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pToken</i> is invalid or bad.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.



ISpObjectWithToken::GetObjectToken

ISpObjectWithToken::GetObjectToken retrieves an object token.

```
HRESULT GetObjectToken(  
    ISpObjectToken **ppToken  
);
```

Parameters

ppToken

[out] Address of an ISpObjectToken interface that receives the object token.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppToken</i> is invalid or bad.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.



ISpResourceManager

The ISpResourceManager interface provides access to the shared resources between different speech components in the same process. The object CLSID_SpResourceManager is a COM singleton. That is, the same resource manager will be shared by all clients in a single process that uses CoCreateInstance to create the object with CLSID_SpResourceManager. This allows for a common point of control for sharing other COM objects.

Applications and engines do not need to use or implement this interface.

This interface inherits from IServiceProvider. The IServiceProvider interface supports a single method, QueryService, which provides access to objects in the resource manager.

Associated Class IDs

The following class IDs (CLSID) may be used with this interface. A complete CLSID listing for all interfaces is in the [Class IDs](#) section.

CLSID_SpResourceManager

Methods in Vtable Order

ISpResourceManager Methods	Description
SetObject	Adds a service object to the current service list.
GetObject	Retrieves a service object from the current service list, or creates one if it does not exist.



ISpResourceManager::SetObject

ISpResourceManager::SetObject adds a COM object to the current service list. If an object is already set for the specified service, the *ISpResourceManager::SetObject* method will replace the current object with the new one. If *pUnkObject* is NULL, the current service object is removed.

```
HRESULT SetObject(  
    REFGUID      guidServiceId,  
    IUnknown    *pUnkObject  
);
```

Parameters

guidServiceId

[in] The unique identifier of the service.

pUnkObject

[in] Address of the IUnknown interface of the object that is setting the service. Any existing service object is removed if this parameter is NULL.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pUnkObject</i> is bad or invalid.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.



ISpResourceManager::GetObject

ISpResourceManager::GetObject retrieves a service object from the current service list, or creates one if it does not exist.

```
HRESULT GetObject(  
    REFGUID    guidServiceId,  
    REFCLSID   ObjectCLSID,  
    REFIID     ObjectIID,  
    BOOL       fReleaseWhenNoRefs,  
    void       **ppObject  
);
```

Parameters

guidServiceId

[in] The unique identifier of the service.

ObjectCLSID

[in] Class identifier of the object.

ObjectIID

[in] Identifier of interface to return in *ppObject*

fReleaseWhenNoRefs

[in] Boolean indicating whether the object should be freed when the last client outside of the resource manager releases the object. If this flag is set, the object specified by *ObjectCLSID* must support aggregation.

ppObject

[out] Address of a pointer that receives the interface pointer of the service.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
E_POINTER	<i>ppObject</i> is bad or invalid.
REGDB_E_CLASSNOTREG	Class is not registered.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.

Remarks

If the object does not exist, the *ObjectCLSID* parameter will be used to `CoCreateInstance` the object. This functionality allows multiple threads to ensure that a single shared object is created atomically by the resource manager. If the *fReleaseWhenNoRefs* flag is set to `TRUE`, the final release of the object will remove it from the service list. If *fReleaseWhenNoRefs* is `FALSE`, the service will remain in the service list until the resource manager is released or the service is explicitly removed through a [SetObject](#) call.



ISpTask

The ISpTask interface is a C++ pure virtual interface, and not a COM interface. It is used by objects to perform atomic operations, which have been optimized for a multiprocessor computer. The [ISpTaskManager](#) can be used to create task objects that support the [ISpNotifySink](#) interface. When the Notify is called on these objects, a thread will be allocated from a thread pool, and ISpTask::Execute will be called on that thread. The client code then performs the necessary operation on that thread and returns from Execute when finished. Applications should avoid blocking on I/O operations because they are consuming a thread from the shared thread pool.

When to Implement

ISpTask is most useful with multiprocessor computers. ISpTask allocates tasks efficiently based on the current availability of processor time. Implement ISpTask on objects that perform tasks that can be broken into smaller tasks.

This is not a COM interface.

Methods in Vtable Order

ISpTask Methods	Description
Execute	Calls on a worker thread to allow the client to perform necessary task operations.



ISpTask::Execute

ISpTask::Execute calls on a worker thread to allow the client to perform necessary task operations.

Implements the work unit for an object. This will be application specific.

```
virtual HRESULT STDMETHODCALLTYPE Execute(  
    void *pvTaskData,  
    volatile const BOOL *pfContinueProcessing  
);
```

Parameters

pvTaskData

[in] The pointer passed to [ISpTaskManager::CreateReoccurringTask](#) *pvTaskData* parameter.

pfContinueProcessing

[in] Boolean indicating if the process should continue. TRUE continues the process; otherwise FALSE. Clients should examine this variable during processing and exit if this flag set to FALSE, as it indicates that another thread has released the reoccurring task object.

Return values

The return value is ignored by the SAPI task manager.



Speech Recognition interfaces

The following section covers:

- [ISpRecoContext](#)
- [ISpRecoGrammar](#)
- [ISpRecoResult](#)
- [ISpRecognizer](#)
- [ISpPhrase](#)
- [ISpPhraseAlt](#)
- [ISpProperties](#)



ISpRecoContext

The ISpRecoContext interface enables applications to create different functional views or contexts of the SR engine. Each ISpRecoContext object can take interest in different SR events (see also [ISpEventSource](#) and [SPEVENTENUM](#)) and use different recognition grammars (see also [ISpRecoGrammar](#)). Applications must have at least one ISpRecoContext instance to receive recognitions. Applications can also create multiple ISpRecoContext instances to separate different types of recognition with their application. For example, a multiple-document-interface (MDI) application could associate a different ISpRecoContext instance with each document pane to localize the grammar and support and event processing.

A new ISpRecoContext object can be created by calling [ISpRecognizer::CreateRecoContext](#).

To use a shared recognizer (see description of [ISpRecognizer](#)), an application can easily create a shared ISpRecoContext by calling `::CoCreateInstance` with `CLSID_SpSharedRecoContext`.

Associated Class IDs

The following class IDs (CLSID) may be used with this interface. A complete CLSID listing for all interfaces is in the [Class IDs](#) section.

CLSID_SpSharedRecoContext

Methods in Vtable Order

ISpRecoContext Methods	Description
ISpEventSource	Inherits from ISpEventSource and those methods are accessible from ISpRecoContext.
GetRecognizer	Returns a reference to the current

	recognizer object associated with this context.
<u>CreateGrammar</u>	Creates an SpGrammar object.
<u>GetStatus</u>	Retrieves current state information associated with a context.
<u>GetMaxAlternates</u>	Retrieves the maximum number of alternates that will be generated for command and control grammars.
<u>SetMaxAlternates</u>	Sets the maximum number of alternates returned for command and control grammars.
<u>SetAudioOptions</u>	Sets the audio options for results from this recognition context.
<u>GetAudioOptions</u>	Retrieves the audio options for the context.
<u>DeserializeResult</u>	Creates a new result object from a serialized result.
<u>Bookmark</u>	Sets a bookmark within the current recognition stream.
<u>SetAdaptationData</u>	Passes a block of text to the SR engine which can be used to adapt the active language models.
<u>Pause</u>	Pauses the engine object to synchronize with the SR engine.
<u>Resume</u>	Resumes the SR engine from the paused state and restarts the recognition process.
<u>SetVoice</u>	Sets the associated ISpVoice to this context.
<u>GetVoice</u>	Retrieves a reference to the associated ISpVoice object.
<u>SetVoicePurgeEvent</u>	Sets the SR engine events that stop audio output, and purges the current speaking queue.
<u>GetVoicePurgeEvent</u>	Retrieves the set of SR engine

	events that stop audio output, and purges the current speaking queue.
<u>SetContextState</u>	Sets the state of the recognition context.
<u>GetContextState</u>	Retrieves the state of the recognition context.



ISpRecoContext::GetRecognizer

ISpRecoContext::GetRecognizer returns a reference to the current recognizer object associated with this context.

```
HRESULT GetRecognizer(  
    ISpRecognizer    **ppRecognizer  
);
```

Parameters

ppRecognizer

[out] Address of a pointer that receives the ISpRecognizer interface. The application must call IUnknown::Release when finished with the interface.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Invalid pointer.
FAILED (hr)	Appropriate error message.

Example

The following code snippet illustrates the use of ISpRecoContext::GetRecognizer with a shared context

```
HRESULT hr = S_OK;  
  
// create a shared recognition context  
hr = cpRecoContext.CoCreateInstance(CLSID_SpSharedRecoCo  
// Check hr  
  
// get a reference to the associated recognizer  
hr = cpRecoContext->GetRecognizer(&cpRecognizer);
```

```
// Check hr

// assert that our shared context has a shared recognize
hr = cpRecognizer->IsSharedInstance();
// Check that hr == S_OK
```



ISpRecoContext::CreateGrammar

ISpRecoContext::CreateGrammar creates an SpRecoGrammar object.

```
HRESULT CreateGrammar(  
    ULONGLONGT          ullGrammarId,  
    ISpRecoGrammar      **ppGrammar  
);
```

Parameters

ullGrammarId

[in] Specifies the grammar identifier. The identifier is used by the application and is not required. This identifier is associated with all result objects from the grammar (see [SPPHRASE.ullGrammarID](#)).

ppGrammar

[out] Address of a pointer which receives the [ISpRecoGrammar](#) object. The application must call IUnknown::Release on the object when finished using it.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppGrammar</i> is invalid.
E_OUTOFMEMORY	Not enough system memory to create a grammar object.
SPERR_SR_ENGINE_EXCEPTION	An exception was thrown by the SR engine during OnCreateGrammar .

FAILED(hr)

Appropriate error message.

Example

The following code snippet illustrates the use of `ISpRecoContext::CreateGrammar`.

```
HRESULT hr = S_OK;

hr = cpRecoContext->CreateGrammar(GRAM_ID, &cpRecoGrammar);
// Check hr

// load a cfg from a file
hr = cpRecoGrammar->LoadCmdFromFile(MY_CFG_FILENAME, SPL);
// Check hr

// activate the top-level rules
hr = cpRecoGrammar->SetRuleState(NULL, NULL, SPRS_ACTIVE);
// Check hr

// get a recognition
// ...

// get the recognized phrase from the recognition result
hr = cpRecoResult->GetPhrase(&pPhrase);
// Check hr

// check the grammar id of the recognition result
SPDBG_ASSERT(GRAM_ID == pPhrase->ullGrammarID);
```



ISpRecoContext::GetStatus

ISpRecoContext::GetStatus retrieves current state information associated with a context (e.g., last SR engine requested UI, audio signal status, etc.).

```
HRESULT GetStatus(  
    SPRECOCONTEXTSTATUS *pStatus  
);
```

Parameters

pStatus

[out] Address of the [SPRECOCONTEXTSTATUS](#) structure that receives the context state information.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pStatus</i> is invalid or bad.
FAILED (hr)	Appropriate error message.

Remarks

A graphical application that is interested in [SPEI_REQUEST_UI](#) events from the SR engine can call `ISpRecoContext::GetStatus`, and check the *szRequestTypeOfUI* field to check the last requested UI-type. After the application has called [ISpRecognizer::DisplayUI](#), the SR engine can clear the *szRequestTypeOfUI* field by calling [ISpSREngineSite::AddEvent](#) with a NULL UI-type.

An application can also periodically query the recognition context status to check the audio signal quality (see also

[SPINTERFERENCE](#)) and respond appropriately. An application can prompt the user to access the SR engine's microphone training UI to improve the audio signal quality (see [SPDUI_MicTraining](#)), or prompt the user to modify the audio settings using Speech properties in Control Panel (see [SPDUI_AudioProperties](#) and [SPDUI_AudioVolume](#)).

Example

The following code snippet illustrates the use of [ISpRecoContext::GetStatus](#) for responding to SR engine UI requests.

```
HRESULT hr = S_OK;

// assume UI request [SPEI_REQUEST_UI] has been received

// check what kind of UI the SR Engine wants
hr = cpRecoContext->GetStatus(&contextStatus);
// Check hr

// get a reference to the SR Engine
hr = cpRecoContext->GetRecognizer(&cpRecognizer);
// Check hr

// sanity check that the UI type is supported
hr = cpRecognizer->IsUISupported(contextStatus.szRequest);
// Check hr

// ask the SR engine to display the UI, and use the default
hr = cpRecognizer->DisplayUI(MY_HWND, NULL, contextStatus);
// Check hr
```



ISpRecoContext::GetMaxAlternates

ISpRecoContext::GetMaxAlternates retrieves the maximum number of alternates that the SR engine will return for command and control or proprietary grammars associated with this context. See Remarks section.

```
HRESULT GetMaxAlternates(  
    ULONG    *pcMaxAlternates  
);
```

Parameters

pcMaxAlternates

[out] The maximum number of alternates. The default value is zero, unless the application specifies it by [ISpRecoContext::SetMaxAlternates](#).

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pcMaxAlternates</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Remarks

The value is the maximum number of alternates that will be returned to the application. For SR engine's that do not support command and control (or proprietary grammar) alternates, this method will succeed, but the alternates returned will always be zero.

For applications and SR engines that are using proprietary grammars and proprietary alternates, **ISpRecoContext::GetMaxAlternates** and

[ISpRecoContext::SetMaxAlternates](#) is the recommended method of coordinating maximum alternate values between the application and SR engine.

The SR engine can query each context's maximum requested alternates value by calling [ISpSREngineSite::GetContextMaxAlternates](#) with the context handle. When using SAPI command and control grammars, the SR engine can call [ISpSREngineSite::GetMaxAlternates](#) with the rule handle.

This method has no effect on dictation alternates. See [ISpRecoResult::GetAlternates](#) for information regarding dictation alternates.

The current version of the Microsoft SR engine does not support command and control alternates.



ISpRecoContext::SetMaxAlternates

ISpRecoContext::SetMaxAlternates sets the maximum number of alternates that the SR engine will return for command and control or proprietary grammars associated with this context. See Remarks section.

```
HRESULT SetMaxAlternates(  
    ULONG    cAlternates  
);
```

Parameters

cAlternates

[in] Specifies the maximum number of alternates the engine will return.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.

Remarks

By default, the maximum alternates value is zero, so an application must call `::SetMaxAlternates` to retrieve alternates.

The value is the maximum number of alternates that will be returned to the application. For SR engine's that do not support command and control (or proprietary grammar) alternates, this method will succeed, but the alternates returned will always be zero.

For applications and SR engines using proprietary grammars

and proprietary alternates, [ISpRecoContext::GetMaxAlternates](#) and [ISpRecoContext::SetMaxAlternates](#) are the recommended methods of coordinating maximum alternate values between the application and the SR engine.

The SR engine can query each context's maximum requested alternates value by calling [ISpSREngineSite::GetContextMaxAlternates](#) with the context handle. When using SAPI command and control grammars, the SR engine can call [ISpSREngineSite::GetMaxAlternates](#) with the rule handle.

This method has no effect on dictation alternates. See [ISpRecoResult::GetAlternates](#) for information regarding dictation alternates.

The current version of the Microsoft speech recognition engine does not support command and control alternates.



ISpRecoContext::SetAudioOptions

ISpRecoContext::SetAudioOptions sets the audio options for result objects from this recognition context. This method also enables or disables the retention of audio with result objects and can change the retained audio format.

```
HRESULT SetAudioOptions(  
    SPAUDIOOPTIONS Options,  
    const GUID *pAudioFormatId,  
    const WAVEFORMATEX *pWaveFormatEx  
);
```

Parameters

Options

[in] Flag of type SPAUDIOOPTIONS indicating the option. It must be one of the following:

Value	
SPA0_NONE	Do not retain audio for results.
SPA0_RETAIN_AUDIO	Retain audio for all future results.

pAudioFormatId

[in] The audio stream format id [of type GUID]. Usually this value is *SPDFID_WaveFormatEx*. If this value is NULL, the retained audio format will not be changed. Reset the retained

audio format to the SR engine's recognition format by setting this value to GUID_NULL and *pWaveFormatEx* to NULL.

pWaveFormatEx

[in] The audio stream wave format [of type [WAVEFORMATEX](#)]. This is only valid if **pAudioFormatId* == *SPFID_WaveFormatEx*.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>Options</i> is not one of the correct types, or the spe format is not valid.
FAILED(hr)	Appropriate error message.

Remarks

If a [WAVEFORMATEX](#)-based retained audio format is specified, but the SR engine and the audio input stream agree on a non-[WAVEFORMATEX](#)-based audio input format (e.g., custom audio object/format), *ISpRecoContext::SetAudioOptions* will return successfully, but will not retain the audio.

By default, SAPI does not retain recognition audio.

By default, when an audio format is not specified, the audio will be retained in the same format that the SR engine used to perform the recognition.

Example

The following code snippet illustrates the use of *ISpRecoContext::SetAudioOptions*.

```
HRESULT hr = S_OK;  
  
// activate retained audio settings with default format
```

```

hr = cpRecoContext->SetAudioOptions(SPAO_RETAIN_AUDIO, &
// Check hr

// deactivate retained audio settings
hr = cpRecoContext->SetAudioOptions(SPAO_NONE, NULL, NULL);
// Check hr

// change the retained audio format to 11 kHz, 16-bit Stereo
hr = cpRecoContext->SetAudioOptions(SPAO_11KHZ_16BIT_STEREO, &
// Check hr

// use the stream format helper to fill in the WAVEFORMATEX structure
CSpStreamFormat sfRetained(SPSF_24kHz16BitStereo, &hr);
// Check hr

// change the settings to the selected stream format
hr = cpRecoContext->SetAudioOptions(SPAO_RETAIN_AUDIO, &
// Check hr

```

Development Helpers

Helper Enumerations and Functions	Description
<u>SPSTREAMFORMAT</u>	SAPI supported stream formats
<u>CSpStreamFormat</u>	Class for managing SAPI supported stream formats and WAVEFORMATEX structures



ISpRecoContext::GetAudioOptions

ISpRecoContext::GetAudioOptions retrieves the audio options for the context.

```
HRESULT GetAudioOptions(  
    SPAUDIOOPTIONS *Options,  
    GUID *pAudioFormatId,  
    WAVEFORMATEX **ppCoMemWFE  
);
```

Parameters

Options

[out] Address that will receive pointer to [SPAUDIOOPTIONS](#) flag, indicating the options set for this context. If this value is not to be retrieved, specify NULL. The flag can be one of the following values:

Value	Meaning
SPA0_NONE	Do not retain audio for results.
SPA0_RETAIN_AUDIO	Retain audio for all future results.

pAudioFormatId

[in] Address that will receive the audio stream format type (i.e., GUID). If the application is not interested in the retained audio format, NULL is specified (i.e., ignore both *pAudioFormatId* and *pWaveFormatEx* parameters).

ppCoMemWFE

[in] Address that will receive a pointer to the audio stream wave format structure (i.e., [WAVEFORMATEX](#)). This can be

NULL if the application is not interested in the retained audio format. If WAVEFORMATEX data is retrieved, it must be freed using `::CoTaskMemFree()`.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	One of the pointers is invalid or bad.
FAILED(hr)	Appropriate error message.

Remarks

The default audio options are none (i.e., [SPAO_NONE](#)). The default retained audio format is the speech recognition engine's recognition format (see [ISpRecognizer::GetFormat](#) with [SPWF_SRENGINE](#)).

See also [ISpRecoContext::SetAudioOptions](#).

Example

The following code snippet illustrates the use of `ISpRecoContext::GetAudioOptions` and querying the different retained audio settings.

```
HRESULT hr = S_OK;

// check if audio is being retained (default is NO)
hr = cpRecoContext->GetAudioOptions(&pAudioOptions, NULL)
// Check hr

// check what audio format would be retained
hr = cpRecoContext->GetAudioOptions(NULL, &guidFormat, &
// Check hr
```

```

// ... do stuff

// free the wave format memory
::CoTaskMemFree(pWaveFormatEx);

// check if audio is being retained, and if so what the
hr = cpRecoContext->GetAudioOptions(&guid, &guidFormat, &
// Check hr

// ... do stuff

// free the wave format memory
::CoTaskMemFree(pWaveFormatEx);

```

Development Helpers

Helper Enumerations and Functions	Description
SPSTREAMFORMAT	SAPI supported stream formats
CSpStreamFormat	Class for managing SAPI supported stream formats and WAVEFORMATEX structures



ISpRecoContext::DeserializeResult

ISpRecoContext::DeserializeResult creates a new result object from a serialized result.

```
HRESULT DeserializeResult(  
    const SPSERIALIZEDRESULT *pSerializedResult,  
    ISpRecoResult **ppResult  
);
```

Parameters

pSerializedResult

[in] Pointer to a serialized result. See also [SPSERIALIZEDRESULT](#)

ppResult

[out] The unserialized result object. The application must call `IUnknown::Release` when finished using the `ISpRecoResult` reference.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pSerializedResult</i> is invalid or bad.
E_POINTER	<i>ppResult</i> is invalid or bad.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.

Remarks

After deserializing the [ISpRecoResult](#) object, the application can

retrieve alternates for the RecoResult, retrieve the retained audio, or examine the recognition.

The same SR engine that was originally used to generate the ISpRecoResult object (or recognition) must be the same SR engine that is associated with the ISpRecoContext that called ::DeserializeResult (see [SPPHRASE.SREngineID](#)). The SR engine requirement ensures that the SR engine is capable of recognizing the original phrase, and that it understands any SR engine private result data (see [SPPHRASE.ulSREnginePrivateDataSize](#)).

Example

The following code snippet illustrates the use ISpRecoContext::Deserialize a previously serialized result object.

```
HRESULT hr = S_OK;

// ... obtain a recognition result object from the recognizer...

SPSERIALIZEDRESULT* pSerializedResult = NULL;
ULONG cbWritten = 0;
ULONG ulSerializedSize = 0;
LARGE_INTEGER liseek;
LARGE_INTEGER li;
CComPtr<IStream> cpStreamWithResult;

hr = CreateStreamOnHGlobal(NULL, true, &cpStreamWithResult);
// Check hr

// Serialize result to memory
hr = cpRecoResult->Serialize(&pSerializedResult);
// Check hr

//serialized to a stream pointer
```

```
hr = cpStreamWithResult->Write(pSerializedResult, pSerializedRes
// Check hr

// free the serialized result
if (pSerializedResult)::CoTaskMemFree(pSerializedResult);

// commit the stream changes
hr = cpStreamWithResult->Commit(STGC_DEFAULT);
// Check hr

// ... persist stream to disk, network share, etc...
// ... shutdown application ....

// ... restart application and get the persisted stream

// reset the stream seek pointer to the start before deserialization
li.QuadPart = 0;
hr = cpStreamWithResult->Seek(li, STREAM_SEEK_SET, NULL);
// Check hr

// find the size of the stream
hr = cpStreamWithResult->Read(&ulSerializedSize, sizeof(ULONG)
// Check hr

// reset the seek pointer
liseek.QuadPart = 0 - sizeof(ULONG);
hr = cpStreamWithResult->Seek(liseek, STREAM_SEEK_CUR, NU
// Check hr

// allocate the memory for the result
pSerializedResult = (SPSERIALIZEDRESULT*)::CoTaskMemAllo
// Check pSerializedResult in case out "out-of-memory"
```

```
// copy the stream into a serialized result object
hr = cpStreamWithResult->Read(pSerializedResult, ulSerializedSize);
// Check hr

// Deserialize result from memory
hr = cpRecoContext->DeserializeResult(pSerializedResult, &cpRecoResult);
// Check hr

// free the pSerializedResult memory
if (pSerializedResult) {
    CoTaskMemFree(pSerializedResult);
}

// As long as the same engine was used to generate
// the original result object, as is now being used,
// applications can now get alternates for the cpRecoResultNew's phi
```



ISpRecoContext::Bookmark

ISpRecoContext::Bookmark sets a bookmark within the current recognition stream. When the engine reaches the specified stream position, a bookmark event is added to the event queue.

```
HRESULT Bookmark(  
    SPBOOKMARKOPTIONS    Options,  
    ULONGLONG            ullStreamPosition,  
    LPARAM                lParamEvent  
);
```

Parameters

Options

[in] Indicates the option associated with the bookmark. Must be one of type [SPBOOKMARKOPTIONS](#).

ullStreamPosition

[in] Specifies the stream position. This value may be anywhere in the stream and will send a Bookmark event when that position is reached. Additionally it may be any one of two special values: [SP_STREAMPOS_ASAP](#) or [SP_STREAMPOS_REALTIME](#). See Remarks section for additional information.

lParamEvent

[in] The *lparam* for the SAPI bookmark event, and can be any value the application wants returned with the bookmark event (e.g., unique identifier, data pointer, NULL, etc.).

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>Options</i> has a bad value.
FAILED(hr)	Appropriate error message.

Remarks

If *Options* is set to SPBO_PAUSE, the [SPEVENT](#) *wParam* variable will be set to SPREF_AutoPause.

An application that wants to implement display a recognition progress/latency meter could use `ISpRecoContext::Bookmark` with SP_STREAMPOS_REALTIME, and update the UI when each bookmark is received. See also [ISpRecognizer::GetStatus](#)

An application that wants to pause the SR engine to perform specific processing could use `ISpRecoContext::Bookmark` with SPBO_PAUSE SP_STREAMPOS_ASAP, which will pause the SR engine automatically, and asynchronously for the application. The application should call [ISpRecoContext::Resume](#) to resume the recognition process. See also [ISpRecoGrammar::SetRuleState](#) regarding "auto-pause" rules.

It is possible to bookmark a stream position before starting a stream. If there is currently no stream and the caller specifies an offset of zero (or as soon as possible) or SP_STREAMPOS_REALTIME (indicating immediately" for a live audio device) the bookmark fires immediately. In both cases the application gets the bookmark as soon as the stream is created. Otherwise, the bookmark is delayed until the next stream reaches the specified offset.

Example

The following code snippet illustrates the use of `ISpRecoContext::Bookmark` with an "auto-pause" bookmark.

```
HRESULT hr = S_OK;
```

```
// setup the recognition context and grammar
// ...

// start listening for recognitions
hr = cpRecoGrammar->SetRuleState( MY_RULE, NULL, SPRS_AC
// Check hr

hr = cpRecoContext->Bookmark( SPBO_PAUSE, SP_STREAMPOS_A

// get the bookmark event in a CSpEvent object
// ...

// assert that the recognition context paused after the
SPDBG_ASSERT(spEvent.IsPaused());

// Since the context was paused from the "auto-pause" ru.
hr = cpRecoContext->Resume( NULL );
// Check hr
```



ISpRecoContext::SetAdaptationData

ISpRecoContext::SetAdaptationData passes a block of text to the SR engine which can be used to adapt the active language models.

```
HRESULT SetAdaptationData(  
    const WCHAR    *pAdaptationData,  
    const ULONG    cch  
);
```

Parameters

pAdaptationData
[in] The string to adapt.

cch
[in] The number of characters in *pAdaptationData*.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pAdaptationData</i> is invalid or <i>cch</i> equals zero.
E_OUTOFMEMORY	Exceeded available memory.
SPERR_SR_ENGINE_EXCEPTION	An exception was thrown by the SR engine during ISpSREngine::SetAdaptationData
FAILED(hr)	Appropriate error message.

Remarks

An application can improve recognition accuracy for dictating

uncommon words, or uncommon word groupings, by training the SR engine for the new words, or word groupings, by creating or obtaining typical text and send the results to the engine using `::SetAdaptationData`.

For example, a word processing application train the engine on previously created text documents. Similarly, an e-mail or collaboration application could train the SR engine on previously sent e-mail or collaborated documents. After training, the SR engine could perform better within specific domains of information, and improve the application's and SAPI's personalization experience.

The SAPI middleware component (and Microsoft) do not store this information - instead it is passed directly to the SR engine using [ISpSREngine::SetAdaptationData](#). The SR engine (vendor) determines what to do with the string data, including how the data is persisted and for how long. Some SR engines may require a significant amount of time to process the string data, so the application may want to break up the data in smaller chunks to send individually.

Applications that use `SetAdaptationData` should break the data into small (1K or less) blocks, call `SetAdaptationData`, and then wait for an [SPEI_ADAPTATION](#) event before sending the next small block of data. This method should not be used with large buffers, because calling `SetAdapataionData` with buffers larger than 32K can cause unpredictable results.

When the SR engine is ready to receive more string data to adapt its language model, it can fire the `SPEI_ADAPTATION` event to the application.

Example

The following code snippet illustrates the use of `ISpRecoContext::SetAdaptationData`

```
HRESULT hr = S_OK;
```

```
// get the "training" data, and break it into manageable
// ...

// set interest in the adaptation event
hr = cpRecoContext->SetInterest(SPFEI(SPEI_ADAPTATION), 0);
// Check hr

// adapt to each chunk of data
for (int i = 0; i < iCountOfDataChunk; i++)
{
    // send each chunk of data the engine
    hr = cpRecoContext->SetAdaptationData(ppwszAdaptation);
    // Check hr

    // wait for the engine to ask for more data
    hr = cpRecoContext->WaitForNotifyEvent(PROCESSING_WAIT);
    // Check hr
}

// SR Engine has adapted its language model to data
```



ISpRecoContext::Pause

ISpRecoContext::Pause pauses the engine object to synchronize with the SR engine.

The SR engine pauses at its synchronization point to allow grammars and rule states to be changed freely. The engine remains paused until the [Resume](#) method is called.

The caller must call Resume once for every call that is made to Pause.

```
HRESULT Pause(  
    DWORD    dwFlags  
);
```

Parameters

dwFlags
[in] Reserved, must be zero.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>dwFlags</i> is not set to zero.

Remarks (when using ISpMMSysAudio as audio input)

Pausing the SR engine will stop recognition, but input audio will continue to be collected and stored by SAPI in an audio buffer. After the application is done with the state change, it should call [ISpRecoContext::Resume](#). SAPI will automatically feed the buffered audio data into the SR engine, ensuring that no real-

time audio data is lost and that the user experience is not interrupted.

However, the SAPI audio buffer has a static limit (see [ISpMMSysAudio::Read](#)) to prevent large amounts of system memory from being consumed by SAPI applications or SR engines. Therefore, Pause should be used by the SR engine for very short periods of time for state changes (e.g., updating grammar, or rule states). Pausing the SR engine will affect all recognition contexts connected to that SR engine including other Speech applications currently running.

If the SR engine is paused too long, and the audio buffer is filled, a buffer overflow will occur. The application can detect this error by setting an event interest in SPEI_END_SR_STREAM (see [SPEVENTENUM](#)), and checking the *LPARAM* of the [SPEVENT](#) structure (see [CSpEvent::EndStreamResult](#)).

SAPI will automatically attempt to restart the SR Engine's recognition thread once the final Resume has been called. Consequently, the audio data collected between the point when the buffer overflow occurred, and when the stream was reactivated, will be completely lost. This would result in a less than optimal user experience, and have a negative effect on all running speech applications, the SR engine, and SAPI.

The following code snippet illustrates the use of [ISpRecoContext::Pause](#)

```
HRESULT hr = S_OK;

// setup the recognition context
// ...

// pause the context so that event notifications are not
hr = cpRecoContext->Pause( NULL );
// Check hr

// [quickly] perform the processing - see ISpRecoContext
// ...
```

```
hr = cpRecoContext->Resume( NULL );  
// Check hr
```

```
// applications will start receiving event notifications
```



ISpRecoContext::Resume

ISpRecoContext::Resume releases the SR engine from the paused state and restarts the recognition process.

```
HRESULT Resume (  
    DWORD dwReserved  
);
```

Parameters

dwReserved
[in] Reserved, must be zero.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>dwFlags</i> is not set to zero.

Remarks

This method must be called after a call to [ISpRecoContext::Pause](#), a bookmark event occurs that pauses the recognition engine, or an auto-pause rule is recognized (see [ISpRecoGrammar::SetRuleState](#)).

The caller must call `Resume` once for every call that is made to [ISpRecoContext::Pause](#).

Example

The following code snippet illustrates the use of `ISpRecoContext::Resume` after a call to [ISpRecoContext::Pause](#)

```
HRESULT hr = S_OK;
```

```

// setup the recognition context
// ...

// pause the context so that event notifications are not
hr = cpRecoContext->Pause( NULL );
// Check hr

// [quickly] perform the processing - see ISpRecoContext
// ...

hr = cpRecoContext->Resume( NULL );
// Check hr

// applications will start receiving event notifications

```

The following code snippet illustrates the use of `ISpRecoContext::Resume` with an "auto-pause" rule.

```

HRESULT hr = S_OK;

// setup the recognition context and grammar
// ...

// activate a top-level rule as an "auto-pause" rule
hr = cpRecoGrammar->SetRuleState( MY_AUTOPAUSE_RULE, NULL );
// Check hr

// get the recognition event for MY_AUTOPAUSE_RULE in a (
// ...

// assert that the recognition context paused after the
SPDBG_ASSERT(spEvent.IsPaused());

// deactivate the "auto-pause" rule
hr = cpRecoGrammar->SetRuleState( MY_AUTOPAUSE_RULE, NULL );
// Check hr

// activate the second rule
hr = cpRecoGrammar->SetRuleState( MY_SECOND_RULE, NULL, S_FALSE );
// Check hr

```

```
// Since the context was paused from the "auto-pause" ru.  
hr = cpRecoContext->Resume( NULL );  
// Check hr  
  
// get the second recognition...
```



ISpRecoContext::SetVoice

ISpRecoContext::SetVoice sets the associated ISpVoice to an object.

```
HRESULT SetVoice(  
    ISpVoice *pVoice,  
    BOOL fAllowFormatChanges  
);
```

Parameters

pVoice

[in] The voice interface to be associated. If NULL, the currently associated Voice is Released.

fAllowFormatChanges

[in] Boolean allowing the voice format alteration by the engine. See Remarks section.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pVoice</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Remarks

If *fAllowFormatChanges* is TRUE, the Voice's output format will be changed to be the same format as the associated SR engine's audio input format (see [ISpRecognizer](#) and [ISpSREngine::GetInputAudioFormat](#)). However, if this voice object has already been bound to a stream which has specific

format, the voice's format will not be changed to the SR engine's audio input format even if *fAllowFormatChanges* is true.

Using the same audio format for input and output source is useful for sound cards that do not support full-duplex audio (i.e., input format must match output format). If the input format quality is lower than the output format quality, the output format quality will be reduced to equal the input quality.

After calling `ISpRecoContext::SetVoice`, an application that calls [ISpRecoContext::GetVoice](#) will retrieve the originally "set" `ISpVoice` interface pointer.

Example

The following code snippet illustrates the use of `ISpRecoContext::SetVoice` and "barge-in" setup.

```
HRESULT hr = S_OK;

// create a shared recognition context
hr = cpRecoContext.CoCreateInstance(CLSID_SpSharedRecoCo
// Check hr

// create a voice
hr = cpVoice.CoCreateInstance(CLSID_SpVoice);
// Check hr

// associate the voice with the context (with same audio
hr = cpRecoContext->SetVoice(cpVoice, TRUE);
// Check hr

// tell the associated Voice to stop speaking when the S
hr = cpRecoContext->SetVoicePurgeEvent(SPFEI(SPEI_SOUND_
// Check hr
```



ISpRecoContext::GetVoice

ISpRecoContext::GetVoice retrieves a reference to an [ISpVoice](#) object that is associated with the [ISpRecoContext](#) object.

```
HRESULT GetVoice(  
    ISpVoice    **ppVoice  
);
```

Parameters

ppVoice

[in] Address of the [ISpVoice](#) interface. `IUnknown::Release` must be called on the `ISpVoice` interface when finished.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Invalid pointer.
FAILED(hr)	Appropriate error message.

Remarks

If an application previously called [ISpRecoContext::SetVoice](#) on the same `ISpRecoContext` object, the `Voice` interface retrieved from `GetVoice` will match that of the `SetVoice` call. `Release` must still call the `ISpVoice` reference for each `GetVoice` call, even though the interface pointer is the same.

The output format of the `ISpVoice` will be the same format as the associated audio input format of the SR engine (see [ISpRecognizer](#) and [ISpSREngine::GetInputAudioFormat](#)). Using the same audio format for input and output source is useful for

sound cards that do not support full-duplex audio (i.e., input format must match output format). If the input format quality is lower than the output format quality, the output format quality will be down-sampled to the lower quality.

Applications implementing a "barge-in" type functionality will need to tie the Voice object to the SR object. Applications can also use `ISpRecoContext::GetVoice` (see [ISpRecoContext::SetVoicePurgeEvent](#)).

Related Samples

The [CoffeeS3](#) Sample application uses `ISpRecoContext::GetVoice` and the "barge-in" functionality.

Example

The following code snippet illustrates the use of `ISpRecoContext::GetVoice` and "barge-in" setup

```
HRESULT hr = S_OK;

// create a shared recognition context
hr = cpRecoContext.CoCreateInstance(CLSID_SpSharedRecoCo
// Check hr

// create a voice from the context (with same audio form
hr = cpRecoContext->GetVoice(&cpVoice);
// Check hr

// tell the associated Voice to stop speaking when the S
hr = cpRecoContext->SetVoicePurgeEvent(SPFEI(SPEI_SOUND_
// Check hr
```



ISpRecoContext::SetVoicePurgeEvent

ISpRecoContext::SetVoicePurgeEvent sets the SR engine events that stop audio output, and purges the current speaking queue.

```
HRESULT SetVoicePurgeEvent(  
    ULONGLONG    ullEventInterest  
);
```

Parameters

ullEventInterest

[in] The set of flags indicating the event interest(s). The event interest(s) must be in the set of speech recognition events (i.e., between SPEI_MIN_SR and SPEI_MAX_SR) (see [SPEVENTENUM](#) and [SPFEI_ALL_SR_EVENTS](#))

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more of the event interests set is not allowed.
FAILED(hr)	Appropriate error message.

Remarks

The ISpRecoContext event interest will be updated to include the Voice Purge Events (see [ISpEventSource::SetInterest](#)).

To find the current Voice Purge Event(s), use [ISpRecoContext::GetVoicePurgeEvent](#)

Applications can call SetVoicePurgeEvent when implementing "barge-in" type functionality. For example, when a user calls a

telephony server, and the server uses TTS Voice prompts, the Voice should stop speaking when the user is speaking. The application would want the associated Voice object of the ISpRecoContext (see [ISpRecoContext::GetVoice](#)) to stop and purge when the SR engine hears a sound (see [SPEI_SOUND_START](#)).

Example

The following code snippet illustrates the use of `ISpRecoContext::SetVoicePurgeEvent` and "barge-in" setup

```
HRESULT hr = S_OK;  
  
// create a shared recognition context  
hr = cpRecoContext.CoCreateInstance(CLSID_SpSharedRecoCo  
// Check hr  
  
// create a voice from the context (with same audio form  
hr = cpRecoContext->GetVoice(&cpVoice);  
// Check hr  
  
// tell the associated Voice to stop speaking when the S  
hr = cpRecoContext->SetVoicePurgeEvent(SPFEI(SPEI_SOUND_  
// Check hr
```



ISpRecoContext::GetVoicePurgeEvent

ISpRecoContext::GetVoicePurgeEvent retrieves the set of SR engine events that stop audio output, and purges the current speaking queue. The events are set by [ISpRecoContext::SetVoicePurgeEvent](#).

```
HRESULT GetVoicePurgeEvent(  
    ULONGLONG    *pullEventInterest  
);
```

Parameters

pullEventInterest

[out] The set of flags indicating the event interests. The event interests will be a member of the SR event set (see [SPFEI_ALL_SR_EVENTS](#)).

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pullEventInterest</i> is invalid or bad.
FAILED(hr)	Appropriate error message.



ISpRecoContext::SetContextState

ISpRecoContext::SetContextState sets the state of the recognition context.

```
HRESULT SetContextState(  
    SPCONTEXTSTATE eContextState  
);
```

Parameters

eContextState

[in] The SPCONTEXTSTATE enumeration sequence specifying the recognition context state.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>eContextState</i> is not one of the correct types.
FAILED(hr)	Appropriate error message.

Remarks

The default recognition context state for an ISpRecoContext object is SPCS_ENABLED.

Applications can use ISpRecoContext::SetContextState to toggle sets of grammars. For example, a multi-document interface application that uses a different ISpRecoContext object for each document could toggle the context states as each document gained and lost the focus.

Applications can use [ISpRecoContext::GetContextState](#) to query the context state.



ISpRecoContext::GetContextState

ISpRecoContext::GetContextState retrieves the state of the recognition context.

```
HRESULT GetContextState(  
    SPCONTEXTSTATE *peContextState  
);
```

Parameters

peContextState

[out] Address of the SPCONTEXTSTATE enumeration that receives the context state information.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	One of the pointers is invalid or bad.
FAILED(hr)	Appropriate error message.

Remarks

The default recognition context state for an ISpRecoContext object is SPCS_ENABLED.

Applications can use [ISpRecoContext::SetContextState](#) to set the context state.



ISpRecoGrammar

The ISpRecoGrammar interface enables applications to manage the words and phrases that the SR engine will recognize.

A single SpRecognizer object can have multiple SpRecoContext objects associated with it. And similarly, a single SpRecoContext object can have multiple SpRecoGrammar objects associated with it. Using a one-to-many relationship with SpRecoContext objects and SpRecoGrammar objects allows applications to separate types of recognizable phrases and content into separate objects for clearer application logic. Each SpRecoGrammar object can also have a context-free grammar (CFG) and a dictation grammar loaded simultaneously (e.g., use the CFG if possible, but back off to dictation if CFG fails to parse).

See [Designing Grammar Rules](#) for examples of how to create context-free grammars.

Methods in Vtable Order

ISpRecoGrammar Methods	Description
ISpGrammarBuilder interface	Inherits from ISpGrammarBuilder and all those methods are accessible from an ISpRecoGrammar object.
GetGrammarId	Retrieves the grammar identifier associated with the application.
GetRecoContext	Retrieves the context object that created this grammar.
LoadCmdFromFile	Loads a command and

	control grammar from a file.
<u>LoadCmdFromObject</u>	Loads a command and control grammar from a COM object.
<u>LoadCmdFromResource</u>	Loads a command and control grammar from a Win32 resource.
<u>LoadCmdFromMemory</u>	Loads a command and control grammar from memory.
<u>LoadCmdFromProprietaryGrammar</u>	Loads an engine proprietary format command and control grammar.
<u>SetRuleState</u>	Activates or deactivates a rule by its rule name.
<u>SetRuleIdState</u>	Activates or deactivates a rule by its rule ID.
<u>LoadDictation</u>	Loads and initializes a dictation topic.
<u>UnloadDictation</u>	Unloads the active dictation topic from the grammar.
<u>SetDictationState</u>	Sets a dictation state to active or inactive.
<u>SetWordSequenceData</u>	Sets a word sequence buffer in the SR engine.
<u>SetTextSelection</u>	Sets the current text selection and insertion point information.
<u>IsPronounceable</u>	Determines if the word has a pronunciation.

<u>SetGrammarState</u>	Sets the grammar state.
<u>SaveCmd</u>	Allows applications using dynamic grammars to save the current grammar state to a stream.
<u>GetGrammarState</u>	Retrieves recognition grammar state information.



ISpRecoGrammar::GetGrammarId

ISpRecoGrammar::GetGrammarId retrieves the identifier associated with the grammar when the grammar was created.

The grammar ID is set by the application by calling [ISpRecoContext::CreateGrammar](#).

```
HRESULT GetGrammarId(  
    ULONGLONG    *pullGrammarId  
);
```

Parameters

pullGrammarId

[out] Address of a ULONGLONG variable to receive the grammar ID.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pullGrammarId</i> is invalid or bad.

Remarks

The grammar ID will be set in the [SPPHRASE](#) object when a recognition is sent to the application (see [ISpPhrase::GetPhrase](#)).



ISpRecoGrammar::GetRecoContext

ISpRecoGrammar::GetRecoContext retrieves the ISpRecoContext object that created this grammar. If this method succeeds, the application using this method must call Release() on the SpRecoContext object returned.

```
HRESULT GetRecoContext(  
    ISpRecoContext **ppRecoCtxt  
);
```

Parameters

ppRecoCtxt

[out] Address of a pointer to an [ISpRecoContext](#) interface that receives the recognition context object pointer.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppRecoCtxt</i> is invalid or bad.



ISpRecoGrammar::LoadCmdFromFile

ISpRecoGrammar::LoadCmdFromFile loads a SAPI 5 command and control grammar from a file. The file can either be a compiled or uncompiled grammar file. To modify the rules of the grammar after it has been loaded, specify `SPLO_DYNAMIC` for the *Options* parameter, otherwise specify the `SPLO_STATIC` flag.

```
HRESULT LoadCmdFromFile(  
    const WCHAR      *pszFileName,  
    SPLOADOPTIONS    Options  
);
```

Parameters

pszFileName

[in, string] The name of the file containing the command and control grammar. SAPI 5 support loading of compiled and static grammars using URL.

Options

[in] Flag of type [SPLOADOPTIONS](#) indicating whether the grammar will be modified dynamically.

Return values

Value	Description
<code>S_OK</code>	Function completed successfully.
<code>E_INVALIDARG</code>	<i>pszFileName</i> is invalid or bad. Alternatively, <i>Options</i> is neither <code>SPLO_STATIC</code> nor <code>SPLO_DYNAMIC</code> .
<code>FAILED(hr)</code>	Appropriate error message.



ISpRecoGrammar::LoadCmdFromObject

ISpRecoGrammar::LoadCmdFromObject loads a CFG from a COM object. The COM object must be located inside of a Windows DLL.

```
HRESULT LoadCmdFromObject(  
    REFCLSID        rcid,  
    const WCHAR     *pszGrammarName,  
    SPLOADOPTIONS   Options  
);
```

Parameters

rcid

[in] The reference class ID of the object containing the command.

pszGrammarName

[in, string] The grammar name of the object containing the command.

Options

[in] Flag of type [SPLOADOPTIONS](#) indicating whether the file should be loaded statically or dynamically.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszGrammarName</i> is invalid or bad. Alternatively, <i>Options</i> is neither SPLO_STATIC nor SPLO_DYNAMIC.
FAILED(hr)	Appropriate error message.

Remarks

When an application calls `::LoadCmdFromObject`, the currently loaded CFG or proprietary grammar will be unloaded.

See also [ISpCFGInterpreter::InitGrammar](#).



ISpRecoGrammar::LoadCmdFromResource

ISpRecoGrammar::LoadCmdFromResource loads a command and control grammar from a Win32 resource.

```
HRESULT LoadCmdFromResource(  
    HMODULE          hModule,  
    const WCHAR     *pszResourceName,  
    const WCHAR     *pszResourceType,  
    WORD            wLanguage,  
    SPLOADOPTIONS   Options  
);
```

Parameters

hModule

[in] Handle to the module whose file name is being requested. If this parameter is NULL, it passes back the path for the file containing the current process.

pszResourceName

[in, string] The name of the resource.

pszResourceType

[in, string] The type of the resource.

wLanguage

[in] The language ID.

Options

[in] Flag of type [SPLOADOPTIONS](#) indicating whether the file should be loaded statically or dynamically.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Either <i>pszResourceName</i> or <i>pszResourceType</i> is invalid or bad. It may also indicate <i>hModule</i> could not be found. Alternatively, <i>Options</i> is neither SPLO_STATIC nor SPLO_DYNAMIC.
FAILED(hr)	Appropriate error message.

Remarks

When an application calls `::LoadCmdFromResource`, the currently loaded context-free grammar or proprietary grammar will be unloaded.

The CFG resource must be a compiled SAPI 5 binary version of a context-free grammar (see [ISpGrammarCompiler::CompileStream](#)).

Example

```
HRESULT hr = S_OK;

// create a new grammar object
hr = cpRecoContext->CreateGrammar(GRAM_ID, &cpRecoGramn
// Check hr

// load a CFG resource from the current module, named SRGRAMM
hr = cpRecoGrammar->LoadCmdFromResource(hModule,
    (const WCHAR*)MAKEINTRESOURCE(
    L"SRGRAMMAR",
    ::SpGetUserDefaultUILanguage()
    , SPLO_STATIC);
```

// Check hr



ISpRecoGrammar::LoadCmdFromMemory

ISpRecoGrammar::LoadCmdFromMemory loads a compiled CFG binary from memory.

```
HRESULT LoadCmdFromMemory(  
    const SPBINARYGRAMMAR *pBinaryData,  
    SPLOADOPTIONS Options  
);
```

Parameters

pBinaryData

[in] The serialized header buffer of type [SPBINARYGRAMMAR](#).

Options

[in] Flag of type [SPLOADOPTIONS](#) indicating whether the file should be loaded statically or dynamically.

Remarks

When an application calls `::LoadCmdFromMemory`, the currently loaded CFG or proprietary grammar will be unloaded.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Either <i>pBinaryData</i> or one of its members is invalid or bad. It may also indicate <i>pBinaryData->FormatId</i> is not SPGDF_ContextFree. Alternatively, <i>Options</i> is neither SPLO_STATIC nor SPLO_DYNAMIC.
FAILED(hr)	Appropriate error message.

Example

The following code snippet illustrates how to use `ISpRecoGrammar::LoadCmdFromMemory` to serialize the CFG from one `SpRecoGrammar` object and deserialize it into another `SpRecoGrammar` object.

```
HRESULT hr = S_OK;

// ... build and use a SpRecoGrammar object

// create a Win32 global stream
hr = ::CreateStreamOnHGlobal(NULL, true, &cpHStream);
// Check hr

// save the current grammar to the global stream
hr = cpRecoGrammar->SaveCmd(cpHStream, NULL);
// Check hr

// create the second grammar to deserialize into
hr = cpRecoContext->CreateGrammar(0, &cpReloadedGrammar);
// Check hr

// get a handle to the stream with the serialized grammar
::GetHGlobalFromStream(cpHStream, &hGrammar);
// Check hr

// deserialize the CFG into a new grammar object
hr = cpReloadedGrammar->LoadCmdFromMemory((SPBINARYG
// Check hr
```



ISpRecoGrammar::LoadCmdFromPropriet

ISpRecoGrammar::LoadCmdFromProprietaryGrammar

loads a proprietary grammar.

```
HRESULT LoadCmdFromProprietaryGrammar(  
    REFGUID          rguidParam,  
    const WCHAR      *pszStringParam,  
    const void       *pvDataParam,  
    ULONG            cbDataSize,  
    SPOLOADOPTIONS    Options  
);
```

Parameters

rguidParam

[in] Unique identifier of the grammar. The GUID will be used by the application and the SR engine to uniquely identify the SR engine for verifying support.

pszStringParam

[in, string] The null-terminated string command. The string can be used by the application and the SR engine to specify which part of a grammar to utilize.

pvDataParam

[in] Additional information for the process. SAPI will handle the marshaling of the data to the SR engine.

cbDataSize

[in] The size, in bytes, of *pvDataParam*. SAPI will handle the marshaling of the data to the SR engine.

Options

[in] Flag of type SPLOADOPTIONS indicating whether the file should be loaded statically or dynamically. This value must be SPLO_STATIC.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszStringParam</i> or <i>pvDataParam</i> is invalid or bad. Alternatively, <i>Options</i> is not SPLO_STATIC.
FAILED(hr)	Appropriate error message.

Remarks

When an application calls `::LoadCmdFromProprietaryGrammar`, the currently loaded CFG or proprietary will be unloaded.

Applications should use `::LoadCmdFromProprietaryGrammar` when using a proprietary grammar format that the SR engine supports (see [ISpSREngine::LoadProprietaryGrammar](#)). If the current SR engine does not support the proprietary grammar format (specified using *rguidParam* and *pszStringParam*), the SR engine may return E_NOTIMPL.

Example

```
HRESULT hr = S_OK;
```

```
// create a new grammar object
```

```
hr = cpRecoContext->CreateGrammar(GRAM_ID, &cpRecoGramm
```

```
// Check hr
```

```
// load our proprietary grammar
```

```
hr = cpRecoGrammar->LoadCmdFromProprietaryGrammar(GUID_
```

```
// Check hr
```



ISpRecoGrammar::SetRuleState

ISpRecoGrammar::SetRuleState activates or deactivates a rule by its rule name.

```
HRESULT SetRuleState(  
    const WCHAR    *pszName,  
    void           *pReserved,  
    SPRULESTATE    NewState  
);
```

Parameters

pszName

[in, string] Address of a null-terminated string containing the rule name. If NULL, all rules with attribute [SPRAF_TopLevel](#) and [SPRAF_Active](#) and set (at rule creation time) are affected.

pReserved

Reserved. Do not use; must be NULL.

NewState

[in] Flag of type [SPRULESTATE](#) indicating the new rule state. See Remarks section.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszName</i> is invalid or bad. Alternatively, <i>pReserved</i> is non-NULL.
SP_STREAM_UNINITIALIZED	ISpRecognizer::SetInput has

	not been called with the InProc recognizer
SPERR_UNINITIALIZED	The object has not been properly initialized.
SPERR_UNSUPPORTED_FORMAT	Audio format is bad or is not recognized. Alternatively, the device driver may be busy by another application and cannot be accessed.
SPERR_NOT_TOPLEVEL_RULE	The rule <i>pszName</i> exists, but is not a top-level rule.
FAILED(hr)	Appropriate error message.

Remarks

The rule name is specified in the XML grammar (using the rule NAME tag), or when [ISpGrammarBuilder::GetRule](#) is called.

See also [ISpSREngine::RuleNotify](#) for information on the how SAPI notifies the SR engine.

An application can use the SPRS_ACTIVE_WITH_AUTO_PAUSE state to pause the engine after each CFG recognition is sent. The application must reactivate the SR engine (see [ISpRecoContext::Resume](#)) to prevent the loss of input audio data (see [ISpSREngineSite::Read](#) and SPERR_AUDIO_BUFFER_OVERFLOW).

By default, the recognizer state ([SPRECOSTATE](#)) is SPRST_ACTIVE, and the recognition will begin as soon as one or more rule are activated. Consequently, an application should not activate the rule state until it is prepared to receive recognitions. An application can also disable the SpRecoContext object (see [ISpRecoContext::SetContextState](#)) or SpRecoGrammar objects (see [ISpRecoGrammar::SetGrammarState](#)) to prevent recognitions from being fired for active rules.

If the recognizer state is `SPRST_ACTIVE`, SAPI will first attempt to open the audio input stream when dictation (or a rule) is activated. Consequently, if the audio device is already in use by another application, or the stream fails to open, the failure code will be returned using `::SetRuleState`. The application should handle this failure gracefully.

If an application uses an InProc recognizer, it must call [ISpRecognizer::SetInput](#) with a non-NULL setting before the recognizer will return recognitions, regardless of how many rules are active.

Example

The following snippet loads a grammar, activate a single rule ("playcard") and then immediately deactivates it.

```
HRESULT hr;

// create a grammar object
hr = cpRecoContext->CreateGrammar(GRAM_ID, &cpRecoGra
//Check return value

// activate the rule
hr = cpRecoGrammar->SetRuleState(L"playcard", NULL, SPRS_
//Check return value

//Deactivate the rule
hr = cpRecoGrammar->SetRuleState(L"playcard", NULL, SPRS_
//Check return value
```



ISpRecoGrammar::SetRuleIdState

ISpRecoGrammar::SetRuleIdState activates or deactivates a rule by its rule ID.

```
HRESULT SetRuleIdState(  
    ULONG          ulRuleId,  
    SPRULESTATE   NewState  
);
```

Parameters

ulRuleId

[in] Value specifying the grammar rule identifier. If zero, all rules with attribute [SPRAF_TopLevel](#) and [SPRAF_Active](#) and set (at rule creation time) are affected.

NewState

[in] Flag of type [SPRULESTATE](#) indicating the new rule state.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>dwRuleId</i> is invalid.
SP_STREAM_UNINITIALIZED	ISpRecognizer::SetInput has not been called with the InProc recognizer.
SPERR_UNINITIALIZED	The object has not been properly initialized.
SPERR_UNSUPPORTED_FORMAT	Audio format is bad or is not recognized. Alternatively, the device driver may be busy by

	another application and cannot be accessed.
SPERR_DEVICE_BUSY	The audio
SPERR_NOT_TOPLEVEL_RULE	The rule ID <i>ulRuleId</i> exists, but is not a top-level rule.
FAILED(hr)	Appropriate error message.

Remarks

An application can use the `SPRS_ACTIVE_WITH_AUTO_PAUSE` state to pause the engine after each dictation recognition is sent. The application must reactivate the SR engine (see [ISpRecoContext::Resume](#)) to prevent the loss of input audio data (see [ISpSREngineSite::Read](#) and `SPERR_AUDIO_BUFFER_OVERFLOW`).

The rule ID is specified in the XML grammar (using the rule ID tag), or when [ISpGrammarBuilder::GetRule](#) is called.

See also [ISpSREngine::RuleNotify](#) for information on the how SAPI notifies the SR engine.

By default, the recognizer state ([SPRECOSTATE](#)) is `SPRST_ACTIVE`, which means that recognition will begin as soon as dictation is activated. Consequently, an application should not activate the dictation state until it is prepared to receive recognitions. An application can also disable the `SpRecoContext` object (see [ISpRecoContext::SetContextState](#)) or `SpRecoGrammar` objects (see [ISpRecoGrammar::SetGrammarState](#)) to prevent recognitions from being fired for active dictation topics.

If the recognizer state is `SPRST_ACTIVE`, SAPI will first attempt to open the audio input stream when a rule (or dictation) is activated. Consequently, if the audio device is already in use by another application, or the stream fails to open, the failure code will be returned using `::SetRuleIdState`. The application should handle this failure gracefully.

If an application uses an InProc recognizer, it must call [ISpRecognizer::SetInput](#) with a non-NULL setting before the recognizer will return recognitions, regardless of the dictation topic state.

Example

The following code snippet illustrates the use of `ISpRecoGrammar::SetRuleIdState` by programmatically creating a rule and activating it

```
HRESULT hr = S_OK;

// create a new rule
SPSTATEHANDLE hStateTravel;
hr = cpRecoGrammar->GetRule(MYRULENAME, MY_RULE_ID);
// Check hr

// .. add word transitions...

// activate the rule by its id
hr = cpRecoGrammar->SetRuleIdState(MY_RULE_ID, SPRS_ACT);
// Check hr
```



ISpRecoGrammar::LoadDictation

ISpRecoGrammar::LoadDictation loads a dictation topic into the SpRecoGrammar object and the SR engine.

See also [ISpSREngine::OnCreateGrammar](#).

```
HRESULT LoadDictation(  
    const WCHAR    *pszTopicName,  
    SPLOADOPTIONS Options  
);
```

Parameters

pszTopicName

[in, optional, string] The null-terminated string containing the topic name. If NULL, the general dictation is loaded. See Remarks section.

Options

[in] Flag of type [SPLOADOPTIONS](#) indicating whether the file should be loaded statically or dynamically. This value must be SPLO_STATIC.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pszTopicName</i> is invalid or bad. Alternatively, <i>Options</i> is not SPLO_STATIC.
FAILED(hr)	Appropriate error message.

Remarks

SAPI currently defines one specialized dictation topic: SPTOPIC_SPELLING. SR engines are not required to support specialized dictation topic (including spelling).

See the SR engine vendor for information on what specialized dictation topics if any are supported.

Example

The following code snippet illustrates the use of ISpRecoGrammar::LoadDictation to load a spelling topic and activate it.

```
HRESULT hr = S_OK;

// create a grammar object
hr = cpRecoContext->CreateGrammar(GRAM_ID, &cpRecoGramm
// Check hr

// load the general dictation topic
hr = cpRecoGrammar->LoadDictation(NULL, SPLO_STATIC);
// Check hr

// activate the dictation topic to receive recognitions
hr = cpRecoGrammar->SetDictationState(SPRS_ACTIVE);
// check hr
```



ISpRecoGrammar::UnloadDictation

ISpRecoGrammar::UnloadDictation unloads the active dictation topic from the grammar.

```
HRESULT UnloadDictation ( void );
```

Parameters

None.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.

Remarks

If an application uses a CFG with dictation tags, and then unloads the dictation grammar component, the dictation tags will default to the generic dictation topic (see [ISpRecoGrammar::LoadDictation](#)).



ISpRecoGrammar::SetDictationState

ISpRecoGrammar::SetDictationState sets the dictation topic state.

The dictation topic is specified by calling [ISpRecoGrammar::LoadDictation](#).

See also [ISpSREngine::SetSLMState](#) for information on how SAPI notifies the SR engine.

```
HRESULT SetDictationState(  
    SPRULESTATE NewState  
);
```

Parameters

NewState

[in] Flag of type [SPRULESTATE](#) indicating the new state of dictation. See Remarks section

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>NewState</i> is not an acceptable value.
SP_STREAM_UNINITIALIZED	ISpRecognizer::SetInput has not been called with the InProc SR engine
SPERR_UNINITIALIZED	A dictation is not currently loaded.
SPERR_UNSUPPORTED_FORMAT	Audio format is bad or is not recognized. Alternatively, the device driver may be busy by

	another application and cannot be accessed.
FAILED(hr)	Appropriate error message.

Remarks

An application can use the `SPRS_ACTIVE_WITH_AUTO_PAUSE` state to pause the engine after each dictation recognition is sent. The application must reactivate the SR engine (see [ISpRecoContext::Resume](#)) to prevent the loss of input audio data (see `ISpSREngineSite::Read` and `SPERR_AUDIO_BUFFER_OVERFLOW`).

By default, the recognizer state ([SPRECOSTATE](#)) is `SPRST_ACTIVE`, which means that recognition will begin as soon as dictation is activated. Consequently, an application should not activate the dictation state until it is prepared to receive recognitions. An application can also disable the `SpRecoContext` object (see [ISpRecoContext::SetContextState](#)) or `SpRecoGrammar` objects (see [ISpRecoGrammar::SetGrammarState](#)) to prevent recognitions from being fired for active dictation topics.

If the recognizer state is `SPRST_ACTIVE`, SAPI will first attempt to open the audio input stream when dictation (or a rule) is activated. Consequently, if the audio device is already in use by another application, or the stream fails to open, the failure code will be returned using `::SetDictationState`. The application should handle this failure gracefully.

If an application uses an InProc recognizer, it must call [ISpRecognizer::SetInput](#) with a non-NULL setting before the recognizer will return recognitions, regardless of the dictation topic state.

Example

The following code snippet illustrates the use of `ISpRecoGrammar::SetDictationState` to load a spelling topic and

activate it.

```
HRESULT hr = S_OK;
```

```
// create a grammar object
```

```
hr = cpRecoContext->CreateGrammar(GRAM_ID, &cpRecoGramn
```

```
// Check hr
```

```
// load the general dictation topic
```

```
hr = cpRecoGrammar->LoadDictation(NULL, SPLO_STATIC);
```

```
// Check hr
```

```
// activate the dictation topic to receive recognitions
```

```
hr = cpRecoGrammar->SetDictationState(SPRS_ACTIVE);
```

```
// check hr
```



ISpRecoGrammar::SetWordSequenceData

ISpRecoGrammar::SetWordSequenceData sets a word sequence buffer in the SR engine.

The command and control grammar can refer to any subsequence of words in this buffer using the <TEXTBUFFER> tag, or the SPRULETRANS_TEXTBUFFER special transition type in [ISpGrammarBuilder::AddRuleTransition\(\)](#).

```
HRESULT SetWordSequenceData(  
    const WCHAR    *pText,  
    ULONG          cchText,  
    const SPTXTSELECTIONINFO *pInfo  
);
```

Parameters

pText

[in] Buffer containing the text to search for possible word sequences. The buffer is double-NULL terminated. The whole buffer could be separated into different groups by '\0'. Any sub-sequence of words in the same group is recognizable, any sub-sequence of words across different groups is not recognizable. The word could be in simple format or complex format: /disp/lex/pron. The SR engines determine where to break words and when to normalize text for better performance. For example, if the buffer displays: "please play\0this new game\0\0", "please play" is recognizable, while "this new game" is not recognizable.

cchText

[in] The number of characters (WCHAR) in pText.

pInfo

[optional, in] Address of the SPTXTSELECTIONINFO structure that contains the selection information. If NULL, the SR engine will use the entire contents of *pText*.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
FAILED(hr)	Appropriate error message.

Remarks

An application that has a text box could enable the user to speak commands into the text box to edit the text. One way to design this functionality would be to create a CFG which supports such commands as "cut the text *", "bold the text *", or "italicize the words *". The grammar would then use a TEXTBUFFER tag in place of the * which would enable the SR engine to recognize the text buffer information. At run time, the application would update the SR engine's view of the text buffer using `::SetWordSequenceData`. So if a user had the text "*hello world*" in the text box, the SR engine could recognize "bold the text *world*".

See also [ISpRecoGrammar::SetTextSelection](#) for information on how to update the text selection information independent of the word sequence data.

See also [ISpSREngine::SetWordSequenceData](#) for information on how SAPI passes the word sequence data to the SR engine.

The SR engine must support text buffer features. Check for the presence of the TextBuffer attribute for the SR engine. Microsoft SR ASR engines support these features although there is no requirement that other manufacturers engines need to. See Recognizers in [Object Tokens and Registry Settings](#) for more

information.

Example

The following code snippet illustrates how an application could send a text buffer to the SR engine using `ISpRecoGrammar::SetWordSequenceData`.

```
HRESULT hr = S_OK;
```

```
// place the contents of text buffer into pwszCoMem and the length
```

```
SPTXTSELECTIONINFO tsi;
```

```
tsi.ulStartActiveOffset = 0;
```

```
tsi.cchActiveChars = cch;
```

```
tsi.ulStartSelection = 0;
```

```
tsi.cchSelection = cch;
```

```
pwszCoMem2 = (WCHAR *)CoTaskMemAlloc(sizeof(WCHAR) * cch);
```

```
if (SUCCEEDED(hr) && pwszCoMem2)
```

```
{
```

```
    // SetWordSequenceData requires double NULL terminator.
```

```
    memcpy(pwszCoMem2, pwszCoMem, sizeof(WCHAR) * cch);
```

```
    pwszCoMem2[cch] = L'\0';
```

```
    pwszCoMem2[cch+1] = L'\0';
```

```
    // set the text buffer data
```

```
    hr = cpRecoGrammar->SetWordSequenceData(pwszCoMem2,
```

```
    // Check hr
```

```
    CoTaskMemFree(pwszCoMem2);
```

```
}
```

```
CoTaskMemFree(pwszCoMem);
```

// the SR engine is now capable of recognizing the contents of the



ISpRecoGrammar::SetTextSelection

ISpRecoGrammar::SetTextSelection sets the current text selection and insertion point information.

```
HRESULT SetTextSelection(  
    const SPTEXTSELECTIONINFO *pInfo  
);
```

Parameters

pInfo

[in] Address of the [SPTEXTSELECTIONINFO](#) structure that contains the text selection and insertion point information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
FAILED(hr)	Appropriate error message.

Remarks

An application that has a text box could enable the user to speak commands into the text box to edit the text. One way to design this functionality would be to create a CFG which supports such commands as "cut the text *", "bold the text *", or "italicize the words *". The grammar would then use a TEXTBUFFER tag in place of the * which would enable the SR engine to recognize the text buffer information. At run time, the application would update the SR engine's view of the text buffer using [ISpRecoGrammar::SetWordSequenceData](#). When the user highlights a selection of text and the text selection using [ISpRecoGrammar::SetTextSelection](#).

If a user had the text "*hello world*" in the text box and no text highlighted, the SR engine could recognize "bold the text *world*". If the user highlighted "*hello*", and the application changed the active text selection only contain "*hello*", "bold the text *world*" would fail to recognize.

The application should change the active text selection when the text highlight changes, rather than the entire word sequence data, to ensure the SR engine has a textual context to help the recognition language model.

See also [ISpRecoGrammar::SetWordSequenceData](#) for information on how to set the text data.

See also [ISpSREngine::SetTextSelection](#) for information on how SAPI passes the text selection information to the SR engine.

The SR engine must support text buffer features. Check for the presence of the TextBuffer attribute in the SR engine. Microsoft SR ASR engines support these features although there is no requirement that other manufacturers engines need to. See Recognizers in [Object Tokens and Registry Settings](#) for more information.

Example

The following code snippet illustrates how to use ISpRecoGrammar::SetTextSelection after sending a text buffer to the SR engine using ISpRecoGrammar::SetWordSequenceData.

```
HRESULT hr = S_OK;
```

```
// place the contents of text buffer into pwszCoMem and the length
```

```
SPTXTSELECTIONINFO tsi;
```

```
tsi.ulStartActiveOffset = 0;
```

```
tsi.cchActiveChars = cch;
```

```
tsi.ulStartSelection = 0;
```

```
tsi.cchSelection = cch;
```

```
pwszCoMem2 = (WCHAR *)CoTaskMemAlloc(sizeof(WCHAR) * cch);

if (SUCCEEDED(hr) && pwszCoMem2)
{
    // SetWordSequenceData requires double NULL terminator.
    memcpy(pwszCoMem2, pwszCoMem, sizeof(WCHAR) * cch);
    pwszCoMem2[cch] = L'\0';
    pwszCoMem2[cch+1] = L'\0';

    // set the text buffer data
    hr = cpRecoGrammar->SetWordSequenceData(pwszCoMem2,
    // Check hr

    // set the text selection information
    hr = cpRecoGrammar->SetTextSelection(&tsi);
    // Check hr

    CoTaskMemFree(pwszCoMem2);
}
CoTaskMemFree(pwszCoMem);

// the SR engine is now capable of recognizing the contents of the
```



ISpRecoGrammar::IsPronounceable

ISpRecoGrammar::IsPronounceable calls the SR engine object to determine if the word has a pronunciation.

```
HRESULT IsPronounceable(  
    const WCHAR          *pszWord,  
    SPWORDPRONOUNCEABLE *pfPronounceable  
);
```

Parameters

pszWord

[in, string] The word to test. Length must be equal to or less than [SP_MAX_WORD_LENGTH](#).

pfPronounceable

[out] Flag, from among the following list, indicating the if the word is pronounceable by the SR engine. See Remarks section.

Value	
SPWP_UNKNOWN_WORD_UNPRONOUNCEABLE	The word is not pronounceable by the SR engine, and is not located in the lexicon and/or the engine's dictionary.
SPWP_UNKNOWN_WORD_PRONOUNCEABLE	The word is pronounceable by the SR

	engine, but is not located in the lexicon and/or the engine's dictionary.
SPWP_KNOWN_WORD_PRONOUNCEABLE	The word is pronounceable by the SR engine, and is located in the lexicon and/or the engine's dictionary.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Either <i>pszWord</i> or <i>pfPronounceable</i> is invalid or bad.
FAILED (hr)	Appropriate error message.

Remarks

The exact implementation and usage for the SR engine's dictionary and pronounceable words may vary between engines. For example, an SR engine may attempt to pronounce all words passed using `::IsPronounceable`, even if it is not located in the lexicon or the dictionary, it would rarely or never, return `SPWP_UNKNOWN_WORD_UNPRONOUNCEABLE`.

Typically, there are two scenarios when an application might use the method `::IsPronounceable`.

If an application is using a number of specialized or uncommon words (e.g., legal, medical, or scientific terms), the application

may want to verify that the words are contained in either the lexicon (see also [ISpLexicon](#)) or the SR engine's dictionary. If the words are not contained in the lexicon or the dictionary (even if they are pronounceable), the application can add them to the lexicon to improve the chances of a successful recognition.

An application may also want to verify that the SR engine will actually recognize the words in a CFG (even though loading the CFG succeeded). If the SR engine returns `SPWP_UNKNOWN_WORD_UNPRONOUNCEABLE`, the application can update the lexicon pronunciation entry (see [ISpLexicon](#)).

See also [ISpSREngine::IsPronounceable](#) for more information on the SR engine's role.

Example

The following code snippet illustrates the use of `ISpRecoGrammar::IsPronounceable`. The words used are examples only, as the pronounceability by different SR engines may vary. See Remarks section.

```
HRESULT hr = S_OK;

// check if a common word is pronounceable
hr = cpRecoGrammar->IsPronounceable(L"hello", &wordPronounceable);
// Check hr

// wordPronounceable is probably equal to SPWP_KNOWN_WORD

// check if an uncommon, or imaginary, word is pronounceable
hr = cpRecoGrammar->IsPronounceable(L"snork", &wordPronounceable);
// Check hr

// wordPronounceable is probably equal to SPWP_UNKNOWN_WORD

// check if a non-word, or imaginary, word is unpronounceable
hr = cpRecoGrammar->IsPronounceable(L"lpdzsd", &wordPronounceable);
// Check hr
```

```
// wordPronounceable is probably equal to SPWP_UNKNOWN_WI
```



ISpRecoGrammar::SetGrammarState

ISpRecoGrammar::SetGrammarState sets the grammar state.

```
HRESULT SetGrammarState(  
    SPGRAMMARSTATE eGrammarState  
);
```

Parameters

eGrammarState

[in] Flag of type SPGRAMMARSTATE indicating the new state of the grammar.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>eGrammarState</i> is not a valid state.
FAILED(hr)	Appropriate error message.

Remarks

If *eGrammarState* is SPGM_DISABLED, SAPI will retain the current rule activation state, so that when the grammar state is set to SPGM_ENABLED, it restores the grammar rules back to each of the original activation states. While the grammar is set to SPGM_DISABLED, the application can still activate and deactivate rule. The effect is not communicated to the SR engine (but retained by SAPI) until the grammar is enabled again.

If *eGrammarState* is SPGM_EXCLUSIVE, SAPI will disable all other

grammars in the system, unless another grammar is already exclusive. Activation and deactivation commands are buffered for all other grammars until the exclusive grammar is set to SPGM_ENABLED again.

The default grammar state is SPGS_ENABLED, meaning the grammar can receive recognitions.

Applications can use the grammar state to control whether it will receive recognitions for rules in that SpRecoGrammar object. For example, an application create a new SpRecoGrammar object, set the grammar state to SPGS_DISABLED, dynamically generate the rules, and finally set the grammar state to SPFS_ENABLED when grammar construction is completed.

See also [ISpRecoGrammar::GetGrammarState](#).

Example

The following code snippet illustrates the use of ISpRecoGrammar::SetGrammarState to set the state of an grammar

```
HRESULT hr = S_OK;

// create a new grammar object
hr = cpRecoContext->CreateGrammar(GRAM_ID, &cpRecoGramn
// Check hr

// disable the grammar, so that recognitions are not received
hr = cpRecoGrammar->SetGrammarState(SPGS_DISABLED);
// Check hr

// ... build the grammar ...

// activate the grammar, so applications start receiving recognitions
hr = cpRecoGrammar->SetGrammarState(SPGS_ENABLED);
```

// Check hr



ISpRecoGrammar::SaveCmd

ISpRecoGrammar::SaveCmd allows applications using dynamic grammars to save the current grammar state to a stream.

```
HRESULT SaveCmd(  
    IStream      *pSaveStream,  
    WCHAR       **ppCoMemErrorText  
);
```

Parameters

pSaveStream

[in] The stream to save the compiler binary grammar into.

ppCoMemErrorText

[out] Optional parameter of a null-terminated string containing error messages that occurred during the save operation.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pSaveStream</i> is invalid or bad.
SPERR_NOT_DYNAMIC_GRAMMAR	Command was loaded but compiler is not available.
SPERR_UNINITIALIZED	Compiler is not available.
E_POINTER	<i>ppCoMemErrorText</i> is invalid or bad.
FAILED (hr)	Appropriate error message.

Remarks

Applications can use `::SaveCmd` to serialize grammar changes that were made at run time for use at a later time. See also [ISpRecoGrammar::LoadCmdFromMemory](#).

Example

The following code snippet illustrates how to use `ISpRecoGrammar::SaveCmd` to serialize the CFG from one `SpRecoGrammar` object and deserialize it into another `SpRecoGrammar` object.

```
HRESULT hr = S_OK;

// ... build and use a SpRecoGrammar object

// create a Win32 global stream
hr = ::CreateStreamOnHGlobal(NULL, true, &cpHStream);
// Check hr

// save the current grammar to the global stream
hr = cpRecoGrammar->SaveCmd(cpHStream, NULL);
// Check hr

// create the second grammar to deserialize into
hr = cpRecoContext->CreateGrammar(0, &cpReloadedGrammar);
// Check hr

// get a handle to the stream with the serialized grammar
::GetHGlobalFromStream(cpHStream, &hGrammar);
// Check hr

// deserialize the CFG into a new grammar object
hr = cpReloadedGrammar->LoadCmdFromMemory((SPBINARYG
```

// Check hr



ISpRecoGrammar::GetGrammarState

ISpRecoGrammar::GetGrammarState retrieves the current state of the recognition grammar.

The default grammar state is SPGS_ENABLED.

See also [ISpRecoGrammar::SetGrammarState](#)

```
HRESULT GetGrammarState(  
    SPGRAMMARSTATE *peGrammarState  
);
```

Parameters

peGrammarState

[out] Address of the [SPGRAMMARSTATE](#) enumeration that receives the grammar state information.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	One of the pointers is invalid or bad.
FAILED(hr)	Appropriate error message.

Example

The following code snippet illustrates the use of `ISpRecoGrammar::GetGrammarState` to query the default state of an `SpRecoGrammar` object.

```
HRESULT hr = S_OK;  
  
// create a new grammar object
```

```
hr = cpRecoContext->CreateGrammar(GRAM_ID, &cpRecoGramn  
// Check hr
```

```
// query the default grammar state
```

```
hr = cpRecoGrammar->GetGrammarState(&grammarState);
```

```
// Check hr
```

```
// ASSERT that grammarState == SPGS_ENABLED
```



ISpRecoResult

The ISpRecoResult interface is used by an application to retrieve information about the SR engine's hypotheses, recognitions, and false recognitions.

The most common use of the ISpRecoResult interface is retrieval of text recognized by the SR engine (see [ISpPhrase::GetText](#)). The ISpRecoResult interface also supports the retrieval of the original audio that the SR engine recognized. SAPI can automatically retain the audio for an application using [ISpRecoContext::SetAudioOptions](#).

An application can set interest in SR engine hypotheses by calling [ISpEventSource::SetInterest](#) with [SPEI_HYPOTHESIS](#). As each hypothesis event is received, the application can examine the proposed text and update its UI to display dynamic run-time recognition status.

An application can set interest in the SR engine's failed recognitions by calling [ISpEventSource::SetInterest](#) with [SPEI_FALSE_RECOGNITION](#). If a false recognition occurs, the application can examine the audio (or even a partial recognition result) to reprocess the recognition or attempt to process the partially recognized text. SAPI does not require that an SR engine send a phrase with the false recognition event.

ISpRecoResult Methods	Description
ISpPhrase	Inherits from ISpPhrase and those methods are accessible from an ISpRecoResult object.
GetResultTimes	Retrieves the time information associated with the result.
GetAlternates	Retrieves an array containing alternate phrases.
GetAudio	Creates an audio stream for a given

	number of elements.
<u>SpeakAudio</u>	Retrieves and speaks the specified audio.
<u>Serialize</u>	Creates a serialized copy of the recognition result object.
<u>ScaleAudio</u>	Converts the format of the retained audio to a different audio format.
<u>GetRecoContext</u>	Returns the recognition context object that is associated with this result.



ISpRecoResult::GetResultTimes

ISpRecoResult::GetResultTimes retrieves the time information associated with the result.

```
HRESULT GetResultTimes(  
    SPRECORESULTTIMES *pTimes  
);
```

Parameters

pTimes

[out] Address of the SPRECORESULTTIMES data structure containing the time information associated with the result.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pTimes</i> is invalid or bad.
SPERR_NOT_FOUND	Interface not found.

Remarks

An application can use `::GetResultTimes` to determine the system time that a recognition occurred, the length in seconds of the recognized phrase, and the length of time between when the SR engine began listening and when the recognition occurred.

Examples Using This Method

SDK: [CoffeeS2](#); [CoffeeS3](#); [CoffeeS4](#).



ISpRecoResult::GetAlternates

ISpRecoResult::GetAlternates retrieves an array of pointers to [ISpPhraseAlt](#) objects containing alternate phrases.

```
HRESULT GetAlternates(  
    ULONG          ulStartElement,  
    ULONG          cElements,  
    ULONG          ulRequestCount,  
    ISpPhraseAlt **ppPhrases,  
    ULONG          *pcPhrasesReturned  
);
```

Parameters

ulStartElement

[in] The starting element to consider for the alternates. This is a zero-based value.

cElements

[in] The number of elements to consider. All elements can be requested using the enumeration value `SPPR_ALL_ELEMENTS` of type [SPPHRASERNG](#).

ulRequestCount

[in] The number of requested alternate phrase elements.

ppPhrases

[out] Address of an array of [ISpPhraseAlt](#) interface pointers that contain the alternate phrases. The elements between the start of the *ulStartElement* element and the end of the *ulStartElement* and *cElements* element combined is the portion that will change. The rest of the elements will be included in each alternate phrase.

pcPhrasesReturned

[out] Pointer to a ULONG that receives the actual number of alternate phrases retrieved.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pcPhrasesReturned</i> is an invalid pointer. However, <i>ppPhrases</i> does not contain <i>ulRequestCount</i> allocations.
E_OUTOFMEMORY	Exceeded available memory.
E_INVALIDARG	<i>ulStartElement</i> is not less than the number of elements in the owning interface. However, the number of expected elements exceeds the number of available elements in the owning interface.
S_FALSE	No analyzer is present or there is no driver data.
FAILED(hr)	Appropriate error message.

Example

The following code snippet illustrates the use `ISpRecoResult::GetAlternates` to retrieve and commit an alternate phrase.

```
HRESULT hr = S_OK;

// ... obtain a recognition result object from the recognizer...

// get the recognized phrase object
hr = cpRecoResult->GetPhrase(&pPhrase);
// Check hr
```

```

// get the phrase's text
hr = pPhrase->GetText(SP_GETWHOLEPHRASE, SP_GETWHOI
// Check hr

// ... check the phrase's text... assume the phrase isn't a correct recogni

// setup MY_MAX_ALTERNATES phrase alternate objects
CComPtr<ISpPhraseAlt> pcpPhrase[MY_MAX_ALTERNATES];
ULONG ulCount;

// get the top MY_MAX_ALTERNATES alternates to the entire reco
hr = cpRecoResult->GetAlternates(pPhrase->Rule.ulFirstElement,
                                pPhrase->Rule.ulCountOfElements,
                                MY_MAX_ALTERNATES,
                                pcpPhraseAlt,
                                &ulCount);

// Check hr

// check each alternate in order of highest likelihood
for (int i = 0; i < ulCount; i++) {
    hr = pcpPhraseAlt[i]->GetText(SP_GETWHOLEPHRASE, SP_(
    // Check hr

    // ... check if this alternate is more appropriate ...

    // if it is more appropriate, then commit the alternate
    if (fMoreAppropriate) {
        hr = pcpPhraseAlt[i]->Commit();
        // Check hr
    }

    // free the alternate text

```

```
    if (pwszAlternate) ::CoTaskMemFree(pwszAlternate);  
}  
  
// free the initial phrase object  
if (pPhrase) ::CoTaskMemFree(pPhrase);
```



ISpRecoResult::GetAudio

ISpRecoResult::GetAudio creates an audio stream of the requested words from the audio data in the result object.

```
HRESULT GetAudio(  
    ULONG          ulStartElement,  
    ULONG          cElements,  
    ISpStreamFormat **ppStream  
);
```

Parameters

ulStartElement

[in] Value specifying from which element in the result data to start the audio stream.

cElements

[in] Value specifying the total number of words.

ppStream

[out] Address that will receive a pointer to an [ISpStreamFormat](#) object containing the audio data requested.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>cElements</i> is zero or the expected number of elements to count exceeds the number available.
E_POINTER	<i>ppStream</i> is an invalid pointer.
SPERR_NO_AUDIO_DATA	This result object does not have any audio data.

FAILED(hr)

Appropriate error message.

Remarks

Even if there are no elements, that is, *ulStartElement* = 0 and *cElements* = 0, the audio will still be played. There are "unrecognized" results that have no elements but have audio.

An application can find the time offsets for each element by examining the [SPPHRASE](#) object retrieved using [ISpRecoResult::GetPhrase](#).

Example

The following code snippet illustrates the use `ISpRecoResult::GetAudio` to retrieve the retained audio.

```
HRESULT hr = S_OK;
```

```
// ... obtain a recognition result object from the recognizer...
```

```
hr = cpRecoResult->GetAudio( 0, 0, &cpStreamFormat );
```

```
// Check hr
```

```
// check the format of the stream for fun...
```

```
hr = cpStreamFormat->GetFormat(&formatId, &pWaveFormatEx);
```

```
// Check hr
```



ISpRecoResult::SpeakAudio

ISpRecoResult::SpeakAudio retrieves and speaks the specified audio. This combines two other methods; first calling [ISpRecoResult::GetAudio](#) and then calling [ISpVoice::SpeakStream](#) on the parent recognition context.

```
HRESULT SpeakAudio(  
    ULONG      ulStartElement,  
    ULONG      cElements,  
    DWORD      dwFlags,  
    ULONG      *pulStreamNumber  
);
```

Parameters

ulStartElement

[in] Value specifying with which element to start.

cElements

[in] Value specifying the number of elements contained in the stream. A value of zero speaks all elements.

dwFlags

[in] Value indicating the attributes of the text stream. These values are contained in the [SPEAKFLAGS](#) enumeration.

pulStreamNumber

[optional, out] Address of a variable containing the stream number information. If NULL, the stream number will not be retrieved.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_NO_AUDIO_DATA	Result does not contain audio data.
E_POINTER	<i>pulStreamNumber</i> is a non-NULL, bad pointer.
FAILED(hr)	Appropriate error message.

Return values may also be those from [ISpVoice::SpeakStream](#).

Remarks

Even if there are no elements, that is, *ulStartElement* = 0 and *cElements* = 0, the audio will still be spoken. These are unrecognized results that have no elements, but do have audio.

If the application did not activate retained audio (see [ISpRecoContext::SetAudioOptions](#)), or make a previous call to [ISpPhrase::Discard](#) and eliminate the retained audio, `::SpeakAudio` will fail with SPERR_NO_AUDIO_DATA.

Example

The following code snippet illustrates the use of `ISpObjectToken::IsUISupported` using [SPGUID_EngineProperties](#).

```

HRESULT hr = S_OK;

// ... get a recognition result object from the SR engine

// replay the user's spoken audio to the user
hr = cpRecoResult->SpeakAudio( 0, 0, 0, &ulStreamNum );
// Check hr

```



ISpRecoResult::Serialize

ISpRecoResult::Serialize creates a serialized copy of the recognition result object. The serialized copy can be saved and later restored using [ISpRecoContext::DeserializeResult](#).

```
HRESULT Serialize(  
    SPSERIALIZEDRESULT    **ppCoMemSerializedResult  
);
```

Parameters

ppCoMemSerializedResult

[out] Address of a pointer to the [SPSERIALIZEDRESULT](#) structure that receives the serialized result information. Call `CoTaskMemFree()` to free the memory associated with the serialized result object.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppCoMemSerializedResult</i> is an invalid pointer.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.

Example

The following code snippet illustrates the use of `ISpRecoResult::Serialize` to serialize a result and deserialize it back into an `ISpRecoContext` object.

```
HRESULT hr = S_OK;
```

```
// ... obtain a recognition result object from the recognizer...

SPSERIALIZEDRESULT* pSerializedResult = NULL;
ULONG cbWritten = 0;
ULONG ulSerializedSize = 0;
LARGE_INTEGER liseek;
LARGE_INTEGER li;
CComPtr<IStream> cpStreamWithResult;

hr = CreateStreamOnHGlobal(NULL, true, &cpStreamWithResult);
// Check hr

// Serialize result to memory
hr = cpRecoResult->Serialize(&pSerializedResult);
// Check hr

//serialized to a stream pointer
hr = cpStreamWithResult->Write(pSerializedResult, pSerializedRes
// Check hr

// free the serialized result
if (pSerializedResult) ::CoTaskMemFree(pSerializedResult);

// commit the stream changes
hr = cpStreamWithResult->Commit(STGC_DEFAULT);
// Check hr

// ... persist stream to disk, network share, etc...
// ... shutdown application ....

// ... restart application and get the persisted stream

// reset the stream seek pointer to the start before deserialization
```

```

li.QuadPart = 0;
hr = cpStreamWithResult->Seek(li, STREAM_SEEK_SET, NULL);
// Check hr

// find the size of the stream
hr = cpStreamWithResult->Read(&ulSerializedSize, sizeof(ULONG)
// Check hr

// reset the seek pointer
liseek.QuadPart = 0 - sizeof(ULONG);
hr = cpStreamWithResult->Seek(liseek, STREAM_SEEK_CUR, NU
// Check hr

// allocate the memory for the result
pSerializedResult = (SPSERIALIZEDRESULT*)::CoTaskMemAllo
// Check pSerializedResult in case out "out-of-memory"

// copy the stream into a serialized result object
hr = cpStreamWithResult->Read(pSerializedResult, ulSerializedSize
// Check hr

// Deserialize result from memory
hr = cpRecoContext->DeserializeResult(pSerializedResult, &cpReco
// Check hr

// free the pSerializedResult memory
if (pSerializedResult) {
    CoTaskMemFree(pSerializedResult);
}

// As long as the same engine was used to generate
// the original result object, as is now being used,
// applications can now get alternates for the cpRecoResultNew's ph

```




ISpRecoResult::ScaleAudio

ISpRecoResult::ScaleAudio converts an existing audio stream into a different audio format.

```
HRESULT ScaleAudio(  
    const GUID          *pAudioFormatId,  
    const WAVEFORMATEX *pWaveFormatEx  
);
```

Parameters

pAudioFormatId

[in] Address of the data format identifier. Typically, this value is *SPDFID_WaveFormatEx*.

pWaveFormatEx

[in] Address of the [WAVEFORMATEX](#) structure that contains the audio format to convert to. This value must be NULL if *pAudioFormatId* is not specified as *SPDFID_WaveFormatEx*.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Either <i>pAudioFormatId</i> or <i>pWaveFormatEx</i> is invalid or bad.
SPERR_NO_AUDIO_DATA	Audio stream is unavailable.
SPERR_UNSUPPORTED_FORMAT	The engine format is non-waveformatex and the retained format the same format.
E_OUTOFMEMORY	Exceeded available memory.

FAILED(hr)

Appropriate error message.

Remarks

Use the [ISpPhrase::Discard](#) method to completely discard audio data associated with a result object.

The application can also set the default retained audio format for the [ISpRecoResult](#) object by calling [ISpRecoContext::SetAudioOptions](#). Calling `::SetAudioOptions` will only apply to all subsequent recognitions, not the current [ISpRecoResult](#) object.

When performing a scaling with a compressed format, it is possible to introduce small rounding errors, since the content of the audio is not used to perform the conversion.

Scaling between certain compressed formats is not supported by the SAPI format converter (See the Remarks section for [ISpStreamFormatConverter](#)).

Example

The following code snippet illustrates the use `ISpRecoResult::ScaleAudio` to scale the audio to a low quality format before serialization to the disk (to save space).

```
HRESULT hr = S_OK;

// ... obtain a recognition result object from the recognizer...

// create a format helper with a very low quality format
CSpStreamFormat ScaleFormat(SPSF_8kHz8BitMono, &hr);
// Check hr

hr = cpRecoResult->ScaleAudio(&(ScaleFormat.FormatId()), ScaleFormat.FormatId());
// Check hr
```

```
// get a result serialization pointer
SPSERIALIZEDRESULT* pSerializedResult;

// serialize the result
hr = cpRecoResult->Serialize(&pSerializedResult);
// Check hr

// ... write pSerializedResult to the disk
```

Development Helpers

Helper Enumerations, Functions and Classes	Description
SPSTREAMFORMAT	SAPI supported stream formats
CSpStreamFormat	Class for managing SAPI supported stream formats and WAVEFORMATEX structures



ISpRecoResult::GetRecoContext

ISpRecoResult::GetRecoContext returns the recognition context object that is associated with this result.

```
HRESULT GetRecoContext(  
    ISpRecoContext **ppRecoContext  
);
```

Parameters

ppRecoContext

[out] A pointer that receives the recognition context interface pointer. The caller must call `::Release` on the [ISpRecoContext](#) references when it is finished.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppRecoContext</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Example

The following code snippet illustrates the use of `ISpRecoResult::GetRecoContext` to retrieve a reference to the `ISpRecoContext` instance that is associated with a recognized phrase and determine the maximum number of CFG alternates that can be generated for it.

```
HRESULT hr = S_OK;  
  
DWORD dwMaxAlternates;
```

```
// ... obtain a recognition result object from the recognizer...

// get the associated ISpRecoContext
hr = cpRecoResult->GetRecoContext(&cpRecoContext);
// Check hr

hr = cpRecoContext->GetMaxAlternates(&dwMaxAlternates);
// Check hr
```



ISpRecognizer

The ISpRecognizer interface enables applications to control aspects of the speech recognition (SR) engine. Each ISpRecognizer interface represents a single SR engine. The application can connect to each recognizer object one or more [recognition contexts](#), from which the application can control the recognition grammars to be used, start and stop recognition, and receive events and recognition results. The ISpRecognizer interface allows some additional control of the SR engine and its audio input. A standard application may not need to call many of the methods on this interface as SAPI tries to set the engine up sensibly by default.

There are two implementations of the ISpRecognizer and ISpRecoContext in SAPI. One is for recognition "in-process" (InProc), where the SR engine is created in the same process as the application. Only this application can connect to this recognizer. The other implementation is the "shared-recognizer," where the SR engine is created in a separate process. There will only be one shared engine running on a system, and all applications using the shared engine connect to the same recognizer. This allows several speech applications to work simultaneously, and allows the user to speak to any application, as recognition is done from the grammars of all applications. For desktop-based speech applications it is recommended to use the shared recognizer because of the way it allows multiple SAPI applications to work at once. For other types of application, such as recognizing from wave files or a telephony server application where multiple SR engines will be required, the InProc recognizer should be used.

When to Use

Call methods of the ISpRecognizer interface to configure or retrieve the attributes of the SR engine.

Implemented By

- This interface is implemented by SAPI. Application developers use this interface but do not implement it.

How Created

There are two objects that implement this interface. These are created by applications by creating a COM object with either of the following CLSIDs:

[SpInprocRecognizer](#) (CLSID_SpInprocRecognizer)

[SpSharedRecognizer](#) (CLSID_SpSharedRecognizer)

Alternatively, the shared recognizer can be created by creating a [SpSharedRecoContext](#) (CLSID_SpSharedRecoContext), and then calling [ISpRecoContext::GetRecognizer](#) on this object to get a reference to the SpSharedRecognizer object.

Methods in Vtable Order

ISpRecognizer Methods	Description
ISpProperties	Inherits from ISpProperties and all those methods are accessible from ISpRecognizer.
SetRecognizer	Specifies the SR engine to be used.
GetRecognizer	Retrieves which SR engine is currently being used.
SetInput	Specifies which input stream the SR engine should use.
GetInputObjectToken	Retrieves the input token object for the stream.
GetInputStream	Retrieves the input stream.
CreateRecoContext	Creates a recognition context for this instance of an SR engine.
GetRecoProfile	Retrieves the current recognition

	profile token.
<u>SetRecoProfile</u>	Sets the recognition profile to be used by the recognizer.
<u>IsSharedInstance</u>	Determines if the recognizer is the shared or InProc implementation.
<u>GetRecoState</u>	Retrieves the state of the recognition engine.
<u>SetRecoState</u>	Sets the state of the recognition engine.
<u>GetStatus</u>	Retrieves current status information for the engine.
<u>GetFormat</u>	Retrieves the format of the current audio input.
<u>IsUISupported</u>	Checks if the SR engine supports a particular user interface component.
<u>DisplayUI</u>	Displays a user interface component.
<u>EmulateRecognition</u>	Emulates a recognition from a text phrase rather than from spoken audio.



ISpRecognizer::SetRecognizer

ISpRecognizer::SetRecognizer specifies the particular speech recognition engine to be used.

```
HRESULT SetRecognizer(  
    ISpObjectToken *pEngineToken  
);
```

Parameters

pEngineToken

[in] The object token referring to the speech recognition engine to be used.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pEngineToken</i> is invalid or bad.
SPERR_ENGINE_BUSY	Recognition is currently running or other applications are connected to the shared recognizer.
FAILED(hr)	Appropriate error message.

Remarks

This method allows the application to select a particular engine object token to be used (For example, the method [SpFindBestToken](#) could be used to find an engine supporting certain attributes, and the resulting token could be passed to this method).

If this method is not called, SAPI will use the current default SR

engine.

If this method is passed NULL, SAPI will switch to the current default SR engine.

This method cannot be called when the current SR engine is already running and processing audio. In addition, when using the shared recognizer, it cannot be called if another application is also using the shared recognizer.



ISpRecognizer::GetRecognizer

ISpRecognizer::GetRecognizer retrieves the current speech recognition engine being used with this ISpRecognizer.

```
HRESULT GetRecognizer(  
    ISpObjectToken **ppEngineToken  
);
```

Parameters

ppEngineToken

[out] The object token representing the current speech recognition engine.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppEngineToken</i> is invalid or bad.
FAILED(hr)	Appropriate error message.



ISpRecognizer::SetInput

ISpRecognizer::SetInput specifies which input stream the SR engine should use.

```
HRESULT SetInput(  
    IUnknown *pUnkInput,  
    BOOL     fAllowFormatChanges  
);
```

Parameters

pUnkInput

[in] The stream object token. See Remarks section.

fAllowFormatChanges

[in] Boolean indicating whether SAPI should try to change the input stream format to the engine's preferred format. This method can normally be set to TRUE; however, when performing both speech recognition and speech output at the same time, some soundcards may require that both input and output are in the same audio format. Setting this to FALSE prevents the audio format on the input device from being changed. Instead, SAPI will try to convert the audio format itself to something the SR engine can use.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pUnkInput</i> is invalid or not a stream.
SPERR_ENGINE_BUSY	The current method cannot be performed while the engine is currently processing audio.
FAILED(hr)	Appropriate error message.

Remarks

This method can be used to switch the input for the recognizer to a wave input stream, a different soundcard device, or to a custom audio object. The *pUnkInput* parameter can be a pointer to an [object token](#) representing an audio input device or a pointer to an actual object implementing [ISpStreamFormat](#).

The input stream object will implement [IStream](#), [ISpStreamFormat](#), and [ISpAudio](#) for real-time streams.

Applications should not use methods on these interfaces that actually change the state of the audio device or read data from it at the same time that the stream is being used by SAPI. For example, reading data from the application with [IStream::Read](#) will prevent the correct data from being passed to the SR engine. Altering the state of the audio using [ISpAudio::SetState](#) will put the audio device into an unexpected state and may cause errors. All control of the audio is done by SAPI.

When using the InProc recognizer, SAPI does not automatically setup the audio input. [ISpRecognizer::SetInput](#) must be called with a non-NULL *pUnkInput* to setup and start the audio input stream. Until [ISpRecognizer::SetInput](#) is called, methods such as [ISpRecoGrammar::SetRuleState](#) will return success code `SP_STREAM_UNINITIALIZED`, but actual recognition will not start.

When using the shared recognizer, SAPI automatically sets up the audio input. However, [ISpRecognizer::SetInput](#) may be called with NULL as the *pUnkInput* parameter to force the recognizer to re-check the default audio input and re-setup the audio input (e.g., the default audio input object changes while recognizing, and the new audio input is to be used).

If the engine is currently processing audio, this call will fail with `SPERR_ENGINE_BUSY`.

Example

The following code snippet illustrates the use of `ISpRecognizer::SetInput`.

```
// setup the inproc recognizer audio input with an audio  
  
// get the default audio input token  
hr = SpGetDefaultTokenFromCategoryId(SPCAT_AUDIOIN, &cpObjToken);  
// Check hr  
  
// set the audio input to our token  
hr = cpRecognizer->SetInput(cpObjToken, TRUE);  
// Check hr  
  
// setup the inproc recognizer audio input with an audio  
  
// create the default audio input object  
hr = SpCreateDefaultObjectFromCategoryId(SPCAT_AUDIOIN, &cpAudio);  
// Check hr  
  
// set the audio input to our object  
hr = cpRecognizer->SetInput(cpAudio, TRUE);  
// Check hr  
  
// ask the shared recognizer to re-check the default audio input  
hr = cpRecognizer->SetInput(NULL, TRUE);  
// Check hr - if SPERR_ENGINE_BUSY, then retry later
```



ISpRecognizer::GetInputObjectToken

ISpRecognizer::GetInputObjectToken retrieves the input token object for the stream currently being used.

GetInputObjectToken will always return the default audio input object token when using a shared recognizer.

```
HRESULT GetInputObjectToken(  
    ISpObjectToken **ppToken  
);
```

Parameters

ppToken

[out] Gets filled in with the current input object token pointer.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	Function completed successfully, but the input stream object has no object token associated with it.
E_POINTER	<i>ppToken</i> is invalid or bad.
SPERR_UNINITIALIZED	No audio input has yet been set with SetInput (InProc engine only).
FAILED(hr)	Appropriate error message.

Remarks

Applications will not normally need to use this method, but it can be used to find out specific details of the object token that was used to create the audio input stream.

If an application receives feedback from the SR engine that

recognition quality is low, (e.g., poor audio signal quality (see [SPEI_INTERFERENCE](#), or that the microphone needs adjustment (see [SPEI_REQUEST_UI](#) for [SPDUI_MicTraining](#)), etc.), it may be helpful to reconfigure the audio input settings. SAPI defines two specific types of audio UI that an audio object token can provide: volume ([SPDUI_AudioVolume](#)) and properties ([SPDUI_AudioProperties](#)). An application can use `::GetInputObjectToken`, [ISpObjectToken::IsUISupported](#), and [ISpObjectToken::DisplayUI](#) to display audio UI in an effort to improve speech recognition.

Example

The following code snippet illustrates the use of `ISpRecognizer::GetInputObjectToken` when displaying audio UI.

```
HRESULT hr = S_OK;

// check if the current recognizer has an object token
hr = cpRecognizer->GetInputObjectToken(&cpObjectToken);
// Check hr == S_OK

// get the object token's UI
hr = cpObjectToken->QueryInterface(&cpTokenUI);
// Check hr

// check if the default audio input object has UI for Vo.
hr = cpTokenUI->IsUISupported(SPDUI_AudioVolume, NULL, NI);
// Check hr

// if fSupported == TRUE, then audio input object has UI
// Display the default audio input object's Volume UI
hr = cpTokenUI->DisplayUI(MY_HWND, MY_AUDIO_DIALOG_TITLE);
// Check hr
```



ISpRecognizer::GetInputStream

ISpRecognizer::GetInputStream retrieves the input stream that is currently being used.

```
HRESULT GetInputStream(  
    ISpStreamFormat **ppStream  
);
```

Parameters

ppStream

[out] Address of a pointer to the [ISpStreamFormat](#) object that receives the input stream information.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppStream</i> invalid or bad.
SPERR_NOT_SUPPORTED_FOR_SHARED_RECOGNIZER	Method is not available when using the shared recognizer.
FAILED(hr)	Appropriate error message.

Remarks

Applications will not normally need to use this method, but it can be used to find the specific audio input stream that is being used. This method can be used only on InProc recognizers, not on the shared recognizer.

The returned object will implement [IStream](#), [ISpStreamFormat](#), and [ISpAudio](#) for real-time streams. Applications should not use methods on these interfaces that actually change the state of the audio device or read data from it. For example, reading data from the application with [IStream::Read](#) will prevent the correct data from being passed to the SR engine. Altering the state of the audio using [ISpAudio::SetState](#) will put the audio device into an unexpected state and may cause errors. All control of the audio is done by SAPI.



ISpRecognizer::CreateRecoContext

ISpRecognizer::CreateRecoContext creates a [recognition context](#) for this instance of an SR engine. The recognition context is used to load recognition grammars, start and stop recognition, and receive events and recognition results.

Each application can have one or more recognition contexts, although normally each application will have only one.

```
HRESULT CreateRecoContext(  
    ISpRecoContext    **ppNewContext  
);
```

Parameters

ppNewContext

[out] Address of a pointer to an ISpRecoContext interface receiving the recognition context.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppNewContext</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Examples Using This Method

SDK: [CoffeeS0](#); [CoffeeS1](#); [CoffeeS2](#).



ISpRecognizer::GetRecoProfile

ISpRecognizer::GetRecoProfile retrieves the current recognition profile token.

```
HRESULT GetRecoProfile(  
    ISpObjectToken **ppToken  
);
```

Parameters

ppToken

[out] Address of a pointer of an [ISpObjectToken](#) that receives the profile information.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	The <i>ppToken</i> is a bad or invalid pointer.
FAILED(hr)	Appropriate error message.

Remarks

A recognition profile represents a single user and training sessions on the system. The user can create, delete, and set the current profile using Speech properties in Control Panel. SAPI will always create the engine using the current default profile. This method can be used to find which profile is currently being used.

If an application needs to store information in a specific recognition profile, it can use the

ISpObjectToken::GetStorageFilename method.

Example

The following code snippet illustrates the use of ISpRecognizer::GetRecoProfile to determine the profile name

```
HRESULT hr = S_OK;

// get the current recognizer's recognition profile token
hr = cpRecognizer->GetRecoProfile(&cpObjectToken);
// Check hr

// get the reco profile name (i.e. the default value of the token)
hr = cpObjectToken->GetStringValue(NULL, &pwszRecoProfileNa
// Check hr
```



ISpRecognizer::SetRecoProfile

ISpRecognizer::SetRecoProfile sets the recognition profile to be used by the recognizer.

```
HRESULT SetRecoProfile(  
    ISpObjectToken *pToken  
);
```

Parameters

pToken

[in] Address of an [ISpObjectToken](#) object that contains the profile information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
FAILED(hr)	Appropriate error message.

Remarks

A recognition profile represents a single user and training sessions on the system. The user can create, delete, and set the current profile using Speech properties in Control Panel. SAPI will always create the engine using the current default profile. This method can be used to set the SR engine to use a profile other than the default.

This method should not be called when the engine is currently processing audio. Calling `::SetRecoProfile` with an active

recognition engine can cause unexpected results, depending on how and when the SR engine reads the profile information.



ISpRecognizer::IsSharedInstance

ISpRecognizer::IsSharedInstance determines the recognizer is the shared or InProc implementation.

```
HRESULT IsSharedInstance ( void );
```

Parameters

None.

Return values

Value	Description
S_OK	Indicates that this instance of the recognition engine is being shared.
S_FALSE	Indicates that this instance of the recognition engine is not being shared.



ISpRecognizer::GetRecoState

ISpRecognizer::GetRecoState retrieves the current state of the recognition engine.

```
HRESULT GetRecoState(  
    SPRECOSTATE *pState  
);
```

Parameters

pState

[out] One of the input state flags contained in the [SPRECOSTATE](#) enumeration.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Invalid pointer.
FAILED(hr)	Appropriate error message.

Remarks

This method determines whether audio is currently being read from the audio input stream and passed to the SR engine.

See also [ISpRecognizer::SetRecoState](#)

The default recognizer state is [SPRST_ACTIVE](#), which means SAPI will activate the audio input stream only when at least one top-level rule is active.

To be notified when the recognizer state changes (e.g. another application changes the shared SR engine's recognizer state),

rather than polling the state with `::GetRecoState`, call [ISpEventSource::SetInterest](#) with [SPEI_RECO_STATE](#).



ISpRecognizer::SetRecoState

ISpRecognizer::SetRecoState sets the state of the recognition engine.

```
HRESULT SetRecoState(  
    SPRECOSTATE    NewState  
);
```

Parameters

NewState

[in] One of the flags contained in the [SPRECOSTATE](#) enumeration.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.
E_INVALIDARG	One or more parameters are invalid.

Remarks

This method should not be called when the engine is currently processing audio. Calling `::SetRecoProfile` with an active recognition engine can cause unexpected results, depending on how and when the SR engine reads the profile information.

A recognition profile represents a single user and training sessions on the system. The user can create, delete, and set the current profile using Speech properties in Control Panel. SAPI will always create the engine using the current default profile. This method can be used to set the SR engine to use a profile other than the default.

When using the shared recognizer, the recognizer state is a global setting. If one application changes the recognizer state, it will affect all other applications connected to the shared recognizer. For this reason, applications using a shared recognizer should take great caution before calling `SetRecoState`.

Changing the recognition state leads to a [SPEI_RECO_STATE_CHANGE](#) event for all interested recognition contexts.



ISpRecognizer::GetStatus

ISpRecognizer::GetStatus retrieves current status information for the engine.

```
HRESULT GetStatus(  
    SPRECOGNIZERSTATUS *pStatus  
);
```

Parameters

pStatus

[out] The current status of the engine.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pStatus</i> is invalid or bad.

Remarks

This method provides static information about the SR engine such as the languages it supports. It also provides dynamic information such as current stream position the engine has recognized up to, and if the stream is actively being sent to the engine.

See [SPRECOGNIZERSTATUS](#) for further explanation of the status information that can be retrieved.



ISpRecognizer::GetFormat

ISpRecognizer::GetFormat retrieves the current input audio format.

```
HRESULT GetFormat(  
    SPSTREAMFORMATTYPE    WaveFormatType,  
    GUID                    *pFormatId,  
    WAVEFORMATEX           **ppCoMemWFEX  
);
```

Parameters

WaveFormatType

[in] One of the wave file format types specified in [SPSTREAMFORMATTYPE](#).

pFormatId

[out] The address of the unique identifier associated with the format type.

ppCoMemWFEX

[out] Address of a pointer to a [WAVEFORMATEX](#) structure that receives the format information. This is set only if the input is of a wave format type. The application must free this data with `CoTaskMemFree` after use.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Invalid pointer.
SPERR_UNINITIALIZED	Audio input not yet set.

FAILED(hr) Appropriate error message.

Remarks

This method can return either the input format or the engine format. Normally these two values will be the same, but if SAPI is using a format converter to convert the input data from the audio input to the engine format these will be different.



ISpRecognizer::IsUISupported

ISpRecognizer::IsUISupported checks if the underlying speech engine implements a certain type of user-interface component.

See the [SR Engine Guide](#) for further information on how an SR engine implements UI.

```
[local] HRESULT IsUISupported(  
    const WCHAR    *pszTypeOfUI,  
    void           *pvExtraData,  
    ULONG          cbExtraData,  
    BOOL           *pfSupported  
);
```

Parameters

pszTypeOfUI

[in] Address of a pointer to a null-terminated string containing the UI type information.

pvExtraData

[in] Additional information for the call. The SR engine implementer dictates the format and usage of the data provided.

cbExtraData

[in] Size, in bytes, of *pvExtraData*. The SR engine implementer dictates the format and usage of the data provided.

pfSupported

[out] Address of a variable that receives the value indicating support for the interface. This value is set to TRUE when this

interface is supported; otherwise set to FALSE. If this value is TRUE, but the return code is S_FALSE, the UI type (*pszTypeOfUI*) is supported, but not with the current parameters or run-time environment. Check with the engine implementer to verify run-time requirements.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	The UI is supported but not with the current run-time environment or parameters.
E_INVALIDARG	<i>pfSupported</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Example

The following code snippet illustrates the use of `ISpRecognizer::IsUISupported` using [SPDUI_UserTraining](#).

```
HRESULT hr = S_OK;

// ask current recognizer if it supports user training
hr = cpRecognizer->IsUISupported(SPDUI_UserTraining, NULL);
// Check hr

// if fSupported == TRUE, then current speech recognizer
```



ISpRecognizer::DisplayUI

ISpRecognizer::DisplayUI displays the requested UI component from the underlying SR engine.

```
[local] HRESULT DisplayUI(  
    HWND          hwndParent,  
    const WCHAR   *pszTitle,  
    const WCHAR   *pszTypeOfUI,  
    void          *pvExtraData,  
    ULONG         cbExtraData  
);
```

Parameters

hwndParent

[in] Specifies the handle of the parent window.

pszTitle

[in] Address of a null-terminated string containing the window title. Set this value to NULL to indicate that the SR engine should use its default window title for this UI type.

pszTypeOfUI

[in] Address of a null-terminated string containing the UI type information.

pvExtraData

[in] Additional information for the call. The SR engine implementer dictates the format and use of the data provided.

cbExtraData

[in] Size, in bytes, of the contents of *pvExtraData*. The SR

engine implementer dictates the format and usage of the data provided.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	The UI is supported but not with the current run-time environment or parameters.
FAILED(hr)	Appropriate error message.

Remarks

SAPI 5 speech recognition engines are capable of sending UI requests back to the application using [SPEI_REQUEST_UI](#). For example, if the SR engine is receiving a poor audio input signal, it may request the user to perform Microphone Training (see [SPDUI_MicTraining](#)). The application can receive these requests by calling [ISpRecognizer::SetInterest](#) with [SPEI_REQUEST_UI](#). When the UI request is received, it can call [ISpRecognizer::DisplayUI](#) at an appropriate point. The typical SR engine UI requests could be User Training (see [SPDUI_UserTraining](#)), Microphone Training (see [SPDUI_MicTraining](#)), and Lexicon Updates (see [SPDUI_AddRemoveWord](#)). An application can call [DisplayUI](#) at any time, and does not necessarily have to wait for a UI request from the SR engine.

To best apply [ISpRecognizer::DisplayUI](#), call [ISpRecognizer::IsUISupported](#) with a specific UI type before calling [DisplayUI](#). (see the [SR Engine Guide](#) for further information on how an SR engine should implement UI.

The call to [DisplayUI](#) is synchronous, so the call will not return until the UI has been closed.

Example

The following code snippet illustrates the use of `ISpRecognizer::DisplayUI` using [SPDUI_UserTraining](#).

```
HRESULT hr = S_OK;  
  
// display user training UI for the current recognizer  
hr = cpRecognizer->DisplayUI(MY_HWND, MY_APP_USER_TRAINING_UI);  
// Check hr
```



ISpRecognizer::EmulateRecognition

ISpRecognizer::EmulateRecognition emulates a recognition from a specified phrase rather than from spoken content.

```
HRESULT EmulateRecognition(  
    ISpPhrase *pPhrase  
);
```

Parameters

pPhrase

[in] The phrase to emulate.

Return values

Value	Description
S_OK	Function completed successfully.
SP_NO_PARSE_FOUND	Function completed successfully but the phrase does not parse through any active rule.
SP_NO_RULES_ACTIVE	Function completed successfully but there are no active rules to parse.
E_POINTER	<i>ppCoMemPhrase</i> is invalid or bad.
SPERR_UNINITIALIZED	Phrase is uninitialized.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.

Remarks

In the case of ambiguous rules or CFG paths, the `::EmulateRecognition` method will return an arbitrary rule or path. For example, if a grammar has two ambiguous rules, the first containing the phrase "a b c", and the second containing only a dictation tag (i.e., `<DICTATION/>`), the rule recognized at run time may not be consistent.

This method can be used for testing applications that use speech recognition by simulating user speech. It can also be used by applications where users have the option to type or speak a command. The phrase can be generated by creating a [phrase builder](#) object and then adding elements representing the text to it. See the SDK Sample Simple Recognition (Reco.exe) for the function `CreatePhraseFromText` as an example of using `ISpPhraseBuilder`.

All the events will be fired back to the application exactly as if a normal recognition had taken place. The result phrase will have the semantic properties set in the same way a real result would. A recognition event will be produced only if the text actually parses through the active rules (if dictation is active, any text will parse). Another application or `ISpRecoContext` containing an active rule that can parse the text can receive the emulated recognition.



ISpPhrase

This is the main interface used to access information contained in a phrase. Using this interface, applications can retrieve recognition information such as the recognized (or hypothesized) text, the recognized rule, and semantic tag or property information. An application can also serialize the phrase data to a stream to enable persisting of recognitions to the disk, the network, or memory.

Methods in Vtable Order

ISpPhrase Methods	Description
<u>GetPhrase</u>	Retrieves data elements associated with a phrase.
<u>GetSerializedPhrase</u>	Returns the phrase information in serialized form.
<u>GetText</u>	Retrieves elements from a text phrase.
<u>Discard</u>	Discards the requested data from the phrase object.



ISpPhrase::GetPhrase

ISpPhrase::GetPhrase retrieves data elements associated with a phrase.

```
HRESULT GetPhrase(  
    SPPHRASE    **ppCoMemPhrase  
);
```

Parameters

ppCoMemPhrase

[out] Address of a pointer to an [SPPHRASE](#) data structure receiving the phrase information. May be NULL if no phrase is recognized. If NULL, no memory is allocated for the structure. It is the caller's responsibility to call `CoTaskMemFree` to free the object; however, the caller does not need to call `CoTaskMemFree` on each of the elements in `SPPHRASE`.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Invalid pointer.
E_OUTOFMEMORY	Exceeded available memory.

Returned data includes all elements associated with this phrase.

Example

The following code snippet illustrates the use of `ISpRecoResult::GetPhrase` as inherited from `ISpPhrase` to retrieve the recognized text, and display the rule recognized and the phrase.

HRESULT hr = S_OK;

// ... obtain a recognition result object from the recognizer...

// get the recognized phrase object

hr = cpRecoResult->GetPhrase(&pPhrase);

// Check hr

// get the phrase's text

hr = pPhrase->GetText(SP_GETWHOLEPHRASE, SP_GETWHOI

// Check hr

// display the recognized text and the rule name in a message box

MessageBoxW(MY_HWND, pwszText, pPhrase->Rule.pszName, M



ISpPhrase::GetSerializedPhrase

ISpPhrase::GetSerializedPhrase returns the phrase information in serialized form.

```
HRESULT GetSerializedPhrase(  
    SPSERIALIZEDPHRASE    **ppCoMemPhrase  
;)
```

Parameters

ppCoMemPhrase

[out] Address of a pointer which will be initialized to point to the serialized phrase data. The block of memory is created by CoTaskMemAlloc and must be manually freed with CoTaskMemFree when no longer needed.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppCoMemPhrase</i> is invalid or bad.
SPERR_UNINITIALIZED	Phrase is uninitialized.
E_OUTOFMEMORY	Exceeded available memory.

Remarks

The caller passes in the address of a pointer which is initialized to point to a block of memory which is allocated using CoTaskMemAlloc. It is the caller's responsibility to call CoTaskMemFree to free this object. The structure returned is defined to be a [SPSERIALIZEDPHRASE](#). However, the actual size of the block is contained in (**ppCoMemPhrase*)-

>*ulSerializedSize*. This size includes the size of the `SPSERIALIZEDPHRASE` structure. The phrase structure can be saved to a file, and later restored by calling [ISpPhraseBuilder::InitFromSerializedPhrase](#).

An application that will not need recognition alternates or retained audio and needs to save space, can serialize only the phrase information (e.g., phrase text, rule name, SR engine ID, etc.).

Example

The following code snippet illustrates the use `ISpRecoResult::GetSerializedPhrase` as inherited from `ISpPhrase` to serialize only the phrase portion of a result object.

```
HRESULT hr = S_OK;
```

```
// ... obtain a recognition result object from the recognizer...
```

```
SPSERIALIZEDPHRASE* pSerializedPhrase = NULL;
```

```
// get the recognized phrase object
```

```
hr = cpRecoResult->GetSerializedPhrase(&pSerializedPhrase);
```

```
// Check hr
```



ISpPhrase::GetText

ISpPhrase::GetText retrieves elements from a text phrase.

```
HRESULT GetText(  
    ULONG      ulStart,  
    ULONG      ulCount,  
    BOOL       fUseTextReplacements,  
    WCHAR      **ppszCoMemText,  
    BYTE       *pbDisplayAttributes  
);
```

Parameters

ulStart

[in] Specifies the first element in the text phrase to retrieve.

ulCount

[in] Specifies the number of elements to retrieve from the text phrase.

fUseTextReplacements

[in] Boolean value that indicates if replacement text should be used. An example of a text replacement is saying "write new check for twenty dollars" and retrieving the *replaced* text as "write new check for \$20". For more information on replacements, see the [SR Engine White Paper](#).

ppszCoMemText

[out] Address of a pointer to the data structure that contains the display text information. It is the caller's responsibility to call `::CoTaskMemFree` to free the memory.

pbDisplayAttributes

[out] Address of the [SPDISPLAYATTRIBUTES](#) enumeration that contains the text display attribute information. Text display attribute information can be used by the application to display the text to the user in a reasonable manner. For example, speaking "hello comma world period" includes a trailing period, so the recognition might include `SPAF_TWO_TRAILING_SPACES` to inform the application without requiring extra text processing logic for the application.

Return values

Value	Description
<code>S_OK</code>	Function completed successfully.
<code>S_FALSE</code>	A phrase that does not contain text or <i>ppszCoMemText</i> is NULL.
<code>E_INVALIDARG</code>	One or more parameters are invalid.
<code>E_POINTER</code>	Invalid pointer.
<code>E_OUTOFMEMORY</code>	Exceeded available memory.

Remarks

The text is the display text of the elements for the phrase and constructs a text string created by `CoTaskMemAlloc` by applying the *pbDisplayAttributes* of each [SPPHRASEELEMENT](#).

Example

The following code snippet illustrates the use `ISpPhrase::GetText` to retrieve parts of the recognized phrase.

```
HRESULT hr = S_OK;
```

```
// ... obtain a recognition result object from the recognizer...
```

```
// get the recognized phrase object
hr = cpRecoResult->GetPhrase(&pPhrase);
// Check hr

// get the phrase's entire text string, including replacements
hr = pPhrase->GetText(SP_GETWHOLEPHRASE, SP_GETWHOI
// Check hr

// get the phrase's first 2 words, excluding replacements
hr = pPhrase->GetText(pPhrase->Rule.ulFirstElement, 2, FALSE, &
```



ISpPhrase::Discard

ISpPhrase::Discard discards the requested data from a phrase object.

```
HRESULT Discard(  
    DWORD    dwValueTypes  
;)
```

Parameters

dwValueTypes

[in] Flags of type [SPVALUETYPE](#) indicating elements to discard. Multiple values may be combined.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>dwValueTypes</i> is not a valid value type flag.
FAILED(hr)	Appropriate error message.

Remarks

Applications that have no use for certain types of retained data, and will be persisting or serializing the phrase or result objects, may discard the unnecessary data. For example, an application performing offline transcription may need to retain only the audio and the final result. It can call `::Discard` with `SPDF_ALTERNATES` to eliminate the alternate data (possibly including a large amount of private engine data).

Note that once retained audio is discarded, a call to [ISpRecoResult::GetAudio](#) will fail.

Example

The following code snippet illustrates the use of `ISpRecoResult::Discard` as inherited from `ISpPhrase` to discard the retained audio.

```
HRESULT hr = S_OK;  
  
// .. get a recognition result object from the SR engine  
  
// discard audio  
hr = cpRecoResult->Discard(SPDF_AUDIO);  
// Check hr  
  
// .. serialize the "shrunk" result to the disk ...
```



ISpPhraseAlt

The ISpPhraseAlt interface is implemented on a phrase alternate object that can be obtained by calling [ISpRecoResult::GetAlternates](#). The ISpPhraseAlt object is the interface that enables applications to retrieve alternate phrase information from an SR engine, and to update the SR engine's language model to reflect committed alternate changes.

Methods in Vtable Order

ISpPhraseAlt Methods	Description
ISpPhrase	Inherits from ISpPhrase and those methods are accessible from an ISpPhraseAlt object.
GetAltInfo	Retrieves data elements associated with an alternate phrase.
Commit	Replaces a section of the parent phrase to which this alternate corresponds.



ISpPhraseAlt::GetAltInfo

ISpPhraseAlt::GetAltInfo retrieves data elements associated with an alternate phrase.

```
HRESULT GetAltInfo(  
    ISpPhrase    **ppParent,  
    ULONG        *pulStartElementInParent,  
    ULONG        *pcElementsInParent,  
    ULONG        *pcElementsInAlt  
);
```

Parameters

ppParent

[out] Address to store the pointer to the parent SpPhrase object.

pulStartElementInParent

[out] Address to store the starting element position within the parent phrase that this alternate applies to.

pcElementsInParent

[out] Address to store the number of elements within the parent that this alternate replaces.

pcElementsInAlt

[out] Address to store the number of elements that the alternate contains.

Return values

Value	Description

S_OK	Function completed successfully.
E_INVALIDARG	At least one of the parameters is invalid or bad.
SPERR_NOT_FOUND	The alternate is not associated with a valid parent phrase object.



ISpPhraseAlt::Commit

ISpPhraseAlt::Commit replaces a section of the parent phrase to which the alternate corresponds.

```
HRESULT Commit ( void );
```

Parameters

None

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_NOT_FOUND	The alternate object is not related to a valid parent phrase object.

Remarks

After an alternate has been committed, the parent phrase will be modified to reflect the substitution.

Upon committing the alternate phrase, the SR engine also has the ability to update its language model to improve future recognitions of the same or similar phrases (see [ISpSRAlternates::Commit](#)).

Example

The following code snippet illustrates the use of `ISpPhraseAlt::Commit` to commit an alternate phrase.

```
HRESULT hr = S_OK;  
  
// ... obtain a recognition result object from the recognizer...
```

```

// get the recognized phrase object
hr = cpRecoResult->GetPhrase(&pPhrase);
// Check hr

// get the phrase's text
hr = pPhrase->GetText(SP_GETWHOLEPHRASE, SP_GETWHOI
// Check hr

// ... check the phrase's text... assume the phrase isn't a correct recogni

// setup MY_MAX_ALTERNATES phrase alternate objects
CComPtr<ISpPhraseAlt> pcpPhrase[MY_MAX_ALTERNATES];
ULONG ulCount;

// get the top MY_MAX_ALTERNATES alternates to the entire reco
hr = cpRecoResult->GetAlternates(pPhrase->Rule.ulFirstElement,
                                pPhrase->Rule.ulCountOfElements,
                                MY_MAX_ALTERNATES,
                                pcpPhraseAlt,
                                &ulCount);

// Check hr

// check each alternate in order of highest likelihood
for (int i = 0; i < ulCount; i++) {
    hr = pcpPhraseAlt[i]->GetText(SP_GETWHOLEPHRASE, SP_(
    // Check hr

    // ... check if this alternate is more appropriate ...

    // if it is more appropriate, then commit the alternate
    if (fMoreAppropriate) {
        hr = pcpPhraseAlt[i]->Commit();
        // Check hr

```

```
}
```

```
// free the alternate text
```

```
if (pwszAlternate) ::CoTaskMemFree(pwszAlternate);
```

```
}
```

```
// free the initial phrase object
```

```
if (pPhrase) ::CoTaskMemFree(pPhrase);
```



ISpProperties

ISpProperties sets and retrieves property attribute information.

This interface is currently implemented only by the SR engine ([ISpRecognizer](#)), and can be used for setting SR engine properties (see [SAPI 5.0 SR Properties White Paper](#)).

Methods in Vtable Order

ISpProperties Methods	Description
SetPropertyNum	Sets a numeric property corresponding to the specified name.
GetPropertyNum	Retrieves a numeric value specified by the named key.
SetPropertyString	Sets a text property corresponding to the specified name.
GetPropertyString	Retrieves the string value corresponding to the specified key name.



ISpProperties::SetPropertyNum

ISpProperties::SetPropertyNum sets a numeric property corresponding to the specified name.

```
HRESULT SetPropertyNum(  
    const WCHAR    *pName,  
    LONG           lValue  
);
```

Parameters

pName

[in] Null-terminated string containing the property name. Valid values are listed in the *SR Properties* section of the [SAPI 5.0 SR Properties White Paper](#).

lValue

[in] The property value to set.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	SR engine does not support specified property name.
E_INVALIDARG	One or more parameters are invalid.
FAILED(hr)	SR engine returned specific error.

Remarks

If the SR engine supports the property, SAPI will fire a property-changed event (see [SPEI_PROPERTY_NUM_CHANGE](#)) to all interested recognizer contexts ([ISpRecoContext](#)). Broadcasting the corresponding event notifies any recognizer contexts that

had interests in the property (see [CSpEvent::PropertyName](#) and [CSpEvent::PropertyNumValue](#)).



ISpProperties::GetPropertyNum

ISpProperties::GetPropertyNum retrieves a numeric value specified by the named key.

```
HRESULT GetPropertyNum(  
    const WCHAR    *pName,  
    LONG           *pIValue  
);
```

Parameters

pName

[in] String containing the property name. Valid values are listed in the *SR Properties* section of the [SAPI 5.0 SR Properties White Paper](#).

pIValue

[out] Address to store the property value.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	SR engine does not support specified property name.
E_INVALIDARG	One or more parameters are invalid.
E_POINTER	Value pointer is invalid.
FAILED(hr)	SR engine returned specific error.



ISpProperties::SetPropertyString

ISpProperties::SetPropertyString sets a text property corresponding to the specified name.

```
HRESULT SetPropertyString(  
    const WCHAR    *pName,  
    const WCHAR    *pValue  
);
```

Parameters

pName

[in, string] Null-terminated string containing the property name.

pValue

[in, string] Null-terminated string containing the property value.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	SR engine does not support specified property name.
E_INVALIDARG	One or more parameters are invalid.
FAILED(hr)	SR engine returned specific error.

Remarks

If the SR engine supports the property, SAPI will fire a property-changed event (see [SPEI_PROPERTY_STRING_CHANGE](#)) to all interested recognizer contexts ([ISpRecoContext](#)). Broadcasting the corresponding event notifies any recognizer contexts that

had interests in the property (see [CSpEvent::PropertyName](#) and [CSpEvent::PropertyStringValue](#)).



ISpProperties::GetPropertyString

ISpProperties::GetPropertyString retrieves the string value corresponding to the specified key name.

```
HRESULT GetPropertyString(  
    const WCHAR    *pName,  
    WCHAR          **ppCoMemValue  
);
```

Parameters

pName

[in] Null-terminated string containing the property name. Valid values are listed in the *SR Properties* section of the [SAPI 5.0 SR Properties White Paper](#).

ppCoMemValue

[out] Address to store the pointer to the string value. The caller must call `CoTaskMemFree()` to free the string pointer.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	SR engine does not support specified property name.
E_INVALIDARG	One or more parameters are invalid.
E_POINTER	Value pointer is invalid.
FAILED(hr)	SR engine returned specific error.



Text-to-speech engine interfaces (API-level)

The following section covers:

- [TTS Overview](#)
- [TTS Engine Characteristics](#)
- [Text Synthesis](#)
- [ISpVoice](#)



Text-to-Speech Overview

ISpVoice Introduction

The central SAPI API for text-to-speech (TTS) is ISpVoice. Using this interface, applications can add TTS support such as speaking text, modifying speech characteristics, changing voices, as well as responding to real-time events while speaking. In fact, most applications should need only this single interface to accomplish everything that is needed for basic TTS support.

Applications obtain access to ISpVoice interface methods by creating a COM object. As the name implies, an ISpVoice object is simply a single instance of a specific TTS voice. Every ISpVoice object is an individual voice. Even if two different ISpVoice objects select the same base voice (for example "Mike"), each of the two voices can be changed and modified independently of the other.

Speaking

When an application first creates an ISpVoice object, the object initializes to the default voice (set in Speech properties of Control Panel). This means that the new object is immediately ready to speak text, no special initialization is needed. At this point, applications can use [Speak](#) or [SpeakStream](#) to speak any Unicode text data.

Synchronous vs. Asynchronous Speaking

The two [speaking functions](#) can generate speech either synchronously (function does not return until text has completely spoken) or asynchronously (function returns immediately but continues speaking as a background process). Asynchronous operation is chosen if the application needs to do something else (highlight text, paint animation, monitor

controls, etc.) while speaking. Otherwise, the simplest case is to speak synchronously.

Getting Status Information

During asynchronous speech, applications can get current status information (text position, speech done state, bookmarks, etc.) in one of two ways. The simplest way is to periodically poll the ISpVoice object using the [GetStatus](#) method. The other way is to initialize the ISpVoice object so that it sends real-time events to the application as they happen.

Flow Control

As a convenience, most TTS applications allow users to temporarily suspend speech output. The [Pause](#) and [Resume](#) methods are typically called in response to a user initiated action.

Modifying Voice Attributes

Often with TTS, voice output needs to be modified from its default setting. There are two ways to do this is; either by calling certain [ISpVoice API methods](#), or by embedding special Extended Markup Language (XML) tags within the spoken text. Typically, the API functions are used as global settings that affect the speech independent of current selected voice or document that is spoken. While the XML tags are usually used in much narrower scope, affecting only the spoken style in a single document.

Audio Output

Although usually the default for desktop applications, audio output for TTS is not restricted to hardware sound card

destinations. SAPI TTS supports, either directly or indirectly, just about any audio configuration an application may require. Whether the destination is a PC sound card, buffer in memory, or a special telephony hardware, ISpVoice has several [audio control methods](#) to change the audio path from its default configuration.

ISpVoice Methods

Speaking Text

Speak	Speaks a text string or file.
SpeakStream	Speaks a text stream or plays an audio (WAV) stream.

Real-time Status

GetStatus	Returns current speech and event status information.
WaitUntilDone	Delays until either the voice has completed speaking or the specified time interval has elapsed.
SpeakCompleteEvent	Returns an event handle that will be signaled when speech is done.

Flow Control

Pause	Pauses the output speech at the nearest alert boundary.
-----------------------	---

Resume	Resumes speaking.
Skip	Skips ahead or backward to a new input text position while speaking.

Changing Voice Attributes

SetRate	Sets the speaking rate in real time.
GetRate	Returns the current speaking rate.
SetVolume	Sets the speech volume level in real time.
GetVolume	Returns the current speech volume level.
SetVoice	Sets the identity of the voice used for synthesis.
GetVoice	Retrieves the object token that identifies the current voice.

Real-time Event Management (inherited from [**ISpEventSource**](#))

SetInterest	Sets the type of events to queue.
GetEvents	Returns the queued events.
GetInfo	Returns information about the event queue.
SetNotifySink	Sets up the instance to make free-threaded calls through <code>ISpNotifySink::Notify</code> .
SetNotifyWindowMessage	Sets a window handle to receive notifications as window messages.

<u>SetNotifyCallbackFunction</u>	Sets a callback function to receive notifications.
<u>SetNotifyCallbackInterface</u>	Sets an object derived from <u>ISpTask</u> to receive notifications.
<u>SetNotifyWin32Event</u>	Sets up a Win32 event object to be used by this instance for notifications.
<u>WaitForNotifyEvent</u>	A blocking call which waits for a notification.
<u>GetNotifyEventHandle</u>	Retrieves Win32 event handle associated with this notify source.

Audio Output Control

<u>SetOutput</u>	Sets the current output object. A value of NULL may be used to select the default audio device.
<u>GetOutputStream</u>	Retrieves a pointer to the current output stream.
<u>GetOutputObjectToken</u>	Retrieves the object token for the current output object.

Miscellaneous

<u>SetPriority</u>	Sets the priority for the voice.
<u>GetPriority</u>	Retrieves the current voice priority level.
<u>SetAlertBoundary</u>	Specifies which event should be used as the insertion point for alerts.
<u>GetAlertBoundary</u>	Retrieves the event that is currently being used as the insertion point for

	alerts.
<u>IsUISupported</u>	Determines if the specified type of UI is supported.
<u>DisplayUI</u>	Displays the requested UI.
<u>SetSyncSpeakTimeout</u>	Sets the timeout interval in milliseconds after which, synchronous Speak and SpeakStream calls to this instance of the voice will timeout.
<u>GetSyncSpeakTimeout</u>	Retrieves the timeout interval for synchronous speech operations for this ISpVoice instance.



TTS Engine Characteristics

Engines use the three characteristics of Volume, Pitch, and Rate to partially define speech traits. At the application level, setting these values is simple; you need only set them to a given number. However, implementation of these traits is more complex for the engine.

Volume

At the application level, volume is a number from zero to 100 where 100 is the maximum value for a voice. It is a linear progression and a value of 50 represents half of the loudest permitted. The increments should be the range divided by 100.

Pitch adjustment

The value can range from -10 to +10. A value of zero sets a voice to speak at its default pitch. A value of -10 sets a voice to speak at three-fourths of its default pitch. A value of +10 sets a voice to speak at four-thirds of its default pitch. Each increment between -10 and +10 is logarithmically distributed such that incrementing or decrementing by 1 is multiplying or dividing the pitch by the 24th root of 2 (about 1.03). Values outside of the -10 and +10 range will be passed to an engine. However, SAPI 5-compliant engines may not support such extremes and may clip the pitch to the maximum or minimum the engine supports. Values of -24 and +24 must lower and raise pitch by 1 octave respectively. All incrementing or decrementing by 1 must multiply or divide the pitch by the 24th root of 2.

Rate adjustment

The value can range from -10 to +10. A value of zero sets a voice to speak at its default rate. A value of -10 sets a voice to speak at one-third of its default rate. A value of +10 sets a voice to speak at three times its default rate. Each increment between -10 and +10 is logarithmically distributed such that incrementing or decrementing by 1 is multiplying or dividing the rate by the 10th root of 3 (about 1.1). Values more extreme than -10 and +10 will be passed to an engine. However, SAPI 5-compliant engines may not support such extremes and may clip the rate to the maximum or minimum rate the engine supports.



Text synthesis

SAPI 5 uses the Extensible Markup Language (XML) to define text synthesis characteristics and application configuration settings.

A text-to-speech (TTS) engine that uses synthesis generates sounds similar to those created by the human voice and applies various filters to simulate throat length, mouth cavity, lip shape, and tongue position. Although the voice produced through text synthesis often sounds less human than a voice produced by diphone concatenation, it is possible to obtain different qualities of voice through modifying TTS configuration settings. SAPI 5-compliant TTS engines can achieve improved synthesized text-to-speech voice qualities using XML to control the configuration settings for text synthesis.

The following section covers:

- [Synthesis markup](#)
- [English Context tag definitions](#)
- [Chinese Context tag definitions](#)
- [Japanese Context tag definitions](#)



Synthesis Markup

SAPI 5 synthesis markup is the collection of XML tags inserted into text to modify the speech synthesis of that text. These XML tags, which provide functionality such as volume control and word emphasis, are inserted into text passed into `ISpVoice::Speak` and text streams of format `SPDFID_XML` which are then passed into `ISpVoice::SpeakStream`. By default, the SAPI XML parser auto-detects XML. In the case of an invalid XML structure, a speak error may be returned to the application. SAPI is not intended to be used to validate the XML structure, as it is the responsibility of the developer to validate the XML with an XML validation tool. Please see [ISpVoice](#) for more information.

SAPI 5 synthesis markup is an application of XML. Every XML element consists of a start tag `<Some_tag>` and an end tag `</Some_tag>` with a case-insensitive tag name and contents between these tags. If the element is empty, it has no contents `<Some_tag></Some_tag>` and the start tag and the end tag might be the same `<Some_tag/>`. More information about XML and the XML specification is available at: <http://www.w3.org/TR/1998/REC-xml-19980210.html>.

The following section covers:

- [SAPI 5 XML tags](#)
- [Attributes](#)
- [Contents](#)
- [Relationship to HTML web pages and SABLE](#)

SAPI 5 XML tags

XML tags in SAPI 5 follow a defined structure program scope and implementation. SAPI 5 XML tags have a specific purpose and affect the input text in a predetermined manner.

The SAPI 5 XML tags are divided into four different scope categories.

1. Non-scoped
2. Scoped
3. Global
4. Scoped/Global

The modification and properties can be controlled through the use of XML tags. Additional information on SAPI 5 XML elements is available at: [SAPI XML Schema](#).

Attributes

Attributes of an XML element appear inside the start tag. Each attribute is in the form of a name, followed by an equal character, followed by a quoted string value. An attribute of a given name may only appear once in a start tag. Exact details on what characters may appear between quotes can be found at <http://www.w3.org/TR/REC-xml#NT-AttValue>.

Briefly, the literal string cannot contain a less than character "<" if the string is surrounded by single quotation marks, it cannot contain a single quotation mark. If the string is surrounded by double quotation marks it cannot contain a double quotation mark. Furthermore, all ampersands (&) can be used only in an entity reference such as & and ">". When a literal string is parsed, the resulting replacement text will resolve all entity references such as ">" into its corresponding text, such as ">".

In this specification, only the resulting replacement text needs to be defined for attribute value strings. The XML specification defines the exact file syntax details. Character references allow entity references in ASCII characters to specify replacement text which has unprintable characters such as extended Unicode characters. The entity reference "ə" specifies the single Unicode character for the International Phonetic Alphabet symbol for a mid-central unrounded vowel. See <http://www.w3.org/TR/1998/REC-xml-19980210#sec-references> for details.

The <LANG> and <VOICE> XML tags are specific to the Microsoft engines and provide support for language and dialect attributes for a given voice.

The following is an example of what 409;9 refers to and how to correctly use it in XML tags:

```
<LANG LANGID="409">This is the US English language</LANG>
<LANG LANGID="9">This is the English Language</LANG>
<VOICE REQUIRED="language=409">This is the required voice fo
<VOICE REQUIRED="language=9">This is the required voice that
```

A speak error will occur when entering voice attribute information as it appears in the Windows Registry:

For example:

```
<LANG LANGID="409;9">Speak this text with the US English lan
<VOICE REQUIRED="language=409;9">Require a voice be used tha
```

In the Windows Registry, the language attribute for the Microsoft SAPI 5 English voices is labeled as '409;9'. The '409' attribute information indicates the voice is specifically US English, and '9' refers to the English language. This language labeling convention for voices may not be followed by all engine manufacturers. For example, the LH voices may use '409' to indicate an English voice, while Microsoft uses '409;9' to specify the voice is specifically US English.

For example:

409;9 = US English

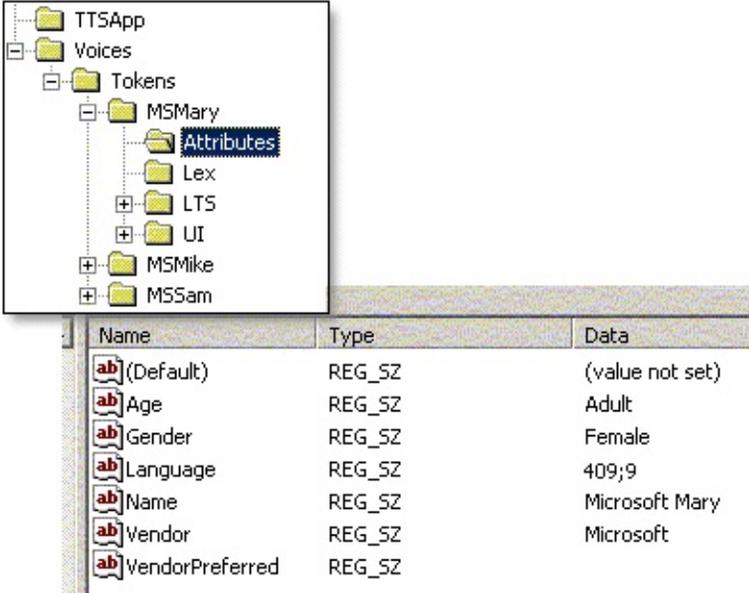
809;9 = British English

Start RegEdit and expand the tree view pane to the following registry key location:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Voices\Tokens

Select one of the available voices and view the corresponding attribute information.

The following is an example of the MSMary voice attributes:



Name	Type	Data
(Default)	REG_SZ	(value not set)
Age	REG_SZ	Adult
Gender	REG_SZ	Female
Language	REG_SZ	409;9
Name	REG_SZ	Microsoft Mary
Vendor	REG_SZ	Microsoft
VendorPreferred	REG_SZ	

Contents

The contents of an element consist of text or sub-elements. With these definitions, the XML specification defines the exact file syntax details.

Relationship to HTML web pages and SABLE

The XML format that SAPI 5 uses is NOT placed inside web pages. Web page authors who want to mark up sections of HTML text so that it is synthesized correctly, should use the W3C Aural Cascading Style Sheets (ACSS). More information is available at: <http://www.w3.org/TR/WD-acss>

SAPI applications that are synthesizing text from a web page will "render" HTML+ACSS into SAPI's synthesis markup format. Programs apply a default ACSS file when synthesizing web pages that do not have an associated ACSS file.

SAPI 5 synthesis markup format is similar to the format published by the SABLE Consortium. However, this format and SABLE version 1.0 are not interoperable. At this time, it's not determined if they will become partially interoperable in the future. More information about the SABLE specification is available at: <http://www.bell-labs.com/project/tts/sable.html>.

[Back to top](#)



English Context tag definitions

The CONTEXT tag specifies the normalization of a block of text. This specification defines the SAPI predefined attributes (ID) for the CONTEXT tag. These IDs are strings. SAPI does not perform any parameter validation on the string passed to the engine, hence, the application can specify engine-specific normalization IDs to the engine. Engine-specific strings begin with the engine vendor's name to avoid confusion between engines.

For example:

```
<CONTEXT ID = "MS_My_Context"> text </CONTEXT>
```

The exact implementation of some of these values is dependent on the engine used in SAPI 5. In order to force a certain normalization, application developers can choose to normalize the text, or use another SAPI tag or engine-specific ID. Each context tag can contain more than one string.

For example:

```
<CONTEXT ID = "date_mdy"> 12/21/99 11/21/99 10/21/99  
</CONTEXT> would be normalized to "December twenty  
first nineteen ninety nine November twenty first nineteen  
ninety nine October twenty first nineteen ninety nine."
```

The following predefined context types are covered in this section:

- [Date](#)
- [Time](#)
- [Number](#)
- [Phone_Number](#)
- [Currency](#)
- [Web](#)

- [E-mail](#)
- [Address](#)

Date

This context specifies that the number passed to the engine is a date. Dates will generally have the format of number [delimiter] number [delimiter] number or number [delimiter] number where the delimiter can be a '.', '/' or '-', and numbers are typically between 01 and 12 for months, 01 and 31 for days. A year is generally a two or four-digit number.

The following are valid string types:

date_mdy

This will normalize the date so that the first group of numbers is the month, the second group is the day, and the third group is the year. In the case where the year is a two-digit number, the engine reads it as a two-digit number or a four-digit number.

For example:

`<context ID = "date_mdy">12/21/99</context>`
will be normalized to "December twenty first ninety nine"
or "December twenty first nineteen ninety nine"

`<context ID = "date_mdy">12/21/1999</context>`
will be normalized to "December twenty first nineteen
ninety nine"

□ [Back to top](#)

date_dmy

This will normalize the date so that the first group of numbers is the day, the second group is the month, and the third group is the year. In the case where the year is a two-

digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is represented as a four-digit year.

For example:

`<context ID = "date_dmy">21.12.99</context>`
will be normalized to "December twenty first ninety nine"
or "December twenty first nineteen ninety nine"

`<context ID = "date_dmy">21-12-1999</context>`
will be normalized to "December twenty first nineteen
ninety nine"

[Back to top](#)

date_ymd

This will normalize the date so that the first group of numbers is the year, the second group is the month, and the third group is the day. In the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is represented as a four-digit year.

For example:

`<context ID = "date_ymd">99-12-21</context>`
will be normalized to "December twenty first ninety nine"
or "December twenty first nineteen ninety nine"

`<context ID = "date_ymd">1999.12.21</context>`
will be normalized to "December twenty first nineteen
ninety nine"

[Back to top](#)

date_ym

This will normalize the date so that the first group of numbers is the year, and the second group is the month. In

the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is represented as a four-digit year.

For example:

```
<context ID = "date_ym">99-12</context>  
will be normalized to "December ninety nine"  
or "December nineteen ninety nine"  
<context ID = "date_ym">1999.12</context>  
will be normalized to "December nineteen ninety nine"
```

☐ [Back to top](#)

date_my

This will normalize the date so that the first group of numbers is the month, and the second group is the year. In the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is represented as a four-digit year.

For example:

```
<context ID = "date_my">12/99</context>  
will be normalized to "December ninety nine"  
or "December nineteen ninety nine"  
<context ID = "date_my">12/1999</context>  
will be normalized to "December nineteen ninety nine"
```

☐ [Back to top](#)

date_dm

This will normalize the date so that the first group of numbers is the day and the second group is the month.

For example:

```
<context ID = "date_dm">21.12</context>  
will be normalized to "December twenty first"
```

[Back to top](#)

date_md

This will normalize the date so that the first group of numbers is the month and the second group is the day.

For example:

`<context ID = "date_md">12/21</context>`
will be normalized to "December twenty first"

[Back to top](#)

date_year

This will normalize the date so that the number is read as a year.

For example:

`<context ID = "date_year">1999</context>`
will be normalized to "nineteen ninety nine"

`<context ID = "date_year">2001</context>`
will be normalized to "Two thousand one"

[Back to top](#)

Time

This context specifies that the number passed to the engine is a time. Times will generally have the format of number [delimiter] number [delimiter] number or number [delimiter] number where the delimiter is ':' or ' ' or ' "' and numbers are typically between 01 and 24 for hours, 01 and 59 for minutes and seconds.

When a zero is present in numbers between 01 and 09, the engine can ignore this, or normalize it as "oh". The engine can place an "and" in the normalized time. The valid string types are:

For example:

`<context ID = "time">12:30</context>`
will be normalized to "twelve thirty"

`<context ID = "time">01:21</context>`
is normalized as "one twenty one"
or "oh one twenty one"

`<context ID = "time">1'21"</context>`
is normalized as "one minute twenty one seconds"
or "one minute and twenty one seconds"

[□ Back to top](#)

Number

number_cardinal

The text is normalized as a number using the regular format of ones, tens, etc. The engine can place "and" in the normalized text.

For example:

`<context ID = "number_cardinal">3432</context>`
will be normalized to "three thousand four hundred thirty two"

`<context ID = "number_cardinal">3432</context>`
will be normalized to "three thousand four hundred and thirty two"

[□ Back to top](#)

number_digit

The text is normalized digit by digit.

For example:

`<context ID = "number_digit">3432</context>`

will be normalized to "three four three two"

[Back to top](#)

number_fraction

The text is normalized as a fraction.

For example:

`<context ID = "number_fraction">3/15</context>`
will be normalized to "three fifteenths" or "three over fifteen"

[Back to top](#)

number_decimal

The text is normalized as a decimal value.

For example:

`<context ID = "number_decimal">423.1243</context>`
will be normalized to "four hundred and twenty three point one two four three"

[Back to top](#)

Phone_Number

The text is normalized as a phone number. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI.

[Back to top](#)

Currency

The text is normalized as a currency. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI.

For example:

<context ID = "currency">\$34.90</context>
will be normalized to "thirty four dollars and ninety cents"

[Back to top](#)

Web

The text is normalized as a URL. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI.

web_url

For example:

<context ID = "web_url">www.Microsoft.com</context>
will be normalized to "is normalized to "w w w dot Microsoft dot com"

[Back to top](#)

E-mail

The text is normalized as e-mail. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI.

E-mail_address

The text is normalized as an e-mail address. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI.

For example:

<context ID = "E-mail_Address">someone@microsoft.com</context>
is normalized to "Someone at Microsoft dot com"

[□ Back to top](#)

Address

The text is normalized as an address. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI.

For example:

```
<context ID = "address">One Microsoft Way, Redmond, WA,  
98052</context>  
will be normalized to "One Microsoft Way Redmond  
Washington nine eight zero five two"
```

address_postal

The text is normalized as a postal address. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI.

For example:

```
<context ID = "address_postal">A2C 4X5</context>  
will be normalized to "A 2 C 4 X 5"
```

[□ Back to top](#)



Chinese Context tag definitions

The CONTEXT tag specifies the normalization of a block of text. The context tag ID attribute contains the text string specifying the type of normalization to apply to the text block. The ID attribute of the Chinese engine specific context tags begin with "CHS_". CONTEXT tag ID attribute strings that do not begin with "CHS_" use English context text normalization.

For example:

```
<CONTEXT ID = "CHS_My_Context"> 12/21/99 </CONTEXT>
```

Each context tag can contain more than one string.

For example:

```
<CONTEXT ID = "date_mdy"> 12/21/99 11/21/99 10/21/99  
</CONTEXT>
```

is normalized to "Nine nine Nian twelve Yue twenty-one Ri nine nine Nian eleven Yue twenty-one Ri nine nine Nian ten Yue twenty-one Ri"

The following topics are covered in this section:

- [Date](#)
- [Time](#)
- [Number](#)
- [Phone Number and Postal Address](#)

Date

This context specifies that the number passed to the engine is a date. Dates generally have the format of number [delimiter] number [delimiter] number or number [delimiter] number where the delimiter may be a '.', '/' or '-', and numbers are typically between 01 and 12 for months, 01 and 31 for days. A year is a two- or four-digit number.

If a date format does not fall within the range shown below, the application cannot expect a consistent result and the engine may interpret it freely. The valid string types are:

- [Date_mdy](#)
- [Date_dmy](#)
- [Date_ymd](#)
- [CHS_Date_ymdhm](#)
- [Date_ym](#)
- [Date_md](#)
- [Date_dm](#)
- [Date_year](#)

[Back to top](#)

Date_mdy

The text specifying the date is normalized so that the first group of numbers is the month, the second group is the day and the third group is the year. In the case where the year is a two-digit number, the engine reads it as a two-digit number. In the case where the year is a four-digit number, the engine reads it as a four-digit number.

This a common tag with the English engine.

Example

```
<context ID = "date_mdy">12/21/99</context>
```

will be normalized to "九九年十二月二十一日(nine nine Nian twelve Yue twenty-one Ri)"

```
<context ID = "date_mdy">12/21/1999</context>
```

will be normalized to "一九九九年十二月二十一日(one nine nine nine Nian twelve Yue twenty-one Ri)"

[Back to top](#)

Date_dmy

The text specifying the date is normalized so that the first group of numbers is the day, the second group is the month and the third group is the year. In the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is represented as a four-digit year.

This a common tag with the English engine.

Example

```
<context ID = "date_dmy">21.12.99</context>
```

will be normalized to "九九年十二月二十一日(nine nine Nian twelve Yue twenty-one Ri)"

```
<context ID = "date_dmy">21-12-1999</context>
```

will be normalized to "一九九九年十二月二十一日(one nine nine nine Nian twelve Yue twenty-one Ri)"

[Back to top](#)

Date_ymd

The text specifying the date is normalized so that the first group of numbers is the year, the second group is the month and the third group is the day. In the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is represented as a four-digit year.

This a common tag with the English engine.

Example

<context ID = "date_ymd">99-12-21</context>

will be normalized to "九九年十二月二十一日(nine nine Nian twelve Yue twenty-one Ri)"

<context ID = "date_ymd">1999.12.21</context>

will be normalized to "一九九九年十二月二十一日(one nine nine nine Nian twelve Yue twenty-one Ri)"

[Back to top](#)

CHS_Date_ymdhm

The text specifying the date is normalized so that the first group of numbers is the year, the second group is the month, the third group is the day, the fourth group is the hour and the fifth group is the minute. In the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is represented as a four-digit year. The text format for time needs to specify the hour as a number between 00 and 23 (in case of with PM/AM before, between 01 and 12), and the minute as a number between 01 and 59.

- This is a Chinese specific context tag.

Example

<context ID = "CHS_date_ymdhm">99-12-21 1:30</context>

will be normalized to "九九年十二月二十一日一时三十分(nine nine Nian twelve Yue twenty-one Ri one Shi thirty Fen)"

<context ID = "CHS_date_ymdhm">99-12-21 1:30 PM</context>

will be normalized to "九九年十二月二十一日下午一时三十分(nine nine Nian twelve Yue twenty-one Ri Xia Wu one Shi thirty Fen)"

<context ID = "CHS_date_ymdhm">99-12-21 1:30 AM</context>

will be normalized to "九九年十二月二十一日上午一时三十分(nine nine Nian twelve Yue twenty-one Ri Shang Wu one Shi thirty Fen)"

[Back to top](#)

Date_ym

The text specifying the date is normalized so that the first group of numbers is the year and the second group is the month. In the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is represented as a four-digit year.

This a common tag with the English engine

Example

```
<context ID = "date_ym">99-12</context>
```

will be normalized to "九九年十二月 (nine nine Nian twelve Yue)"

```
<context ID = "date_ym">1999.12</context>
```

will be normalized to "一九九九年十二月 (one nine nine nine Nian twelve Yue)"

[Back to top](#)

Date_my

The text specifying the date is normalized so that the first group of numbers is the month and the second group is the year. In the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is represented as a four-digit year.

This a common tag with the English engine.

Example

```
<context ID = "date_my">12/99</context>
```

will be normalized to "九九年十二月 (nine nine Nian twelve Yue)"

```
<context ID = "date_my">12/1999</context>
```

will be normalized to "一九九九年十二月 (one nine nine nine Nian twelve Yue)"

[Back to top](#)

Date_md

The text specifying the date is normalized so that the first group of numbers is the month and the second group is the day.

This a common tag with the English engine.

Example

```
<context ID = "date_md">12/21</context>
```

will be normalized to "十二月二十一日 (twelve Yue twenty-one Ri)"

```
<context ID = "date_md">12-21</context>
```

will be normalized to "十二月二十一日 (twelve Yue twenty-one Ri)"

[Back to top](#)

Date_dm

The text specifying the date is normalized so that the first group of numbers is the day and the second group is the month.

This a common tag with the English engine.

Example

```
<context ID = "date_dm">21/12</context>
```

will be normalized to "十二月二十一日(twelve Yue twenty-one Ri)"

[Back to top](#)

Date_year

The text specifying the date is normalized so that the number is read as a year. This year should be a four-digit number.

This a common tag with the English engine.

Example

```
<context ID = "date_year">1999</context>
```

will be normalized to "一九九九(one nine nine nine)"

[Back to top](#)

Time

This context specifies that the number passed to the engine is a time. Times generally have the format of number [delimiter] number [delimiter] number or number [delimiter] number where the delimiter is ':' and numbers are typically between 00 and 23 for hours (in case of with PM/AM before, it needs to be between 01 and 12), 00 and 59 for minutes and seconds.

- [CHS_time_hms](#)
- [CHS_time_hm](#)

When a zero is present in numbers between 01 and 09, the engine ignores it.

- This is a Chinese engine specific context tag.

example:
<context ID = "CHS_time_hm"> 12:30</context>
will be normalized to "十二时三十分"

□ [Back to top](#)

CHS_time_hms

The text specifying the time is normalized so that the first group of numbers is the hour, the second group is the minute and the third group is the second.

- This is a Chinese engine specific context tag.

Example
<context ID = "CHS_time_hms">13:30:24</context>
will be normalized to "十三时三十分二十四秒(thirteen Shi thirty Fen twenty-four Miao)"

Example
<context ID = "CHS_time_hms">13:30:24 AM</context>
will be normalized to "下午十三时三十分二十四秒(Xia Wu thirteen Shi thirty Fen twenty-four Miao)"

□ [Back to top](#)

CHS_time_hm

The text specifying the time is normalized so that the first group of numbers is the hour and the second group is the minute.

- This is a Chinese engine specific context tag.

Example

```
<context ID = "CHS_time_hm">13:30</context>
```

will be normalized to "十三时三十分(thirteen Shi thirty Fen)"

[Back to top](#)

Number

This context specifies how to read the number passed to the engine.

- [number_cardinal](#)
- [number_digit](#)
- [number_fraction](#)
- [number_decimal](#)
- [CHS_number_percentage](#)
- [CHS_number_scientific](#)

number_cardinal

The text is normalized as a number using the regular format of ones, tens, etc.

This a common tag with the English engine.

Example:

```
<context ID = "number_cardinal"> 3432</context>
```

will be normalized to "三千四百三十二(three thousands four hundreds and thirty-two)"

Example:

```
<context ID = "number_cardinal"> 3,432</context>
```

will be normalized to "三千四百三十二(three thousands four hundreds and thirty-two)"

Example:

```
<context ID = "number_cardinal"> -3432</context>
```

will be normalized to "负三千四百三十二(Fu three thousands four hundreds and thirty-two)"

[Back to top](#)

number_digit

The text is normalized digit by digit.

This a common tag with the English engine.

example:
<context ID = "number_digit"> 3432</context>
will be normalized to "三四三二(three four three two)"

□ [Back to top](#)

number_fraction

The text is normalized as a fraction

This a common tag with the English engine.

example:
<context ID = "number_fraction"> 3/15</context>
will be normalized to "十五分之三 (fifteen Fen Zhi three)"

□ [Back to top](#)

number_decimal

The text is normalized as a decimal value.

This a common tag with the English engine.

example:
<context ID = "number_decimal">423.1243</context>
will be normalized to "四百二十三点一二四三(four hundreds twenty-three point one two four three)"

example:
<context ID = "number_decimal">-423.1243</context>
will be normalized to "负四百二十三点一二四三(Fu four hundreds twenty-three point one two four three)"

example:
<context ID = "number_decimal">-1,423.1243</context>
will be normalized to "负一千四百二十三点一二四三(Fu one thousand four hundreds and twenty-three point one two four three)"

□ [Back to top](#)

CHS_number_percentage

The text is normalized as a percentage value.

- This is a Chinese engine specific context tag.

example:

```
<context ID = "CHS_number_percentage">42%</context>
```

will be normalized to "百分之四十二(Bai Fen Zhi forty-two)"

example:

```
<context ID = "CHS_number_percentage">-42%</context>
```

will be normalized to "负百分之四十二(Fu Bai Fen Zhi forty-two)"

example:

```
<context ID = "CHS_number_decimal">42.1%</context>
```

will be normalized to "百分之四十二点一(Bai Fen Zhi forty-two point one)"

[Back to top](#)

CHS_number_scientific

The text is normalized as a scientific value.

- This is a Chinese engine specific context tag.

example:

```
<context ID = "CHS_number_scientific">1.23E+02</context>
```

will be normalized to "一点二三乘以十的二次方(one point two three Cheng Yi ten De two Ci Fang)"

example:

```
<context ID = "CHS_number_scientific">1.23E-12</context>
```

will be normalized to "一点二三乘以十的负十二次方(one point two three Cheng Yi ten De Fu twelve Ci Fang)"

[Back to top](#)

Phone_Number/address_postal

The text is normalized as a phone number or address postal.

- [CHS_phone_postal](#)

CHS_phone_postal

The phone number and address postal is almost the same as that of number_cardinal--the only difference is that "1" is read as "yao1"

- This is a Chinese engine specific context tag.

Example:

```
<context ID = "CHS_phone_postal"> 62617711</context>
```

will be normalized to "六二六幺七七幺幺(six two six Yao seven seven Yao Yao)"

[Back to top](#)



Japanese Context tag definitions

The CONTEXT tag specifies the normalization of a block of text. This specification defines the SAPI predefined attributes (ID) for the CONTEXT tag. These IDs are strings. SAPI does not validate any parameters on the string passed to the engine, and hence, the application can specify engine-specific normalization IDs to the engine. Engine-specific strings begin with the engine vendor's name to avoid confusion between engines.

For example:

```
<CONTEXT ID = "MS_My_Context"> text </CONTEXT>
```

The exact implementation of some of these values is dependent on the engine in SAPI 5. In order to force a certain normalization, application developers can choose to normalize the text, or use another SAPI tag or engine-specific ID. Each context tag can contain more than one string.

For example:

```
<CONTEXT ID = "MS_My_Context"> text1 text2 text3  
</CONTEXT>
```

A clearer example is shown in the following:

example:

```
<CONTEXT ID = "date_mdy"> 12/21/99 11/21/99 10/21/99 </CONTEXT>  
may be normalized to 九十九年十二月二十一日  
(kyuu juu kyuu nen juu ni gatsu ni juu ichi nichi)  
or 千九百九十九年十二月二十一日  
(sen kyuu hyaku kyuu juu kyuu nen juu ni gatsu ni juu ichi nichi)
```

The following topics are covered in this section:

- [Date](#)
- [Time](#)
- [Number](#)
- [Phone_Number](#)

- [Currency](#)
- [Web](#)
- [Address](#)

Date

This context specifies that the number passed to the engine is a date. Dates generally have the format of number [delimiter] number [delimiter] number or number [delimiter] number where the delimiter may be a '.', '/' or '-', and numbers are typically between 01 and 12 for months, 01 and 31 for days. A year is generally a two- or four-digit number. If a date format does not fall within the range shown below, the application cannot expect a consistent result and the engine may interpret it freely. The valid string types are:

- [date_mdy](#)
- [date_dmy](#)
- [date_ymd](#)
- [date_ym](#)
- [date_my](#)
- [date_dm](#)
- [date_md](#)
- [date_year](#)

[Back to top](#)

date_mdy

The text specifying the date is normalized so that the first group of numbers is the month, the second group is the day and the third group is the year. In the case where the year is a two-digit number, the engine reads it as a two-digit number or a four-digit number.

example:

```
<context ID = "date_mdy"> 12/21/99 </context>
```

will be normalized to 九十九年十二月二十一日

(kyuu juu kyuu nen juu ni gatsu ni juu ichi nichi) or 千九百九十九年十二月二十一日

(sen kyuu hyaku kyuu juu kyuu nen juu ni gatsu ni juu ichi nichi)

```
<context ID = "date_mdy"> 12/21/1999 </context>
```

will be normalized to 千九百九十九年十二月二十一日

(sen kyuu hyaku kyuu juu kyuu nen juu ni gatsu ni juu ichi nichi)

[Back to top](#)

date_dmy

The text specifying the date is normalized so that the first group of numbers is the day, the second group is the month and the third group is the year. In the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is be represented as a four-digit year.

example:

```
<context ID = "date_dmy"> 21.12.99 </context>
```

will be normalized to 九十九年十二月二十一日

(kyuu juu kyuu ne'n juu ni gatsu ni juu ichi nichi) or 千九百九十九年十二月二十一日

(sen kyuu hyaku kyuu juu kyuu nen juu ni gatsu ni juu ichi nichi)

```
<context ID = "date_dmy"> 21-12-1999 </context>
```

will be normalized to 千九百九十九年十二月二十一日

(sen kyuu hyaku kyuu juu kyuu nen juu ni gatsu ni juu ichi nichi)

[Back to top](#)

date_ymd

The text specifying the date is normalized so that the first group of numbers is the year, the second group is the month and the third group is the day. In the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is be represented as a four-digit year.

example:

```
<context ID = "date_ymd"> 99-12-21</context>
```

will be normalized to 九十九年十二月二十一日

(kyuu juu kyuu nen juu ni gatsu ni juu ichi nichi) or 千九百九十九年十二月二十一日

(sen kyuu hyaku kyuu juu kyuu nen juu ni gatsu ni juu ichi nichi)

```
<context ID = "date_ymd"> 1999.12.21</context>
```

will be normalized to 千九百九十九年十二月二十一日

(sen kyuu hyaku kyuu juu kyuu nen juu ni gatsu ni juu ichi nichi)

[Back to top](#)

date_ym

The text specifying the date is normalized so that the first group of numbers is the year and the second group is the month. In the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is be represented as a four-digit year.

example:

```
<context ID = "date_ym"> 99-12</context>
```

will be normalized to 九十九年十二月

(kyuu juu kyuu nen juu ni gatsu) or 千九百九十九年十二月

(sen kyuu hyaku kyuu juu kyuu nen juu ni gatsu)

```
<context ID = "date_ym"> 1999.12</context>
```

will be normalized to 千九百九十九年十二月

(sen kyuu hyaku kyuu juu kyuu nen juu ni gatsu)

[Back to top](#)

date_my

The text specifying the date is normalized so that the first group of numbers is the month and the second group is the year. In the case where the year is a two-digit number, the engine reads it as a two-digit number. If the year is represented as a four-digit number, it is be represented as a four-digit year.

example:

```
<context ID = "date_ym"> 99-12</context>
```

will be normalized to 九十九年十二月

(kyuu juu kyuu nen juu ni gatsu) or 千九百九十九年十二月

(sen kyuu hyaku kyuu juu kyuu nen juu ni gatsu)

```
<context ID = "date_ym"> 1999.12</context>
```

will be normalized to 千九百九十九年十二月

(sen kyuu hyaku kyuu juu kyuu nen juu ni gatsu)

[Back to top](#)

date_dm

The text specifying the date is normalized so that the first group of numbers is the day and the second group is the month.

example:

```
<context ID = "date_dm"> 21.12</context>
```

will be normalized to 十二月二十一日 (juu ni gatsu ni juu ichi nichi)

[Back to top](#)

date_md

The text specifying the date is normalized so that the first group of numbers is the month and the second group is the day.

example:

```
<context ID = "date_md"> 12/21</context>
```

will be normalized to 十二月二十一日 (juu ni gatsu ni juu ichi nichi)

[Back to top](#)

date_year

The text specifying the date is normalized so that the number is read as a year.

example:

<context ID = "date_year"> 1999</context>

will be normalized to 千九百九十九年 (sen kyuu hyaku kyuu juu kyuu nen)

example:

<context ID = "date_year"> 2001</context>

will be normalized to 二千一年 (ni sen ichi nen)

[Back to top](#)

Time

This context specifies that the number passed to the engine is a time. Times generally have the format of number [delimiter] number [delimiter] number or number [delimiter] number where the delimiter is ':' or '"' or ' "' and numbers are typically between 01 and 24 for hours, 01 and 59 for minutes and seconds.

When a zero is present in numbers between 01 and 09, the engine can ignore this, or normalize it as "oh". The engine can also place an "and" in the normalized time. The valid string types are:

example:
<context ID = "time"> 12:30</context>
will be normalized to 十二時三十分 (juu ni ji san juppun)

example:
<context ID = "time"> 01:21</context>
may be normalized as 一時二十一分 (ichi ji ni juu ippun)

example:
<context ID = "time"> 1'21</context>
may be normalized as 一分二十一秒 (ip pun ni juu ichi byou)

[Back to top](#)

Number

- [number_cardinal](#)
- [number_digit](#)
- [number_fraction](#)
- [number_decimal](#)

□ [Back to top](#)

number_cardinal

The text is normalized as a number using the regular format of ones, tens, etc. The engine can place "and" in the normalized text.

example:

```
<context ID = "number_cardinal"> 3432</context>
```

will be normalized to 三千四百三十二 (san zen yon hyaku san juu ni)

example:

```
<context ID = "number_cardinal"> 3432</context>
```

will be normalized to 三千四百三十二 (san zen yon hyaku san juu ni)

□ [Back to top](#)

number_digit

The text is normalized digit by digit.

example:

```
<context ID = "number_digit"> 3432</context>
```

will be normalized to 三四三二 (san yon san ni)

□ [Back to top](#)

number_fraction

The text is normalized as a fraction.

example:

`<context ID = "number_fraction"> 3/15</context>`

will be normalized to 十五分の三 (juu go bun no san)

[Back to top](#)

number_decimal

The text is normalized as a decimal value.

example:

`<context ID = "number_decimal">423.1243</context>`

will be normalized to 四百二十三点一二四三 (yon hyaku ni juu san ten ichi ni yon san)

[Back to top](#)

Phone_Number

The text is normalized as a phone number. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI. An example is provided below:

example:
<context ID = "phone_number">423.1243</context>
may be normalized to 四二三一二四三
(yon ni san ichi ni yon san) or 四二三の一二四三
(yon ni san no ichi ni yon san)

[Back to top](#)

Currency

The text is normalized as a currency. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI. An example is provided below:

example:

```
<context ID = "currency">$34.90</context>
```

may be normalized to 三十四ドル九十セント (san juu yon doru kyuu jussento)

[□ Back to top](#)

Web

web_url

The text is normalized as a URL. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI. An example is provided below:

example:

```
<context ID = "web_url">www.Microsoft.com</context>
```

may be normalized to

ダブリエウダブリエウダブリエウドットマイクrosoftドットコム

(daburyuu daburyuu daburyuu dotto maikurosofuto dotto komu)

[Back to top](#)

E-mail

E-mail_address

The text is normalized as e-mail. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI.

<context ID = "email_address">bob@microsoft.com</context>
may be normalized to ビーオービーアットマイクロソフトドットコム
(bii oo bii atto maikurosofuto dotto komu)

[Back to top](#)

Address

The text is normalized as an address. The exact implementation of this is left to the engine developer and may be defined in a future release of SAPI. An example is provided below:

Example:

```
<context ID = "address"> 151-8553 東京都渋谷区笹塚1 - 50 - 1 </context> may be  
normalized to 郵便番号一五一の八五五三東京都渋谷区笹塚一の五十の一  
(yuubin bangou ichi go ichi no hachi go go san toukyouto shibuyaku sasazuka ichi no gojuu no  
ichi)
```

[Back to top](#)



ISpVoice

The ISpVoice interface enables an application to perform text synthesis operations. Applications can speak text strings and text files, or play audio files through this interface. All of these can be done synchronously or asynchronously.

Applications can choose a specific TTS voice using `ISpVoice::SetVoice`. The state of the voice (for example, rate, pitch, and volume), can be modified using SAPI XML tags that are embedded into the spoken text. Some attributes, like rate and volume, can be changed in real time using `ISpVoice::SetRate` and `ISpVoice::SetVolume`. Voices can be set to different priorities using `ISpVoice::SetPriority`.

ISpVoice inherits from the ISpEventSource interface. An ISpVoice object forwards events back to the application when the corresponding audio data has been rendered to the output device.

Associated Class IDs

The following class IDs (CLSID) may be used with this interface. A complete CLSID listing for all interfaces is in the Class IDs section.

CLSID_SpVoice

Methods in Vtable Order

ISpVoice Methods	Description
ISpEventSource inherited methods	All methods of ISpEventSource are accessible from this interface
SetOutput	Sets the current output object. A value of NULL may be used to select the default audio device.
GetOutputObjectToken	Retrieves the object token for the

	current audio output object.
<u>GetOutputStream</u>	Retrieves a pointer to the current output stream.
<u>Pause</u>	Pauses the voice at the nearest alert boundary and closes the output device.
<u>Resume</u>	Sets the output device to the RUN state and resumes rendering.
<u>SetVoice</u>	Sets the identity of the voice used for text synthesis. By default, <i>ISpVoice</i> will use the voice information set in Speech properties in Control Panel.
<u>GetVoice</u>	Retrieves the object token that identifies the voice used in text synthesis.
<u>Speak</u>	Speaks the contents of a text string or file.
<u>SpeakStream</u>	Speaks the contents of a stream.
<u>GetStatus</u>	Retrieves the current rendering and event status associated with this <i>ISpVoice</i> instance.
<u>Skip</u>	Causes the voice to skip forward or backward the specified number of items within the text of the current speak call.
<u>SetPriority</u>	Sets the priority for the voice. Normal, Alert, Over.
<u>GetPriority</u>	Retrieves the current voice priority level.
<u>SetAlertBoundary</u>	Specifies which event should be used as the insertion point for alerts.
<u>GetAlertBoundary</u>	Retrieves the event that is currently being used as the insertion point for alerts.

<u>SetRate</u>	Sets the text rendering rate adjustment in real time.
<u>GetRate</u>	Retrieves the current text rendering rate adjustment.
<u>SetVolume</u>	Sets the synthesizer output volume level in real time.
<u>GetVolume</u>	Retrieves the current output volume level of the synthesizer.
<u>WaitUntilDone</u>	Blocks the caller until either the voice has completed speaking or the specified time interval has elapsed.
<u>SetSyncSpeakTimeout</u>	Sets the timeout interval in milliseconds after which, synchronous Speak and SpeakStream calls to this instance of the voice will timeout.
<u>GetSyncSpeakTimeout</u>	Retrieves the timeout interval for synchronous speech operations for this ISpVoice instance.
<u>SpeakCompleteEvent</u>	Returns an event handle that will be signaled when the voice has completed speaking all pending requests.
<u>IsUISupported</u>	Determines if the specified type of UI is supported.
<u>DisplayUI</u>	Displays the requested UI.



ISpVoice::SetOutput

ISpVoice::SetOutput sets the current output object. The object may either be a stream, audio device, or an object token for an output audio device. If *pUnkOutput* is NULL the default audio device will be used.

```
HRESULT SetOutput(  
    IUnknown *pUnkOutput,  
    BOOL      fAllowFormatChanges  
);
```

Parameters

pUnkOutput

[in] IUnknown pointer to output object. The pointer must point to an object that implements ISpStreamFormat (a stream or audio device), or an object that implements ISpObjectToken. If a token is provided, this method will create the object described by the token and use it. If *pUnkOutput* is NULL, the default audio out device will be used.

fAllowFormatChanges

[in] Flag specifying whether the voice is allowed to change the format of the audio output object to match that of the engine, or a wav stream being spoken. If FALSE, the voice will use the SAPI format converter to translate between the data being rendered and the format of the output object. This should be set to TRUE if using the default audio device and the output format is of no consequence. If *pUnkOutput* is an ISpStreamFormat object, *fAllowFormatChanges* is ignored. In this case, the voice instance will render the output audio data in the format of the specified stream to the application.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
SPERR_UNINITIALIZED	<i>pUnkOutput</i> is an uninitialized ISpStream object.
E_OUTOFMEMORY	Exceeded available memory.

Example:

The following is an example of how to enumerate all the available audio output devices registered under HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\AudioOutput

```

HRESULT                hr = S_OK;
CComPtr<ISpObjectToken>    cpAudioOutToken;
CComPtr<IEnumSpObjectTokens>    cpEnum;
CComPtr<ISpVoice>        cpVoice;
ULONG                    ulCount = 0;

// Create the SAPI voice
if(SUCCEEDED(hr))
    hr = cpVoice.CoCreateInstance( CLSID_SpVoice );

//Enumerate the available audio output devices
if(SUCCEEDED(hr))
    hr = SpEnumTokens( SPCAT_AUDIOOUT, NULL, NULL, &cpEnum;

//Get the number of audio output devices
if(SUCCEEDED(hr))
    hr = cpEnum->GetCount( &ulCount; );

// Obtain a list of available audio output tokens, set the o
while ( SUCCEEDED(hr) && ulCount-- )
{
    if(SUCCEEDED(hr))
        hr = cpEnum->Next( 1, &cpAudioOutToken;, NULL );

    if(SUCCEEDED(hr))
        hr = cpVoice->SetOutput( cpAudioOutToken, TRUE );

    if(SUCCEEDED(hr))

```

```
} hr = cpVoice->Speak( L"How are you?", SPF_DEFAULT, NI
```



ISpVoice::GetOutputObjectToken

ISpVoice::GetOutputObjectToken retrieves the object token for the current audio output object.

```
HRESULT GetOutputObjectToken(  
    ISpObjectToken **ppObjectToken  
);
```

Parameters

ppObjectToken

[out] Address of an [ISpObjectToken](#) pointer that receives the audio output object token. If the current output is set to an ISpStream object, GetOutputObjectToken will return S_FALSE and *ppObjectToken* will be NULL.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppObjectToken</i> is invalid.
S_FALSE	The current output stream does not have an object token



ISpVoice::GetOutputStream

ISpVoice::GetOutputStream retrieves a pointer to the current output stream.

```
HRESULT GetOutputStream(  
    ISpStreamFormat **ppStream  
);
```

Parameters

ppStream

[out] Address of a pointer to an [ISpStreamFormat](#) object that receives the output stream.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppstream</i> is invalid.



ISpVoice::Pause

ISpVoice::Pause pauses the voice at the nearest alert boundary and closes the output device, allowing access to pending speak requests from other voices.

```
HRESULT Pause ( void );
```

Parameters

None.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message

Remarks:

Calling **ISpVoice::Pause** while the voice is not speaking increments the pause count and will put the voice into a paused state until **ISpVoice::Resume** is called the same number of times.

The voice maintains a pause count, so each call to pause must be balanced with a corresponding call to [ISpVoice::Resume](#).

The default alert boundary is at the beginning of each word. See [ISpVoice::SetAlertBoundary](#) for details.



ISpVoice::Resume

ISpVoice::Resume sets the output device to the RUN state and resumes rendering.

Decrements the pause count (which is incremented by [ISpVoice::Pause](#)) if the voice is currently paused. If the pause count hits zero, Resume attempts to reclaim the output device and resumes rendering. This method has no effect if the voice was not in a paused state.

```
HRESULT Resume ( void );
```

Parameters

None.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.



ISpVoice::SetVoice

ISpVoice::SetVoice sets the identity of the voice used for text synthesis. ISpVoice normally uses the default voice, which is set through Speech properties in Control Panel.

```
HRESULT SetVoice(  
    ISpObjectToken *pToken  
);
```

Parameters

pToken

[in] Pointer to token that describes the requested voice. If *pToken* is NULL, the system default voice is used.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.

Remarks

Changing the voice selection will preserve the same volume and rate levels for an ISpVoice object.

Example

The following is an example to enumerate all the available voices registered under HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Voices.

```

HRESULT                hr = S_OK;
CComPtr<ISpObjectToken> cpVoiceToken;
CComPtr<IEnumSpObjectTokens> cpEnum;
CComPtr<ISpVoice>      cpVoice;
ULONG                  ulCount = 0;

// Create the SAPI voice
if(SUCCEEDED(hr))
    hr = cpVoice.CoCreateInstance( CLSID_SpVoice );

//Enumerate the available voices
if(SUCCEEDED(hr))
    hr = SpEnumTokens(SPCAT_VOICES, NULL, NULL, &cpEnum);

//Get the number of voices
if(SUCCEEDED(hr))
    hr = cpEnum->GetCount(&ulCount);

// Obtain a list of available voice tokens, set the voice to the token, and
while (SUCCEEDED(hr) && ulCount -- )
{
    cpVoiceToken.Release();
    if(SUCCEEDED(hr))
        hr = cpEnum->Next( 1, &cpVoiceToken, NULL );

    if(SUCCEEDED(hr))
        hr = cpVoice->SetVoice(cpVoiceToken);

    if(SUCCEEDED(hr))
        hr = cpVoice->Speak( L"How are you?", SPF_DEFAULT, NULL );
}

```




ISpVoice::GetVoice

ISpVoice::GetVoice retrieves the object token that identifies the voice currently in use. If there is not a voice currently in use, this method will return the token for the default voice.

```
HRESULT GetVoice(  
    ISpObjectToken **ppToken  
);
```

Parameters

ppToken

[out] Pointer which will be set to point to the current voice's object token.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>ppToken</i> is invalid.



ISpVoice::Speak

ISpVoice::Speak speaks the contents of a text string or file.

```
HRESULT Speak(  
    const WCHAR *pwcs,  
    DWORD dwFlags,  
    ULONG *pulStreamNumber  
);
```

Parameters

pwcs

[in, string] Pointer to the null-terminated text string (possibly containing XML markup) to be synthesized. This value can be NULL when *dwFlags* is set to `SPF_PURGEBEFORESPEAK` indicating that any remaining data to be synthesized should be discarded. If *dwFlags* is set to `SPF_IS_FILENAME`, this value should point to a null-terminated, fully qualified path to a file.

dwFlags

[in] Flags used to control the rendering process for this call. The flag values are contained in the [SPEAKFLAGS](#) enumeration.

pulStreamNumber

[out] Pointer to a ULONG which receives the current input stream number associated with this Speak request. Each time a string is spoken, an associated stream number is returned. Events queued back to the application related to this string will contain this number. If NULL, no value is passed back.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
E_POINTER	Invalid pointer.
E_OUTOFMEMORY	Exceeded available memory.
SPERR_INVALID_FLAGS	Invalid flags specified for this operation.
SPERR_DEVICE_BUSY	Timeout occurred on synchronous call.

Remarks

Normally, *pulStreamNumber* will just be 1. If, however, several asynchronous Speak (or [SpeakStream](#)) calls are received and must be queued, the stream number will be incremented for each call.



ISpVoice::SpeakStream

ISpVoice::SpeakStream speaks the contents of a stream.

```
HRESULT SpeakStream(  
    IStream      *pStream,  
    DWORD        dwFlags,  
    ULONG        *pulStreamNumber  
);
```

Parameters

pStream

[in] Address of an IStream interface containing the input stream. If the [ISpStreamFormat](#) interface is not implemented by the input stream, the format type is assumed to be SPDFID_Text.

dwFlags

[in] Flags used to control the rendering process for this call. The flag values are contained in the [SPEAKFLAGS](#) enumeration, however the SPF_IS_FILENAME flag is not used for SpeakStream.

pulStreamNumber

[out] Pointer to a ULONG which receives the current input stream number associated with this SpeakStream request. Each time a string is spoken, an associated stream number is returned. Events queued back to the application related to this string will contain this number.

Return values

Value	Description
S_OK	Function completed successfully.

E_INVALIDARG	One or more parameters are invalid.
E_POINTER	Invalid pointer.
E_OUTOFMEMORY	Exceeded available memory.
SPERR_INVALID_FLAGS	Invalid flags specified for this operation.
SPERR_DEVICE_BUSY	Timeout on synchronous call.

Remarks

If the input stream is wave data, it is sent directly to the output object. For more information about connecting an input stream to a voice, see `ISpStream::BindToFile`.

If the input stream is text data, it is processed by the text-to-speech engine.

Normally, *puStreamNumber* will just be 1. If, however, several asynchronous `SpeakStream` (or [Speak](#)) calls are received and must be queued, the stream number will be incremented for each call.



ISpVoice::GetStatus

ISpVoice::GetStatus retrieves the current rendering and event status associated with this voice.

```
HRESULT GetStatus(  
    SPVOICESTATUS *pStatus,  
    WCHAR **ppszLastBookmark  
);
```

Parameters

pStatus

[out] Pointer to an SPVOICESTATUS structure which receives the status information. This pointer can be NULL if the caller does not want this information.

ppszLastBookmark

[out, string] Pointer to a pointer which receives a CoTaskMemAlloc allocated null-terminated string containing the text of the last bookmark reached. If there is no last bookmark, NULL will be returned. Applications calling this method must call CoTaskMemFree() to free the returned string. This pointer can be NULL if the caller does not want this information.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Invalid pointer.
E_OUTOFMEMORY	Exceeded available memory.

Remarks

Because the SPVOICESTATUS structure is closely associated with audio device status, GetStatus will not return an active status for a voice speaking to an audio output stream.



ISpVoice::Skip

ISpVoice::Skip causes the voice to skip forward or backward the specified number of items within the text of the current speak call.

```
HRESULT Skip(  
    WCHAR    *pItemType,  
    long     lNumItems,  
    ULONG    *pulNumSkipped  
);
```

Parameters

pItemType

[in,string] Specifies the type of item to skip. Currently "SENTENCE" is the only type supported.

lNumItems

[in] Specifies the number of items to skip in the current speak request. If *lNumItems* is a positive number, the voice will skip forward, and if it is negative, the voice will skip backward. If *lNumItems* is 0, the voice will skip back to the beginning of the current item.

pulNumSkipped

[out] Pointer to a ULONG which will be set to the actual number of items skipped.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pItemType</i> is invalid or bad.

E_POINTER	<i>pulNumSkipped</i> is invalid or bad.
SPERR_VOICE_PAUSED	Voice is in a paused state and may not be skipped.



ISpVoice::SetPriority

ISpVoice::SetPriority sets the priority for the voice. The default priority is SPVPRI_NORMAL.

```
HRESULT SetPriority(  
    SPVPRIORITY ePriority  
);
```

Parameters

ePriority

[in] Priority of type [SPVPRIORITY](#) associated with the current voice.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>ePriority</i> is not an acceptable priority value.

Remarks

Assuming an output object which implements [ISpAudio](#), speak requests of similar priority voices are queued, and are spoken one at a time in the order they are issued. That is, speak requests from normal priority voices are put in one queue, while speak requests from alert priority voices (with priority SPVPRI_ALERT) are put in another queue.

Alert priority voices take priority over normal voices. If one or more speak requests from alert priority voices are pending, a normal voice that is speaking will be interrupted on the next alert boundary (see [ISpVoice::SetAlertBoundary](#)). When all the queued alert priority voice speak requests have been processed,

the normal voice will continue.

Voices with the SPVPRI_OVER priority speak over (mix with) all other audio in the system with no synchronization. SPVPRI_OVER priority voices only mix on Windows 2000.

If the output object does not implement ISpAudio, no serialization will occur, and all voices will be treated as if their priority is SPVPRI_OVER.



ISpVoice::GetPriority

ISpVoice::GetPriority retrieves the current voice priority level. See [ISpVoice::SetPriority](#) for more information on voice priorities.

```
HRESULT GetPriority(  
    SPVPRIORITY *pePriority  
);
```

Parameters

pePriority
[out] Pointer to priority information of type [SPVPRIORITY](#).

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Invalid pointer.



ISpVoice::SetAlertBoundary

ISpVoice::SetAlertBoundary specifies which event should be used as the insertion point for alert priority voice interruptions and [ISpVoice::Pause](#) calls.

```
HRESULT SetAlertBoundary(  
    SPEVENTENUM eBoundary  
);
```

Parameters

eBoundary

[in] [SPEVENTENUM](#) enumeration value that specifies which event to use for the alert insertion point. Appropriate events to use for this purpose include SPEI_WORD_BOUNDARY, SPEI_SENTENCE_BOUNDARY, SPEI_PHONEME, SPEI_VISEME, and possibly SPEI_VOICE_CHANGE or SPEI_TTS_BOOKMARK.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>eBoundary</i> is invalid.

Remarks

Events, such as word and sentence boundaries, are queued by the TTS engine during text rendering. Alert priority voices will only be able to interrupt normal priority voices, and pauses will only be able to occur on the specified event boundaries.

The default alert boundary event is SPEI_WORD_BOUNDARY.

See [ISpVoice::SetPriority](#) for more details on voice priorities.



ISpVoice::GetAlertBoundary

ISpVoice::GetAlertBoundary retrieves the event that is currently being used as the insertion point for alerts. For more information on alert boundaries, see [ISpVoice::SetAlertBoundary](#).

```
HRESULT GetAlertBoundary(  
    SPEVENTENUM *peBoundary  
);
```

Parameters

peBoundary

[out] Address of an [SPEVENTENUM](#) that receives the event type of the alert boundary.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Invalid pointer.



ISpVoice::SetRate

ISpVoice::SetRate retrieves the current text rendering rate adjustment. The default rate for a voice is set through Speech properties in Control Panel.

```
HRESULT SetRate(  
    long    RateAdjust  
);
```

Parameters

RateAdjust

[in] Value specifying the speaking rate of the voice. Supported values range from -10 to 10 - values outside this range may be truncated.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.

Remarks

Voices do not all have the same default rate.

The granularity of the rate is engine dependent.

Applications can adjust the rate of a voice either through this function call, or through XML passed to the voice with the input text of a speak call (see the [XML Schema : SAPI](#) white paper). The voice should combine rate adjustments made in these two ways to arrive at a final rate.



ISpVoice::GetRate

ISpVoice::GetRate retrieves the current base rate (either the default rate or the last value set by [ISpVoice::SetRate](#)).

```
HRESULT GetRate(  
    long    *pRateAdjust  
);
```

Parameters

pRateAdjust

[out] Pointer to a long which receives the base rate.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pRateAdjust</i> is invalid.



ISpVoice::SetVolume

ISpVoice::SetVolume sets the synthesizer output volume level of the voice in real time. The default base volume for all voices is 100.

```
HRESULT SetVolume(  
    USHORT          usVolume  
);
```

Parameters

usVolume

[in] Value containing the requested volume level. Volume levels are specified in percentage values ranging from zero to 100 - values outside this range may be truncated.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.

Remarks

Volume is specified as a percentage of the maximum volume of the current voice. Different voices may have different maximum volume levels.

Applications can adjust the volume of a voice either through this function call, or through XML grammar passed to the voice with the input text of a speak call (see the [XML Schema : SAPI](#) white paper). The voice should combine volume adjustments made in these two ways to arrive at a final volume.



ISpVoice::GetVolume

ISpVoice::GetVolume retrieves the current output volume level (either the default volume level or the last level set by [ISpVoice::SetVolume](#)).

```
HRESULT GetVolume(  
    USHORT    *pusVolume  
);
```

Parameters

pusVolume

[out] Address to receive the current base volume level.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pusVolume</i> is invalid.



ISpVoice::WaitUntilDone

ISpVoice::WaitUntilDone blocks the caller until either the voice has completed speaking or the specified time interval has elapsed.

```
HRESULT WaitUntilDone(  
    ULONG    msTimeout  
);
```

Parameters

msTimeout

[in] Timeout period in milliseconds. INFINITE may be used to prevent this method from timing out.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	Wait time interval was exceeded.

Remarks

This call may be used after a single asynchronous [Speak](#) (or [SpeakStream](#)) call, or after several calls have been queued. In either case it will return only after all pending calls have been completed (or after the specified time interval has elapsed).



ISpVoice::SetSyncSpeakTimeout

ISpVoice::SetSyncSpeakTimeout sets the timeout interval in milliseconds after which synchronous [Speak](#) and [SpeakStream](#) calls to this voice will timeout.

```
HRESULT SetSyncSpeakTimeout(  
    ULONG    msTimeout  
);
```

Parameters

msTimeout

[in] Value specifying the timeout interval in milliseconds. The default is 10 seconds. INFINITE may also be used to prevent timeouts.

Return values

Value	Description
S_OK	Function completed successfully.

Remarks:

Timeouts occur when waiting for access to the output object. This means that for a normal priority voice (see [ISpVoice::SetPriority](#) for more information on priorities) and an output device which implements [ISpAudio](#), a timeout may occur while waiting to reacquire the output object after an interruption by an alert priority voice. For voices of both normal and alert priorities, a timeout may also occur while waiting to reacquire the output object after the voice has been paused and resumed (see [ISpVoice::Pause](#) and [ISpVoice::Resume](#)).

Wait times are not accumulated - that is, if a voice waits for *n* milliseconds to initially acquire the output object, and is then

paused and resumed, it will again wait for up to `msTimeout` milliseconds to reacquire the output object, *not* `msTimeout - n` milliseconds.



ISpVoice::GetSyncSpeakTimeout

ISpVoice::GetSyncSpeakTimeout retrieves the timeout interval for synchronous speech operations for this voice. For more information on timeouts, see [ISpVoice::SetSyncSpeakTimeout](#).

```
HRESULT GetSyncSpeakTimeout(  
    ULONG    *pmsTimeout  
);
```

Parameters

pmsTimeout

[out] Pointer to a ULONG which receives the timeout interval in milliseconds for synchronous speech operations.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pmsTimeout</i> is invalid.



ISpVoice::SpeakCompleteEvent

ISpVoice::SpeakCompleteEvent returns an event handle that will be signaled when the voice has completed speaking all pending requests. This is similar to the functionality provided by [ISpVoice::WaitUntilDone](#), but allows the caller to wait on the event handle.

```
[local] HANDLE SpeakCompleteEvent ( void );
```

Parameters

None.

Return values

Value	Description
Event Handle	For WAIT operation.

Remarks:

The caller should not call CloseHandle(), nor should the caller ever use the handle after releasing the COM reference to this voice.



ISpVoice::IsUISupported

ISpVoice::IsUISupported checks if the underlying text-to-speech engine's object token supports the requested UI.

```
[local] HRESULT IsUISupported(  
    const WCHAR    *pszTypeOfUI,  
    void           *pvExtraData,  
    ULONG          cbExtraData,  
    BOOL           *pfSupported  
);
```

Parameters

pszTypeOfUI

[in] Address of the null-terminated string containing the UI type that is being queried.

pvExtraData

[in] Pointer to additional information needed for the object. The TTS Engine implementer dictates the format and usage of the data provided.

cbExtraData

[in] Size, in bytes, of the ExtraData. The TTS Engine implementer dictates the format and usage of the data provided.

pfSupported

[out] Flag specifying whether the specified UI is supported. TRUE indicates the UI is supported, and FALSE indicates the UI is not supported. If this value is TRUE, but the return code is S_FALSE, the UI type (*pszTypeOfUI*) is supported, but not with the current parameters or run-time environment. Check

with the engine implementer to verify run-time requirements.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	The UI is supported but not with the current run-time environment or parameters.
E_INVALIDARG	One or more parameters are invalid.
FAILED(hr)	Appropriate error message.

Remarks

See the [TTS Engine White Paper](#) for further information on how a TTS Engine should implement UI.

Example

The following code snippet illustrates the use of `ISpVoice::IsUISupported` using [SPDUI_EngineProperties](#).

```
HRESULT hr = S_OK;

// display properties UI for the current TTS engine
hr = cpVoice->IsUISupported(SPDUI_EngineProperties, NULL
// Check hr

// if fSupported == TRUE, then current TTS engine support
```



ISpVoice::DisplayUI

ISpVoice::DisplayUI displays the UI from the underlying text-to-speech engine's object token.

```
[local] HRESULT DisplayUI(  
    HWND          hwndParent,  
    const WCHAR   *pszTitle,  
    const WCHAR   *pszTypeOfUI,  
    void          *pvExtraData,  
    ULONG         cbExtraData  
);
```

Parameters

hwndParent

[in] Specifies the parent window handle information.

pszTitle

[in] Address of a null-terminated string containing the window title information. Set this value to NULL to indicate that the TTS engine should use its default window title for this UI type.

pszTypeOfUI

[in] Address of the null-terminated string containing the requested UI type to display.

pvExtraData

[in] Pointer to additional information needed for the object. The TTS Engine implementer dictates the format and usage of the data provided.

cbExtraData

[in] Size, in bytes, of the ExtraData. The TTS Engine implementer dictates the format and usage of the data provided.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	The UI is supported but not with the current run-time environment or parameters.
E_INVALIDARG	One or more parameters are invalid.
FAILED(hr)	Appropriate error message.

Remarks

The best practice for using `ISpVoice::DisplayUI` is to call [ISpVoice::IsUISupported](#) with a specific UI type before calling `DisplayUI`.

See the [TTS Engine White Paper](#) for further information on how a TTS Engine should implement UI.

The call to `DisplayUI` is synchronous, so the call will not return until the UI has been closed.

Example

The following code snippet illustrates the use of `ISpVoice::DisplayUI` using [SPDUI_EngineProperties](#).

```
HRESULT hr = S_OK;

// display properties UI for the current TTS engine
hr = cpVoice->DisplayUI(MY_HWND, MY_APP_VOICE_PROPERTIES)
// Check hr
```



Engine-Level Interfaces

This section describes the interfaces and methods for incorporating speech engines into applications. They are intended for use at the DDI or device driver interface level. Some managers or interfaces may have entries also in the [Application-Level Interfaces](#) section. However, entries listed here apply only to the device driver or engine level.

- [Grammar Compiler interfaces](#)
- [Resource interfaces](#)
- [Speech Recognition interfaces](#)
- [Speech Recognition Engine interfaces](#)
- [Text-to-Speech Engine interfaces](#)



Grammar compiler interfaces (DDI-level)

The following section covers:

- [ISpErrorLog](#)
- [ISpGramCompBackend](#)
- [ISpGrammarCompiler](#)
- [ISpITNProcessor](#)
- [ISpCFGInterpreter](#)
- [ISpCFGInterpreterSite](#)



ISpErrorLog

An object implementing this interface can be supplied to [ISpGrammarCompiler::CompileStream\(\)](#) to receive compilation error messages.

Methods in Vtable Order

ISpErrorLog Methods	Description
AddError	Writes an error to the log file.



ISpErrorLog::AddError

ISpErrorLog::AddError writes an error to the log file. Applications can implement this method to process the compilation error messages.

```
HRESULT AddError(  
    const long      lLineNumber,  
    HRESULT         hr,  
    const WCHAR     *pszDescription,  
    const WCHAR     *pszHelpFile,  
    DWORD           dwHelpContext  
);
```

Parameters

lLineNumber

The line number of the error in the XML grammar file.

hr

The error code being logged.

pszDescription

A textual description of the error.

pszHelpFile

The file being written to.

dwHelpContext

Flags providing additional information for the log.

Return values

Value	Description
-------	-------------

S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message. Currently any hr returned from AddError will be ignored by SAPI.

Because this method is application defined, the return value may change. See specific vendor documentation for details.



ISpGramCompBackend

ISpGramCompBackend inherits from the [ISpGrammarBuilder](#) interface.

Methods in Vtable Order

ISpGramCompBackend Methods	Description
SetSaveObjects	Sets the storage location of the binary grammar.
InitFromBinaryGrammar	Initializes a grammar from binary data.



ISpGramCompBackend::SetSaveObjects

ISpGramCompBackend::SetSaveObjects sets the storage location of the binary grammar.

```
HRESULT SetSaveObjects(  
    IStream      *pStream,  
    ISpErrorLog *pErrorLog  
);
```

Parameters

pStream

Address of the IStream that receives the binary grammar.

pErrorLog

Address of the [ISpErrorLog](#) interface that receives the error information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	Either <i>pStream</i> or <i>pErrorLog</i> is bad or invalid.
FAILED(hr)	Appropriate error message.

Remarks

When [ISpGrammarBuilder::Commit](#) is called, the grammar compiler back end writes the binary grammar to the location of *pStream*. When calling the SetSaveObjects method multiple times, the last call made before calling Commit, receives the binary grammar.



ISpGramCompBackend::InitFromBinaryGrammar

ISpGramCompBackend::InitFromBinaryGrammar initializes a grammar from binary data.

```
HRESULT InitFromBinaryGrammar(  
    const SPBINARYGRAMMAR *pBinaryData  
);
```

Parameters

pBinaryData

Pointer to the grammar list.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pBinaryData</i> is invalid or bad.
E_OUTOFMEMORY	Exceeded available memory.
FAILED (hr)	Appropriate error message.



ISpGrammarCompiler

SAPI 5 Text Grammar compiler.

Methods in Vtable Order

ISpGrammarCompiler Methods	Description
<u>CompileStream</u>	Loads the XML grammar and produces the binary grammar format.



ISpGrammarCompiler::CompileStream

ISpGrammarCompiler::CompileStream loads the XML grammar and produces the binary grammar format.

Compiles the SAPI 5 Speech Text Grammar pointed to by *pSource* stream and writes the output to the *pDest* stream. It can optionally generate C/C++ header information from the <DEFINE> <ID> tags.

```
HRESULT CompileStream(  
    IStream          *pSource,  
    IStream          *pDest,  
    IStream          *pHeader,  
    IUnknown         *pReserved,  
    ISpErrorLog     *pErrorLog,  
    DWORD            dwFlags  
);
```

Parameters

pSource

Pointer to the source of the XML grammar text.

pDest

Pointer to the destination stream for the binary grammar.

pHeader

Pointer to the stream to write the C/C++ header information (from the <DEFINE> tags) to (e.g., #define myterm 3).

pReserved

Reserved. Do not use.

pErrorLog

Pointer to the error log receiving the messages.

dwFlags

[in] Not currently used. Must be zero.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One of the parameters is bad or invalid.
FAILED (hr)	Appropriate error message.



ISpITNProcessor

SAPI implements the ISpITNProcessor interface to perform Inverse Text Normalization (ITN).

Methods in Vtable Order

ISpITNProcessor Methods	Description
<u>LoadITNGrammar</u>	Loads an inverse text normalization grammar.
<u>ITNPhrase</u>	Parses an inverse text normalization phrase.



ISpITNProcessor::LoadITNGrammar

ISpITNProcessor::LoadITNGrammar loads an inverse text normalization (ITN) grammar. The loaded grammar can be used by either SAPI or the speech recognition (SR) engine.

```
HRESULT LoadITNGrammar(  
    WCHAR    *pszCLSID  
);
```

Parameters

pszCLSID

Address of the null-terminated string containing the CLSID of the ITN grammar object implementing [ISpCFGInterpreter](#).

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pszCLSID</i> is invalid or bad.
FAILED(hr)	Appropriate error message.



ISpITNProcessor::ITNPhrase

ISpITNProcessor::ITNPhrase parses an inverse text normalization (ITN) on a previously loaded grammar.

```
HRESULT ITNPhrase(  
    ISpPhraseBuilder *pPhrase  
);
```

Parameters

pPhrase

Address of the phrase to parse.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	No grammar is loaded.
E_INVALIDARG	No words are available.
SP_NO_RULE_ACTIVE	No rule is active by default in ITN grammar.
E_OUTOFMEMORY	Not enough memory to complete operation.
FAILED(hr)	Appropriate error message.

Remarks

The ITNPhrase will attempt to parse the *pPhrase* passed in using the ITN grammar loaded by [ISpITNProcessor::LoadITNGrammar](#). If a parse is found, the ITN grammar will add the display text replacement. For example, AddReplacement "\$100" for "one hundred dollars".



ISpCFGInterpreter

ISpCFGInterpreter interface supports loading compiled grammars and modifying semantic properties and text replacements.

When to use

The ISpCFGInterpreter interface does not need to be directly called by applications or speech recognition (SR) engines. SAPI will create the interpreter and call the appropriate methods as needed. See the individual methods for scenarios when the methods will be called.

When to implement

The ISpCFGInterpreter interface should be implemented by applications and SR engine vendors who need to either load compiled grammars from COM objects (e.g., distributed COM objects on a server), or when an application scenario requires that the semantic properties or display text be replaced with dynamic information.

For example, the context-free grammar (CFG) interpreter could be written to dynamically run code depending on the order and structure of the semantic properties in a recognized phrase, or it could replace the text "today" with the actual day of the week.

Methods in Vtable Order

ISpCFGInterpreter Methods	Description
<u>InitGrammar</u>	Initializes a grammar that is loaded from an object or DLL.
<u>Interpret</u>	Examines a grammar and generates new properties and text replacements.



ISpCFGInterpreter::InitGrammar

ISpCFGInterpreter_InitGrammar initializes a grammar that is loaded from an object or DLL.

When to call

InitGrammar is typically called by SAPI only when a grammar includes a rule reference (e.g., ProgId). The following grammar includes XML:

```
<RULEREf NAME="MyRule" OBJECT="MyObject"/>
```

SAPI will internally call `::CoCreateInstance` on the COM object registered under the programmatic identifier (ProgId), "MyObject", and query for the [ISpCFGInterpreter](#) interface. SAPI will call `ISpCFGInterpreter::InitGrammar` and pass in the string identifier "MyRule" so that the COM object's implementation can return a pointer to the compiled grammar binary (typically stored in a resource).

When to implement

Application writers and engine vendors can use the `ISpCFGInterpreter` interface to create COM objects that contain compiled grammars. For example, the Microsoft Speech Recognition engine (that ships in the Microsoft Speech SDK) implements inverse text normalization (INT) as a compiled SAPI context-free grammar (CFG), which is retrievable using the `ISpCFGInterpreter::InitGrammar` method call.

```
HRESULT InitGrammar(  
    const WCHAR    *pszGrammarName,  
    const void     **pvGrammarData  
);
```

Parameters

pszGrammarName

[in] Address of a null-terminated string of the grammar to be loaded.

pvGrammarData

[in] Address of a pointer to the serialized binary grammar.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.

Example

The following code snippet illustrates how an application writer or engine vendor might implement the InitGrammar method.

```
STDMETHODIMP CMyCFGInterpreter::InitGrammar(const WCHAR* pPath)
{
    HRESULT hr = S_OK;

    // find the resource data of type COMPILED_CFG, which has the
    HRSRC hResInfo = ::FindResource(_Module.GetModuleInstance(),
    if (hResInfo)
    {
        // Load the resource into a global handle
        HGLOBAL hData = ::LoadResource(_Module.GetModuleInstance(),
        if (hData)
        {
            // return/store a pointer to the compiled grammar
            *pvGrammarData = ::LockResource(hData);
            if (*pvGrammarData == NULL)
            {
                hr = HRESULT_FROM_WIN32(::GetLastError());
            }
        }
        else
        {
            hr = HRESULT_FROM_WIN32(::GetLastError());
        }
    }
    else
    {
        hr = HRESULT_FROM_WIN32(::GetLastError());
    }
}
```

```
    return hr;  
}
```



ISpCFGInterpreter::Interpret

ISpCFGInterpreter::Interpret examines a grammar and generates new properties and text replacements.

Interpret is typically called by SAPI only when a grammar includes a reference to a context-free grammar (CFG) interpreter.

```
<RULE NAME="MyRule" INTERPRETER="TRUE"/>
```

SAPI will internally call `ISpCFGInterpreter_Interpret` whenever the speech recognition engine calls `ISpSREngineSite_ParseFromTransitions` with a rule handle matching "MyRule".

Application writers and engine vendors can use the [ISpCFGInterpreter](#) interface to manipulate the semantic properties returned to the application. For example, the CFG interpreter could detect whenever the grammar contained a semantic property called "Today" and replace the semantic property value with the actual system date and time.

```
HRESULT Interpret(  
    ISpPhraseBuilder          *pPhrase,  
    const ULONG              ulFirstElement,  
    const ULONG              ulCountOfElements,  
    ISpCFGInterpreterSite    *pSite  
);
```

Parameters

pPhrase

[in] Address of the [ISpPhraseBuilder](#) interface containing the phrase information.

ulFirstElement

[in] Value specifying the location of the first element in *pPhrase*.

ulCountOfElements

[in] Value specifying the number of phrase elements in *pPhrase*.

pSite

[in] Address of the [ISpCFGInterpreterSite](#) interface containing methods that can be used to attach semantic properties or text replacements to the parent phrase.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.



ISpCFGInterpreterSite

This interface is used by rule interpreters to set properties and text replacements in the parent phrase.

When To Use

The **ISpCFGInterpreterSite** interface allows context-free grammar (CFG) interpreters (see [ISpCFGInterpreter](#)) to access the semantic property tree and the phrase element structure, so that it can modify and update the properties or text replacements.

When To Implement

The **ISpCFGInterpreterSite** interface is implemented by SAPI, and is sent to the CFG interpreter (see [ISpCFGInterpreter](#)) when the speech recognition engine recognizes a rule that uses the interpreter (see [ISpSREngineSite::ParseFromTransitions](#)).

Methods in Vtable Order

ISpCFGInterpreterSite Methods	Description
AddTextReplacement	Adds one text replacement to the phrase.
AddProperty	Adds a property entry to the phrase object.
GetResourceValue	Retrieves the resource information for a grammar.



ISpCFGInterpreterSite::AddTextReplacement

ISpCFGInterpreterSite::AddTextReplacement adds one text replacement to the phrase. The object must have been initialized by calling SetPhrase prior to calling this method.

```
HRESULT AddTextReplacement(  
    SPPHRASEREPLACEMENT *pReplace  
);
```

Parameters

pReplace

[in] Address of the [SPPHRASEREPLACEMENT](#) that contains the replacement text.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pReplace</i> is invalid or bad.
SPERR_UNINITIALIZED	The object is uninitialized.
FAILED(hr)	Appropriate error message.



ISpCFGInterpreterSite::AddProperty

ISpCFGInterpreterSite::AddProperty adds a property entry to the phrase object.

```
HRESULT AddProperty(  
    const SPPHRASEPROPERTY *pProperty  
);
```

Parameters

pProperty

[in] Address of the [SPPHRASEPROPERTY](#) structure that contains the property information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pProperty</i> is bad or invalid.
SPERR_UNINITIALIZED	The object is uninitialized.
FAILED(hr)	Appropriate error message.



ISpCFGInterpreterSite::GetResourceValue

ISpCFGInterpreterSite::GetResourceValue retrieves the resource information for a grammar.

```
HRESULT GetResourceValue(  
    const WCHAR      *pszResourceName,  
    WCHAR            **ppCoMemResource  
);
```

Parameters

pszResourceName

[in] The name of the resource from which to retrieve the grammar information.

ppCoMemResource

[out] Pointer containing the passed back resource value. Applications implementing this method must call CoTaskMemFree() to free memory associated with this resource.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One of the parameters is bad or invalid.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.



Resource Manager interfaces (DDI-level)

The following section covers:

- [ISpObjectTokenEnumBuilder](#)
- [ISpTokenUI](#)
- [ISpTaskManager](#)
- [ISpThreadControl](#)
- [ISpThreadTask](#)



ISpObjectTokenEnumBuilder

This interface inherits from [IEnumSpObjectTokens](#).

Methods in Vtable Order

ISpObjectTokenEnumBuilder Methods	Description
SetAttribs	Sets the required and optional token enumerator attribute information.
AddTokens	Adds tokens to the object token enumerator.
AddTokensFromDataKey	Adds a new token using specified subkey and CategoryId information.
AddTokensFromTokenEnum	Adds a new token from an enumerated list of object tokens.
Sort	Sorts the list of enumerated object tokens.



ISpObjectTokenEnumBuilder::SetAttribs

ISpObjectTokenEnumBuilder::SetAttribs sets the required and optional token enumerator attribute information. This function can be called only once for the same object.

```
HRESULT SetAttribs(  
    const WCHAR    *pszReqAttribs,  
    const WCHAR    *pszOptAttribs  
);
```

Parameters

pszReqAttribs

Address of a null-terminated string containing the required attribute information.

pszOptAttribs

Address of a null-terminated string containing the optional attribute information.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_ALREADY_INITIALIZED	The object has already been initialized.
E_OUTOFMEMORY	Exceeded available memory.
FAILED(hr)	Appropriate error message.



ISpObjectTokenEnumBuilder::AddTokens

ISpObjectTokenEnumBuilder::AddTokens adds tokens to the object token enumerator.

```
HRESULT AddTokens(  
    ULONG          cTokens,  
    ISpObjectToken **pToken  
);
```

Parameters

cTokens

The number of object tokens being added to the sequence.

pToken

Address of a pointer to an *ISpObjectToken* object containing the information associated with the tokens being added.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
E_POINTER	Invalid pointer.
E_OUTOFMEMORY	Exceeded available memory.
SPERR_UNINITIALIZED	The object has not been properly initialized.
FAILED(hr)	Appropriate error message.



ISpObjectTokenEnumBuilder::AddTokensF

ISpObjectTokenEnumBuilder::AddTokensFromDataKey

adds a new token using specified subkey and CategoryId information.

```
HRESULT AddTokensFromDataKey(  
    ISpDataKey *pDataKey,  
    const WCHAR *pszSubKey,  
    const WCHAR *pszCategoryId  
);
```

Parameters

pDataKey

Address of an ISpDataKey interface that specifies the system registry key from which to create the token.

pszSubKey

Address of a null-terminated string containing the system registry subkey name.

pszCategoryId

Address of a null-terminated string containing the category identifier information for the system registry subkey.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
SPERR_UNINITIALIZED	The object has not been properly initialized.
FAILED(hr)	Appropriate error message.



ISpObjectTokenEnumBuilder::AddTokensF

ISpObjectTokenEnumBuilder::AddTokensFromTokenEnum

adds new tokens from an enumerated list of object tokens.

```
HRESULT AddTokensFromTokenEnum(  
    IEnumSpObjectTokens *pTokenEnum  
);
```

Parameters

pTokenEnum

Address of an [IEnumSpObjectTokens](#) interface containing the list of enumerated object tokens to add.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
SPERR_UNINITIALIZED	The object has not been properly initialized.
FAILED(hr)	Appropriate error message.



ISpObjectTokenEnumBuilder::Sort

ISpObjectTokenEnumBuilder::Sort sorts the list of enumerated object tokens.

```
HRESULT Sort(  
    const WCHAR    *pszTokenIdToListFirst  
);
```

Parameters

pszTokenIdToListFirst

Address of a null-terminated string of tokenId for the first token in the sorted list.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Invalid pointer.
SPERR_UNINITIALIZED	The object has not been properly initialized.
FAILED(hr)	Appropriate error message.

If the optional attributes of the EnumBuilder have been set, the first token in the EnumBuilder after the Sort call will be the token that matches the optional attributes best, not the token that *pszTokenIdToListFirst* refers to.



ISpTokenUI

Provides developers with a means to programmatically manage user-interface associated with an [ISpObjectToken](#).

When To Implement

The ISpTokenUI interface should be implemented so that the object can allow other applications to display the UI. For example, an SR engine (see [ISpRecognizer](#)) has a UI for Training (see [SPDUI_UserTraining](#)), and it would be helpful for an application to be able to display the Training UI as appropriate.

Example

The following code snippet illustrates the use of ISpTokenUI using QueryInterface.

```
HRESULT hr = S_OK;

// find the preferred multimedia input object token
hr = SpFindBestToken(SPCAT_AUDIOIN, L"Technology=MMSys",
// Check hr

// get the multimedia object token's UI
hr = cpObjectToken->QueryInterface(&cpTokenUI);
// Check hr
```

The following code snippet illustrates the use of ISpTokenUI using CoCreateInstance. The user must know the exact CLSID of the intended UI object.

```
HRESULT hr = S_OK;

// create the Token UI for the UI object CLSID_MY_TOKEN_UI
hr = cpTokenUI.CoCreateInstance(CLSID_MY_TOKEN_UI);
// Check hr
```

Methods in Vtable Order

ISpTokenUI Methods	Description
<u>IsUISupported</u>	Determines if the specified UI type is supported by the token.
<u>DisplayUI</u>	Displays the UI associated with the object token.

Development Helpers

Helper Functions	Description
<u>SpCreateBestObject</u>	Creates the most appropriate object based on specific criteria.
<u>SpFindBestToken</u>	Finds the most appropriate ISpObjectToken based on specific criteria.
<u>SpCreateObjectFromToken</u>	Creates an object based on a specified ISpObjectToken.
<u>SpCreateDefaultObjectFromCategoryId</u>	Creates the default object from a specific category.
<u>SpGetTokenFromId</u>	Creates an ISpObjectToken based on a token id.
<u>SpGetDefaultTokenFromCategoryId</u>	Creates the default ISpObjectToken from a specific category.



ISpTokenUI::IsUISupported

ISpTokenUI::IsUISupported determines if the specified UI type is supported by the token.

```
[local] HRESULT IsUISupported(  
    const WCHAR    *pszTypeOfUI,  
    void           *pvExtraData,  
    ULONG          cbExtraData,  
    IUnknown       *punkObject,  
    BOOL          *pfSupported  
);
```

Parameters

pszTypeOfUI

[in] Address of a null-terminated string containing the object's UI type.

pvExtraData

[in] Pointer to additional information needed for the object. The [ISpTokenUI](#) object implementer dictates the format and usage of the data provided.

cbExtraData

[in] Size, in bytes, of the ExtraData. The [ISpTokenUI](#) object implementer dictates the format and usage of the data provided.

punkObject

[in] Address of the object's IUnknown interface. See Remarks section.

pfSupported

[out] Address of a variable that receives the value indicating support for the interface. This value is set to TRUE when this interface is supported and FALSE otherwise. If this value is TRUE, but the return code is S_FALSE, the UI type (*pszTypeOfUI*) is supported, but not with the current parameters or run-time environment. Check with the implementer of the UI object to verify run-time requirements.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	The UI is supported but not with the current run-time environment or parameters.
E_INVALIDARG	One or more parameters are invalid.
E_POINTER	Invalid or bad pointer.
FAILED(hr)	Error returned by UI object.

Remarks

When asking a token to display a particular piece of UI, the token may require extra functionality that only it understands. Common implementation practice for accessing this functionality is to QueryInterface off of a known [IUnknown](#) interface. The caller of `ISpTokenUI::IsUISupported` can set the *punkObject* parameter with the necessary [IUnknown](#) interface. For example, asking to display Speech Recognition Training UI requires that a specific SR engine be used.

Example

The following code snippet illustrates the use of `ISpTokenUI::IsUISupported` using [SPDUI_AudioProperties](#).

```
HRESULT hr = S_OK;
```

```
// get the default input audio object token
hr = SpGetDefaultTokenFromCategoryId(SPCAT_AUDIOIN, &cpObj
// Check hr

// get the object token's UI
hr = cpObjectToken->QueryInterface(&cpTokenUI);
// Check hr

// check if the default audio input object has UI for Pro
hr = cpTokenUI->IsUISupported(SPDUI_AudioProperties, NUL
// Check hr

// if fSupported == TRUE, then default audio input objec
```



ISpTokenUI::DisplayUI

ISpTokenUI::DisplayUI displays the UI associated with the object token.

```
[local] HRESULT DisplayUI(  
    HWND                hwndParent,  
    const WCHAR         *pszTitle,  
    const WCHAR         *pszTypeOfUI,  
    void                *pvExtraData,  
    ULONG               cbExtraData,  
    ISpObjectToken      *pToken,  
    IUnknown           *punkObject  
);
```

Parameters

hwndParent

[in] Specifies the handle of the parent window.

pszTitle

[in] Address of a null-terminated string containing the window title to display on the UI. This value can be set to NULL to indicate that the TokenUI object should use its default window title.

pszTypeOfUI

[in] Address of a null-terminated string containing the UI type to display.

pvExtraData

[in] Pointer to additional information needed for the object. The [ISpTokenUI](#) object implementer dictates the format and usage of the data provided.

cbExtraData

[in] Size, in bytes, of the ExtraData. The [ISpTokenUI](#) object implementer dictates the format and usage of the data provided.

pToken

[in] Address of the ISpObjectToken containing the object token identifier. See Remarks section.

punkObject

[in] Address of the IUnknown interface pointer. See Remarks section.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	The UI is supported but not with the current run-time environment or parameters.
E_INVALIDARG	One or more parameters are invalid.
E_POINTER	Invalid or bad pointer.
FAILED(hr)	Error returned by UI object.

Remarks

The best-practice for using [ISpTokenUI](#) is to call [ISpTokenUI::IsUISupported](#) with a specific UI type before calling DisplayUI.

The call to DisplayUI is synchronous, so the call will not return until the UI has been closed.

The token may require extra functionality that only it understands in order to display a particular piece of UI. Common implementation practice for accessing this functionality is to use

QueryInterface of a known [IUnknown](#) interface or create the object associated a known [ISpObjectToken](#) instance. The caller of `ISpTokenUI::DisplayUI` can set *punkObject* with the necessary [IUnknown](#) interface or set *pToken* with the necessary [ISpObjectToken](#) interface. For example, asking to display Speech Recognition Training UI requires that a specific SR engine be used.

Example

The following code snippet illustrates the use of `ISpTokenUI::DisplayUI` using [SPDUI_AudioVolume](#).

```
HRESULT hr = S_OK;

// get the default input audio object token
hr = SpGetDefaultTokenFromCategoryId(SPCAT_AUDIOIN, &cpObj
// Check hr

// get the object token's UI
hr = cpObjectToken->QueryInterface(&cpTokenUI);
// Check hr

// Check if default audio input object has UI for volume
hr = cpTokenUI->IsUISupported(SPDUI_AudioVolume, NULL, NI
// Check hr

// if fSupported == TRUE, then default audio input objec

// Display the default audio input object's Volume UI
hr = cpTokenUI->DisplayUI(MY_HWND, MY_AUDIO_DIALOG_TITLE
// Check hr
```



ISpTaskManager

When to Implement

This interface is used to implement a task management service provider to optimize thread usage.

Associated Class IDs

The following class IDs (CLSID) may be used with this interface. A complete CLSID listing for all interfaces is in the [Class IDs](#) section.

CLSID_SpResourceManager

Methods in Vtable Order

ISpTaskManager Methods	Description
SetThreadPoolInfo	Sets the attributes for thread pool management.
GetThreadPoolInfo	Retrieves the current thread pool management attributes.
QueueTask	Adds a task to the queue for asynchronous task processing.
CreateReoccurringTask	Creates a task entry that will be processed on a high priority thread.
CreateThreadControl	Creates a thread control object.
TerminateTask	Interrupts a specified task.
TerminateTaskGroup	Terminates a group of tasks that match a specific group identifier.



ISpTaskManager::SetThreadPoolInfo

ISpTaskManager::SetThreadPoolInfo sets the attributes for thread pool management.

```
HRESULT SetThreadPoolInfo(  
    const SPTMTHREADINFO *pPoolInfo  
);
```

Parameters

pPoolInfo

[in] Address of an SPTMTHREADINFO structure that receives the thread management information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pPoolInfo</i> is invalid or <i>pPoolInfo->IPoolSize</i> size is less than -1.
FAILED (hr)	Appropriate error message.



ISpTaskManager::GetThreadPoolInfo

ISpTaskManager::GetThreadPoolInfo retrieves the current thread pool management attributes.

```
HRESULT GetThreadPoolInfo(  
    SPTMTHREADINFO *pPoolInfo  
);
```

Parameters

pPoolInfo

[out] Address of an SPTMTHREADINFO structure that contains the current thread management information.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pPoolInfo</i> is invalid or bad.
FAILED (hr)	Appropriate error message.



ISpTaskManager::QueueTask

ISpTaskManager::QueueTask adds a task to the queue for asynchronous task processing.

```
HRESULT QueueTask(  
    ISpTask    pTask,  
    void      *pvTaskData,  
    HANDLE    hCompEvent,  
    DWORD*    *pdwGroupId,  
    DWORD*    *pTaskID  
);
```

Parameters

pTask

[in] Address of an ISpTask interface containing the task.

pvTaskData

[in] Address of the task data that will be passed to the [ISpTask::Execute](#) method.

hCompEvent

[in] Handle of the task completion event. This event will be set when the Execute method returns. This parameter can be NULL.

pdwGroupId

[in, out] Value specifying the identifier for the task group. This value may be NULL. This can be used to cancel a group of pending tasks.

pTaskID

[out] Value specifying the task identifier. This parameter can

be NULL if this information is not needed.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pTask</i> is invalid or bad.
E_POINTER	<i>pTaskId</i> or <i>pdwGroupId</i> is invalid or bad.
FAILED (hr)	Appropriate error message.



ISpTaskManager::CreateReoccurringTask

ISpTaskManager::CreateReoccurringTask creates a task entry that will be processed on a thread when the [ISpNotifySink::Notify](#) method is called on the task control object.

```
HRESULT CreateReoccurringTask(  
    ISpTask          *pTask,  
    void            *pvTaskData,  
    HANDLE         hCompEvent,  
    ISpNotifySink   **ppTaskCtrl  
);
```

Parameters

pTask

[in] Address of an [ISpTask](#) interface containing the task.

pvTaskData

[in] Pointer that will be passed to the [ISpTask::Execute](#) method.

hCompEvent

[in] Handle of the task completion event. This is optional and can be NULL. If non-NULL, this event handle will be signaled when the Execute method returns

ppTaskCtrl

[out] Address of a pointer to an [ISpNotifySink](#) interface. Call the Notify() method on this object to cause the task to be scheduled.

Return values



Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pTask</i> is invalid or bad.
E_POINTER	<i>ppTaskCtrl</i> is invalid or bad.
FAILED (hr)	Appropriate error message.



ISpTaskManager::CreateThreadControl

ISpTaskManager::CreateThreadControl allocates a thread control object and does not allocate a thread.

```
HRESULT CreateThreadControl(  
    ISpThreadTask          *pTask,  
    void                  *pvTaskData,  
    long                  nPriority,  
    ISpThreadControl     **ppThreadCtrl  
);
```

Parameters

pTask

[in] Address of the ISpThreadTask interface that is used to initialize and execute the task thread.

pvTaskData

[in] Data passed to all ISpThreadTask member functions. This value can be NULL.

nPriority

[in] The Win32 priority for the allocated thread.

ppThreadCtrl

[out] Address of a pointer to an ISpThreadControl interface that receives the thread control.

Return values

Value	Description
S_OK	Function completed

	successfully.
E_INVALIDARG	<i>pTask</i> is invalid or bad.
E_POINTER	<i>ppThreadCtrl</i> is invalid or bad.
E_OUTOFMEMORY	Exceeded available memory.
FAILED (hr)	Appropriate error message.



ISpTaskManager::TerminateTask

ISpTaskManager::TerminateTask interrupts the specified task.

```
HRESULT TerminateTask(  
    DWORD    dwTaskId,  
    ULONG    ulWaitPeriod  
);
```

Parameters

dwTaskId

[in] Value specifying the identifier of the task to interrupt.

ulWaitPeriod

[in] Number of milliseconds to wait before interrupting the task.

Return values

Value	Description
S_OK	Function completed successfully.
S_FALSE	Method timed out.
FAILED (hr)	Appropriate error message.



ISpTaskManager::TerminateTaskGroup

ISpTaskManager::TerminateTaskGroup terminates a group of tasks that match a specific group identifier.

```
HRESULT TerminateTaskGroup(  
    DWORD    dwGroupId,  
    ULONG    ulWaitPeriod  
);
```

Parameters

dwGroupId

[in] Value specifying the identifier for the task group to interrupt.

ulWaitPeriod

[in] Number of milliseconds to wait before interrupting the task group.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



ISpThreadControl

The ISpThreadControl interface inherits from the [ISpNotifySink](#) interface.

Methods in Vtable Order

ISpThreadControl Methods	Description
StartThread	Initializes a thread and returns a window handle.
WaitForThreadDone	Specifies the time interval to wait before ending thread processing.
TerminateThread	Forces immediate termination of the thread.
ThreadHandle	Returns the thread handle of the allocated thread.
ThreadId	Returns the thread ID of the allocated thread.
NotifyEvent	Returns the Win32 event handle that will be set when the Notify() method is called.
WindowHandle	Returns the window handle associated with the thread.
ThreadCompleteEvent	Returns an event that the client can use to wait until the thread processing has completed.
ExitThreadEvent	Returns the event passed to the ISpThreadTask::ThreadProc method.



ISpThreadControl::StartThread

ISpThreadControl::StartThread initializes a thread and optionally returns a window handle.

```
HRESULT StartThread(  
    DWORD    dwFlags,  
    HWND     *phwnd  
);
```

Parameters

dwFlags

Reserved. Must be zero.

phwnd

Optional address of an handle to a window. The handle of the new window will be returned to *phwnd* if this parameter is non-NULL. A window will not be created if this parameter is NULL.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
E_POINTER	Invalid pointer.
E_OUTOFMEMORY	Exceeded available memory.



ISpThreadControl::WaitForThreadDone

ISpThreadControl::WaitForThreadDone specifies the time interval to wait before ending thread processing.

A thread can be forced to stop running or can wait until the thread stops running.

```
HRESULT WaitForThreadDone(  
    BOOL        fForceStop,  
    HRESULT     *p hrThreadResult,  
    ULONG       msTimeout  
);
```

Parameters

fForceStop

Flag specifies to stop thread processing. If this is TRUE, [ISpThreadTask::ThreadProc](#) will be called on the thread proc of the worker thread. The ThreadProc parameters of *hExitThreadEvent* handle will be filled out and *pfContinueProcessing* flag set to FALSE.

p hrThreadResult

If this function returns S_OK, this address will contain the value returned from the thread proc.

msTimeout

Time-out interval in milliseconds to wait before timing out the wait operation.

Return values

Value	Description
S_OK	Function completed successfully.

E_INVALIDARG

One or more parameters are invalid.

Remarks

Specifying *fForceStop*=TRUE with a timeout of zero will simply request that the thread exit and this method will return immediately. The caller could then wait for the [ThreadCompleteEvent](#) using a Win32 wait function.



ISpThreadControl::TerminateThread

ISpThreadControl::TerminateThread forces immediate termination of the thread.

```
HRESULT TerminateThread ( void );
```

Parameters

None.

Return values

Value	Description
S_OK	Function completed successfully.

Remarks

This function should be used with caution, as unintended results can occur. It should only be used to force a thread to stop when all other attempts to stop the thread have been unsuccessful.



ISpThreadControl::ThreadHandle

ISpThreadControl::ThreadHandle returns the thread handle of the allocated thread. This handle should not be used to wait for the completion of the thread procedure. Use [ISpThreadControl::ThreadCompleteEvent](#), or [ISpThreadControl::WaitForThreadDone](#).

```
HANDLE ThreadHandle ( void );
```

Parameters

None.

Return values

Returns the thread handle, or NULL if [ISpThreadControl::StartThread](#) has not been called for this thread control object.



ISpThreadControl::ThreadId

ISpThreadControl::ThreadId returns the thread ID of the allocated thread.

Returns the Win32 thread ID of the thread associated with this thread control object. If [StartThread](#) has not been called, this returns zero.

```
DWORD ThreadId ( void );
```

Parameters

None.

Return values

The ID of the thread.



ISpThreadControl::NotifyEvent

ISpThreadControl::NotifyEvent returns the Win32 event handle that will be set when Notify() is called.

The event may be passed to [ISpThreadTask::ThreadProc](#).

```
HANDLE NotifyEvent ( void );
```

Parameters

None.

Return values

Returns the event handle.



ISpThreadControl::WindowHandle

ISpThreadControl::WindowHandle returns the window handle associated with this ISpThreadControl object. This will be NULL unless the caller of [ISpThreadControl::StartThread](#) specified a non-NULL HWND pointer.

```
HWND WindowHandle ( void );
```

Parameters

None.

Return values

Returns the window handle. NULL if no handle is associated with this object.



ISpThreadControl::ThreadCompleteEvent

ISpThreadControl::ThreadCompleteEvent returns a Win32 event handle that can be used to wait for the completion of the thread.

```
HANDLE ThreadCompleteEvent ( void );
```

Parameters

None.

Return values

Returns the Win32 event handle.



ISpThreadControl::ExitThreadEvent

ISpThreadControl::ExitThreadEvent returns the event handle passed to the [ISpThreadTask::ThreadProc](#). This should never be set manually. Use [WaitForThreadDone](#) to force a thread to exit.

```
HANDLE    ExitThreadEvent(void);
```

Parameters

none.

Return values

Win32 event handle.



ISpThreadTask

The ISpThreadTask interface simplifies thread-based operations. Clients can implement this virtual C++ interface to simplify the management of threads. SAPI provides coordination methods for starting, initializing, and stopping the thread. See the [ISpThreadControl](#) documentation for more information. Clients simply implement three methods which will be called on the allocated thread.

When to Implement

Use SpThreadControl objects to allocate threads for an object.

This is not a COM interface.

Methods in Vtable Order

ISpThreadTask Methods	Description
InitThread	Method called on the allocated thread. If the method returns an error, the ISpThreadControl::StartThread method will fail.
ThreadProc	Implements the processing of the thread.
WindowMessage	Implements the processing of window messages.



ISpThreadTask::InitThread

ISpThreadTask::InitThread is called on the allocated thread. If the method returns an error, the [ISpThreadControl::StartThread](#) method will fail.

This method is called on the newly allocated thread when [ISpThreadControl::StartThread](#) is called.

```
virtual HRESULT STDMETHODCALLTYPE InitThread(  
    void *pvTaskData,  
    HWND hwnd  
) = 0;
```

Parameters

pvTaskData

[in] The same pointer is passed to [ISpTaskManager::CreateThreadControl](#).

hwnd

[in] A window handle if, and only if, the caller to [ISpTaskManager::CreateThreadControl](#) specified that a window handle be created by passing a non-NULL HWND pointer to CreateThreadControl. Otherwise this parameter is NULL.

Return values

S_OK	Function completed successfully.
Other success	Success code will be returned to caller of the StartThread method and the thread will continue.
Failure code	Function failed. Failure code will be returned to caller of the StartThread

method and the thread will not continue.

Remarks

The caller of StartThread method will be blocked until the InitThread method completes, and the HRESULT returned from this method will be returned from StartThread. If the return code from this method indicates failure, the thread will be terminated, and [ThreadProc](#) and [WindowMessage](#) will never be called.



ISpThreadTask::ThreadProc

ISpThreadTask::ThreadProc implements the main processing loop of the thread. This method will be application specific.

```
virtual HRESULT STDMETHODCALLTYPE ThreadProc(  
    void                *pvTaskData,  
    HANDLE              hExitThreadEvent,  
    HANDLE              hNotifyEvent,  
    HWND               hwndWorker,  
    volatile const BOOL *pfContinueProcessing  
    ) = 0;
```

Parameters

pvTaskData

[in] Pointer passed to [ISpTaskManager::CreateThreadControl](#).

hExitThreadEvent

[in, out] An event handle which when signaled indicates that the thread process should exit.

hNotifyEvent

[in] A handle to an auto-reset event object that will be set if the [ISpThreadControl::Notify](#) method is called. This functionality is provided for any notification event the client determines, or it can optionally be ignored if it is not needed.

hwndWorker

[in] A window handle. This parameter will be NULL if the caller of [ISpThreadControl::StartThread](#) passed a NULL HWND pointer to StartThread.

pfContinueProcessing

[in] Boolean flag indicating whether to continue processing. TRUE indicates the process should continue; FALSE otherwise. This mirrors the functionality of the *hExitThreadEvent*, but provides a lightweight mechanism for checking for a request to exit without calling a Win32 WaitForxxx object function.

Return values

S_OK	Function completed successfully.
S_FAILED	Function failed.



ISpThreadTask::WindowMessage

ISpThreadTask::WindowMessage implements the processing of window messages.

```
virtual HRESULT STDMETHODCALLTYPE WindowMessage(  
    void      *pvTaskData,  
    HWND      hWnd,  
    UINT      Msg,  
    WPARAM    wParam,  
    LPARAM    lParam  
    ) = 0;
```

Parameters

pvTaskData

[in] Pointer passed to [ISpTaskManager::CreateThreadControl](#).

hWnd

[in] A window handle.

Msg

[in] The type of window message.

wParam

Message-specific information. This will change based on the *Msg* value.

lParam

Message-specific information. This will change based on the *Msg* value.

Return values

The return value is message specific.

Remarks

Not all applications will need a window and this method may be left unimplemented. If the caller of [ISpThreadControl::StartThread](#) passes a non-NULL HWND pointer, the client must implement this function and must also use a MessageWaitForMultipleObjects() loop in the ThreadProc.



Speech Recognition interfaces (DDI-level)

The following section covers:

- [ISpPhraseBuilder](#)



ISpPhraseBuilder

The ISpPhraseBuilder interface inherits from [ISpPhrase](#).

Methods in Vtable Order

ISpPhraseBuilder Methods	Description
InitFromPhrase	Initializes from a phrase.
InitFromSerializedPhrase	Initializes a phrase from a serialized phrase.
AddElements	Adds a copy of the given element to the end of this object's element list.
AddRules	Adds phrase rules to the phrase object.
AddProperties	Adds property entries to the phrase object.
AddReplacements	Adds one or more text replacements to the phrase.



ISpPhraseBuilder::InitFromPhrase

ISpPhraseBuilder::InitFromPhrase initializes from a phrase.

```
HRESULT InitFromPhrase(  
    const SPPHRASE *pSrcPhrase  
);
```

Parameters

pSrcPhrase

Address of a [SPPHRASE](#) data structure containing the phrase information. If *pSrcPhrase* is NULL, the object is reset to its initial state.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pSrcPhrase</i> or <i>pSrcPhrase->Rule.pNextSibling</i> is invalid or bad. Alternatively, <i>pSrcPhrase->LangID</i> may be zero or <i>pSrcPhrase->cbSize</i> does not indicate the same size as <i>pSrcPhrase</i> .
FAILED(hr)	Appropriate error message.

Example

The following code snippet demonstrates creating and initializing from a phrase.

```
HRESULT hr;
```

```
CComPtr<ISpPhraseBuilder> cpPhraseBuilder;  
CComPtr<ISpPhrase> cpPhrase;
```

```
SPPHRASE    Phrase;

hr = cpPhraseBuilder.CoCreateInstance( CLSID_SpPhraseBuilder

if (SUCCEEDED(hr))
{
    //We initialize the Phrase data structure
}

if (SUCCEEDED(hr))
{
    hr = cpPhraseBuilder->InitFromPhrase( &Phrase; );
}
```



ISpPhraseBuilder::InitFromSerializedPhra

ISpPhraseBuilder::InitFromSerializedPhrase initializes a phrase from a serialized phrase.

```
HRESULT InitFromSerializedPhrase(  
    const SPSERIALIZEDPHRASE *pPhrase  
);
```

Parameters

pPhrase

Address of the [SPSERIALIZEDPHRASE](#) structure that contains the phrase information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pSrcPhrase</i> or <i>pSrcPhrase->cbSerializedSize</i> is invalid or bad.
FAILED(hr)	Appropriate error message.

Example

The following code fragment demonstrates `InitFromSerializedPhrase`.

```
HRESULT hr;  
CComPtr<ISpRecoResult>      RecoResult;  
CComPtr<ISpPhraseBuilder>  pPhraseBuilder;  
SPSERIALIZEDPHRASE         *pSerializedPhrase=NULL;  
CComPtr<ISpStream>         cpStream;  
  
LARGE_INTEGER  liZero = {0,0};  
hr = SPBindToFile(L"SerializedPhrase.sp", SPFM_OPEN_READONLY  
if (hr == S_OK)
```

```

{
    hr = cpStream->Seek(liZero, STREAM_SEEK_SET, NULL);
    ULONG ulSerializedSize = 0;
    hr = cpStream->Read(&ulSerializedSize, sizeof(ULONG), NU
    if (SUCCEEDED(hr))
    {
        //We need to seek back and read the whole chunk of d
        LARGE_INTEGER liseek;
        liseek.QuadPart -= sizeof(ULONG);
        hr = cpStream->Seek(liseek, STREAM_SEEK_CUR, NULL);

        pSerializedPhrase = (SPSERIALIZEDPHRASE*)::CoTaskMem
        if (SUCCEEDED(hr) && pSerializedPhrase)
        {
            hr = cpStream->Read(pSerializedPhrase, ulSer.
        }
        if (SUCCEEDED(hr))
        {
            CComPtr<ISpPhraseBuilder> cpPhraseBuilder;
            hr = cpPhraseBuilder.CoCreateInstance(CLSID_
            if (SUCCEEDED(hr))
            {
                hr = cpPhraseBuilder->InitFromSerialized
            }
        }
        ::CoTaskMemFree( pSerializedPhrase );
    }
}

```



ISpPhraseBuilder::AddElements

ISpPhraseBuilder::AddElements adds a copy of the given element to the end of this object's element list.

```
HRESULT AddElements(  
    ULONG                cElements,  
    const SPPHRASEELEMENT *pElement  
);
```

Parameters

cElements

Specifies the number of phrase elements to add.

pElement

Address of the [SPPHRASEELEMENT](#) data structure containing the phrase element to add.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
SPERR_UNINITIALIZED	The object has not been properly initialized.
FAILED(hr)	Appropriate error message.



ISpPhraseBuilder::AddRules

ISpPhraseBuilder::AddRules adds phrase rules to the phrase object.

```
HRESULT AddRules(  
    const SPPHRASERULEHANDLE    hParent,  
    const SPPHRASERULE        *pRule,  
    SPPHRASERULEHANDLE          *phNewRule  
);
```

Parameters

hParent

[in] Handle to the parent phrase rule.

pRule

[in] Address of the [SPPHRASERULE](#) structure that contains the phrase rule information.

phNewRule

[out] Address of the handle of SPPHRASERULEHANDLE that contains the new phrase rule information.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Invalid pointer.
SPERR_UNINITIALIZED	The object has not been properly initialized.
FAILED(hr)	Appropriate error message.



ISpPhraseBuilder::AddProperties

ISpPhraseBuilder::AddProperties adds property entries to the phrase object.

```
HRESULT AddProperties(  
    const SPPHRASEPROPERTYHANDLE hParent,  
    const SPPHRASEPROPERTY      *pProperty,  
    SPPHRASEPROPERTYHANDLE      *phNewProperty  
);
```

Parameters

hParent

[in] Handle to the parent phrase element.

pProperty

[in] Address of the [SPPHRASEPROPERTY](#) structure that contains the property information.

phNewProperty

[out] Address of the handle of SPPHRASEPROPERTY that contains the new property information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
E_POINTER	Invalid pointer.
SPERR_UNINITIALIZED	The object has not been properly initialized.
SPERR_ALREADY_INITIALIZED	The object has already been

initialized.

FAILED(hr)

Appropriate error message.



ISpPhraseBuilder::AddReplacements

ISpPhraseBuilder::AddReplacements adds one or more text replacements to the phrase.

```
HRESULT AddReplacements(  
    ULONG cReplacements,  
    const SPPHRASEREPLACEMENT *pReplacements  
);
```

Parameters

cReplacements

The number of replacement phrase elements.

pReplacements

Address of the [SPPHRASEREPLACEMENT](#) structure that contains the phrase element replacement information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
SPERR_UNINITIALIZED	The object has not been properly initialized.
FAILED(hr)	Appropriate error message.



Speech Recognition Engine interfaces (DDI-level)

The following section covers:

- [ISpPrivateEngineCall](#)
- [ISpSREngine](#)
- [ISpSREngineSite](#)
- [ISpSRAlternates](#)



_ISpPrivateEngineCall

This interface is obtained by calling the QueryInterface method of the [ISpRecoContext](#) interface. However, applications should not directly use _ISpPrivateEngineCall interface.

Applications should call the QueryInterface method of the recognition context for a particular engine extension interface that is implemented by their engine. SAPI will create the engine extension class identified in ExtensionCLSID attribute of the engine object token. This interface should call the QueryInterface method for _ISpPrivateEngineCall, where it can then call [CallEngine](#) to make a private call to the engine. The data passed into the CallEngine method is passed to the engine's PrivateCall method.

When to Implement

Implemented by SAPI and inherits from [ISpRecoContext](#).

Associated Class IDs

The following class IDs (CLSID) may be used with this interface. A complete CLSID listing for all interfaces is in the [Class IDs](#) section.

CLSID_SpSharedRecoContext

Methods in Vtable Order

_ISpPrivateEngineCall Methods	Description
CallEngine	Allows an engine-specific call.
CallEngineEx	Returns the non-fixed size data block response information associated with the SR engine.



ISpPrivateEngineCall::CallEngine

ISpPrivateEngineCall::CallEngine allows an engine-specific call.

It is called from the engine extension object to the engine object. Data passed into this call is given to the main SR engine through the [ISpSREngine::PrivateCall](#) method.

```
HRESULT CallEngine(  
    VOID    *pCallFrame,  
    ULONG   ulCallFrameSize  
);
```

Parameters

pCallFrame

[in, out] The engine-specific structured block of memory parameters. This block will be marshaled in the shared engine case and must not contain pointers to other memory allocations. It must be fully self-contained and relative only to itself.

ulCallFrameSize

[in] Size, in bytes, of the *pCallFrame* structure.

Return values

Value	Description
S_OK	Function completed successfully.
E_FAILED	No engine could be found.
FAILED (hr)	Appropriate error message.



ISpPrivateEngineCall::CallEngineEx

ISpPrivateEngineCall::CallEngineEx returns the non-fixed size data block response information associated with the SR engine.

Applications implementing this method must call `CoTaskMemFree()` to free memory associated with the returned response.

```
HRESULT CallEngineEx(  
    const void *pInFrame,  
    ULONG      ulInFrameSize,  
    void      **ppCoMemOutFrame,  
    ULONG      *pulOutFrameSize  
);
```

Parameters

pInFrame

[in] Address of the recognition engine data.

ulInFrameSize

[in] Size, in bytes, of the *pInCallFrame* structure.

ppCoMemOutFrame

[out] Address of a pointer to the data block information associated with the SR engine.

pulOutFrameSize

[out] Size, in bytes, of the *ppCoMemOutFrame* structure.

Return values

Value	Description
-------	-------------

S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
E_POINTER	Invalid pointer.
FAILED(hr)	Appropriate error message.



ISpSREngine

This is the main interface that engine developers must implement to have a working SAPI SR engine. Engine vendors need to implement this interface in an InProc COM object with the threading model “Both”. This object can also implement the ISpObjectWithToken interface. SR engines may also have additional objects implementing ISpTokenUI, ISpSRAlternates, and a custom engine-extension object.

The [SR Engine Guide](#) describes in detail how an SR engine interacts with SAPI and how the methods of this interface are implemented.

Applications do not directly call the methods on this interface, all calls are made by SAPI.

Implemented By

- SR engine developers as the main interface between their SR engine and SAPI.

How Created

- Only SAPI will directly create this object. When an application creates a [RecoContext](#) object, either shared or InProc, then SAPI will find the appropriate recognizer object token and create the object implementing this interface from the CLSID in the object token.

Methods in Vtable Order

ISpSREngine Methods	Description
SetSite	Sets the ISpSREngineSite interface for the engine to use.
GetInputAudioFormat	Gets the audio format that the SR engine supports.
RecognizeStream	Begins recognition processing

	on a stream.
<u>SetRecoProfile</u>	Passes the current active user profile to the engine.
<u>OnCreateGrammar</u>	Creates a new recognition grammar.
<u>OnDeleteGrammar</u>	Informs the engine of the deletion of a grammar.
<u>LoadProprietaryGrammar</u>	Instructs the engine to load a grammar in an engine-specific format.
<u>UnloadProprietaryGrammar</u>	Unloads an engine-specific grammar.
<u>SetProprietaryRuleState</u>	Sets the proprietary grammar rule state.
<u>SetProprietaryRuleIdState</u>	Sets the proprietary grammar rule ID state.
<u>LoadSLM</u>	Instructs the engine to load a dictation statistical language model (SLM).
<u>UnloadSLM</u>	Instructs the engine to unload an SLM.
<u>SetSLMState</u>	Sets the recognition state of the SLM to active or inactive.
<u>SetWordSequenceData</u>	Sets the text buffer information.
<u>SetTextSelection</u>	Informs the engine of the displayed and selected areas of the text buffer.
<u>IsPronounceable</u>	Determines if a word can be recognized by the engine.
<u>OnCreateRecoContext</u>	Informs the engine of the creation of a recognition context.
<u>OnDeleteRecoContext</u>	Notifies the engine that a recognition context is being destroyed.

<u>PrivateCall</u>	Engine-specific call to the engine.
<u>SetAdaptationData</u>	Provides text data for language model adaptation to the engine.
<u>SetPropertyNum</u>	Sets a numerical property attribute on the SR engine.
<u>GetPropertyNum</u>	Retrieves a numerical property attribute from the SR engine.
<u>SetPropertyString</u>	Sets a text property attribute on the SR engine.
<u>GetPropertyString</u>	Retrieves a text property attribute from the SR engine.
<u>SetGrammarState</u>	Informs the engine if a grammar has been activated or deactivated.
<u>WordNotify</u>	Notifies the SR engine of the words in CFG grammars.
<u>RuleNotify</u>	Notifies the SR engine of the rule information in CFG grammars.
<u>PrivateCallEx</u>	Calls the engine which allows a variable sized data block can be returned from the engine to the engine extension object..
<u>SetContextState</u>	Indicates that a recognition context has been deactivated or activated.



ISpSREngine::SetSite

ISpSREngine::SetSite sets the ISpSREngineSite interface for the engine to use. The SR engine can call back to SAPI using the methods in the ISpSREngineSite.

```
HRESULT SetSite(  
    ISpSREngineSite *pSite  
);
```

Parameters

pSite

Pointer to the ISpEngineSite interface for the engine to use to call back to SAPI.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.



ISpSREngine::GetInputAudioFormat

ISpSREngine::GetInputAudioFormat gets the audio format that the SR engine supports.

```
HRESULT GetInputAudioFormat(  
    const GUID          *pguidSourceFormatId,  
    const WAVEFORMATEX *pSourceWaveFormatEx,  
    GUID                *pguidDesiredFormatId,  
    WAVEFORMATEX       **ppCoMemDesiredWaveFormatEx,  
);
```

Parameters

pguidSourceFormatId

[in] The GUID of the audio format. SAPI determines if the engine can support this. It will either be NULL, indicating that the engine should select its preferred audio format, or it will be set to a given format that the engine determines if it can support.

pSourceWFEX

[in] Address of the WAVEFORMATEX structure containing information about the audio format SAPI is querying the engine about. This will only be set if *pSourceFormatId* is equal to *SPDFID_WaveFormatEx*.

pguidDesiredFormatId

[out] The GUID of the format that the engine can support.

ppCoMemDesiredWFEX

[out] The engine should call a WAVEFORMATEX structure with CoTaskMemAlloc to determine the format the engine

supports. The address of the structure should be placed in this parameter. For non-wave formats, this parameter should be NULL.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Remarks

Audio formats in SAPI are described by two parameters: a GUID, and for wave formats, a WAVEFORMATEX structure.

The first two parameters in this method define the audio format that SAPI queries the engine about. If these parameters are set to a specific format, the engine determines if it can support this format, and returns that format or the nearest format that the engine can support using the second pair of parameters to this function. Alternatively, if the first two parameters are NULL, the engine should return its preferred audio format in the second pair of parameters.



ISpSREngine::RecognizeStream

ISpSREngine::RecognizeStream begins recognition processing on a stream. From this point on, the engine can read data, perform recognition, and send results and events back to SAPI. When all the data has been recognized and read, or the application has deactivated recognition, the engine finishes processing and returns from this method.

```
HRESULT RecognizeStream(  
    REFGUID          rguidFmtId,  
    const WAVEFORMATEX *pWaveFormatEx,  
    HANDLE          hRequestSync,  
    HANDLE          hDataAvailable,  
    HANDLE          hExit,  
    BOOL            fNewAudioStream,  
    BOOL            fRealTimeAudio,  
    ISpObjectToken *pAudioObjectToken  
);
```

Parameters

rguidFmtId

[in] The REFGUID of the input audio format to recognize. This will be SPDFID_WaveFormatEx for wave format files.

pWaveFormatEx

[in] The WAVEFORMATEX structure describing the input format (if it is a wave format). Only a format that the engine has already indicated it can process (by returning the format from [ISpSREngine::GetInputAudioFormat](#)) will be used.

hRequestSync

[in] This is a Win32 event handle that is set whenever there are tasks (such as grammar changes etc.) waiting for the

engine to respond to. The tasks get processed whenever the engine calls Synchronize. The engine can call Synchronize regularly or do so only when this event is set.

hDataAvailable

[in] This is a Win32 event handle that is set when data is available for reading. The amount of data to be available before this event is set can be controlled by calling [ISpSREngineSite::SetBufferNotifySize](#). By default, this event will be set whenever any amount of data is available. This event can be used as an alternative to ISpSREngineSite::DataAvailable.

hExit

[in] This is a Win32 event handle indicating when the engine should exit. The engine on one of two conditions:

- When there is no more data in the stream and it has finished processing, or
- If this event is set. Recognition or Synchronize calls returning S_FALSE indicate that this event has been set.

fNewAudioStream

[in] Indicates whether the input is a new stream. TRUE indicates it is a newly created stream; FALSE otherwise. For example, if an application deactivates the rules, RecognizeStream returns, and later the application activates some rules, the RecognizeStream call will have this parameter set as FALSE because the stream had exited previously. Only if the application calls [ISpRecognizer::SetInput](#) to create a new stream, will this return TRUE. Some engines will find this information useful if resetting channel adaptation, for example, a new telephone call.

fRealTimeAudio

[in] Indicates whether the input is real time audio. TRUE means it is real time audio; FALSE otherwise. Real-time inputs in SAPI are those that implement the ISpAudio interface - for example the standard multi-media microphone input. Non-real time streams are those that only implement ISpStreamFormat - for example input from wave files using the ISpStream object. With non real-time streams all the data is available for reading immediately. The *hDataAvailable* event is always set and the DataAvailable method will always return INFINITE.

pAudioObjectToken

[in] The object token interface for the audio object that the stream was created from. Engines do not need to do anything with this parameter, but it may be useful in some circumstances.

Return values

Value	Description
S_OK	Function completed successfully. This should be returned if the engine is exiting because the stream has ended, or because it was signaled to exit by SAPI.
FAILED (hr)	Appropriate error message if the engine is terminating for an unexpected reason.

Remarks

The engine can read audio data using [ISpSREngineSite::Read](#). The engine determines how much data is available for reading with [ISpSREngineSite::DataAvailable](#), or the *hDataAvailable* event handle. The engine does not have direct access to the

input audio device and will perform in a consistent way regardless of whether input is from desktop audio, wave files, or a custom audio device. The audio format is given by the *rguidFmtId* and *pWaveFormatEx* parameters, and additional details of the audio device can be found from the *fNewAudioStream*, *fRealTimeAudio*, and *pAudioObjectToken* parameters. When a Read call indicates that there is no more data available, the engine should complete processing on the data it has and return from the RecognizeStream method.

The engine recognizes from all rules and/or dictation grammars that have been activated. If there are multiple active rules and/or dictations, the engine is expected to recognize from all things "in parallel." That is, the user is able to say something from any rule that is active. It is possible for this method to be called with nothing active. In this case, the engine can just read data and then discard it, or use it to gather environmental noise information.

Because the engine remains in the RecognizeStream method all the time that it is recognizing, SAPI has effectively given the engine one thread on which to perform recognition. It is possible to write an engine which processes everything on this one thread and thus does not require any additional threads, critical sections, or other thread-locking.

It is also possible to have alternative arrangements with additional threads. For example, one thread could read data, while another thread could do the actual recognition processing. SAPI makes no restrictions about which threads call which methods or whether they are called simultaneously.

The engine uses [ISpSREngineSite::Synchronize](#) to be notified of any grammar or other changes that are pending, and uses [ISpSREngineSite::UpdateRecoPos](#) to keep SAPI informed of how much of the stream has been recognized. The engine passes details of events and recognition results back to SAPI with [ISpSREngineSite::AddEvent](#) and [ISpSREngineSite::Recognition](#).



ISpSREngine::SetRecoProfile

ISpSREngine::SetRecoProfile passes the current active user profile to the engine.

```
HRESULT SetRecoProfile(  
    ISpObjectToken *pProfile  
);
```

Parameters

pProfile

Address of an [ISpObjectToken](#) object that contains the recognition profile token information.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	The <i>pProfile</i> parameter is not a valid ISpObjectToken interface pointer.
FAILED(hr)	Appropriate error message.

Remarks

RecoProfiles are added or removed using Speech properties in Control Panel.

Engines can query the token and include any information needed about this profile, for example, enrollment or training information. Engine specific information should be stored in a subkey of the recognition profile token named with the engine class ID. This is to avoid conflicts with other engines using the same profile.

In order to provide user enrollment the engine can implement a UI component with the name "UserTraining." To instantiate, click Control Panel->Speech properties->SR tab->Train Profile.

SAPI does not restrict applications from calling `ISpRecognizer::SetRecoProfile` during the middle of recognition (see [ISpSREngine::RecognizeStream](#)). If an SR engine does not allow recognition profile changes during recognition, it should fail the `::SetRecoProfile` call gracefully (e.g., return `SPERR_ENGINE_BUSY`).



ISpSREngine::OnCreateGrammar

ISpSREngine::OnCreateGrammar creates a new recognition grammar. Each grammar belongs to a recognition context and can contain dictation, CFG, or proprietary grammar information. The engine can associate a pointer with each grammar that is then passed back to the engine in other methods using this grammar.

```
HRESULT OnCreateGrammar(  
    void *pvEngineRecoContext,  
    SPGRAMMARHANDLE hSAPIGrammar,  
    void **ppvEngineGrammar  
);
```

Parameters

pvEngineRecoContext

[in] The engine's recognition context pointer indicating the context this grammar belongs to. This is the value that the engine passes back to SAPI in the [OnCreateRecoContext](#) method.

hSAPIGrammar

[in] Unique handle to the grammar.

ppvEngineGrammar

[out] The engine should set the contents of this to an arbitrary pointer containing any information the engine has associated with this grammar.

Return values

Value	Description
S_OK	Function completed

	successfully.
FAILED(hr)	Appropriate error message.



ISpSREngine::OnDeleteGrammar

ISpSREngine::OnDeleteGrammar notifies the engine that a recognition grammar is deleted. By the time this method is called, any rules or a loaded SLM in this grammar will already have been deleted.

```
HRESULT OnDeleteGrammar(  
    void    *pvEngineGrammar  
);
```

Parameters

pvEngineGrammar

[in] The engine's grammar pointer for this grammar, as returned from a previous call to the [OnCreateGrammar](#) method.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.



ISpSREngine::LoadProprietaryGrammar

ISpSREngine::LoadProprietaryGrammar instructs the engine to load a grammar in an engine-specific format. This is used to load grammars that are not in the standard SAPI CFG format.

```
HRESULT LoadProprietaryGrammar(  
    void          *pvEngineGrammar,  
    REFGUID       rguidParam,  
    const WCHAR   *pszStringParam,  
    const void    *pvDataParam,  
    ULONG         ulDataSize,  
    SPLOADOPTIONS Options  
);
```

Parameters

pvEngineGrammar

[in] The engine's grammar pointer, as returned from the [OnCreateGrammar](#) method.

rguidParam

[in] Unique identifier for the grammar.

pszStringParam

[in, string] Null-terminated string containing proprietary grammar string data.

pvDataParam

[in] Pointer to grammar image data.

ulDataSize

[in] Size, in bytes, of the grammar image data.

Options

[in] One of the grammar loading options specified in the [SPLOADOPTIONS](#) enumeration sequence.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Remarks

The application can supply the engine with either a GUID, string data, or binary data or some combination of these, in order to describe the grammar. SAPI does nothing with this data apart from correctly marshaling it to the SR engine.



ISpSREngine::UnloadProprietaryGrammar

ISpSREngine::UnloadProprietaryGrammar unloads an engine-specific grammar.

```
HRESULT UnloadProprietaryGrammar(  
    void    *pvEngineGrammar  
);
```

Parameters

pvEngineGrammar

[in] Address of the engine's grammar pointer.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



ISpSREngine::SetProprietaryRuleState

ISpSREngine::SetProprietaryRuleState sets the proprietary grammar rule state. This is used to activate or deactivate rules in non-standard proprietary grammars, where each rule is identified by a string name.

```
HRESULT SetProprietaryRuleState(  
    void                *pvEngineGrammar,  
    const WCHAR         *pszName,  
    const WCHAR         *pszValue,  
    SPRULESTATE         NewState,  
    ULONG               *pcRulesChanged  
);
```

Parameters

pvEngineGrammar

[in] The engine's grammar pointer for this grammar, as returned from a previous call to the [OnCreateGrammar](#) method.

pszName

[in, string] Null-terminated string that contains the grammar rule name, or NULL to indicate all top-level rules should be activated in this grammar.

pszValue

[in, string] Null-terminated string that contains rule value information (Currently always NULL).

NewState

[in] One of the grammar rule states specified in the [SPRULESTATE](#) enumeration sequence.

pcRulesChanged

[out] The number of rules whose state has been changed.
This should be set to 1 if a specific rule name was supplied.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.

Remarks

If `::SetProprietaryRuleState` is called with the rule name set to NULL, the engine should activate or deactivate all top-level rules in this grammar. The *pcRulesChanged* parameter must be set to the number of rules whose state has changed.



ISpSREngine::SetProprietaryRuleIdState

ISpSREngine::SetProprietaryRuleIdState sets the proprietary grammar rule ID state. This is used to activate or deactivate rules in non-standard proprietary grammars where each rule is identified by an ID.

```
HRESULT SetProprietaryRuleIdState(  
    void          *pvEngineGrammar,  
    DWORD         dwRuleId,  
    SPRULESTATE  NewState  
);
```

Parameters

pvEngineGrammar

[in] The engine's grammar pointer for this grammar, as returned from a previous call to the [OnCreateGrammar](#) method.

dwRuleId

[in] The engine proprietary grammar rule identifier.

NewState

[in] One of the grammar rule states specified in the [SPRULESTATE](#) enumeration sequence.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.



ISpSREngine::LoadSLM

ISpSREngine::LoadSLM instructs the engine to load a statistical language model (SLM) for dictation.

```
HRESULT LoadSLM(  
    void          *pvEngineGrammar,  
    const WCHAR  *pszTopicName  
);
```

Parameters

pvEngineGrammar

[in] The engine's grammar pointer, as returned from the [OnCreateGrammar](#) method.

pszTopicName

[in, string] Null-terminated string that specifies a topic name. The default SLM should be loaded if the value of *pszTopicName* is NULL.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.



ISpSREngine::UnloadSLM

ISpSREngine::UnloadSLM instructs the engine to unload an SLM.

```
HRESULT UnloadSLM(  
    void    *pvEngineGrammar  
);
```

Parameters

pvEngineGrammar

[in] The engine's grammar pointer, as returned from the [OnCreateGrammar](#) method.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.



ISpSREngine::SetSLMState

ISpSREngine::SetSLMState activates or deactivates dictation on this grammar.

```
HRESULT SetSLMState(  
    void          *pvEngineGrammar,  
    SPRULESTATE   NewState  
);
```

Parameters

pvEngineGrammar

[in] The engine's grammar pointer for this grammar, as returned from a previous call to the [OnCreateGrammar](#) method.

NewState

[in] One of the grammar rule states specified in the [SPRULESTATE](#) enumeration sequence. This can be `SPRS_ACTIVE` to indicate that dictation is being activated; `SPRS_INACTIVE` to indicate dictation is being deactivated, and `SPRS_ACTIVE_WITH_AUTO_PAUSE` to indicate dictation is being activated with auto-pause. This means that the engine will be put into the paused state each time it returns a final recognition result on this grammar. This is just for information and can be handled in the same way as `SPRS_ACTIVE`.

Return values

Value	Description
<code>S_OK</code>	Function completed successfully.
<code>FAILED(hr)</code>	Appropriate error message.



ISpSREngine::SetWordSequenceData

ISpSREngine::SetWordSequenceData sets a text buffer that the engine can use for recognition.

```
HRESULT SetWordSequenceData(  
    void                                *pvEngineGrammar,  
    const WCHAR                         *pText,  
    ULONG                               cchText,  
    const SPTXTSELECTIONINFO         *pInfo  
);
```

Parameters

pvEngineGrammar

[in] The engine's grammar pointer for this grammar, as returned from a previous call to the [OnCreateGrammar](#) method.

pText

[in] The text buffer.

cchText

[in] The length, in characters, of the text buffer.

pInfo

[in] Address of the [SPTXTSELECTIONINFO](#) structure that contains information of which parts of the buffer are active and currently selected.

Return values

Value	Description

S_OK	Function completed successfully.
E_NOTIMPL	Engine does not support text-buffer functionality
FAILED(hr)	Other appropriate error message.

Remarks

Whenever a text-buffer transition is reached in a CFG, the engine should attempt to recognize a sub-string of words from the current text buffer. This provides a very simple way for applications to select from a set of text.

This method is called when an application calls [ISpRecoGrammar::SetWordSequenceData](#). The format of the buffer is a sequence of one or more null-terminated strings, with a double null-termination at the end. The engine recognizes any sub-string of words from any of the strings in the buffer. It is up to the SR engines to perform word breaking and text normalization for better performance.

It is also possible for the application to alter the areas of the buffer that are used for recognition. The initial range can be set with the structure `SPTXTSELECTIONINFO`, and later calls to [ISpSREngine::SetTextSelection](#) can alter this without changing the actual buffer. The *ulStartActiveOffset* and *cchActiveChars* indicate which area of the buffer should be active for recognition.

The other two fields of the `SPTXTSELECTIONINFO`, *ulStartSelection* and *cchSelection* can be used with dictation. These could indicate, on screen for example, which area of the buffer is currently selected. If *cchSelection* is zero, this could display the current location of the insertion point. The engine can use `SPTXTSELECTIONINFO` to get extra language model context from the preceding words in the dictated text.

This text buffer feature is optional for engines, and support for it

is determined using the *TextBuffer* attribute in the engine object token. If this method is called on an engine that does not support this feature, the engine should return E_NOTIMPL.



ISpSREngine::SetTextSelection

ISpSREngine::SetTextSelection informs the engine of the displayed and selected areas of the text buffer. (See [ISpSREngine::SetWordSequenceData](#)). Once a text buffer has been supplied to the engine, this method can be used to control which parts of the buffer are active for recognition using a text-buffer transition in a CFG. This method can communicate to the engine the location of the current text insertion point for dictation.

```
HRESULT SetTextSelection(  
    void *pvEngineGrammar,  
    const SPTXTSELECTIONINFO *pInfo  
);
```

Parameters

pvEngineGrammar

[in] The engine's grammar pointer for this grammar, as returned from a previous call to the [OnCreateGrammar](#) method.

pInfo

[in] Pointer to the text selection information structure.

Return values

Value	Description
S_OK	Function completed successfully.
E_NOTIMPL	Engine does not support text-buffer functionality
FAILED(hr)	Other appropriate error message.

Remarks

The first two fields of the `SPTXTSELECTIONINFO` structure, *ulStartActiveOffset* and *cchActiveChars* indicate the area of the buffer that should be active for recognition when using a text-buffer transition in a CFG.

The other two fields of the `SPTXTSELECTIONINFO`, *ulStartSelection* and *cchSelection* can be used with dictation. These could indicate, for example, which area of the buffer is currently selected on the screen. If *cchSelection* is zero, this could display the current location of the insertion point. This can be used by the engine to get extra language model context from preceding words in the dictated text.

This text buffer feature is optional for engines, and support for it is indicated using the *TextBuffer* attribute. If this method is called on an engine that does not support this feature, the engine should return `E_NOTIMPL`.



ISpSREngine::IsPronounceable

ISpSREngine::IsPronounceable determines if a word can be recognized by the engine.

This method is called by SAPI after an application calls [ISpRecoGrammar::IsPronounceable](#).

```
HRESULT IsPronounceable(  
    void *pvDrvGrammar,  
    const WCHAR *pszWord,  
    SPWORDPRONOUNCEABLE *pfPronounceable  
);
```

Parameters

pvDrvGrammar

[in] The engine's grammar pointer, as returned from the [OnCreateGrammar](#) method.

pszWord

[in] The word to test.

pfPronounceable

[out] Address of the [SPWORDPRONOUNCEABLE](#) enumeration indicating the results of the test. If the SR engine can generate a reasonable pronunciation for the given word, it should return TRUE in the *pfPronounceable* pointer; otherwise FALSE.

Value	
SPWP_UNKNOWN_WORD_UNPRONOUNCEABLE	The word is not pronounceable

	by the SR engine, and is not located in the lexicon and/or the engine's dictionary.
SPWP_UNKNOWN_WORD_PRONOUNCEABLE	The word is pronounceable by the SR engine, but is not located in the lexicon and/or the engine's dictionary.
SPWP_KNOWN_WORD_PRONOUNCEABLE	The word is pronounceable by the SR engine, and is located in the lexicon and/or the engine's dictionary.

Return values

Value	Description
S_OK	Method completed successfully.
FAILED(hr)	Appropriate error message.

See Also

[ISpRecoGrammar::IsPronounceable](#)



ISpSREngine::OnCreateRecoContext

ISpSREngine::OnCreateRecoContext notifies the engine that a recognition context is being created.

```
HRESULT OnCreateRecoContext(  
    SPRECOCONTEXTHANDLE    hSAPIRecoContext,  
    void                    **ppvEngineContext  
);
```

Parameters

hSAPIRecoContext

[in] Unique handle to the recognition context.

ppvEngineContext

[out] The engine should set the contents of this to a pointer to any engine-specific information it wishes to associate with this context.

Return values

Value	Description
NOERROR	No error is possible with this function.

Each application connected with the SR engine can have one or more recognition contexts. The engine can associate a pointer with each recognition context that is then passed back to the engine in other methods using this context.



ISpSREngine::OnDeleteRecoContext

ISpSREngine::OnDeleteRecoContext notifies the engine that a recognition context is being destroyed. By the time this call is made, all grammars associated with this context will have been deleted.

```
HRESULT OnDeleteRecoContext(  
    void    *pvEngineContext  
);
```

Parameters

pvEngineContext

[in] Pointer to the engine's data for this context, as returned from a previous call to [OnCreateRecoContext](#).

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.



ISpSREngine::PrivateCall

ISpSREngine::PrivateCall makes an engine-specific extension call to the SR engine. This method is called when the engine extension COM object calls `_ISpPrivateEngineCall::CallEngine`. SAPI marshals the data to the main engine object and calls this method.

```
HRESULT PrivateCall(  
    void      *pvEngineContext,  
    PVOID     *pCallFrame,  
    ULONG     ulCallFrameSize,  
);
```

Parameters

pvEngineContext

[in] Pointer to the engine's pointer for this context, as returned from a previous call to [OnCreateRecoContext](#).

pCallFrame

[in] Pointer to the engine-specific data. This can be used both to pass in and return data from the engine. The returned data must be the same size as the passed in data.

ulCallFrameSize

[in] Size, in bytes, of the engine-specific data.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Remarks

The engine must implement an engine-extension COM object and implement whatever interfaces the engine wants. Then an application can call QueryInterface for these interfaces on the recognition context object. The engine extension object can query for the `_ISpPrivateEngineCall` interface from the recognition context. If the engine must directly communicate with the main engine object, it can call the `CallEngine` method, which passes the data to this method. See the [SR Engine Guide](#) for more details on this process.



ISpSREngine::SetAdaptationData

ISpSREngine::SetAdaptationData provides the SR engine with text data from the application for language-model adaptation purposes. This method is called when an application calls [ISpRecoContext::SetAdaptationData](#). If the engine does not support this type of adaptation, it should do nothing in this function and return S_OK.

```
HRESULT SetAdaptationData(  
    void          *pvEngineContext,  
    const WCHAR  *pAdaptationData,  
    const ULONG  cch  
);
```

Parameters

pvEngineContext

[in] Engine's recognition context pointer for the context that is sending the data.

pAdaptationData

[in] Buffer containing the adaptation data. Applications should copy this data before returning from this function if they are going to use this data.

cch

[in] The size, in WCHARs, of the adaptation data in *pAdaptationData*.

Return values

Value	Description
S_OK	Function completed successfully.

FAILED(hr)

Appropriate error message.

Remarks

Some engines may take considerable processing time to perform language model adaptation. Thus, applications should submit adaptation data in chunks to the engine. The engine can then fire an event `SPEI_ADAPTATION` to indicate that it can receive more adaptation data.

Engines can either persist the adaptation in the current `RecoProfile` or reset it every session.



ISpSREngine::SetPropertyNum

ISpSREngine::SetPropertyNum sets a numerical property value on the SR engine.

```
HRESULT SetPropertyNum(  
    SPPROPSRC          eSrc,  
    void              *pvSrcObj,  
    const WCHAR       *pName,  
    LONG              lValue  
);
```

Parameters

eSrc

[in] One of the types specified in the [SPPROPSRC](#) enumeration sequence. (This will currently always be SPPROPSRC_RECO_INST).

pvSrcObj

[in] Pointer to additional information. (Currently always NULL).

pName

[in] String containing the property name.

lValue

[in] Value that the property should be set to.

Return values

Value	Description
S_OK	Function completed successfully. Engine supports this property

	attribute and has set it to the requested value.
S_FALSE	Function completed successfully but engine does not support this property.
FAILED(hr)	Appropriate error message.

Remarks

Applications can use properties to control run-time results of the SR engine. The application can set and get values for specific attributes on the engine. Some values are predefined by SAPI and others may be added by an engine. See [SAPI 5.0 SR Properties White Paper](#) for more details.

This method is called on the engine by SAPI when the application calls `ISpProperties::SetPropertyNum` on its recognition context object. If the engine returns `S_OK` from this method, indicating that it supports this property and has changed the value for it, SAPI will send an `SPEI_PROPERTY_NUM_CHANGE` event to all contexts to inform them of this change.



ISpSREngine::GetPropertyNum

ISpSREngine::GetPropertyNum retrieves a numerical property value from the SR engine.

```
HRESULT GetPropertyNum(  
    SPPROPSRC      eSrc,  
    void          *pvSrcObj,  
    const WCHAR  *pName,  
    LONG         *lValue  
);
```

Parameters

eSrc

[in] One of the types specified in the [SPPROPSRC](#) enumeration sequence. (This will currently always be SPPROPSRC_RECO_INST).

pvSrcObj

[in] Pointer to additional information. (Currently always NULL).

pName

[in] String containing the property name.

lValue

[out] Pointer that the SR engine supplies with the property value information. If the engine does not support this property attribute, it should set the contents of this pointer to zero.

Return values

Value	Description
S_OK	Function completed successfully. Engine supports this property attribute and has returned a value for it.
S_FALSE	Function completed successfully but engine does not support this property.
FAILED(hr)	Appropriate error message.

Remarks

Applications can use properties to control run-time results of the SR engine. The application can set and retrieve values for specific attributes on the engine. Some values are predefined by SAPI and others may be added by an engine. See [SAPI 5.0 SR Properties White Paper](#) for more details.

This method is called on the engine by SAPI when the application calls `ISpProperties::GetPropertyNum` on its recognition context object.



ISpSREngine::SetPropertyString

ISpSREngine::SetPropertyString sets a string property value on the SR engine.

```
HRESULT SetPropertyString(  
    SPPROPSRC          eSrc,  
    void              *pvSrcObj,  
    const WCHAR       *pName,  
    const WCHAR       *pValue  
);
```

Parameters

eSrc

[in] One of the types specified in the [SPPROPSRC](#) enumeration sequence. (This will currently always be SPPROPSRC_RECO_INST).

pvSrcObj

[in] Pointer to additional information. (Currently always NULL).

pName

[in] String containing the property name.

pValue

[in] String value that the property should be set to.

Return values

Value	Description
S_OK	Function completed successfully. Engine supports this property

	attribute and has set it to the requested value.
S_FALSE	Function completed successfully but engine does not support this property.
FAILED(hr)	Appropriate error message.

Remarks

Applications can use properties to control run-time results of the SR engine. The application can set and get values for specific attributes on the engine. Some values are predefined by SAPI and others may be added by an engine. See [SAPI 5.0 SR Properties White Paper](#) for more details.

This method is called on the engine by SAPI when the application calls `ISpProperties:: SetPropertyString` on its recognition context object. If the engine returns `S_OK` from this method, indicating that it supports this property and has changed the value for it, SAPI will send an `SPEI_PROPERTY_STRING_CHANGE` event to all contexts to inform them of this change.



ISpSREngine::GetPropertyString

ISpSREngine::GetPropertyString retrieves a string property value from the SR engine.

```
HRESULT GetPropertyString(  
    SPPROPSRC          eSrc,  
    void              *pvSrcObj,  
    const WCHAR       *pName,  
    WCHAR             **ppCoMemValue  
);
```

Parameters

eSrc

[in] One of the types specified in the [SPPROPSRC](#) enumeration sequence. (This will currently always be SPPROPSRC_RECO_INST).

pvSrcObj

[in] Pointer to additional information. (Currently always NULL).

pName

[in] String containing the property name.

ppCoMemValue

[out] Pointer to a string that the SR engine should supply with the property value string. The string should be allocated with CoTaskMemAlloc; SAPI will delete the allocated memory after return from this function. If the engine does not support this property attribute, the parameter should be NULL.

Return values

Value	Description
S_OK	Function completed successfully. Engine supports this property attribute and has returned a value for it.
S_FALSE	Function completed successfully but engine does not support this property.
FAILED(hr)	Appropriate error message.

Remarks

Applications can use properties to control run-time results of the SR engine. The application can set and retrieve values for specific attributes on the engine. Some values are predefined by SAPI and others may be added by an engine. See [SAPI 5.0 SR Properties White Paper](#) for more details.

SAPI calls this method on an engine when the application calls `ISpProperties::GetPropertyString` using its recognition context object.



ISpSREngine::SetGrammarState

ISpSREngine::SetGrammarState indicates that a grammar has been activated or deactivated by the application calling [ISpRecoGrammar::SetGrammarState](#).

```
HRESULT SetGrammarState(  
    void          *pvEngineGrammar,  
    SPGRAMMARSTATE *eGrammarState  
);
```

Parameters

pvEngineGrammar

[in] The engine's grammar pointer for this grammar, as returned from a previous call to the [OnCreateGrammar](#) method.

eGrammarState

[in] Flag of type SPGRAMMARSTATE indicating the new state of the grammar. This will either be SPGS_DISABLED or SPGS_ENABLED to indicate the grammar is being disabled or enabled; or SPGS_EXCLUSIVE to indicate this grammar has been enabled exclusively and other non-exclusive grammars will be disabled.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.

Remarks

When using standard CFG and dictation grammars, the engine may not need to be informed of the grammar state, as SAPI will automatically activate and deactivate the grammars as necessary. However, when using proprietary grammars, it may be necessary to look at this information.



ISpSREngine::WordNotify

ISpSREngine::WordNotify notifies the SR engine when words in command and control grammars are being added or removed.

```
HRESULT WordNotify(  
    SPCFGNOTIFY           Action,  
    ULONG                cWords,  
    const SPWORDENTRY * pWords  
);
```

Parameters

Action

The SPCFGNOTIFY enumeration value specifying which action, add or delete, is occurring. This will either be SPCFGN_ADD when words are added, or SPCFGN_REMOVE when they are deleted.

cWords

The number of words contained in *pWords*.

pWords

Array of SPWORDENTRY structures containing information on each word.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_NO_WORD_PRONUNCIATION	The engine could not generate a pronunciation

	for some of the words added.
FAILED(hr)	Other appropriate error message.

Remarks

This method is called by SAPI when words are added or removed when the application loads, unloads, or modifies grammars. SAPI internally keeps a reference count and each word will be added only if it is not already present in any existing grammar.



ISpSREngine::RuleNotify

ISpSREngine::RuleNotify notifies the SR engine when CFG rules are added, changed, or removed.

```
HRESULT RuleNotify(  
    SPCFGNOTIFY           Action,  
    ULONG                cRules,  
    const SPRULEENTRY    *pRules  
);
```

Parameters

Action

The [SPCFGNOTIFY](#) enumeration value specifying the action that is occurring.

cRules

The number of rules contained in *pRules*.

pRules

Array of [SPRULEENTRY](#) structures containing information on each rule.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.

Remarks

The engine recognizes each CFG grammar as containing one or more rules. Rules can be top-level, indicating that they can be activated for recognition. Each rule has an initial state and additional states, connected by transitions. Each transition can be one of several types: a word transition indicating a word to be recognized, a rule transition indicating a reference to a sub-rule, an epsilon (null) transition, and some special transitions for such features as embedding dictation within a CFG.

References to sub-rules can be recursive – i.e., rules can reference themselves, either directly or indirectly. Left recursion is not supported and SAPI will reject such grammars upon loading. Inside a grammar, transitions can have semantic properties, although the engine does not normally need to identify these.

SAPI takes full control of loading of a grammar when an application asks it to. The loading can be from a file, a URL, a resource, or from memory, and can involve loading either binary or XML forms of the grammar, and resolving imports. SAPI then notifies the SR engine about the contents of the grammar through various DDI methods.

There are five actions that are performed on rules:

- New rules can be added (SPCFGN_ADD) or existing rules removed (SPCFGN_REMOVE).
- Rules can be activated (SPCFGN_ACTIVATE) or deactivated (SPCFGN_DEACTIVATE) for recognition.
- Rule can be invalidated (SPCFGN_INVALIDATE), which means the rule has been edited by the application and thus the engine needs to reread the contents of the rule.

Each rule is represented by an [SPRULEENTRY](#) structure. This contains an *hRule*, which gives a unique handle identifying the rule. The *pvClientRuleContext* is a pointer that the engine can set using [ISpSREngineSite::SetRuleContext](#). Then, subsequent calls to the [ISpSREngineSite::GetRuleInfo](#) method will return the same structure but with the *pvClientRuleContext* field filled in. The *pvClientGrammarContext* is the pointer that

the engine set in the [ISpSREngine::OnCreateGrammar](#) method. This indicates which grammar the rule belongs to. The *Attributes* field, of type SPCFGRULEATTRIBUTES, contains some flags giving extra information about the rule:

SPRAF_TopLevel if the rule is top level and thus can be activated for recognition.

SPRAF_Active if the rule is currently activated.

SPRAF_Interpreter if the rule is associated with an Interpreter object for semantic processing.

SPRAF_AutoPause if the rule is autopause.

The *hInitialState* gives the initial state of the rule.

In order for the SR engine to find out the full contents of the rule (either immediately, or later during recognition), it can use the [ISpSREngineSite::GetStateInfo](#) method. This gives information about all the subsequent transitions and states following on from any given state (starting with the initial state of the rule).



ISpSREngine::PrivateCallEx

ISpSREngine::PrivateCallEx performs the same task as [ISpSREngine::PrivateCall](#), except that a variable sized data block can be returned from the engine to the engine extension object.

```
HRESULT PrivateCallEx(  
    void          *pvEngineContext,  
    const void    *pInCallFrame,  
    ULONG         ulInCallFrameSize,  
    void          **ppvCoMemResponse,  
    ULONG         *pulResponseSize  
);
```

Parameters

pvEngineContext

[in] The engine's pointer for this context, as returned from a previous call to [OnCreateRecoContext](#).

pInCallFrame

[in] Address of the engine-specific input data.

ulInCallFrameSize

[in] Size, in bytes, of the engine-specific data contained in *pInCallFrame*.

ppvCoMemResponse

[out] Address of a pointer to the response block information from the SR engine. This must be allocated with `CoTaskMemAlloc`.

pulResponseSize

[out] Size, in bytes, of the *ppvCoMemResponse* data.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.



ISpSREngine::SetContextState

ISpSREngine::SetContextState indicates that a recognition context has been activated or deactivated by the application calling [ISpRecoContext::SetContextState](#).

```
HRESULT SetContextState(  
    void * pvEngineContext,  
    SPCONTEXTSTATE eContextState  
);
```

Parameters

pvEngineContext

[in] Pointer to the engine's data for this context, as returned from a previous call to [OnCreateRecoContext](#).

eContextState

[in] The SPCONTEXTSTATE enumeration value specifying the new recognition context state.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED(hr)	Appropriate error message.

Remarks

When using standard CFG and dictation grammars, the engine may not need to be informed of the context state, as SAPI will automatically activate and deactivate the grammars as necessary. However, when using proprietary grammars, it may

be necessary to look at this information.

Microsoft Speech SDK

SAPI 5.1



ISpSREngineSite

The interface ISpEngineSite is implemented by SAPI and is called by the SR engine. It is used by the SR engine to get audio data, retrieve grammar information, send event events, and return recognition information to SAPI.

Full details on how an SR engine interacts with SAPI and how it should call the methods on this interface are given in the "[SR Engine Guide](#)" in the white papers section of the Help files.

Implemented By

- SAPI. Applications or engines do not implement this method.

Created By

- SAPI. The SR engine obtains a pointer to this interface when the [ISpSREngine::SetSite](#) is called, after SAPI creates the SR engine.

Methods in Vtable Order

ISpSREngineSite Methods	Description
Read	Reads audio data from the input stream.
DataAvailable	Retrieves the amount of data that can be read using ISpSREngineSite::Read without blocking.
SetBufferNotifySize	Sets the amount of data to be available before data available event is set.
ParseFromTransitions	Produces an ISpPhraseBuilder

	result from a list of transitions.
<u>Recognition</u>	Returns a recognition result (final, partial, or false) to SAPI.
<u>AddEvent</u>	Sends an event back from the engine to applications.
<u>Synchronize</u>	Informs SAPI that the engine is ready to process changes in its grammars.
<u>GetWordInfo</u>	Retrieves information about a word in a CFG grammar.
<u>SetWordClientContext</u>	Sets an engine-defined pointer on a CFG word.
<u>GetRuleInfo</u>	Retrieves information about a CFG rule.
<u>SetRuleClientContext</u>	Sets an engine-defined pointer on a CFG rule.
<u>GetStateInfo</u>	Retrieves information on the transitions from a CFG state.
<u>GetResource</u>	Retrieves a named resource from a grammar.
<u>GetTransitionProperty</u>	Retrieves semantic property information for a transition in a grammar.
<u>IsAlternate</u>	Determines whether one rule is a valid alternate of another.
<u>GetMaxAlternates</u>	Returns the maximum number of alternates that should be generated for the specified rule.
<u>GetContextMaxAlternates</u>	Returns the maximum number of alternates that should be generated for the specified recognition context.
<u>UpdateRecoPos</u>	Informs SAPI of the current position of the recognizer in the stream to SAPI.



ISpSREngineSite::Read

ISpSREngineSite::Read retrieves a chunk of audio data for the SR engine to convert to text.

```
HRESULT Read(  
    void      *pv,  
    ULONG     cb,  
    ULONG     *pcbRead  
);
```

Parameters

pv

[in] Pointer to the buffer into which the audio input stream data is read.

cb

[in] Specifies the number of bytes of data to attempt to read from the audio input stream.

pcbRead

[out] Pointer to a ULONG variable that receives the actual number of bytes read from the audio input stream.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_AUDIO_BUFFER_OVERFLOW	SAPI's internal audio buffer has filled, and the device has been closed. See Remarks section.

SPERR_AUDIO_BUFFER_UNDERFLOW	The audio object has not received audio data from the device quickly enough, and the device has been closed. See Remarks section.
SPERR_AUDIO_STOPPED	Audio device state has been set to stop.
SPERR_STREAM_NOT_ACTIVE	Method called when engine is not inside RecognizeStream call.
E_OUTOFMEMORY	Exceeded available memory
E_POINTER	At least one of <i>pcbRead</i> or <i>pv</i> are invalid or bad.
FAILED (hr)	Other appropriate error message.

Remarks

The engine requests a certain amount of data to read and supplies a buffer of this size. SAPI will read this amount of data from the audio input. If the amount of data that is requested is not available immediately, SAPI will block this call until the requested amount is available.

If this call returns with a failure code or if the amount that was read is less than the amount requested, this indicates that the stream has ended. The engine should return from the RecognizeStream method after it has finished processing all data.

This method can only be called while the SR engine is inside a [ISpSREngine::RecognizeStream](#) call, although it can be called on any thread.

When the SR engine calls ISpSREngineSite::Read, SAPI will ultimately call ISpAudio::Read and pass the return code back to

the SR engine using `ISpSREngineSite::Read`. If an error code is returned and the error is recoverable, SAPI will automatically detect that an audio error occurred and attempt to reactivate the audio device. The SR engine will then receive a new call, [ISpSREngine::RecognizeStream](#), that will enable it to continue recognizing with minimal audio data loss.

When using a real-time audio device as input, it is important for an SR engine to call `Read` as often as it is able to avoid an audio buffer overflow (for example see [ISpMMSysAudio::Read](#)).

In all cases, if the `Read` call returns a failure code, the engine should not return this as the return value of the `RecognizeStream` method. This failure code can occur for normal conditions indicating the audio has finished.



ISpSREngineSite::DataAvailable

ISpSREngineSite::DataAvailable retrieves the amount of data that can be read using [ISpSREngineSite::Read](#) without blocking.

This method can only be called while the SR engine is inside a [ISpSREngine::RecognizeStream](#) call, although it can be called on any thread.

```
HRESULT DataAvailable(  
    ULONG    *pcb  
);
```

Parameters

pcb

[out] The amount, in bytes, of data available. For real-time audio streams this is the actual amount of data currently available. For non real-time streams this method will always return the value INFINITE. Using the *fRealTimeAudio* parameter on the RecognizeStream method, engines can determine whether this is a real-time stream.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pcb</i> is a bad write pointer.
SPERR_STREAM_NOT_ACTIVE	Method called when engine is not inside RecognizeStream call.
FAILED(hr)	Other appropriate error message.



ISpSREngineSite::SetBufferNotifySize

ISpSREngineSite::SetBufferNotifySize

controls how much data will be available to read before this event gets set. This method is used in conjunction with the *hDataAvailable* Win32 event that is passed as a parameter in [ISpSREngine::RecognizeStream](#).

```
HRESULT SetBufferNotifySize(  
    ULONG    cbSize  
);
```

Parameters

cbSize

[in] The minimum amount of data that should be available before the data available event is set.

Return values

Value	Description
S_OK	Function completed successfully.
SP_UNSUPPORTED_ON_STREAM_INPUT	Function call has no effect as this is a non-real time audio stream.
FAILED(hr)	Other appropriate error message.

Remarks

This can be used if an engine calls Read only when at least a certain amount of data is available.

On non-real time streams, for example when reading from a wave file, this event will always be set, as all the data in the file

is always available for reading. This method will return `SP_UNSUPPORTED_ON_STREAM_INPUT` in this case.

This method can only be called while the SR engine is inside a [ISpSREngine::RecognizeStream](#) call, although it can be called on any thread.



ISpSREngineSite::ParseFromTransitions

ISpSREngineSite::ParseFromTransitions parses an [ISpPhraseBuilder](#) result from a list of CFG transitions.

```
HRESULT ParseFromTransitions(  
    const SPPARSEINFO *pParseInfo,  
    ISpPhraseBuilder **ppPhrase  
);
```

Parameters

pParseInfo

[in] Address of the [SPPARSEINFO](#) structure containing phrase information.

ppPhrase

[out] Address of a pointer to an [ISpPhraseBuilder](#) interface that receives the phrase information.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Remarks

This method is called to produce a phrase which can be used to send CFG results back to SAPI. For more details on this method, see the [SR Engine Guide](#).



ISpSREngineSite::Recognition

ISpSREngineSite::Recognition returns a recognition result to SAPI.

```
HRESULT Recognition(  
    SPRECORESULTINFO *pResultInfo  
);
```

Parameters

pResultInfo

[in] Pointer to type `SPRECORESULTINFO` indicating the results.

Return values

Value	Description
S_OK	Function completed successfully and to continue recognition.
S_FALSE	Function completed successfully and the engine can terminate recognition without reading the rest of the stream.
FAILED (hr)	Appropriate error message.

Remarks

The phrase can be either a hypothesis or a final result. If it is a hypothesis, a hypothesis event is issued to all interested recognition contexts. A final result event is issued to the target grammar that the result refers to. An engine can also send a false recognition with this method, indicating it has low confidence in the result.

If the return value from this call is `S_FALSE`, there are no more

active rules and the engine is free to exit the RecognizeStream call without reading or processing any more data. Otherwise, the engine should continue reading data and continue recognition.

An [ISpSREngineSite::AddEvent](#) call with an SPEI_PHRASE_START parameter as the event type must precede the call to Recognition. For more details on this method, see the [SR Engine Guide](#).



ISpSREngineSite::AddEvent

ISpSREngineSite::AddEvent sends an event back from the engine to applications.

For more details on this method and the different events that can be fired, see the [SR Engine Guide](#).

```
HRESULT AddEvent(  
    const SPEVENT *pEvent,  
    SPRECOCONTEXTHANDLE hContext  
);
```

Parameters

pEvent

[in] Address of the [SPEVENT](#) structure containing the event information.

hContext

[in] The RecoContext is the event handle passed to SR engine from SAPI through [ISpSREngine::OnCreateRecoContext](#). This value should normally be set to NULL indicating the event is a global one.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	At least one of <i>pEvent</i> or <i>hContext</i> is invalid or bad. Alternatively, it indicates that an event is being added to an inappropriate mode.
E_POINTER	Invalid pointer.

SPERR_STREAM_POS_INVALID	The current audio stream offset is greater than either the current seek position or the last sync position. Alternatively, if the event stream is not initialized, the stream position is not zero.
FAILED(hr)	Appropriate error message.



ISpSREngineSite::Synchronize

ISpSREngineSite::Synchronize informs SAPI that the engine is ready to process changes in its grammars.

```
HRESULT Synchronize(  
    ULONGLONG    ullStreamPos  
);
```

Parameters

ullStreamPos

[in] The position within the audio stream that the engine has completed recognizing. SAPI discards its stored audio up to this point. The engine cannot fire more events prior to this position. However, the engine will still be informed of current grammar changes regardless of the value of the parameter.

Return values

Value	Description
S_OK	Function completed successfully; recognition should continue.
S_FALSE	Function completed successfully and the engine can terminate recognition without reading the rest of the stream.
SPERR_STREAM_NOT_ACTIVE	Stream is not initialized.
SPERR_STREAM_POS_INVALID	Stream position is either greater than the current seek position or less than the last synchronized position.
FAILED (hr)	Appropriate error message.

Remarks

If there are any changes pending, SAPI will call back to the engine to inform it of any changes using `WordNotify` or `RuleNotify`. When the engine returns from these methods, SAPI will return back from the `Synchronize` call. If the return value from this call is `S_FALSE`, and there are no more active rules, the engine is free to exit the `RecognizeStream` call without reading or processing more data.

The engine can choose when to call `Synchronize`. Often an engine will respond to state changes when no speech is detected, but it will not respond when the user is speaking. It is important, however, to periodically, if not routinely, call `Synchronize`. Specifically, if an application attempts to release its final reference to SAPI, and no other applications are connected, SAPI will attempt to shutdown the SR engine. However, the shutdown process will wait indefinitely for the SR engine to reach its next synchronization point (e.g., the speech recognition engine calls [ISpSREngineSite::Synchronize](#)).

This method can only be called while the SR engine is inside a [ISpSREngine::RecognizeStream](#) call, although it can be called on any thread. For more details on this method, see the [SR Engine Guide](#).

An example of the synchronization for the user would be starting dictation mode, and activating a non-silence noise source in the background that will generate a single continuous recognition. When the user attempts to exit the application, the exit will be blocked until the SR engine finishes recognizing the audio stream. For this reason, the SR engine should call [ISpSREngineSite::Synchronize](#) periodically to prevent extended delays in state changes (e.g., application shutdown, grammar changes, etc.), even when performing a long recognition. This ensures that the SR engine is able to properly clean up and exit its [ISpSREngine::RecognizeStream](#) method.



ISpSREngineSite::GetWordInfo

ISpSREngineSite::GetWordInfo retrieves information about a word in a CFG grammar.

```
HRESULT GetWordInfo(  
    SPWORDENTRY *pWordEntry,  
    SPWORDINFOOPT Options  
);
```

Parameters

pWordEntry

[in, out] Address of the [SPWORDENTRY](#) structure that contains the grammar word entry information. This can be called with only the *pWordEntry->hWord* word handle set. The following members may be allocated with `CoTaskMemAlloc()` and if so, each must be freed by the engine with `CoTaskMemTaskFree()` when no longer required.

<i>pWordEntry->pszDisplayText</i>
<i>pWordEntry->pszLexicalForm</i>
<i>pWordEntry->aPhoneld</i>

Options

[in] One of the grammar word options specified in the [SPWORDINFOOPT](#) enumeration. If `SPWIO_NONE`, the *LangID* and *pvClientContext* are filled in. If `SPWIO_WANT_TEXT`, the display and lexical text and pronunciation are also filled in.

Return values

Value	Description
<code>S_OK</code>	Function completed successfully.
<code>E_POINTER</code>	<i>pWordEntry</i> points to invalid memory.

E_INVALIDARG	Either invalid <i>pWordEntry</i> -> <i>hWord</i> word handle or <i>Options</i> contains invalid flags.
E_OUTOFMEMORY	Not enough memory to complete the operation.
FAILED (hr)	Appropriate error message.



ISpSREngineSite::SetWordClientContext

ISpSREngineSite::SetWordClientContext sets an engine-defined context pointer on a CFG word.

This allows an engine to associate a pointer to its own data with each word. This can be quickly recovered with the [ISpSREngineSite::GetWordInfo](#) method.

```
HRESULT SetWordClientContext(  
    SPWORDHANDLE    hWord,  
    void            *pvClientContext  
);
```

Parameters

hWord

[in] The handle for a word.

pvClientContext

[in] Pointer to the engine's data it wishes to associate with this word.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_INVALID_HANDLE	Invalid word handle parameter.
FAILED (hr)	Other appropriate error message.



ISpSREngineSite::GetRuleInfo

ISpSREngineSite::GetRuleInfo retrieves information about a CFG rule.

```
HRESULT GetRuleInfo(  
    SPRULEENTRY *pRuleEntry,  
    SPRULEINFOPT Options  
);
```

Parameters

pRuleEntry

[in, out] Address of the [SPRULEENTRY](#) structure that contains the grammar rule entry information.

Options

[in] One of the grammar rule options specified in the [SPRULEINFOPT](#) enumeration sequence. (This should always be set to SPRIO_NONE).

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pRuleEntry</i> points to invalid memory.
E_INVALIDARG	Either invalid <i>hRule</i> rule handle or <i>Options</i> contains invalid flags.
FAILED (hr)	Appropriate error message.

Remarks

This method can be called after the engine has been informed of a rule with the [ISpSREngine::RuleNotify](#) method. This method can be used to recover rule information from the rule handle.

Only the *hRule* rule handle field in the SPRULEENTRY needs to be filled in by the engine when calling this method. The engine will fill in:

- The *pvClientRuleContext*, which is a pointer that the engine can set using [ISpSREngineSite::SetRuleContext](#).
- The *pvClientGrammarContext*, which is the pointer that the engine set in the [ISpSREngine::OnCreateGrammar](#) method. This indicates which grammar the rule belongs to.
- The *Attributes* field, of type [SPCFGRULEATTRIBUTES](#), which contains some flags giving extra information about the rule:

SPAudioBufferInfo.ulMsMinNotification cannot be larger than one quarter the size of SPAudioBufferInfo.ulMsBufferSize and must not be zero.

SPRAF_TopLevel if the rule is top-level and thus can be activated for recognition.

SPRAF_Interpreter if the rule is associated with an [Interpreter](#) object for semantic processing.

SPRAF_AutoPause if the rule is auto-pause.

- The *hInitialState* field, which gives the initial state of the rule. Information on this and subsequent states can be obtained by calling the [ISpSREngineSite::GetStateInfo](#) method.



ISpSREngineSite::SetRuleClientContext

ISpSREngineSite::SetRuleClientContext sets an engine-defined pointer on a CFG rule.

This allows an engine to associate a pointer to its own data with each rule. This can be quickly recovered with [ISpSREngineSite::GetRuleInfo](#).

```
HRESULT SetRuleClientContext(  
    SPRULEHANDLE    hRule,  
    void            *pvClientContext  
);
```

Parameters

hRule

[in] Handle of a rule.

pvClientContext

[in] Pointer to the engine's data it wishes to associate with this rule.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_INVALID_HANDLE	Invalid rule handle parameter.
FAILED (hr)	Other appropriate error message.



ISpSREngineSite::GetStateInfo

ISpSREngineSite::GetStateInfo retrieves information on the transitions from a CFG state.

```
HRESULT GetStateInfo(  
    SPSTATEHANDLE    hState,  
    SPSTATEINFO      *pStateInfo  
);
```

Parameters

hState
[in] Handle to the current state.

pStateInfo
[out] The state information.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Either <i>pStateInfo</i> or <i>pTransitions</i> in <i>pStateInfo</i> points to invalid memory.
E_OUTOFMEMORY	Not enough memory to complete the operation.
SPERR_INVALID_HANDLE	Invalid state handle parameter.
FAILED (hr)	Appropriate error message.

Remarks

This method is called so that the SR engine can discern the full contents of the rule. `GetStateInfo` can be called immediately upon receiving a [ISpSREngine::RuleNotify](#) call about a rule, or

later during recognition. This method supplies information about all the subsequent states from any given state.

The engine passes this method a state handle (starting with the *hInitialState* of the rule), and a pointer to an SPSTATEINFO structure with all its fields initially zeroed. This structure is filled out with information on all of the transitions out of that state in the *pTransitions* array. SAPI uses CoTaskMemAlloc to allocate this array. The engine can call this method again on each of the states following the current state, and so on, in order to get information about all of the states in the rule. Loop-back transitions are possible in a rule and the engine may need to confirm that it has not visited the state before.

When the engine calls GetStateInfo subsequent times, it can call it with the *cAllocatedEntries* and *pTransitions* fields unchanged from the last call. If possible, SAPI will then re-use the memory from the transition array rather than re-allocating it.

Alternatively, the engine can use CoTaskMemFree to free the *pTransitions* memory, set these fields to NULL and SAPI will re-allocate the memory every time.

Each transition represents a link from one state to another state. Each transition is represented by an SPTRANSITIONENTRY structure. This structure contains an ID field that uniquely identifies the transition, an *hNextState* handle that indicates the state the transition is connected to, and a Type field that indicates the type of transition.

There are three common types of transition, which all engines should support:

- Word transitions (SPTRANSWORD). These represent single words that the recognizer will recognize before advancing to the next state. The handle to the word and the engine's word pointer are supplied inside the SPTRANSITIONENTRY structure. To produce recognition results, the engine needs to keep track of the transition IDs of word transitions as they are used in the ParseFromTransitions method.

- Rule transitions (SPTRANSRULE). These represent transitions into sub-rules. This transition is only passed when a path through the sub-rule has been recognized. The rule handle, engine's rule pointer and initial state of the sub-rule are supplied. Rules can be recursive (not left recursive).
- Epsilon transitions (SPTRANSEPSILON). These are NULL transitions that can be traversed without recognizing anything.

A state that has a transition to a null state handle indicates the end of a rule. There can also be 'void' states, which block and indicate that there is no recognition path from this state. A void state has zero transitions out of it.

There are also a number of special transitions, which may not be supported by all engines.

See the [SR Engine Guide](#) document for more information on CFG grammars.



ISpSREngineSite::GetResource

ISpSREngineSite::GetResource retrieves a named resource from a grammar.

```
HRESULT GetResource(  
    SPRULEHANDLE    hRule,  
    WCHAR           *pszResourceName,  
    WCHAR           **ppCoMemResource  
);
```

Parameters

hRule

[in] The rule handle.

pszResourceName

[in] Null-terminated string containing the name of the resource to recover.

ppCoMemResource

[out] The resource associated with the rule. Applications calling this method must call CoTaskMemFree() to free memory associated with this resource.

Return values

Value	Description
S_OK	Function completed successfully and the rule contained a resource of the correct name.
S_FALSE	Function completed successfully but no resource was found.
E_INVALIDARG	<i>pszResourceName</i> points to invalid

	string.
E_POINTER	<i>ppCoMemResource</i> invalid or bad.
E_NOTIMPL	Method is not implemented.
SPERR_INVALID_HANDLE	Invalid <i>hRule</i> handle.
FAILED (hr)	Appropriate error message.

Remarks

Within a CFG, each rule can contain one or more named strings containing arbitrary string data. The engine can recover this data using `::GetResource` and passing in the rule handle and resource name.



ISpSREngineSite::GetTransitionProperty

ISpSREngineSite::GetTransitionProperty retrieves semantic property information for a transition in a grammar.

```
HRESULT GetTransitionProperty(  
    SPTRANSITIONID          ID,  
    SPTRANSITIONPROPERTY **ppCoMemProperty  
);
```

Parameters

ID

[in] The transition identifier.

ppCoMemProperty

[out] Address of a pointer to a [SPTRANSITIONPROPERTY](#) that receives the property information. Applications calling this method must call `CoTaskMemFree()` to free memory returned. If the transition does not have a semantic property, this will be set to point to `NULL`, and `S_FALSE` will be returned.

Return values

Value	Description
<code>S_OK</code>	Function completed successfully and transition has a property.
<code>S_FALSE</code>	Function completed successfully but transition does not have a property.
<code>E_INVALIDARG</code>	One or more parameters are invalid.
<code>E_OUTOFMEMORY</code>	Exceeded available memory.
<code>FAILED(hr)</code>	Appropriate error message.

Remarks

CFG grammars can contain properties (also known as 'semantic tags') within a grammar. This provides a means for semantic information to be embedded inside a grammar.

By default, the engine does not recognize these properties. Typically, an engine will recognize only the speech from the words in the grammar, and SAPI will parse and add the property information in the [ISpSREngineSite::ParseFromTransitions](#) call. However, it is possible for an engine to recognize this information by calling this method on any transition. If there is a property on this transition, the property name or ID and value is returned in the SPTRANSITIONPROPERTY structure, which must be freed after each use using CoTaskMemFree.

An engine can find out if a transition has a semantic property attached before calling this method by looking at the *fHasProperty* flag in the SPTRANSITIONENTRY structure associated with this transition.



ISpSREngineSite::IsAlternate

ISpSREngineSite::IsAlternate determines whether one rule is an alternate of another.

```
HRESULT IsAlternate(  
    SPRULEHANDLE    hPriRule,  
    SPRULEHANDLE    hAltRule  
);
```

Parameters

hPriRule

[in] The primary rule.

hAltRule

[in] The alternate rule to be checked.

Return values

Value	Description
S_OK	<i>hAltRule</i> is an alternate of <i>hPriRule</i> .
S_FALSE	<i>hAltRule</i> is not an alternate of <i>hPriRule</i> .
FAILED (hr)	Appropriate error message.

Remarks

This method is used because it is not possible to return alternates for CFG rules from different recognition contexts. This method provides an easy way for engines to determine whether two rules belong to the same context.



ISpSREngineSite::GetMaxAlternates

ISpSREngineSite::GetMaxAlternates passes back the maximum number of alternates that should be generated for the specified CFG rule.

```
HRESULT GetMaxAlternates(  
    SPRULEHANDLE    hRule,  
    ULONG           *pu1NumAlts  
);
```

Parameters

hRule

[in] The rule to check.

pu1NumAlts

[out] The maximum number of alternates for the rule.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pu1NumAlts</i> is invalid or bad.
FAILED (hr)	Appropriate error message.



ISpSREngineSite::GetContextMaxAlternates

ISpSREngineSite::GetContextMaxAlternates passes back the maximum number of alternates that should be generated for the specified recognition context.

```
HRESULT GetContextMaxAlternates(  
    SPRECOCONTEXTHANDLE    hContext,  
    ULONG                   *pulNumAlts  
);
```

Parameters

hContext

[in] Handle to the context.

pulNumAlts

[out] The maximum number of alternates to generate.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pulNumAlts</i> is invalid or bad.
FAILED (hr)	Appropriate error message.

Remarks

For SAPI command and control grammars, it is usually easier to use the [ISpSREngineSite::GetMaxAlternates](#) method.

This method currently applies to command and control and proprietary grammars, but not to dictation grammars.



ISpSREngineSite::UpdateRecoPos

ISpSREngineSite::UpdateRecoPos returns the current position of the recognizer in the stream to SAPI. An engine should call this regularly, up to several times a second, regardless of whether it is recognizing speech or silence.

This method can only be called while the SR engine is inside a [ISpSREngine::RecognizeStream](#) call, although it can be called on any thread.

```
HRESULT UpdateRecoPos(  
    ULONGLONG    ullStreamPos  
);
```

Parameters

ullStreamPos

[out] The current stream position of the recognizer.

Return values

Value	Description
S_OK	Function completed successfully.



ISpSRAlternates

ISpSRAlternates defines the engine-level interface by which SAPI communicates with a speech recognition engine's alternate analyzer. The two main features of an alternate analyzer are the ability to generate alternate phrases for a recognized phrase (see [GetAlternates](#)) and the ability to update the speech recognition engine's acoustic and/or language models based on alternate selection (see [Commit](#)).

When to Use

The ISpSRAlternates interface is an engine-level interface, called by SAPI when an SR application calls the respective application-level interface (e.g., [ISpRecoResult::GetAlternates](#) for [ISpSRAlternates::GetAlternates](#), [ISpPhraseAlt::Commit](#) for [ISpSRAlternates::Commit](#)). The application should not call the ISpSRAlternates interface directly.

When to Implement

The ISpSRAlternates interface should be implemented as a COM object by the speech recognition engine vendor. The object will be used by SAPI to generate alternates of phrases recognized by the vendor's engine and to commit updates to the engine's acoustic and/or language model.

See the [Speech Recognition Engine Porting Guide](#) for more information on how to implement the alternate analyzer object.

Methods in Vtable Order

ISpSRAlternates Methods	Description
GetAlternates	Retrieves a list of alternate phrases.
Commit	Instructs the alternate analyzer to update the speech recognition (SR) engine's acoustic and/or language

model based on the selected alternative phrase, versus the original phrase.



ISpSRAlternates::GetAlternates

GetAlternates retrieves a list of alternate phrases.

SAPI calls **GetAlternates** when an application calls [ISpRecoResult::GetAlternates](#). The alternate analyzer uses the [SPPHRASEALTREQUEST](#) information to generate at most, the requested number of alternate phrases, returned using the [SPPHRASEALT](#) pointer.

If the alternate analyze needs information that is not included in the phrase structure (see [SPPHRASE](#)), the speech recognition (SR) engine can store a custom, private block of data that is sent to the alternate analyze in the [SPPHRASEALTREQUEST.pvResultExtra](#) field. See also [SPPARSEINFO.pSREnginePrivateData](#) and [SPPHRASE.pSREnginePrivateData](#).

The alternate analyzer communicates with the SR engine by retrieving the [ISpRecoContext](#) interface from the [SPPHRASEALTREQUEST.pRecoContext](#) field, and querying (see [IUnknown::QueryInterface](#)) for the SR engine's private extension (see [ISpPrivateEngineCall](#)).

```
HRESULT GetAlternates(  
    SPPHRASEALTREQUEST    *pAltRequest,  
    SPPHRASEALT            **ppAlts,  
    ULONG                  *pcAlts  
);
```

Parameters

pAltRequest

[in] Pointer to a structure of type, [SPPHRASEALTREQUEST](#), which points to information about the alternate request (e.g., original phrase, number of alternates requested, private SR engine data, etc.).

ppAlts

[out] Pointer to a list of structures of type [SPPHRASEALT](#) for alternate phrases. The alternate analyzer uses the ISpPhraseBuilder interface to create the alternate phrases and return pointers to the alternates in the SPPHRASEALT.pPhrase field.

pcAlts

[out] The actual number of alternates in *ppAltslist*. The alternate analyzer returns the number of alternates it actually generated. If it cannot generate alternates for the original phrase, it should return zero. The number should be less than or equal to the number requested by the application (see [SPPHRASEALTREQUEST.ulRequestAltCount](#)).

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



ISpSRAlternates::Commit

ISpSRAlternates::Commit instructs the alternate analyzer to update the speech recognition (SR) engine's acoustic and/or language model based on the selected alternate phrase, versus the original phrase.

SAPI calls Commit when an application calls [ISpPhraseAlt::Commit](#). The alternate analyzer compares the original phrase (see [SPPHRASEALTREQUEST.pPhrase](#)) and the new alternate phrase (see [SPPHRASEALT](#)) to improve recognition accuracy.

Depending on the manufacturer, the SR engine might provide a custom related data block (see [SPPHRASEALTREQUEST.pvResultExtra](#)) so that the analyzer can synchronize the data block with the new alternate phrase. If the user wants to re-analyze the new phrase (after being serialized and deserialized), or if the user chooses a different alternate, SAPI provides the revised data block to the analyzer.

The alternate analyzer communicates with the SR engine by retrieving the ISpRecoContext interface from the [SPPHRASEALTREQUEST.pRecoContext](#) field, and querying (see [IUnknown::QueryInterface](#)) for the SR engine's private extension (see [ISpPrivateEngineCall](#)).

```
HRESULT Commit(  
    SPPHRASEALTREQUEST *pAltRequest,  
    SPPHRASEALT *pAlt,  
    void **ppvResultExtra,  
    ULONG *pcbResultExtra  
);
```

Parameters

pAltRequest

[in] A pointer to the structure of type [SPPHRASEALTREQUEST](#)

that specifies the original alternate request and phrase information.

pAlt

[in] A pointer to the structure of type [SPPHRASEALT](#) that specifies the application-chosen alternate phrase.

ppvResultExtra

[out] Additional engine-defined information that should be included with recognition result.

pcbResultExtra

[out] Size, in bytes, of *ppvResultExtra*.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



Text-to-speech engine interfaces (DDI-level)

The following section covers:

- [ISpTTSEngine](#)
- [ISpTTSEngineSite](#)



ISpTTS Engine

The SAPI speech synthesis (text-to-speech, or TTS) engine implements the ISpTTS Engine interface.

[ISpTTS Engine::Speak](#) is the primary method called by SAPI to perform speech rendering. SAPI, rather than the engine, performs XML parsing of the input text stream. The Speak method receives a linked list of text fragments with their associated XML attribute states. The Speak method also receives a pointer to the ISpVoice [ISpTTS EngineSite](#) interface. The TTS engine uses this interface to queue events and to write the output audio data.

Even though SAPI 5 is a free-threaded architecture, TTS engine instances will always be called by SAPI on a single thread. TTS engines are never directly accessed by applications. SAPI ensures that all parameter validation and thread synchronization has been performed properly before calling the TTS engine. All calls to the TTS engine in the release build of SAPI are within a try or except block to protect applications from faulting.

Methods in Vtable Order

ISpTTS Engine Methods	Description
Speak	Renders the specified text fragment list in the specified output format.
GetOutputFormat	Queries the engine about a specific output format.



ISpTTSEngine::Speak

ISpTTSEngine::Speak renders the specified text fragment list in the specified output format.

```
HRESULT Speak(  
    DWORD                dwSpeakFlags,  
    REFGUID              rguidFormatId,  
    const WaveFormatEx*  pWaveFormatEx,  
    const SPVTEXTFRAG*  pTextFragList,  
    ISpTTSEngineSite*   pOutputSite  
);
```

Parameters

dwSpeakFlags

[in] Possible values are contained in the [SPEAKFLAGS](#) enumeration, but all values other than `SPF_NLP_SPEAK_PUNC` will be masked off. If `SPF_NLP_SPEAK_PUNC` is set, the engine should speak all punctuation (e.g., "This is a sentence." should be expanded to "This is a sentence period").

rguidFormatId

[in] The stream format identifier describing the required output format. This format is guaranteed to be one that the engine specified as supported in a previous [GetOutputFormat](#) call.

SPDFID_Text

Engines are *not* required to support this format, nor are they required to do anything specific with this format if they do support it. It is provided merely for

	debugging purposes.
SPDFID_WaveFormatEx	<i>pWaveFormatEx</i> will be a WAVEFORMATEX structure describing an output format specified by the engine in a previous GetOutputFormat call.

pWaveFormatEx

[in] Pointer to a [WAVEFORMATEX](#) structure describing the output format. Will be NULL if rguidFormatID is SPDFID_Text.

pTextFragList

[in] A linked list of [SPVTEXTFRAGs](#) to synthesize. See the [TTS Engine Vendor Porting Guide](#) for more information.

pOutputSite

[in] Pointer to an [ISpTTSEngineSite](#) where audio data and events should be written.



ISpTTSEngine::GetOutputFormat

ISpTTSEngine::GetOutputFormat queries the engine about a specific output format. The engine should examine the requested output format, and return the closest format that it supports.

```
HRESULT GetOutputFormat(  
    const GUID          *pTargetFmtId,  
    const WAVEFORMATEX *pTargetWaveFormatEx,  
    GUID               *pOutputFormatId,  
    WAVEFORMATEX      **ppCoMemOutputWaveFormatEx  
);
```

Parameters

pTargetFmtId

[in] Address of the GUID describing the requested output format.

SPDFID_Text	Engines are <i>not</i> required to support this format, nor are they required to do anything specific with this format if they do support it. It is provided merely for debugging purposes.
SPDFID_WaveFormatEx	<i>pWaveFormatEx</i> will be a WAVEFORMATEX structure.

pTargetWaveFormatEx

[in] Pointer to the WAVEFORMATEX structure describing the

requested output format.
Will be NULL if *pTargetFmtId* is SPDFID_Text.

pOutputFormatId

[out] Address of a GUID to receive the engine's supported output format identifier.

SPDFID_Text	If the engine does support SPDFID_Text, SPDFID_Text should be used, and <i>ppCoMemOutputWaveFormatEx</i> should be set to NULL.
SPDFID_WaveFormatEx	Output format will be described by a WAVEFORMATEX structure.

ppCoMemOutputWaveFormatEx

[out] The engine allocates space for a WAVEFORMATEX structure using CoTaskMemAlloc. This structure describes the supported output format.



ISpTTSEngineSite

The ISpTTSEngineSite interface is implemented by the SAPI SpVoice object. It is used to write audio data and events. See the [TTS Engine Vendor Porting Guide](#) for more information.

The ISpTTSEngineSite interface inherits from [ISpEventSink](#). [AddEvents](#) and [GetEventInterest](#) are included in this interface.

Methods in Vtable Order

ISpTTSEngineSite Methods	Description
ISpEventSink inherited methods.	All the methods of ISpEventSink are accessible from this interface.
GetActions	Queries the ISpVoice to determine what action(s) to perform.
Write	Sends output data (normally audio) to SAPI.
GetRate	Retrieves the current TTS rendering rate adjustment that should be used by the engine.
GetVolume	Retrieves the base output volume level the engine should use during synthesis.
GetSkipInfo	Retrieves the number and type of items to be skipped in the text stream.
CompleteSkip	Notifies the SpVoice object that the last skip request has been completed and to pass it the results.



ISpTTSEngineSite::GetActions

ISpTTSEngineSite::GetActions queries the SpVoice object to determine which real-time action(s) to perform. An engine should call this method frequently during the rendering process to be as responsive as possible. SAPI returns a DWORD indicating which action(s) contained in the [SPVESACTIONS](#) enumeration should be performed. See [GetRate](#), [GetVolume](#), and [GetSkipInfo](#) for more information.

```
DWORD GetActions ( void );
```

Parameters

None.

Return values

DWORD containing one or more values from SPVESACTIONS specifying the action(s) to perform.



ISpTTSEngineSite::Write

ISpTTSEngineSite::Write sends output data (normally audio) to SAPI.

```
HRESULT Write(  
    const void    *pBuff,  
    ULONG        cb,  
    ULONG        *pcbWritten  
);
```

Parameters

pBuff

Pointer to synthesized speech audio data. The output format is specified by SAPI as a parameter to the [ISpTTSEngine::Speak](#) call.

cb

The buffer size, in bytes (*not* samples), of *pBuff*.

pcbWritten

Pointer to a ULONG which receives the number of bytes actually copied.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pBuff</i> is bad or invalid.
E_POINTER	<i>pcbWritten</i> is bad or invalid.
SPERR_UNINITIALIZED	Output stream cannot be initialized.

Remarks

SAPI handles sending the audio data to the correct output destination. It is important that any events associated with the audio data are queued by calling [ISpEventSink::AddEvents](#) prior to calling this method. This ensures proper synchronization of event firing and audio rendering.



ISpTTSEngineSite::GetRate

ISpTTSEngineSite::GetRate retrieves the current TTS rendering rate adjustment that should be used by the engine.

```
HRESULT GetRate(  
    long    *pRateAdjust  
);
```

Parameters

pRateAdjust

[out] Pointer to a long which specifies the baseline rate.

Return values

Value	Description
E_POINTER	<i>pRateAdjust</i> is invalid
S_OK	Function completed successfully.

Remarks

This function should be called when a call to [GetActions](#) returns SPVES_RATE. The retrieved value establishes a baseline rate. Additional rate adjustments in the XML state should be combined with this value to determine the actual absolute rate adjustment.



ISpTTSEngineSite::GetVolume

ISpTTSEngineSite::GetVolume retrieves the base output volume level the engine should use during synthesis.

```
HRESULT GetVolume(  
    USHORT* pusVolume  
);
```

Parameters

pusVolume

[out] Pointer to a USHORT which specifies the baseline volume level.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	<i>pusVolume</i> is invalid.

Remarks

This function should be called when a call to [GetActions](#) returns SPVES_VOLUME. The retrieved value establishes a baseline volume. Additional volume adjustments in the XML state should be combined with this value to determine the actual absolute volume level.



ISpTTEngineSite::GetSkipInfo

ISpTTEngineSite::GetSkipInfo retrieves the number and type of items to be skipped in the text stream.

```
HRESULT GetSkipInfo(  
    SPVSKIPTYPE *peType,  
    long *plNumItems  
);
```

Parameters

peType

[out] Pointer to an [SPVSKIPTYPE](#) which specifies the type of item to skip. Currently only sentences.

plNumItems

[out] Pointer to a long that specifies the number of items to skip.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	One of the return addresses is invalid.

Remarks

This function should be called when a call to [GetActions](#) returns SPVES_SKIP. *plNumItems* can be positive, signifying a forward skip, or negative, signifying a backward skip, or zero, signifying a skip to the beginning of the current item.

After the engine has skipped as many items as possible, it must call [ISpTTEngineSite::CompleteSkip](#) to inform SAPI of how

many items were successfully skipped. If the engine was unable to skip the requested number of items, it should end its current Speak call immediately.



ISpTTSEngineSite::CompleteSkip

ISpTTSEngineSite::CompleteSkip notifies the SpVoice object that the last skip request has been completed and to pass it the results.

```
HRESULT CompleteSkip(  
    long    INumSkipped  
);
```

Parameters

INumSkipped

[in] Specifies the number of items that actually were skipped. It is invalid to skip more items than were requested.

Return values

Value	Description
E_INVALIDARG	Engine skipped more items than requested.
S_OK	Function completed successfully.



Structures

The following structures are used with SAPI 5.

- [SPAUDIOBUFFERINFO](#)
- [SPAUDIOSTATUS](#)
- [SPBINARYGRAMMAR](#)
- [SPEVENT](#)
- [SPEVENTSOURCEINFO](#)
- [SPPARSEINFO](#)
- [SPPATHENTRY](#)
- [SPPHRASE](#)
- [SPPHRASEALT](#)
- [SPPHRASEALTREQUEST](#)
- [SPPHRASEELEMENT](#)
- [SPPHRASEPROPERTY](#)
- [SPPHRASEREPLACEMENT](#)
- [SPPHRASERULE](#)
- [SPPROPERTYINFO](#)
- [SPRECOCONTEXTSTATUS](#)
- [SPRECOGNIZERSTATUS](#)
- [SPRECORESULTINFO](#)
- [SPRECORESULTTIMES](#)
- [SPRULEENTRY](#)
- [SPSERIALIZEDEVENT](#)

- [SPSERIALIZEDEVENT64](#)
- [SPSERIALIZEDPHRASE](#)
- [SPSERIALIZEDRESULT](#)
- [SPSTATEINFO](#)
- [SPTTEXTSELECTIONINFO](#)
- [SPTMTHREADINFO](#)
- [SPTRANSITIONENTRY](#)
- [SPTRANSITIONPROPERTY](#)
- [SPVCONTEXT](#)
- [SPVOICESTATUS](#)
- [SPVPITCH](#)
- [SPVSTATE](#)
- [SPVTEXTFRAG](#)
- [SPWORD](#)
- [SPWORDENTRY](#)
- [SPWORDLIST](#)
- [SPWORDPRONUNCIATION](#)
- [SPWORDPRONUNCIATIONLIST](#)
- [WAVEFORMATEX](#)



SPAUDIOBUFFERINFO

SPAUDIOBUFFERINFO contains the audio stream buffer information.

Real-time audio objects (e.g., ISpMMSysAudio or a custom ISpAudio-based audio object) should use a buffer to allow for latency in the audio stream, either during reads or writes. The SPAUDIOBUFFERINFO structure contains elements that define how the buffer should be used at run time (e.g., buffer size, event latency, etc.)

```
typedef struct SPAUDIOBUFFERINFO
{
    ULONG          ulMsMinNotification;
    ULONG          ulMsBufferSize;
    ULONG          ulMsEventBias;
} SPAUDIOBUFFERINFO;
```

Members

ulMsMinNotification

The minimum preferred time, in milliseconds, between the actual time an event notification occurs and the ideal time. More CPU resources are needed when the amount of time is shorter; however, the event notifications are more timely. This value must be greater than zero and no more than one quarter the size of the *ulMsBufferSize*. A reasonable default is 50ms.

ulMsBufferSize

The size of the audio object's buffer, in milliseconds. For readable audio objects, this is simply a preferred size; readable objects will automatically expand their buffers to accommodate data. For writable audio objects, this is the amount of audio data that will be buffered before a call to

Write will block. This value must be greater than or equal to 200 milliseconds.

A reasonable default is 500ms.

ulMsEventBias

The amount of time, in milliseconds, by which event notifications precede the actual occurrence of the events. For example, setting a value of 100 for the event bias would cause all events to be notified 100 milliseconds prior to the audio data being played. This can be useful for applications needing time to animate mouths for TTS voices. This value cannot be larger than *ulMsBufferSize*.

A reasonable default is 0ms; applications should set *ulMsEventBias* based on specific circumstances.



SPAUDIOSTATUS

```
typedef struct SPAUDIOSTATUS
{
    long          cbFreeBuffSpace;
    ULONG        cbNonBlockingIO;
    SPAUDIOSTATE State;
    ULONGLONG    CurSeekPos;
    ULONGLONG    CurDevicePos;
    DWORD        dwReserved1;
    DWORD        dwReserved2;
} SPAUDIOSTATUS;
```

Members

cbFreeBuffSpace

Size, in bytes, of free space for reading and/or writing in the audio object.

cbNonBlockingIO

The amount of data which can be read from or written to a device without blocking.

State

The state (of type [SPAUDIOSTATE](#)) of the audio device.

CurSeekPos

The current seek position, in bytes, within the audio stream. This is the position in the stream where the next read or write will be performed.

CurDevicePos

The current read position, in bytes, of the device. This is the position in the stream where the device is currently reading

or writing. For readable streams, this value will always be greater than or equal to *CurSeekPos*. For writable streams, this value will always be less than or equal to *CurSeekPos*.

dwReserved1

Reserved for future expansion.

dwReserved2

Reserved for future expansion.



SPBINARYGRAMMAR

SPBINARYGRAMMAR contains the grammar size information. A binary grammar is the resulting data after calling either [ISpGrammarBuilder::Commit\(\)](#) or [ISpGrammarCompiler::CompileStream\(\)](#).

```
typedef struct SPBINARYGRAMMAR
{
    ULONG      ulTotalSerializedSize;
} SPBINARYGRAMMAR;
```

Members

ulTotalSerializedSize

Total size, in bytes, of the serialized grammar.



SPEVENT

SPEVENT contains information about an event. Events are passed from the TTS or SR engines or audio devices back to applications.

```
typedef struct SPEVENT
{
    WORD          eEventId;
    WORD          eiParamType;
    ULONG         ulStreamNum;
    ULONGLONG    ullAudioStreamOffset;
    WPARAM        wParam;
    LPARAM        lParam;
} SPEVENT;
```

Members

eEventId

The event ID of type [SPEVENTENUM](#).

eiParamType

The signature of the associated data in the lParam parameter. The user may need to release associated data after using the event. See [SPEVENTLPARAMTYPE](#) for more information about associated data.

ulStreamNum

The stream number associated with the event. For text-to-speech (i.e., output streams), the stream number is incremented each time a new speak call (e.g. [ISpVoice::SpeakStream](#), [ISpVoice::Speak](#)) is made. For speech recognition (i.e., input streams), the stream is incremented each time an audio stream is opened (i.e., [ISpSREngine::RecognizeStream](#)). Note that a single audio

input object can be opened multiple times (e.g., buffer overflow, device error, recognition state change).

ullAudioStreamOffset

The byte offset into the audio stream associated with the event at which the event was fired. For synthesis, the output stream is the synthesized data. For recognition, this indicates the position in the input audio stream.

wParam

The generic word field. For event IDs with the `SPFEI_LPARAM_IS_POINTER` set, this is the size, in bytes, for the data pointed to by *lParam*. In some cases, the type of event will change the function of this parameter. See [SPEVENTENUM](#) for information about specific events. See the helper [SpClearEvent](#) for more information about releasing objects or memory attached to an event.

lParam

The generic event field. For event IDs with the `SPFEI_LPARAM_IS_POINTER` set, this points to the data allocated by *CoTaskMemAlloc*. The caller is responsible for freeing this memory using *CoTaskMemFree*(). In some cases, the type of event will change the function of this parameter. See [SPEVENTENUM](#) for information about specific events. See the helper [SpClearEvent](#) for more information about releasing objects or memory attached to an event.



SPEVENTSOURCEINFO

SPEVENTSOURCEINFO is used by [ISpEventSource::GetInfo](#) to pass back information about the event source.

Event sources contain a queue, which hold events until a caller retrieves the events using `::GetEvents`.

```
typedef struct SPEVENTSOURCEINFO
{
    ULONGLONG    ullEventInterest;
    ULONGLONG    ullQueuedInterest;
    ULONG        ulCount;
} SPEVENTSOURCEINFO;
```

Members

ullEventInterest

Set of event Id flags of type [SPEVENTENUM](#) defining which events should trigger a notification (e.g. callback, signaled event, window message, etc.).

ullQueuedInterest

Set of event Id flags of type [SPEVENTENUM](#) defining which events should be stored in the source's event queue until the caller uses [ISpEventSource::GetEvents](#) to remove them.

ulCount

Number of events currently waiting in the event queue.

Remarks

Note that event interest (*ullEventInterest*) only specifies which events should cause a notification. The queued interest (*ullQueuedInterest*) specifies which events should be stored in the event queue.

For example, due to graphics performance issues, it might not be optimal for an application to redraw every viseme that occurs. Instead, it would draw the current viseme whenever a viseme event occurs. Instead of storing viseme events in the event queue, the application would set only the event interest to include `SPEI_VISEME` and not the queued interest. Whenever the TTS engine fires a viseme event, the application would receive a notification, and would then check the current viseme by calling [ISpVoice::GetStatus](#) to check the `Visemeld`.

The queued interest (*ullQueuedInterest*) always includes the event interest (*ullEventInterest*) to ensure that a notification is sent whenever an event is queued.

See Also

Helper macro [SPFEI](#) for combining [SPEVENTENUM](#) flags.



SPPARSEINFO

SPPARSEINFO is filled in by the speech recognition (SR) engine and sent to SAPI. SAPI uses the information to build a phrase object that can be sent in a hypothesis or recognition (see [ISpSREngineSite::ParseFromTransitions](#)).

```
typedef struct SPPARSEINFO
{
    ULONG          cbSize;
    SPRULEHANDLE   hRule;
    ULONGLONG      ullAudioStreamPosition;
    ULONG          ulAudioSize;
    ULONG          cTransitions;
    SPPATHENTRY   *pPath;
    BOOL           fHypothesis;
    GUID           SREngineID;
    ULONG          ulSREnginePrivateDataSize;
    const BYTE     *pSREnginePrivateData;
} SPPARSEINFO;
```

Members

cbSize

The size of the SPPARSEINFO structure, as set by the SR engine.

hRule

The handle of the top-level rule for the recognition or hypothesis.

ullAudioStreamPosition

The position in the stream where the recognition started. If downsampling an audio stream, *ullAudioStreamPosition* will be the byte position within the original stream.

ulAudioSize

The size in bytes of the portion of the stream that is recognized.

cTransitions

The number of word transitions in the *pPath* array.

pPath

The transition path through the CFG that is recognized.

fHypothesis

TRUE if a hypothesis, false if a final recognition.

SREngineID

The unique identifier of the SR engine. Can be NULL.

ulSREnginePrivateDataSize

The size of private SR engine data that will be sent to the SR engine alternates analyzer. Can be zero.

pSREnginePrivateData

A pointer the private SR engine data (memory) that will be sent to the SR engine alternates analyzer. Can be NULL. SAPI will handle the serialization and marshaling of the data for the SR engine.



SPPATHENTRY

SPPATHENTRY array is passed by the SR engine to [ISpSREngineSite::ParseFromTransitions](#) as part of [SPPARSEINFO](#)

```
typedef struct SPPATHENTRY
{
    union
    {
        SPTRANSITIONID      hTransition;
        SPPHRASEELEMENT    elem;
    };
} SPPATHENTRY;
```

Members

hTransition

Handle of the transition that was recognized at this point.

elem

Element information--can be left blank. In this case, SAPI will fill in the information from the grammar.



SPPHRASE

SPPHRASE

contains information about speech recognition information, including hypotheses, false recognitions, recognitions, and alternate recognitions. The information in the phrase includes, language, audio and event timing, text (display and lexicon), inverse text replacements, semantic tags (i.e., properties), and depending on the engine, an optional block of engine-specific phrase data.

SAPI typically provides the application with a pointer to a block of memory that has been allocated by *CoTaskMemAlloc*, which the application must free using *CoTaskMemFree* when it is finished with the phrase information.

```
typedef struct SPPHRASE
{
    ULONG                cbSize;
    LANGID               LangID;
    WORD                 wReserved;
    ULONGLONG            ullGrammarID;
    ULONGLONG            ftStartTime;
    ULONGLONG            ullAudioStreamPosition;
    ULONG                ulAudioSizeBytes;
    ULONG                ulRetainedSizeBytes;
    ULONG                ulAudioSizeTime;
    SPPHRASERULE         Rule;
    const SPPHRASEPROPERTY *pProperties;
    const SPPHRASEELEMENT *pElements;
    ULONG                cReplacements;
    const SPPHRASEREPLACEMENT *pReplacements;
    GUID                 SREngineID;
    ULONG                ulSREnginePrivateDataSize;
    const BYTE           *pSREnginePrivateData;
} SPPHRASE;
```

Members

cbSize

The size of this structure in bytes.

LangID

The language ID of the phrase elements.

wReserved

Reserved for future use.

ullGrammarID

ID of the grammar that contains the top-level rule used to recognize this phrase.

ftStartTime

Absolute time for start of phrase audio as a 64-bit value based on the Win32 APIs, *SystemTimeToFileTime* and *GetSystemTime*. When an application uses wav file input, SAPI sets the stream position and start time information to zero.

ullAudioStreamPosition

The starting offset of the phrase in bytes relative to the start of the audio stream. If downsampling an audio stream, *ullAudioStreamPosition* will be the byte position within the original stream.

ulAudioSizeBytes

Size of audio data, in bytes, for this phrase.

ulRetainedSizeBytes

Size, in bytes, of the retained audio data (in the user-specified retained-audio format).

See also [ISpRecoContext::SetAudioOptions](#) for more information about specifying the retained audio format

ulAudioSizeTime

Length of phrase audio in 100-nanosecond units.

Rule

Information about the top-level rule (and rule-reference hierarchy) used to recognize this phrase.

pProperties

Pointer to the root of the semantic-tag property tree.

pElements

Pointer to the array of phrase elements (the number of elements is contained in Rule). Each phrase element includes position and text information, including lexical and display forms.

cReplacements

Number of text replacements. Text replacements are generally based on engine-defined Inverse Text Normalization rules (e.g. recognize "five dollars" as "\$5").

pReplacements

Pointer to the array of text replacements.

SREngineID

GUID that identifies the particular speech recognition (SR) engine that recognized this phrase.

ulSREnginePrivateDataSize

Size of the engine's private data, in bytes.

pSREnginePrivateData

Pointer to the engine's private data.

Engine private data is specific to each SR engine, and the format and structure of the data is not defined by SAPI.



SPPHRASEALT

SPPHRASEALT is used by SAPI and the speech recognition (SR) engine's alternate analyzer to exchange alternate information.

When a speech recognition application requests phrase alternates from the SR engine (see [ISpRecoResult::GetAlternates](#)), SAPI passes an empty SPPHRASEALT structure to the alternate analyzer (see [ISpSRAlternates::GetAlternates](#)). If it can generate alternate phrases based on the application's request (see [SPPHRASEALTREQUEST](#)), the alternate analyzer must fill in the SPPHRASEALT structure and return it to SAPI. SAPI will return the alternate phrases to the application as an array of pointers to [ISpPhraseAlt](#) interfaces.

If the application selects an alternate as the preferred phrase (see [ISpPhraseAlt::Commit](#)), then SAPI will pass the respective SPPHRASEALT structure back to the alternate analyzer to update the language or acoustic models (see [ISpSRAlternates::Commit](#)).

```
typedef struct tagSPPHRASEALT
{
    ISpPhraseBuilder    *pPhrase;
    ULONG                ulStartElementInParent;
    ULONG                cElementsInParent;
    ULONG                cElementsInAlternate;
    void                *pvAltExtra;
    ULONG                cbAltExtra;
} SPPHRASEALT;
```

Members

pPhrase

The alternate phrase.

ulStartElementInParent

The index of the starting element for the change in the original phrase.

cElementsInParent

The number of elements in the original recognition.

cElementsInAlternate

The number of elements that are changed in the alternate.

pvAltExtra

A pointer to the private SR engine alternate analyzer data.
Can be NULL if no data is provided.

cbAltExtra

The size of the private SR engine alternate analyzer data.
Can be zero if no data is provided.



SPPHRASEALTREQUEST

SPPHRASEALTREQUEST contains information relevant to an application calling [ISpRecoResult::GetAlternates](#) (e.g., number of alternates requested, original phrase information, private engine data, etc.).

```
typedef struct tagSPPHRASEALTREQUEST
{
    ULONG          ulStartElement;
    ULONG          cElements;
    ULONG          ulRequestAltCount;
    void           *pvResultExtra;
    ULONG          cbResultExtra;
    ISpPhrase      *pPhrase;
    ISpRecoContext *pRecoContext;
} SPPHRASEALTREQUEST;
```

Members

ulStartElement

Based on the original phrase, the starting element of the span from which to retrieve alternates.

cElements

Based on the original phrase, the number of elements in the span from which to retrieve alternates.

ulRequestAltCount

The maximum number of alternates that an application requests.

pvResultExtra

Pointer to the private SR engine data as sent from the SR engine. Can be NULL if no data was supplied.

cbResultExtra

The size of the private SR engine data as sent from the SR engine. Can be zero if no data was supplied.

pPhrase

The original recognition as sent from the SR engine.

pRecoContext

Pointer to an [ISpRecoContext](#) interface that allows the alternate analyzer to communicate with itself and the SR engine. The alternate analyzer can use `IUnknown::QueryInterface` to query the context for the private SR engine extension (see [ISpPrivateEngineCall](#) for more information about private engine extensions).



SPPHRASEELEMENT

SPPHRASEELEMENT contains the information for a spoken word.

```
typedef struct SPPHRASEELEMENT
{
    ULONG          ulAudioTimeOffset;
    ULONG          ulAudioSizeTime;
    ULONG          ulAudioStreamOffset;
    ULONG          ulAudioSizeBytes;
    ULONG          ulRetainedStreamOffset;
    ULONG          ulRetainedSizeBytes;
    const WCHAR    *pszDisplayText;
    const WCHAR    *pszLexicalForm;
    const SPPHONEID *pszPronunciation;
    BYTE           bDisplayAttributes;
    char           RequiredConfidence;
    char           ActualConfidence;
    BYTE           Reserved;
    float          SREngineConfidence;
} SPPHRASEELEMENT;
```

Members

ulAudioTimeOffset

This is the starting offset of the element in 100-nanosecond units of time relative to the start of the phrase.

ulAudioSizeTime

This is the length of the element in 100-nanosecond units of time.

ulAudioStreamOffset

This is the starting offset of the element in bytes relative to the start of the phrase in the original input stream.

ulAudioSizeBytes

This is the size of the element in bytes in the original input stream.

ulRetainedStreamOffset

This is the starting offset of the element in bytes relative to the start of the phrase in the retained audio stream

ulRetainedSizeBytes

This is the size of the element in bytes in the retained audio stream.

pszDisplayText

The display text for this element (e.g., ",").

pszLexicalForm

The lexical form of this element (e.g., "comma" for ",").

pszPronunciation

The pronunciation for this element as a null-terminated array of [SPPHONEID](#).

bDisplayAttributes

A bit field of [SPDISPLAYATTRIBUTES](#) defining extra display information which the application should honor when displaying this word.

RequiredConfidence

The required confidence for this element (either SP_LOW_CONFIDENCE, SP_NORMAL_CONFIDENCE, or SP_HIGH_CONFIDENCE). If a word is prefixed with a '-' (minus), the RequiredConfidence is SP_LOW_CONFIDENCE,

and '+' (plus) will set this field to SP_HIGH_CONFIDENCE (e.g., "This -is -a +test"). See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details.

ActualConfidence

The actual confidence for this element (either SP_LOW_CONFIDENCE, SP_NORMAL_CONFIDENCE, or SP_HIGH_CONFIDENCE). This is always at least the RequiredConfidence. See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details.

Reserved

Reserved for future use.

SREngineConfidence

The confidence score computed by the SR engine. The value range is engine dependent. It can be used to optimize an application's performance with a specific engine. Using this value will improve the application with a particular speech engine but more than likely will make it worse with other engines and should be used with care. This value is more useful with speaker-independent engines because it allows a large corpus of recorded usage to correctly optimize the overall accuracy of the application. See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details.



SPPHRASEPROPERTY

SPPHRASEPROPERTY stores the information for one semantic property. It can be used to construct a semantic property tree.

See also [Designing Grammar Rules](#) for more information about semantic properties.

```
struct SPPHRASEPROPERTY
{
    const    WCHAR                *pszName;
    ULONG    ulId;
    const    WCHAR                *pszValue;
    VARIANT  vValue;
    ULONG    ulFirstElement;
    ULONG    ulCountOfElements;
    const    SPPHRASEPROPERTY     *pNextSibling;
    const    SPPHRASEPROPERTY     *pFirstChild;
    float    SREngineConfidence;
    signed   char                  Confidence;
};
```

Members

pszName

Name of the null-terminated string of the semantic property (in the Speech Text Grammar Format set using the PROPNAME attribute).

ulId

ID of the semantic property (in the Speech Text Grammar Format set using the PROPID attribute).

pszValue

Null-terminated string value of the semantic property (in the Speech Text Grammar Format set using the VALSTR

attribute).

vValue

VARIANT value of a semantic property. The type has to be one of the following: VT_BOOL, VT_I4, VT_R4, VT_R8, or VT_BYREF (only for dynamic grammars). This is set using the VAL attribute in the Speech Text Grammar Format.

ulFirstElement

The first spoken element spanned by this property.

ulCountOfElements

The number of spoken elements spanned by this property.

pNextSibling

Pointer to next sibling in property tree.

pFirstChild

Pointer to the first child of this semantic property.

SREngineConfidence

Confidence value for this semantic property computed by the SR engine. The value range is specific to each SR engine. See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details.

Confidence

Confidence value for this semantic property computed by SAPI. The value is either SP_LOW_CONFIDENCE, SP_NORMAL_CONFIDENCE, or SP_HIGH_CONFIDENCE. See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details.



SPPHRASEREPLACEMENT

SPPHRASEREPLACEMENT replaces the display text of one or more of the spoken words. This is used by speech recognition engines to perform Inverse Text Normalization (ITN). For example the spoken words "twenty" and "three" are replaced by the replacement text "23."

```
typedef struct tagSPPHRASEREPLACEMENT
{
    BYTE                bDisplayAttributes;
    const WCHAR        *pszReplacementText;
    ULONG              ulFirstElement;
    ULONG              ulCountOfElements;
} SPPHRASEREPLACEMENT;
```

Members

bDisplayAttributes

One or more [SPDISPLAYATTRIBUTES](#) for the replacement text.

pszReplacementText

Text for the replacement.

ulFirstElement

Offset of the first spoken element to be replaced.

ulCountOfElements

Number of spoken elements to replace.



SPPHRASERULE

SPPHRASERULE contains the information for a rule in a grammar result. SAPI uses the *pFirstChild* and *pNextSibling* pointers to represent the parse tree. SPPHRASE.Rule is the root node of the parse tree.

```
struct tagSPPHRASERULE
{
    const    WCHAR          *pszName;
    ULONG
    ULONG          ulId;
    ULONG          ulFirstElement;
    ULONG          ulCountOfElements;
    const    SPPHRASERULE *pNextSibling;
    const    SPPHRASERULE *pFirstChild;
    float          SREngineConfidence;
    signed char    Confidence;
};
```

Members

pszName

Name of this rule (in Speech Text Grammar Format set using <RULE NAME="MyName">).

ulId

ID of this rule (set using <RULE ID="123">).

ulFirstElement

The index of the first spoken element (word) of this rule.

ulCountOfElements

Number of spoken elements (words) spanned by this rule.

pNextSibling

Pointer to the next sibling in the parse tree.

pFirstChild

Pointer to the first child node in the parse tree.

SREngineConfidence

Confidence for this rule computed by the SR engine. The value is engine dependent and not standardized across multiple SR engines. See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details.

Confidence

Confidence for this rule computed by SAPI. The value is either SP_LOW_CONFIDENCE, SP_NORMAL_CONFIDENCE, or SP_HIGH_CONFIDENCE. See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details.



SPPROPERTYINFO

SPPROPERTYINFO contains the information for a semantic property.

```
typedef struct tagSPPROPERTYINFO
{
    const    WCHAR    *pszName;
    ULONG    ulId;
    const    WCHAR    *pszValue;
    VARIANT  vValue;
} SPPROPERTYINFO;
```

Members

pszName

Pointer to the null-terminated string that contains the name information of the property. This is set using the PROPNAME attribute in the Speech Text Grammar Format.

ulId

Identifier associated with the property. This is set using the PROPID attribute in the Speech Text Grammar Format.

pszValue

Pointer to the null-terminated string that contains the value information of the property. This is set using the VALSTR attribute in the Speech Text Grammar Format.

vValue

Must be one of the following: VT_BOOL, VT_I4, VT_R4, VT_R8, or VT_BYREF (for dynamic grammars only.) This is set using the VAL attribute in the Speech Text Grammar Format.



SPRECOCONTEXTSTATUS

```
typedef struct SPRECOCONTEXTSTATUS
{
    SPINTERFERENCE    eInterference;
    WCHAR             szRequestTypeOfUI[255];
    DWORD             dwReserved1;
    DWORD             dwReserved2;
} SPRECOCONTEXTSTATUS;
```

Members

eInterference

One of the interference types contained in the [SPINTERFERENCE](#) enumeration. An application can check this value for the input audio signal quality.

szRequestTypeOfUI[255]

Specifies the type of UI requested. If the first byte is NULL, no UI is requested. See [ISpRecognizer::DisplayUI](#) and [SPDUI_*](#) for a list of the SAPI-defined UI types.

dwReserved1

Reserved for future expansion.

dwReserved2

Reserved for future expansion.



SPRECOGNIZERSTATUS

```
typedef struct SPRECOGNIZERSTATUS
{
    SPAUDIOSTATUS    AudioStatus;
    ULONGLONG        ullRecognitionStreamPos;
    ULONG            ulStreamNumber;
    ULONG            ulNumActive;
    CLSID            clsidEngine;
    ULONG            cLangIDs;
    LANGID           aLangID[SP_MAX_LANGIDS];
    DWORD            dwReserved1;
    DWORD            dwReserved2;
} SPRECOGNIZERSTATUS;
```

Members

AudioStatus

The [SPAUDIOSTATUS](#) structure containing the current audio device information.

ullRecognitionStreamPos

The current stream position the engine has recognized to. Stream positions are measured in bytes. This value can be used to check the engine's progress using the audio data.

ulStreamNumber

This value is incremented every time SAPI starts or stops recognition on an engine (see [SPEI_START_SR_STREAM](#) and [SPEI_END_SR_STREAM](#)). Each time this happens the ullRecognitionStreamPos gets reset to zero. [Events](#) fired from the engine have equivalent stream number and position information also.

ulNumActive

The current engine's number of active rules.

clsidEngine

The unique identifier associated with the current engine.

cLangIDs

The number of languages that the current engine supports.

aLangID

Array containing the languages that the current engine supports.

dwReserved1

Reserved for future expansion.

dwReserved2

Reserved for future expansion.



SPRECORESULTINFO

SPRECORESULTINFO is the result structure passed from the engine to SAPI.

```
typedef struct SPRECORESULTINFO
{
    ULONG                cbSize;
    SPRESULTTYPE        eResultType;
    BOOL                 fHypothesis;
    BOOL                 fProprietaryAutoPause;
    ULONGLONG           ullStreamPosStart;
    ULONGLONG           ullStreamPosEnd;
    SPGRAMMARHANDLE     hGrammar;
    ULONG               ulSizeEngineData;
    void                *pvEngineData;
    ISpPhraseBuilder    *pPhrase;
    SPPHRASEALT         *aPhraseAlts;
    ULONG               ulNumAlts;
} SPRECORESULTINFO;
```

Members

cbSize

Total size, in bytes, of this structure.

eResultType

Type of result object (CFG, SLM, or Proprietary).

For example, the result type can be SPRT_SLM |

SPRT_FALSE_RECOGNITION if the speech recognition engine fails to recognize a dictation phrase.

fHypothesis

If TRUE, this recognition is a hypothesis.

fProprietaryAutoPause

This field is only used for SPRT_PROPRIETARY grammars. If TRUE, the recognition will pause.

ullStreamPosStart

Starting position within the input stream. If downsampling an audio stream, *ullStreamPosStart* will be the byte position within the original stream.

ullStreamPosEnd

Ending position within the input stream. If downsampling an audio stream, *ullStreamPosEnd* will be the byte position within the original stream.

hGrammar

Required for SPRT_SLM and SPRT_PROPRIETARY, otherwise this value is NULL.

ulSizeEngineData

Specifies the size of *pvEngineData*.

pvEngineData

Pointer to the engine data.

pPhrase

Pointer to phrase object.

aPhraseAlts

An array containing the alternate phrases.

ulNumAlts

The number of alternate phrases contained in *aPhraseAlts*.



SPRECORESULTTIMES

SPRECORESULTTIMES contains the time information for speech recognition. This data structure is used by the [ISpRecoResult::GetResultTimes](#) method.

```
typedef struct SPRECORESULTTIMES
{
    FILETIME      ftStreamTime;
    ULONGLONG     ullLength;
    DWORD         dwTickCount;
    ULONGLONG     ullStart;
} SPRECORESULTTIMES;
```

Members

ftStreamTime

Absolute time for start of phrase audio as a 64-bit value based on the Win32 APIs, *SystemTimeToFileTime* and *GetSystemTime*. When an application uses wav file input, SAPI sets the stream position and start time information to zero.

ullLength

Value containing the length of the phrase specified in 100 nanosecond units.

dwTickCount

Number of milliseconds elapsed from the start of the system to the start of the current result. This variable is set to zero if the wave file input is used.

ullStart

Value containing the total 100 nanosecond units from the

start of the stream to the start of the phrase.



SPRULEENTRY

SPRULEENTRY contains information about a rule.

```
typedef struct SPRULEENTRY
{
    SPRULEHANDLE    hRule;
    SPSTATEHANDLE  hInitialState;
    DWORD          Attributes;
    void           *pvClientContext;
    void           *pvClientGrammarContext;
} SPRULEENTRY;
```

Members

hRule

Handle to the rule.

hInitialState

Handle to the rule's initial state.

Attributes

The rule's attribute.

pvClientContext

The client context pointer passed using
[ISpSREngineSite::SetRuleClientContext](#)

pvClientGrammarContext

The client's grammar context pointer.



SPSERIALIZEDEVENT

SPSERIALIZEDEVENT serializes a SAPI event.

See [SPSERIALIZEDEVENT64](#) for a 64-bit version of the serialized event structure.

See also [SPEVENT](#) for further information on the related event variable fields.

```
typedef struct SPSEIALIZEDEVENT
{
    WORD        eEventId;
    WORD        eIParamType;
    ULONG       ulStreamNum;
    ULONGLONG   ullAudioStreamOffset;
    ULONG       SerializedwParam;
    LONG        SerializedIParam;
} SPSEIALIZEDEVENT;
```

Members

eEventId

The event ID of type [SPEVENTENUM](#).

eIParamType

The signature of the *IParam* parameter of type [SPEVENTLPARAMTYPE](#).

ulStreamNum

The stream number associated with the event.

For text-to-speech (i.e., output streams), the stream number is incremented each time a new speak call (e.g., [ISpVoice::SpeakStream](#), [ISpVoice::Speak](#)) is made.

For speech recognition (i.e., input streams), the stream is incremented each time an audio stream is opened (i.e.,

[ISpSREngine::RecognizeStream](#)). Note that a single audio input object can be opened multiple times (e.g., buffer overflow, device error, recognition state change).

ullAudioStreamOffset

The byte offset into the audio stream associated with the event at which the event was fired. For synthesis, the output stream is the synthesized data. For recognition, this indicates the position in the input audio stream.

SerializedwParam

The *wParam* value that was included in the event. See [SPEVENTENUM](#) for further information on possible *wParam* values.

SerializedIParam

The *IParam* value that was included in the event. See [SPEVENTENUM](#) for further information on possible *IParam* values.



SPSERIALIZEDEVENT64

SPSERIALIZEDEVENT64 serializes a SAPI event.

```
typedef struct SPSEIALIZEDEVENT64
{
    WORD            eEventId;
    WORD            eiParamType;
    ULONG           ulStreamNum;
    ULONGLONG       ullAudioStreamOffset;
    ULONGLONG       SerializediParam;
    LONGLONG        SerializediParam;
} SPSEIALIZEDEVENT64;
```

Members

eEventId

The event ID of type [SPEVENTENUM](#).

eiParamType

The signature of the *iParam* parameter of type [SPEVENTLPARAMTYPE](#).

ulStreamNum

The stream number associated with the event.

For text-to-speech (i.e., output streams), the stream number is incremented each time a new speak call (e.g., [ISpVoice::SpeakStream](#), [ISpVoice::Speak](#)) is made.

For speech recognition (i.e., input streams), the stream is incremented each time an audio stream is opened (i.e., [ISpSREngine::RecognizeStream](#)). Note that a single audio input object can be opened multiple times (e.g., buffer overflow, device error, recognition state change).

ullAudioStreamOffset

The byte offset into the audio stream associated with the event at which the event was fired. For synthesis, the output stream is the synthesized data. For recognition, this indicates the position in the input audio stream.

SerializedwParam

The *wParam* value that was included in the event. The variable uses a 64-bit data type for large addresses and values. See [SPEVENTENUM](#) for further information on possible *wParam* values.

SerializediParam

The *IParam* value that was included in the event. The variable uses a 64-bit data type for large addresses and values. See [SPEVENTENUM](#) for further information on possible *IParam* values.

Remarks

The 64 label signifies the use of 64-bit values for the *wParam* and *IParam* variables. Normally applications will not need to use 64-bit values, unless large pointers or sizes were stored in the original event structure. See also [SPSERIALIZEDEVENT](#) for the non-64-bit version.

See also [SPEVENT](#) for further information on the related event variable fields.



SPSERIALIZEDPHRASE

```
typedef struct tagSPSERIALIZEDPHRASE
{
    ULONG      ulSerializedSize;
} SPSERIALIZEDPHRASE;
```

Members

ulSerializedSize

Value specifying the size of the serialized phrase in bytes.



SPSERIALIZEDRESULT

SPSERIALIZEDRESULT contains the phrase size information.

```
typedef struct SPSEIALIZEDRESULT
{
    ULONG    ulSerializedSize;
} SPSEIALIZEDRESULT;
```

Members

ulSerializedSize

The size of the entire phrase in bytes, including this ULONG.



SPSTATEINFO

SPSTATEINFO contains information about the transitions from a single state in a context-free grammar.

SAPI provides information when the speech recognition engine requests grammar state information (see [ISpSREngineSite::GetStateInfo](#)).

```
typedef struct SPSTATEINFO
{
    ULONG                cAllocatedEntries;
    SPTRANSITIONENTRY *pTransitions;
    ULONG                cEpsilons;
    ULONG                cRules;
    ULONG                cWords;
    ULONG                cSpecialTransitions;
} SPSTATEINFO;
```

Members

cAllocatedEntries

Total number of entries in *pTransitions*

pTransitions

Pointer to a [SPTRANSITIONENTRY](#) structure.

cEpsilons

Number of SPTRANSEPSILON transitions in *pTransitions*.

cRules

Number of SPTRANSRULE transitions in *pTransitions*.

cWords

Number of SPTRANSWORD transitions in *pTransitions*.

cSpecialTransitions

Number of special transitions (SPTRANSTEXTBUF, SPTRANSWILDCARD, or SPTRANSDICTATION) in *pTransitions*.



SPTTEXTSELECTIONINFO

SPTTEXTSELECTIONINFO combines text selection information that the SR engine can use to improve the recognition accuracy (using [ISpRecoGrammar::SetWordSequenceData\(\)](#) and [ISpRecoGrammar::SetTextSelection\(\)](#)).

```
typedef struct tagSPTTEXTSELECTIONINFO
{
    ULONG        ulStartActiveOffset;
    ULONG        cchActiveChars;
    ULONG        ulStartSelection;
    ULONG        cchSelection;
} SPTTEXTSELECTIONINFO;
```

Members

ulStartActiveOffset

Count of characters from the start of the WordSequenceData buffer. The word containing the character pointed to is the first word of the active text selection buffer.

cchActiveChars

Count of characters for the active range of the text selection buffer.

ulStartSelection

Start of the selected text (e.g., the user is selecting part of the previously dictated text that he/she is going to edit or correct).

cchSelection

Count of characters of the user selection.



SPTMTHREADINFO

SPTMTHREADINFO contains thread management information implemented by the [ISpTaskManager](#) interface.

```
typedef struct SPTMTHREADINFO
{
    long    lPoolSize;
    long    lPriority;
    ULONG   ulConcurrencyLimit;
    ULONG   ulMaxQuickAllocThreads;
} SPTMTHREADINFO;
```

Members

lPoolSize

Number of threads in pool (-1 default).

lPriority

Priority of threads in pool.

ulConcurrencyLimit

Number of threads allowed to concurrently execute (0 default).

ulMaxQuickAllocThreads

Maximum number of dedicated threads retained.



SPTRANSITIONENTRY

```
typedef struct SPTRANSITIONENTRY
{
    SPTRANSITIONID          ID;
    SPSTATEHANDLE           hNextState;
    BYTE                    Type;           // SPTRANSITIONTYPE
    char                    RequiredConfidence;
    struct
    {
        DWORD              fHasProperty;
    };
    float                   Weight;
    union
    {
        struct
        {
            SPSTATEHANDLE  hRuleInitialState; // Only if T
            SPRULEHANDLE    hRule;
            void            *pvClientRuleContext;
        };
        struct
        {
            SPWORDHANDLE    hWord;           // Only if T
            void            *pvClientWordContext;
        };
        struct
        {
            void            *pvGrammarCookie; // Only if T
        };
    };
} SPTRANSITIONENTRY;
```

Members

ID

ID of this transition.

hNextState

Handle to the end state of this transition

Type

Type of this transition.

RequiredConfidence

Required confidence for this transition.

fHasProperty

Flag to indicate if this transition has a semantic property associated with it.

Weight

The relative weight of this transition (relative to other transitions out of the same state).

hRuleInitialState

If this is an SPRULETRANS, it points to the rule's initial state.

hRule

If this is an SPRULETRANS, it contains the rule's handle.

pvClientRuleContext

Client context set using

[ISpSREngineSite::SetRuleContext.htm](#)

hWord

If this is an SPWORDTRANS, it contains the word handle.

pvClientWordContext

Client context set using
[ISpSREngineSite::SetWordClientContext.htm](#)

pvGrammarCookie

Grammar cookie needed to associate a text buffer with this transition.



SPTRANSITIONPROPERTY

SPTRANSITIONPROPERTY contains transition property information.

```
typedef struct SPTRANSITIONPROPERTY
{
    const WCHAR    *pszName;
    ULONG          ulId;
    const WCHAR    *pszValue;
    VARIANT        vValue;
} SPTRANSITIONPROPERTY;
```

Members

pszName

Address of a null-terminated string containing the name information.

ulId

Identifier associated with the transition property.

pszValue

Address of a null-terminated string containing the value information.

vValue

For dynamic grammars this value will be VT_BOOL, VT_I4, VT_R4, VT_R8, or VT_BYREF.



SPVCONTEXT

SPVCONTEXT contains strings passed into an [ISpVoice::Speak](#) call using a <Context> XML tag. See the [XML Schema : SAPI](#) white paper for more details.

```
typedef struct SPVCONTEXT
{
    LPCWSTR    pCategory;
    LPCWSTR    pBefore;
    LPCWSTR    pAfter;
} SPVCONTEXT;
```

Members

pCategory

String passed in with the ID attribute.

pBefore

String passed in with the Before attribute.

pAfter

String passed in with the After attribute.



SPVOICESTATUS

SPVOICESTATUS contains voice status information. This structure is returned by [ISpVoice::GetStatus](#).

```
typedef struct SPVOICESTATUS
{
    ULONG        ulCurrentStream;
    ULONG        ulLastStreamQueued;
    HRESULT      hrLastResult;
    DWORD        dwRunningState;
    ULONG        ulInputWordPos;
    ULONG        ulInputWordLen;
    ULONG        ulInputSentPos;
    ULONG        ulInputSentLen;
    LONG         lBookmarkId;
    SPPHONEID    PhonemeId;
    SPVISEMES    VisemeId;
    DWORD        dwReserved1;
    DWORD        dwReserved2;
} SPVOICESTATUS;
```

Members

ulCurrentStream

The number of the current stream being synthesized or receiving output (see [ISpVoice::Speak](#) for more information on stream numbers).

ulLastStreamQueued

The number of the last stream queued.

hrLastResult

Result of the last [Speak](#) or [Speakstream](#) call.

dwRunningState

Indicates the status of the voice.. That is, whether it is currently speaking, is done with all pending speak requests, or is currently waiting to speak. The possible flag values are contained in the [SPRUNSTATE](#) enumeration. A value of zero indicates that the voice is currently waiting to speak.

ullInputWordPos

Character position within the input text of the word currently being processed.

ullInputWordLen

Length of the word currently being processed.

ullInputSentPos

Character position within the input text of the sentence currently being processed.

ullInputSentLen

Length of the sentence currently being processed.

lBookmarkId

Current bookmark string (in base 10) converted to a long integer. If the string of the current bookmark does not begin with an integer, *lBookmarkId* will be zero. For example, if the bookmark name is "123Bookmark", the *lBookmarkId* is "123"; and if the bookmark name is "hello", the *lBookmarkId* is zero.

Phonemeld

Current phoneme ID - see [SAPI Phoneme set](#)

Visemeld

Current viseme ID - see [SAPI Viseme set](#)

dwReserved1

Reserved for future expansion.

dwReserved2

Reserved for future expansion.



SPVPITCH

SPVPITCH contains a long value passed into an [ISpVoice::Speak](#) call using a <Pitch> XML tag. See the [XML Schema : SAPI](#) white paper for more details.

```
typedef struct SPVPITCH
{
    long    MiddleAdj;
    long    RangeAdj;
} SPVPITCH;
```

MiddleAdj

Value passed in with the Middle or AbsMiddle attributes. Supported values are -10 to 10 with a value of zero being the default pitch. Values outside of this range may be truncated.

RangeAdj

Reserved for future use.



SPVSTATE

SPVSTATE is a member of [SPVTEXTFRAG](#), and contains information about the XML state associated with a text string. A linked list of SPVTEXTFRAGs is passed into every [ISpTTSEngine::Speak](#).

```
typedef struct SPVSTATE
{
    //--- Action
    SPVACTIONS      eAction;
    //--- Running state values
    LANGID           LangID;
    WORD             wReserved;
    long             EmphAdj;
    long             RateAdj;
    ULONG           Volume;
    SPVPITCH         PitchAdj;
    ULONG           SilenceMSecs;
    SPPHONEID*      pPhoneIds;
    SPPARTOFSPEECH ePartOfSpeech;
    SPVCONTEXT      Context;
} SPVSTATE;
```

Members

eAction

Describes the action to be performed with the associated text fragment. The normal action is to Speak (SPVA_Speak) the fragment.

LangID

The language ID associated with this text. Set using the <Lang> XML tag.

wReserved

Reserved for future use.

EmphAdj

Specifies whether the text should be emphasized - zero indicates no emphasis, one indicates emphasis. Set using the <Emph> XML tag.

RateAdj

The rate associated with this text. Set using the <Rate> XML tag. This value should be combined with the baseline rate (either the default, or a value set by [ISpVoice::SetRate](#)) to yield the final rate value.

Volume

The volume associated with this text. Set using the <Volume> XML tag. This value should be combined with the baseline volume (either the default, or a value set by [ISpVoice::SetVolume](#)) to yield the final volume value.

PitchAdj

The pitch associated with this text. Set using the <Pitch> XML tag.

SilenceMSecs

The length of a silence, in milliseconds, to be inserted into the audio output. *SilenceMSecs* is always zero unless *eAction* is SPVA_Silence. Set using the <Silence> XML tag.

pPhonelds

A pronunciation (possibly associated with text) to be inserted into the audio output. This value is a pointer to a null-terminated array of SPPHONEIDs. Set using the <Pron> XML tag.

ePartOfSpeech

An SPPARTOFSPEECH value to be associated with this text.
Set using the <PartOfSp> XML tag.

Context

An SPVCONTEXT to be associated with this text. Set using
the <Context> XML tag.



SPVTEXTFRAG

SPVTEXTFRAG is the structure through which input text is passed to the TTS engine. The text passed to an [ISpVoice::Speak](#) call will normally be parsed for SAPI 5 XML, and the resulting list of SPVTEXTFRAGs will be passed to an [ISpTTSEngine::Speak](#) call.

```
typedef struct SPVTEXTFRAG
{
    struct SPVTEXTFRAG    *pNext;
    SPVSTATE                State;
    LPCWSTR                pTextStart;
    ULONG                  ulTextLen;
    ULONG                  ulTextSrcOffset;
} SPVTEXTFRAG;
```

Members

pNext

Pointer to the next text fragment in list. A NULL value indicates the end of the list.

State

The current XML attribute state.

pTextStart

Pointer to the beginning of the text string associated with this fragment.

ulTextLen

The length, in characters, of the text string associated with this fragment.

ulTextSrcOffset

Original character offset of *pTextStart* within the text string passed to `ISpVoice::Speak`.



SPWORD

SPWORD defines changes to the words in a lexicon and is used with `IspLexicon`. It is used in connection with [SPWORDLIST](#) which in turn is used by [IspLexicon::GetWords](#) and [IspLexicon::GetGenerationChange](#).

```
typedef struct SPWORD
{
    struct SPWORD          *pNextWord;
    LANGID                 LangID;
    WORD                   wReserved;
    SPWORDTYPE            eWordType;
    WCHAR                  *pszWord;
    SPWORDPRONUNCIATION *pFirstWordPronunciation;
} SPWORD;
```

Members

pNextWord
Pointer to the next word in the list.

LangID
The language ID of the word.

wReserved
Reserved for future use.

eWordType
Flag of type [SPWORDTYPE](#) indicating whether the word has been added or deleted.

pszWord
Pointer to the null-terminated word.

pFirstWordPronunciation

Pointer to the first possible pronunciation of the word.



SPWORDENTRY

SPWORDENTRY is used by SAPI and the speech recognition (SR) engine to exchange information about a word that can be recognized.

When new words are added to a grammar by the application, SAPI informs the SR engine by calling [ISpSREngine::WordNotify](#) for each word, with an associated SPWORDENTRY structure which the SR engine can examine.

When the SR engine needs to query information from SAPI about a particular word in the application grammar, the engine can call [ISpSREngineSite::GetWordInfo](#).

```
typedef struct SPWORDENTRY
{
    SPWORDHANDLE    hWord;
    LANGID          LangID;
    const WCHAR     *pszDisplayText;
    const WCHAR     *pszLexicalForm;
    SPPHONEID      *aPhoneId;
    void            *pvClientContext;
} SPWORDENTRY;
```

Members

hWord
Handle to the current word.

LangID
Language identifier.

pszDisplayText
Pointer to a null-terminated string containing the display text information.

pszLexicalForm

Pointer to a null-terminated string containing the lexical text information.

aPhoneld

Pointer to a null-terminated array containing the phoneme identifier.

pvClientContext

Pointer to a string representing the client context data.



SPWORDLIST

SPWORDLIST receives words currently in the lexicon or that have changed since the last time an engine was checked. It is used in conjunction with `ISpLexicon`.

```
typedef struct SPWORDLIST
{
    ULONG        ulSize;
    BYTE         *pvBuffer;
    SPWORD       *pFirstWord;
} SPWORDLIST;
```

Members

ulSize

The size of the buffer for all of the words, in bytes.

pvBuffer

Pointer to the buffer for all word information or changes.

pFirstWord

Pointer to the first word in the list.

Remarks

This structure is the beginning of a linked list of [SPWORD](#) structures and contains the size and actual buffer of all subsequent word operations. It is used with [ISpLexicon::GetWords](#) and [ISpLexicon::GetGenerationChange](#).

Call `ZeroMemory` before using `SPWORDLIST` to initialize it, and call `CoTaskMemFree(spWordList.pvBuffer)` to free the buffer allocated during the calls. The *pvBuffer* need not (and should not) be freed between the calls. [ISpLexicon::GetWords](#) and [ISpLexicon::GetGenerationChange](#) will reuse the buffer for

efficiency and reallocate when necessary.

Examples

The following example is a code fragment demonstrating the use and creation of SPWORDLIST. The code initializes the structure prior to use. Note that the returned SPWORDLIST has a CoTaskMemAllocate buffer attached to it. This should be freed after all operations using the list. If not all words are returned from [ISpLexicon::GetWords](#), the structure should not be wiped and can be passed into subsequent calls to efficiently reuse the allocated memory block.

```
SPWORDLIST SPWordList;
hr = ZeroMemory(&SPWordList, sizeof(SPWordList));
// Check return value here. Handle error.
hr = S_FALSE;
while (hr == S_FALSE)
{
    hr = pLex->GetWords(eLEXTYPE_USER, &dwGen, &dwCookie, &SPI
    // Do something with the received words.
    // S_FALSE is returned if there are still words remaining
}
// Have finished with the list. Free the enclosed buffer.
::CoTaskMemFree(SPWordList.pvBuffer);
```

The following helper class will ensure the correct usage of SPWORDLIST.

```
class CSpWordList : public SPWORDLIST
{
public:
    CSpWordList()
    {
        ZeroMemory(static_cast<SPWORDLIST*>(this), sizeof(SPI
    }
    ~CSpWordList()
    {
        CoTaskMemFree(pvBuffer);
    }
};
```

Using the helper class, the above sample becomes:

```
CSpWordList SPWordList;
hr = S_FALSE;
while (hr == S_FALSE)
{
    hr = pLex->GetWords(eLEXTYPE_USER, &dwGen, &dwCookie, &SPI
    // Do something with the received words.
    // S_FALSE is returned if there are still words remaining
}
```



SPWORDPRONUNCIATION

SPWORDPRONUNCIATION is used by [ISpLexicon](#) for words with possible variations in pronunciation. SPWORDPRONUNCIATION contains a single pronunciation for a word.

```
typedef struct SPWORDPRONUNCIATION
{
    struct SPWORDPRONUNCIATION    *pNextWordPronunciation;
    SPLEXICONTYPE                eLexiconType;
    LANGID                          LangID;
    WORD                             wReserved;
    SPPARTOFSPEECH              ePartOfSpeech;
    SPPHONEID                       szPronunciation[1];
} SPWORDPRONUNCIATION;
```

Members

pNextWordPronunciation

Pointer to the next possible pronunciation. May be NULL.

eLexiconType

Flag of type [SPLEXICONTYPE](#) indicating which lexicon this pronunciation and part of speech was obtained from.

LangID

The language identifier.

wReserved

Reserved for future use.

ePartOfSpeech

The part of speech used by this particular variation.

szPronunciation[1]

This is a null-terminated array of SPPHONID elements defining the pronunciation. It runs off the end of the SPWORDPRONUNCIATION structure and is part of data buffer in the containing [SPWORDPRONUNCIATIONLIST](#) structure.



SPWORDPRONUNCIATIONLIST

SPWORDPRONUNCIATIONLIST is used with [ISpLexicon::GetPronunciations](#) to list possible variations in pronunciation for a given word.

```
typedef struct SPWORDPRONUNCIATIONLIST
{
    ULONG                ulSize;
    BYTE                 *pvBuffer;
    SPWORDPRONUNCIATION *pFirstWordPronunciation;
} SPWORDPRONUNCIATIONLIST;
```

Members

ulSize

Size of the pronunciation buffer, in bytes.

pvBuffer

Pointer to the buffer for all pronunciations.

pFirstWordPronunciation

Pointer to the first in a linked list of [SPWORDPRONUNCIATION](#) structures within *pvBuffer*.

Remarks

This structure is the start of a linked list of [SPWORDPRONUNCIATION](#) structures and contains the size and actual buffer of all subsequent pronunciations.

Call `ZeroMemory` before using `SPWORDPRONUNCIATIONLIST` to initialize it, and call `CoTaskMemFree(spwordpronlist.pvBuffer)` to free the buffer allocated during the calls. The *pvBuffer* need not (and should not) be freed between the calls.

[ISpLexicon::GetPronunciations](#) will reuse the buffer for efficiency

and reallocate when necessary.

Example

The following example is a code fragment demonstrating the use and creation of SPWORDPRONUNCIATIONLIST.

```
SPWORDPRONUNCIATIONLIST spwordpronlist;
memset(spwordpronlist, 0, sizeof(spwordpronlist));

pISpLexicon->GetPronunciations(L"resume", 0, 0, &spwordp
for (SPWORDPRONUNCIATION *pwordpron = spwordpronlist.pFi
    wordpron != NULL;
    wordpron = pwordpron->pNextWordPronunciation)
{
    DoSomethingWith(pwordpron->ePartOfSpeech, pwordpron->
}

pISpLexicon->GetPronunciations(L"record", 0, 0, &spwordp
// repeat the for loop above to process the pronunciation

CoTaskMemFree(spwordpronlist.pvBuffer);
```

The following helper class will ensure the correct usage of SPWORDPRONUNCIATIONLIST.

```
class CSpPronList : public SPWORDPRONUNCIATIONLIST
{
public:
    CSpPronList()
    {
        ZeroMemory(static_cast<SPWORDPRONUNCIATIONLIST*>(this));
    }
    ~CSpPronList()
    {
        CoTaskMemFree(pvBuffer);
    }
};
```

Using the helper class, the above sample becomes:

```
CSpPronList spwordpronlist;

pISpLexicon->GetPronunciations(L"resume", 0, 0, &spwordp
for (SPWORDPRONUNCIATION *pwordpron = spwordpronlist.pFi
    wordpron != NULL;
    wordpron = pwordpron->pNextWordPronunciation)
{
    DoSomethingWith(pwordpron->ePartOfSpeech, pwordpron->
}

pISpLexicon->GetPronunciations(L"record", 0, 0, &spwordp
// repeat the for loop above to process the pronunciations
```



WAVEFORMATEX

WAVEFORMATEX defines the format of waveform-audio data. Only format information common to all waveform-audio data formats is included in this structure. For formats requiring additional information, this structure is included as the first member in another structure, along with the additional information.

```
typedef struct WAVEFORMATEX
{
    WORD    wFormatTag;
    WORD    nChannels;
    DWORD   nSamplesPerSec;
    DWORD   nAvgBytesPerSec;
    WORD    nBlockAlign;
    WORD    wBitsPerSample;
    WORD    cbSize;
} WAVEFORMATEX;
```

Members

wFormatTag

Waveform-audio format type. Format tags are registered with Microsoft Corporation for many compression algorithms. A complete list of format tags is located in the `Mmsystem.h` header file.

nChannels

Number of channels in the waveform-audio data. Monaural data uses one channel and stereo data uses two channels.

nSamplesPerSec

Sample rate, in samples per second (hertz) at which each channel should be played or recorded. If *wFormatTag* is

WAVE_FORMAT_PCM, common values for *nSamplesPerSec* are 8.0 kHz, 11.025 kHz, 22.05 kHz, and 44.1 kHz. For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

nAvgBytesPerSec

Required average data-transfer rate, in bytes per second, for the format tag. If *wFormatTag* is WAVE_FORMAT_PCM, *nAvgBytesPerSec* should be equal to the product of *nSamplesPerSec* and *nBlockAlign*. For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

Playback and record software can estimate buffer sizes using the *nAvgBytesPerSec* member.

nBlockAlign

Block alignment, in bytes. The block alignment is the minimum atomic unit of data for the *wFormatTag* format type. If *wFormatTag* is WAVE_FORMAT_PCM, *nBlockAlign* should be equal to the product of *nChannels* and *wBitsPerSample* divided by 8 (bits per byte). For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

Playback and record software must process a multiple of *nBlockAlign* bytes of data at a time. Data written and read from a device must always start at the beginning of a block. For example, it is illegal to start playback of PCM data in the middle of a sample (that is, on a non-block-aligned boundary).

wBitsPerSample

Bits per sample for the *wFormatTag* format type. If *wFormatTag* is WAVE_FORMAT_PCM, *wBitsPerSample* should be equal to 8 or 16. For non-PCM formats, this member must

be set according to the manufacturer's specification of the format tag. Note that some compression schemes cannot define a value for *wBitsPerSample*, so this member can be zero.

cbSize

Size, in bytes, of extra format information appended to the end of the WAVEFORMATEX structure. This information can be used by non-PCM formats to store extra attributes for the *wFormatTag*. If no extra information is required by the *wFormatTag*, this member must be set to zero. For WAVE_FORMAT_PCM formats only, this member is ignored.



Enumerations

The following enumerations are used with SAPI 5.

- [SPAUDIOOPTIONS](#)
- [SPAUDIOSTATE](#)
- [SPBOOKMARKOPTIONS](#)
- [SPCFGNOTIFY](#)
- [SPCFGRULEATTRIBUTES](#)
- [SPCONTEXTSTATE](#)
- [SPDATAKEYLOCATION](#)
- [SPDISPLAYATTRIBUTES](#)
- [SPEAKFLAGS](#)
- [SPENDSRSTREAMFLAGS](#)
- [SPEVENTENUM](#)
- [SPEVENTLPARAMTYPE](#)
- [SPFILEMODE](#)
- [SPGRAMMARSTATE](#)
- [SPGRAMMARWORDTYPE](#)
- [SPINTERFERENCE](#)
- [SPLEXICONTYPE](#)
- [SPLOADOPTIONS](#)
- [SPPARTOFSPEECH](#)
- [SPPHRASERNG](#)
- [SPPROPSRC](#)

- [SPRECOEVENTFLAGS](#)
- [SPRECOSTATE](#)
- [SPRESULTTYPE](#)
- [SPRULEINFOOPT](#)
- [SPRULESTATE](#)
- [SPRUNSTATE](#)
- [SPSTREAMFORMAT](#)
- [SPTRANSITIONTYPE](#)
- [SPVACTIONS](#)
- [SPVALUETYPE](#)
- [SPVESACTIONS](#)
- [SPVFEATURE](#)
- [SPVISEMES](#)
- [SPVLIMITS](#)
- [SPVPRIORITY](#)
- [SPVSKIPTYPE](#)
- [SPWAVEFORMATTYPE](#)
- [SPWORDINFOOPT](#)
- [SPWORDPRONOUNCEABLE](#)
- [SPWORDTYPE](#)



SPAUDIOOPTIONS

SPAUDIOOPTIONS lists the options for an audio stream.

See also the [ISpRecoContext::SetAudioOptions](#) and [ISpRecoContext::GetAudioOptions](#) methods.

```
typedef enum SPAUDIOOPTIONS
{
    SPAO_NONE,
    SPAO_RETAIN_AUDIO
} SPAUDIOOPTIONS;
```

Elements

SPAO_NONE

Flag indicating no options for the audio stream should be used.

SPAO_RETAIN_AUDIO

Flag indicating the audio stream should be retained (e.g. serialization of recognition object, playback of recognized audio, etc.). See also [ISpRecoResult::SpeakAudio](#) and [ISpRecoResult::GetAudio](#).



SPAUDIOSTATE

SPAUDIOSTATE sets the audio input or output state to one of four possible states.

Used directly by the [ISpAudio::SetState](#) method and as a member of the [SPAUDIOSTATUS](#) structure.

```
typedef enum _SPAUDIOSTATE
{
    SPAS_CLOSED,
    SPAS_STOP,
    SPAS_PAUSE,
    SPAS_RUN
} SPAUDIOSTATE;
```

Elements

SPAS_CLOSED

Audio is stopped and closed. For multimedia audio input devices (sound cards etc.), the device will be released. It can be opened by other processes and potentially made unavailable to SAPI.

SPAS_STOP

Audio is stopped. For multimedia audio input devices (sound cards etc.), the audio device will not be closed. This guarantees that it can be restarted by SAPI without an intervening process opening it.

SPAS_PAUSE

Audio is paused. Staying in this state for too long a period will cause audio loss.

SPAS_RUN

Audio is enabled.



SPBOOKMARKOPTIONS

SPBOOKMARKOPTIONS is used at the creation of a bookmark to specify whether the bookmark will pause a recognition context.

```
typedef enum SPBOOKMARKOPTIONS
{
    SPBO_NONE,
    SPBO_PAUSE
} SPBOOKMARKOPTIONS;
```

Members

SPBO_NONE

The recognition context will not be paused when the associated bookmark event occurs.

SPBO_PAUSE

The recognition context will be paused when the associated bookmark event occurs. See also [ISpRecoContext::Pause](#).



SPCFGNOTIFY

SPCFGNOTIFY lists event notification information related to the addition, deletion, invalidation, activation, or deactivation of words and rules in the loaded grammars.

```
typedef enum SPCFGNOTIFY
{
    SPCFGN_ADD,
    SPCFGN_REMOVE,
    SPCFGN_INVALIDATE,
    SPCFGN_ACTIVATE,
    SPCFGN_DEACTIVATE
} SPCFGNOTIFY;
```

Elements

SPCFGN_ADD

Flag indicating that the grammar rule should be added.

SPCFGN_REMOVE

Flag indicating that the grammar rule should be removed.

SPCFGN_INVALIDATE

Flag indicating that the grammar rule should be invalidated.

SPCFGN_ACTIVATE

Flag indicating that the grammar rule should be activated.

SPCFGN_DEACTIVATE

Flag indicating that the grammar rule should be deactivated.



SPCFGRULEATTRIBUTES

SPCFGRULEATTRIBUTES lists the attribute information of grammar rules.

```
typedef enum SPCFGRULEATTRIBUTES
{
    SPRAF_TopLevel,
    SPRAF_Active,
    SPRAF_Export,
    SPRAF_Import,
    SPRAF_Interpreter,
    SPRAF_Dynamic,
    SPRAF_AutoPause
} SPCFGRULEATTRIBUTES;
```

Elements

SPRAF_TopLevel

Flag specifying that the rule is defined as a top-level rule. Top-level rules are the entry points into the grammar and can be de-/activated programmatically. This can be set using the TOPELVEL attribute in the Speech Text Grammar Format.

SPRAF_Active

Flag specifying that the rule is defined as a top-level rule that is activated by default. These rules can be de-/activated by calling De-/ActivateRule(NULL, 0, ...). This can be set using the TOPLEVEL="ACTIVE" attribute-value pair in the Speech Text Grammar Format.

SPRAF_Export

Flag specifying that the rule is exported and hence can be referred to by a rule in another grammar. This can be set using the EXPORT="YES" attribute-value pair in the Speech

Text Grammar Format.

SPRAF_Import

Flag specifying that the rule is imported from another grammar and is therefore not defined in this grammar.

SPRAF_Interpreter

Flag specifying that the rule has an interpreter (custom C/C++ code implementing the [ISpCFGInterpreter](#) interface) associated with it.

SPRAF_Dynamic

Flag specifying that the rule is dynamic (can be changed programmatically through the [ISpGrammarBuilder](#) interface). Note that the CFG must be loaded with the [SPLO_DYNAMIC](#) flag to enable changes at run time.

SPRAF_AutoPause

Flag specifying the grammar attributes as AutoPause. This flag is only valid at run time as part of a rule state and is not valid to pass as part of a rule definition.



SPCONTEXTSTATE

SPCONTEXTSTATE lists controls for setting and restoring recognition states on a per-context basis.

```
typedef enum SPCONTEXTSTATE
{
    SPCS_DISABLED,
    SPCS_ENABLED
} SPCONTEXTSTATE;
```

Elements

SPCS_DISABLED

Specifies that grammars associated with this recognition context are disabled. When an application sets the context state to `SPCS_DISABLED`, all rules in all grammars owned by that context are disabled, even if the grammar state is set to exclusive.

SPCS_ENABLED

Specifies that grammars associated with this recognition context are enabled. By default recognition contexts are created with `SPCS_ENABLED`.



SPDATAKEYLOCATION

SPDATAKEYLOCATION lists top-level registry keys. It is used for data key locations with [ISpObjectTokenCategory::GetDataKey](#)

```
typedef enum SPDATAKEYLOCATION
{
    SPDKL_DefaultLocation,
    SPDKL_CurrentUser,
    SPDKL_LocalMachine,
    SPDKL_CurrentConfig
} SPDATAKEYLOCATION;
```

Elements

SPDKL_DefaultLocation

The default location set by ISpObjectTokenCategory.

SPDKL_CurrentUser

The registry key HKEY_CURRENT_USER.

SPDKL_LocalMachine

The registry key HKEY_LOCAL_MACHINE.

SPDKL_CurrentConfig

The registry key HKEY_CURRENT_CONFIG.



SPDISPLAYATTRIBUTES

SPDISPLAYATTRIBUTES lists the display text of [phrase elements](#).

```
typedef enum tagSPDISPLYATTRIBUTES
{
    SPAF_ONE_TRAILING_SPACE,
    SPAF_TWO_TRAILING_SPACES,
    SPAF_CONSUME_LEADING_SPACES,
    SPAF_ALL
} SPDISPLAYATTRIBUTES;
```

Elements

SPAF_ONE_TRAILING_SPACE

Inserts one trailing space, used for most words.

SPAF_TWO_TRAILING_SPACES

Insert two trailing spaces, often used after a sentence final period.

SPAF_CONSUME_LEADING_SPACES

Consume leading space, often used for periods. If this is absent, the word should have a leading space by default.

SPAF_ALL

A combination of all of the above flags.



SPEAKFLAGS

SPEAKFLAGS lists the [ISpVoice::Speak](#) call also the [ISpTTSVoice::Speak](#) call.

```
typedef enum SPEAKFLAGS
{
    //--- SpVoice flags
    SPF_DEFAULT,
    SPF_ASYNC,
    SPF_PURGEBEFORESPEAK,
    SPF_IS_FILENAME,
    SPF_IS_XML,
    SPF_IS_NOT_XML,
    SPF_PERSIST_XML,

    //--- Normalizer flags
    SPF_NLP_SPEAK_PUNC,

    //--- Masks
    SPF_NLP_MASK,
    SPF_VOICE_MASK,
    SPF_UNUSED_FLAGS
} SPEAKFLAGS;
```

Elements

SPF_DEFAULT

Specifies that the default settings should be used. The defaults are:

- Speak the given text string synchronously
- Not purge pending speak requests
- Parse the text as XML only if the first character is a left-angle-bracket (<)
- Not persist global XML state changes across speak

calls

- Not expand punctuation characters into words.

To override this default, use the other flag values given below.

SPF_ASYNC

Specifies that the Speak call should be asynchronous. That is, it will return immediately after the speak request is queued.

SPF_PURGEBEFORESPEAK

Purges all pending speak requests prior to this speak call.

SPF_IS_FILENAME

The string passed to [ISpVoice::Speak](#) is a file name, and the file text should be spoken.

SPF_IS_XML

The input text will be parsed for XML markup.

SPF_IS_NOT_XML

The input text will not be parsed for XML markup.

SPF_PERSIST_XML

Global state changes in the XML markup will persist across speak calls.

SPF_NLP_SPEAK_PUNC

Punctuation characters should be expanded into words (e.g. "This is a sentence." would become "This is a sentence period").

SPF_NLP_MASK

This mask is used to remove the SAPI handled flags before ISpTTSEngine::Speak is called. The only flag which the TTS engine must handle is SPF_NLP_SPEAK_PUNC.

SPF_VOICE_MASK

This mask has every flag bit set.

SPF_UNUSED_FLAGS

This mask has every unused bit set.



SPENDSRSTREAMFLAGS

SPENDSRSTREAMFLAGS enables an application to query for state changes when the end of a speech recognition (SR) stream is encountered.

```
typedef enum SPENDSRSTREAMFLAGS
{
    SPESF_NONE,
    SPESF_STREAM_RELEASED
} SPENDSRSTREAMFLAGS;
```

Elements

SPESF_NONE

No flags are associated with the end of stream event.

SPESF_STREAM_RELEASED

The input stream object was released upon reaching the end of the current stream. For example, a wave file is a finite stream of data, and once the end of the stream, and file, is reached, the stream object is released. See also [CSpEvent::InputStreamReleased](#).

Microsoft Speech SDK

SAPI 5.1



SPEVENTENUM

SPEVENTENUM lists the events possible from SAPI.

It is recommended that developers use the helper class [CSpEvent](#) to easily and clearly decode events.

```
typedef enum SPEVENTENUM
{
    SPEI_UNDEFINED,

    //--- TTS engine
    SPEI_START_INPUT_STREAM,
    SPEI_END_INPUT_STREAM,
    SPEI_VOICE_CHANGE,
    SPEI_TTS_BOOKMARK,
    SPEI_WORD_BOUNDARY,
    SPEI_PHONEME,
    SPEI_SENTENCE_BOUNDARY,
    SPEI_VISEME,
    SPEI_TTS_AUDIO_LEVEL,

    //--- Engine vendors use these reserved bits
    SPEI_TTS_PRIVATE,

    SPEI_MIN_TTS,
    SPEI_MAX_TTS,

    //--- Speech Recognition
    SPEI_END_SR_STREAM,
    SPEI_SOUND_START,
    SPEI_SOUND_END,
    SPEI_PHRASE_START,
    SPEI_RECOGNITION,
    SPEI_HYPOTHESIS,
    SPEI_SR_BOOKMARK,
    SPEI_PROPERTY_NUM_CHANGE,
    SPEI_PROPERTY_STRING_CHANGE,
    SPEI_FALSE_RECOGNITION,
    SPEI_INTERFERENCE,
    SPEI_REQUEST_UI,
    SPEI_RECO_STATE_CHANGE,
```

```

    SPEI_ADAPTATION,
    SPEI_START_SR_STREAM,
    SPEI_RECO_OTHER_CONTEXT,
    SPEI_SR_AUDIO_LEVEL,

    //--- Engine vendors use these reserved bits
    SPEI_SR_PRIVATE,

    SPEI_MIN_SR,
    SPEI_MAX_SR,

    SPEI_RESERVED1,
    SPEI_RESERVED2,
    SPEI_RESERVED3
} SPEVENTENUM;

```

Elements

SPEI_START_INPUT_STREAM

The input stream (text or audio) from a `Speak` or `SpeakStream` call has begun synthesizing to the output. The event is fired by SAPI.

SPEI_END_INPUT_STREAM

The input stream (text or audio) from a `Speak` or `SpeakStream` call has finished synthesizing to the output. The event is fired by SAPI.

SPEI_VOICE_CHANGE

SAPI fires this event for voice changes within a single input stream of a `Speak` call. *wParam* is either zero or the `SPF_PERSIST_XML`. If the current `Speak` call takes `SPF_PERSIST_XML`, *wparam* is `SPF_PERSIST_XML`. Otherwise, zero. *lParam* is the current voice object token. *elParamType* has to be `SPET_LPARAM_IS_TOKEN`.

SPEI_TTS_BOOKMARK

The bookmark element is used to insert a bookmark into the input stream. If an application specifies interest in bookmark events, it will receive the bookmark events during synthesis. *wParam* is the current bookmark name (in base 10) converted to a long integer. If name of current bookmark is not an integer, *wParam* will be zero. *lParam* is the bookmark string. *elParamType* has to be `SPET_LPARAM_IS_STRING`.

SPEI_WORD_BOUNDARY

A word is beginning to synthesize. Markup language (XML) markers are counted in the boundaries and offsets. *wParam* is the character length of the word in the current input stream being synthesized. *lParam* is the character position within the current text input stream of the word being synthesized.

SPEI_PHONEME

Phoneme was returned by the TTS engine. The high word of *wParam* is the duration, in milliseconds, of the current phoneme element. The low word is the id of the next phoneme element. The high word of *lParam* is the phoneme element feature defined in [SPVFEATURE](#). This value will be zero if the current phoneme element is not a primary stress or emphasis. The low word of *lParam* is the id for the current phoneme element being synthesized.

When the engine synthesizes a phoneme comprised of more than one phoneme element, it raises an event for each element. For example, when a Japanese TTS engine speaks the phoneme "KYA," which is comprised of the phoneme elements "KI" and "XYA," it raises an `SPEI_PHONEME` event for each element. Because the element "KI" in this case modifies the sound of the element following it, rather than initiating a sound, the duration of its `SPEI_PHONEME` event is zero.

SPEI_SENTENCE_BOUNDARY

A sentence is beginning to synthesize. *wParam* is the character length of the sentence including punctuation in the current input stream being synthesized. *lParam* is the character position within the current text input stream of the sentence being synthesized.

SPEI_VISEME

Viseme was determined by synthesis engine. The high word of *wParam* is the duration, in milliseconds, of the current viseme. The low word is for the next viseme of type [SPVISEMES](#). The high word of *lParam* is the viseme feature defined in [SPVFEATURE](#). This value will be zero if the current viseme is not primary stress or emphasis. The low word of *lParam* is the current viseme being synthesized.

SPEI_TTS_AUDIO_LEVEL

This event is fired by SAPI. *lParam* is 0, and *wParam* is the current audio level from zero to 100.

SPEI_TTS_PRIVATE

Reserved for private/internal use by the TTS Engine.

SPEI_MIN_TTS

Minimum event enumeration value for TTS events.

SPEI_MAX_TTS

Maximum event enumeration value for TTS events.

SPEI_END_SR_STREAM

The SR engine has finished receiving an audio input stream. *LPARAM* points to the SR engine's final HRESULT code (see [CSpEvent::EndStreamResult](#)). *WPARAM* points to a Boolean

value signifying whether the audio input stream object was released (see [CSpEvent::InputStreamReleased](#)).

SPEI_SOUND_START

The SR engine determined that audible sound is available through the input stream.

SPEI_SOUND_END

The SR engine has determined that audible sound is no longer available through the input stream, or that the sound stream has been inactive for a period.

SPEI_PHRASE_START

The SR engine is starting to recognize a phrase. Note that this **MUST** be followed by either an **SPEI_FALSE_RECOGNITION** or **SPEI_RECOGNITION** event.

SPEI_RECOGNITION

The SR engine is returning a full recognition - its best guess at a text representation of the audio data. *LParam* is a pointer to an [ISpRecoResult](#) object (see [CSpEvent::RecoResult](#)).

SPEI_HYPOTHESIS

The SR engine is returning a partial phrase recognition - effectively its best guess up to that point in the stream. *LParam* is a pointer to an [ISpRecoResult](#) object (see [CSpEvent::RecoResult](#)).

SPEI_SR_BOOKMARK

A Bookmark event is returned when the SR engine has processed to the stream position of a bookmark. *lParam* is an application specified value set using

ISpRecoContext::Bookmark. *wParam* is SPREF_AutoPause if ISpRecoContext::Bookmark was called with SPBO_PAUSE, and NULL otherwise.

SPEI_PROPERTY_NUM_CHANGE

An SR engine supported property was changed. *LPARAM* is a string pointer to the property name that changed (see [CSpEvent::PropertyName](#)). *WPARAM* contains the new value (see [CSpEvent::PropertyNumValue](#)).

SPEI_PROPERTY_STRING_CHANGE

LPARAM is a string pointer to the property name that changed (see [CSpEvent::PropertyName](#)). Immediately following the NULL-termination of the property name is the new property value (see [CSpEvent::PropertyStringValue](#)).

SPEI_FALSE_RECOGNITION

Apparent speech without valid recognition. An SR engine can optionally return a result object, which will be referenced by the *LPARAM* member (see [CSpEvent::RecoResult](#)).

SPEI_INTERFERENCE

The SR engine determined that the sound stream has a hindrance and is preventing a successful recognition. *IParam* is any combination of [SPINTERFERENCE](#) flags (See [CSpEvent::Interference](#)).

SPEI_REQUEST_UI

The SR engine's request to display a specific user interface. *LPARAM* is a null-terminated string (see [CSpEvent::RequestTypeOfUI](#)).

SPEI_RECO_STATE_CHANGE

The recognizer state has changed. *WPARAM* is the new recognizer state (see [SPRECOSTATE](#) and [CSpEvent::RecoState](#)).

SPEI_ADAPTATION

The SR engine is ready to process the adaptation buffer.

SPEI_START_SR_STREAM

The SR engine has reached the start of a new audio stream.

SPEI_SR_AUDIO_LEVEL

The audio input stream object fires this event. *wParam* is the current audio level from zero to 100.

SPEI_RECO_OTHER_CONTEXT

A recognition was sent to another context.

SPEI_SR_PRIVATE

Reserved for private/internal use by the SR engine.

SPEI_MIN_SR

Minimum event enumeration value for speech recognition events.

SPEI_MAX_SR

Maximum event enumeration value for speech recognition events.

SPEI_RESERVED1

Reserved for SAPI internal use. See [SPFEI](#) Remarks section.

SPEI_RESERVED2

Reserved for SAPI internal use. See [SPFEI](#) Remarks section.

SPEI_RESERVED3

Reserved for future use, do not use.



SPEVENTLPARAMTYPE

SPEVENTLPARAMTYPE lists objects and data pointers attached to events. **SPEVENTLPARAMTYPE** specifies the type of the attached data, enabling the user to determine how to access the data, and how to release the data when finished with the event.

See the helper [CSpEvent](#) for more information about C++ class that has properties for accessing event-specific data, and a destructor to automatically cleanup attached data.

See the helper [SpClearEvent](#) for more information about releasing objects or memory attached to an event.

```
typedef enum SPEVENTLPARAMTYPE
{
    SPET_LPARAM_IS_UNDEFINED,
    SPET_LPARAM_IS_TOKEN,
    SPET_LPARAM_IS_OBJECT,
    SPET_LPARAM_IS_POINTER,
    SPET_LPARAM_IS_STRING
} SPEVENTLPARAMTYPE;
```

Elements

SPET_LPARAM_IS_UNDEFINED

The `SPEVENT.IParam` value represents an undefined value. For example, all TTS events, except `SPEI_VOICE_CHANGE` and `SPEI_TTS_BOOKMARK`, do not have attached data, so the event type is `SPET_LPARAM_IS_UNDEFINED`.

The user does **not** need to release associated data, since there is no associated data.

SPET_LPARAM_IS_TOKEN

The `SPEVENT.IParam` value represents a pointer to an `ISpObjectToken` object.

For example, the TTS voice change event (i.e., SPEI_VOICE_CHANGE) includes a pointer to the new voice's object token, so the *IParam* type is SPET_LPARAM_IS_TOKEN. The user **must** call *IUnknown::Release* on the *IParam* member (as pointer) to release the associated object token.

SPET_LPARAM_IS_OBJECT

The SPEVENT.*IParam* value represents a pointer to an object. For example, the speech recognition event (i.e., SPEI_RECOGNITION) includes a pointer to the recognition result (e.g. ISpRecoResult), so the *IParam* type is SPET_LPARAM_IS_OBJECT. The user **must** call *IUnknown::Release* on the *IParam* member (as pointer) to release the associated object token.

SPET_LPARAM_IS_POINTER

The SPEVENT.*IParam* value represents a memory pointer. For example, the property string change event (i.e., SPEI_PROPERTY_STRING_CHANGE) includes a pointer to a block of memory that contains a string (e.g. WCHAR*), so the *IParam* type is SPET_LPARAM_IS_POINTER. The user **must** call *CoTaskMemFree* on the *IParam* member (as pointer) to release the associated memory.

SPET_LPARAM_IS_STRING

The SPEVENT.*IParam* value represents a pointer to a string. For example, the TTS bookmark event (i.e., SPEI_TTS_BOOKMARK) includes a pointer the bookmark name, so the *IParam* type is SPET_LPARAM_IS_STRING. The user **must** call *CoTaskMemFree* on the *IParam* member (as pointer) to release the associated memory.



SPFILEMODE

SPFILEMODE lists the file opening state used by [ISpStream::BindToFile](#) and the helper function [SPBindToFile](#).

```
typedef enum SPFILEMODE
{
    SPFM_OPEN_READONLY,
    SPFM_OPEN_READWRITE,
    SPFM_CREATE,
    SPFM_CREATE_ALWAYS,
    SPFM_NUM_MODES
} SPFILEMODE;
```

Elements

SPFM_OPEN_READONLY

Opens the existing file in read-only mode. This will fail if the file does not exist.

SPFM_OPEN_READWRITE

Opens the existing file in read-write mode. This will fail if the file does not exist.

SPFM_CREATE

Opens the file if one exists, or creates the file if one does not exist. This flag indicates that the file will be opened in read-write mode.

SPFM_CREATE_ALWAYS

Creates the file, even if the file already exists and deletes the previous file. This flag indicates that the file will be opened in read-write mode.

SPFM_NUM_MODES

This flag is used for limit checking.



SPGRAMMARSTATE

SPGRAMMARSTATE lists controls for setting and restoring grammar states.

```
typedef enum SPGRAMMARSTATE
{
    SPGS_ENABLED = 0,
    SPGS_DISABLED = 1,
    SPGS_EXCLUSIVE = 3,
} SPGRAMMARSTATE;
```

Elements

SPGS_ENABLED

Activates all the top-level rules in the grammar for the SR engine.

Note that a rule must have an active recognition context, active grammar, and active top-level rule in order to be recognized.

SPGS_DISABLED

Deactivates all the top-level rules in the grammar for the SR engine.

SPGS_EXCLUSIVE

Turns off all top-level rules that are not part of this grammar.

For example, an application that needs modal-like input exclusivity can change the grammar to exclusive, which disables all other non-exclusive grammars.



SPGRAMMARWORDTYPE

SPGRAMMARWORDTYPE lists the type of the word(s) to be added to a grammar. This type is either specified in the Speech Text Grammar Format as <GRAMMAR WORDTYPE="LEXICAL"> or as a parameter to [ISpGrammarBuilder::AddWordTransition\(\)](#). SAPI currently allows only SPWT_LEXICAL.

```
typedef enum SPGRAMMARWORDTYPE
{
    SPWT_DISPLAY,
    SPWT_LEXICAL,
    SPWT_PRONUNCIATION
} SPGRAMMARWORDTYPE;
```

Elements

SPWT_DISPLAY

Each word to be added is in display form. That is, it possibly will have to be converted into lexical form(s). For example, the word "23" (display form) would have to be converted into "twenty three" (lexical form). This is currently not implemented in SAPI.

SPWT_LEXICAL

Each word to be added is in lexical form and can be used to access the lexicon.

SPWT_PRONUNCIATION

Each word is specified solely by its pronunciation. This is currently not implemented.



SPINTERFERENCE

SPINTERFERENCE lists possible causes of interference or poor recognition with the input stream.

```
typedef enum SPINTERFERENCE
{
    SPINTERFERENCE_NONE,
    SPINTERFERENCE_NOISE,
    SPINTERFERENCE_NOSIGNAL,
    SPINTERFERENCE_TOOLOUD,
    SPINTERFERENCE_TOOQUIET,
    SPINTERFERENCE_TOOFAST,
    SPINTERFERENCE_TOOSLOW
} SPINTERFERENCE;
```

Elements

SPINTERFERENCE_NONE

Private event. Do not use.

SPINTERFERENCE_NOISE

The sound received is interpreted by the speech recognition engine as noise. This event is generated when there is a **SOUND_START** followed by a **SOUND_END** without an intervening **PHRASE_START**. The event will be also generated during dictation if, after a series of hypotheses, it is determined that the signal is noise.

SPINTERFERENCE_NOSIGNAL

A sound is received but it is of a constant intensity. This also includes the microphone being unplugged or muted.

SPINTERFERENCE_TOOLOUD

A sound is received but the stream intensity is too high for discrete recognition.

SPINTERFERENCE_TOOQUIET

A sound is received but the stream intensity is too low for discrete recognition.

SPINTERFERENCE_TOOFAST

The words are spoken too quickly for discrete recognition.

SPINTERFERENCE_TOOSLOW

The words are spoken too slowly and indicates excessive time between words.



SPEXICONTYPE

SPEXICONTYPE lists the allowed lexicon types. Currently there are only two types in use--user and application lexicons. There are ample reserved types for future expansion and ample private types for potential private use by applications and engines.

```
typedef enum SPEXICONTYPE
```

```
{  
    eLEXTYPE_USER,  
    eLEXTYPE_APP,  
    eLEXTYPE_RESERVED1,  
    eLEXTYPE_RESERVED2,  
    eLEXTYPE_RESERVED3,  
    eLEXTYPE_RESERVED4,  
    eLEXTYPE_RESERVED5,  
    eLEXTYPE_RESERVED6,  
    eLEXTYPE_RESERVED7,  
    eLEXTYPE_RESERVED8,  
    eLEXTYPE_RESERVED9,  
    eLEXTYPE_RESERVED10,  
    eLEXTYPE_PRIVATE1,  
    eLEXTYPE_PRIVATE2,  
    eLEXTYPE_PRIVATE3,  
    eLEXTYPE_PRIVATE4,  
    eLEXTYPE_PRIVATE5,  
    eLEXTYPE_PRIVATE6,  
    eLEXTYPE_PRIVATE7,  
    eLEXTYPE_PRIVATE8,  
    eLEXTYPE_PRIVATE9,  
    eLEXTYPE_PRIVATE10,  
    eLEXTYPE_PRIVATE11,  
    eLEXTYPE_PRIVATE12,  
    eLEXTYPE_PRIVATE13,  
    eLEXTYPE_PRIVATE14,  
    eLEXTYPE_PRIVATE15,  
    eLEXTYPE_PRIVATE16,  
    eLEXTYPE_PRIVATE17,  
    eLEXTYPE_PRIVATE18,  
    eLEXTYPE_PRIVATE19,  
}
```

```
    eLEXTYPE_PRIVATE20  
} SPLEXICONTYPE;
```

Elements

eLEXTYPE_USER

Indicates the user lexicon. Each Windows user has a unique user lexicon.

eLEXTYPE_APP

Indicates the application lexicons. Application lexicon is shared by all users.

eLEXTYPE_RESERVED1 through **eLEXTYPE_RESERVED10**

Reserved for future use.

eLEXTYPE_PRIVATE1 through **eLEXTYPE_PRIVATE20**

Indicates the private lexicons. Engines can call `ISpContainerLexicon::AddLexicon` with one of these flags to add its private lexicons to the container lexicon for consistent access. If these lexicon types are used the lexicons must fully comply with the `ISpLexicon` interface.



SPLOADOPTIONS

SPLOADOPTIONS indicates how a grammar is loaded. This is used by the [ISpRecoGrammar](#) interface.

```
typedef enum SPLOADOPTIONS
{
    SPLO_STATIC,
    SPLO_DYNAMIC
} SPLOADOPTIONS;
```

Elements

SPLO_STATIC

Flag specifying that the grammar is loaded statically.

SPLO_DYNAMIC

Flag specifying that the grammar is loaded dynamically, meaning that rules can be modified and committed at run-time.



SPPARTOFSPEECH

SPPARTOFSPEECH lists the parts-of-speech categories used in SAPI 5. This list of known parts-of-speech types is intentionally small and broad and will be expanded and refined in future releases. SPPARTOFSPEECH in its minimal form is required to support look ups from the standard SAPI lexicon. This information is useful to TTS engines to determine the correct pronunciation for ambiguous words based on their context.

```
typedef enum SPPARTOFSPEECH
{
    //--- SAPI5 public POS category values (bits 28-31)
    SPPS_NotOverriden,
    SPPS_Unknown,
    SPPS_Noun,
    SPPS_Verb,
    SPPS_Modifier,
    SPPS_Function,
    SPPS_Interjection
} SPPARTOFSPEECH;
```

Elements

SPPS_NotOverriden

Flag indicating that the part of speech already present in the lexicon should not be overridden.

SPPS_Unknown

Flag indicating that the part of speech is unknown and is probably from the user lexicon.

SPPS_Noun

Flag indicating that the part of speech is a noun.

SPPS_Verb

Flag indicating that the part of speech is a verb.

SPPS_Modifier

Flag indicating that the part of speech is a modifier.

SPPS_Function

Flag indicating that the part of speech is a function.

SPPS_Interjection

Flag indicating that the part of speech is an interjection.



SPPHRASERNG

```
typedef enum SPPHRASERNG
{
    SPPR_ALL_ELEMENTS
} SPPHRASERNG;
```

Elements

SPPR_ALL_ELEMENTS

Indicates all elements of an alternate phrase may be used.



SPPROPSRC

SPPROPSRC lists the source object type of a property change method call.

```
typedef enum SPPROPSRC
{
    SPPROPSRC_RECO_INST,
    SPPROPSRC_RECO_CTX,
    SPPROPSRC_RECO_GRAMMAR
} SPPROPSRC;
```

Elements

SPPROPSRC_RECO_INST

The source of the property change call was an ISpRecognizer or ISpeechRecognizer-based object.

SPPROPSRC_RECO_CTX

The source of the property change call was an ISpRecoContext or ISpeechRecoContext-based object.

Currently, not used by SAPI.

SPPROPSRC_RECO_GRAMMAR

The source of the property change call was an ISpRecoGrammar or ISpeechRecoGrammar-based object.

Currently, not used by SAPI.



SPRECOEVENTFLAGS

SPRECOEVENTFLAGS lists the states of the SR engine.

```
typedef enum SPRECOEVENTFLAGS
{
    SPREF_AutoPause,
    SPREF_EmuLated
} SPRECOEVENTFLAGS;
```

Elements

SPREF_AutoPause

Indicates that the engine is in the auto-paused state.

SPREF_EmuLated

Indicates that the engine is in emulation.



SPRECOSTATE

SPRECOSTATE lists the various states of the recognition engine.

```
typedef enum SPRECOSTATE
{
    SPRST_INACTIVE,
    SPRST_ACTIVE,
    SPRST_ACTIVE_ALWAYS,
    SPRST_INACTIVE_WITH_PURGE,
    SPRST_NUM_STATES
} SPRECOSTATE;
```

Elements

SPRST_INACTIVE

The engine and audio input are inactive and no audio is being read, even if there rules active. The audio device will be closed in this state. Normally an application should not set the state to `SPRST_INACTIVE` because when using the shared engine, recognition will be stopped for all applications, not just this one. An application can easily disable recognition on its contexts by calling [ISpRecoContext::SetContextState](#).

SPRST_ACTIVE

This state is the default and indicates that recognition will take place if there are any active rules. If a rule is active, audio will be read and passed to the SR engine and recognition will happen.

SPRST_ACTIVE_ALWAYS

Indicates the audio is running regardless of the rule state. Even if there are no active rules, audio will still be read and passed to the engine. This state can be useful for

applications if they want to receive volume level events ([SPEI_SR_AUDIO_LEVEL](#)), in order to display a VU-meter or similar.

SPRST_INACTIVE_WITH_PURGE

Indicates the engine state will be set to inactive, but all active audio data is purged. This state is used when an application wishes to shut an engine down as quickly as possible, without waiting for it to finish processing any audio data that is currently buffered. This state should be used with care because it will affect all applications in the shared case.

SPRST_NUM_STATES

To be provided in a future release.



SPRESULTTYPE

SPRESULTTYPE lists the result object type information.

```
typedef enum SPRESULTTYPE
{
    SPRT_CFG,
    SPRT_SLM,
    SPRT_PROPRIETARY,
    SPRT_FALSE_RECOGNITION
} SPRESULTTYPE;
```

Elements

SPRT_CFG

Flag specifying that the result object is a context-free grammar type (e.g. command and control grammar).

SPRT_SLM

Flag specifying that the result object is a statistical-language model type (e.g. dictation).

SPRT_PROPRIETARY

Flag specifying that the result object is a proprietary grammar type.

SPRT_FALSE_RECOGNITION

Flag specifying that the result object is a false recognition type. The speech recognition engine can combine the other values with false recognition to inform applications that it failed to recognize specific type of grammar.



SPRULEINFOPT

SPRULEINFOPT lists grammar rule options.

```
typedef enum SPRULEINFOPT
{
    SPRIO_NONE
} SPRULEINFOPT;
```

Elements

SPRIO_NONE

Flag specifying the SPRIO_NONE option.



SPRULESTATE

SPRULESTATE lists the states of a grammar rule.

```
typedef enum SPRULESTATE
{
    SPRS_INACTIVE,
    SPRS_ACTIVE,
    SPRS_ACTIVE_WITH_AUTO_PAUSE
} SPRULESTATE;
```

Elements

SPRS_INACTIVE

Grammar rule is inactive.

SPRS_ACTIVE

Grammar rule is active.

SPRS_ACTIVE_WITH_AUTO_PAUSE

SR engine will be placed in a paused state when the grammar rule is recognized. Also known as an "auto-pause" rule.



SPRUNSTATE

SPRUNSTATE lists the voice running states.

```
typedef enum SPRUNSTATE
{
    SPRS_DONE,
    SPRS_IS_SPEAKING
} SPRUNSTATE;
```

Elements

SPRS_DONE

The voice has completed processing all queued streams.

SPRS_IS_SPEAKING

The voice instance currently has the audio claimed.



SPSTREAMFORMAT

SPSTREAMFORMAT lists the supported stream formats.

These enumeration elements are all common audio formats ranging from the uncompressed PCM formats to highly compressed formats. They are available as standard formats on the Windows operating systems and are supported by SAPI 5.

```
typedef enum SPSTREAMFORMAT
{
    SPSF_Default,
    SPSF_NoAssignedFormat,
    SPSF_Text,
    SPSF_NonStandardFormat,
    SPSF_ExtendedAudioFormat,
    // Standard PCM wave formats
    SPSF_8kHz8BitMono,
    SPSF_8kHz8BitStereo,
    SPSF_8kHz16BitMono,
    SPSF_8kHz16BitStereo,
    SPSF_11kHz8BitMono,
    SPSF_11kHz8BitStereo,
    SPSF_11kHz16BitMono,
    SPSF_11kHz16BitStereo,
    SPSF_12kHz8BitMono,
    SPSF_12kHz8BitStereo,
    SPSF_12kHz16BitMono,
    SPSF_12kHz16BitStereo,
    SPSF_16kHz8BitMono,
    SPSF_16kHz8BitStereo,
    SPSF_16kHz16BitMono,
    SPSF_16kHz16BitStereo,
    SPSF_22kHz8BitMono,
    SPSF_22kHz8BitStereo,
    SPSF_22kHz16BitMono,
    SPSF_22kHz16BitStereo,
    SPSF_24kHz8BitMono,
    SPSF_24kHz8BitStereo,
    SPSF_24kHz16BitMono,
```

SPSF_24kHz16BitStereo,
SPSF_32kHz8BitMono,
SPSF_32kHz8BitStereo,
SPSF_32kHz16BitMono,
SPSF_32kHz16BitStereo,
SPSF_44kHz8BitMono,
SPSF_44kHz8BitStereo,
SPSF_44kHz16BitMono,
SPSF_44kHz16BitStereo,
SPSF_48kHz8BitMono,
SPSF_48kHz8BitStereo,
SPSF_48kHz16BitMono,
SPSF_48kHz16BitStereo,
// TrueSpeech format
SPSF_TrueSpeech_8kHz1BitMono,
// A-Law formats
SPSF_CCITT_ALaw_8kHzMono,
SPSF_CCITT_ALaw_8kHzStereo,
SPSF_CCITT_ALaw_11kHzMono,
SPSF_CCITT_ALaw_11kHzStereo,
SPSF_CCITT_ALaw_22kHzMono,
SPSF_CCITT_ALaw_22kHzStereo,
SPSF_CCITT_ALaw_44kHzMono,
SPSF_CCITT_ALaw_44kHzStereo,
// u-Law formats
SPSF_CCITT_uLaw_8kHzMono,
SPSF_CCITT_uLaw_8kHzStereo,
SPSF_CCITT_uLaw_11kHzMono,
SPSF_CCITT_uLaw_11kHzStereo,
SPSF_CCITT_uLaw_22kHzMono,
SPSF_CCITT_uLaw_22kHzStereo,
SPSF_CCITT_uLaw_44kHzMono,
SPSF_CCITT_uLaw_44kHzStereo,
// ADPCM formats
SPSF_ADPCM_8kHzMono,
SPSF_ADPCM_8kHzStereo,
SPSF_ADPCM_11kHzMono,
SPSF_ADPCM_11kHzStereo,
SPSF_ADPCM_22kHzMono,
SPSF_ADPCM_22kHzStereo,
SPSF_ADPCM_44kHzMono,
SPSF_ADPCM_44kHzStereo,

```
// GSM 6.10 formats
SPSF_GSM610_8kHzMono,
SPSF_GSM610_11kHzMono,
SPSF_GSM610_22kHzMono,
SPSF_GSM610_44kHzMono,
SPSF_NUM_FORMATS
} SPSTREAMFORMAT;
```



SPTRANSITIONTYPE

SPTRANSITIONTYPE lists grammar rule transition information.

```
typedef enum SPTRANSITIONTYPE
{
    SPTRANSEPSILON,
    SPTRANSWORD,
    SPTRANSRULE,
    SPTRANSTEXTBUF,
    SPTRANSWILDCARD,
    SPTRANSDICTATION
} SPTRANSITIONTYPE;
```

Elements

SPTRANSEPSILON

Flag specifying that the grammar rule is an SPTRANSEPSILON type.

SPTRANSWORD

Flag specifying that the grammar rule is an SPTRANSWORD type.

SPTRANSRULE

Flag specifying that the grammar rule is an SPTRANSRULE type.

SPTRANSTEXTBUF

Flag specifying that the grammar rule is an SPTRANSTEXTBUF type.

SPTRANSWILDCARD

Flag specifying that the grammar rule is an

SPTRANSWILDCARD type.

SPTRANSDICTATION

Flag specifying that the grammar rule is an SPTRANSDICTATION type.



SPVACTIONS

SPVACTIONS is a member of [SPVSTATE](#), which is a member of [SPVTEXTFRAG](#). This enumeration specifies the action that should be taken for the text fragment with which it is associated.

```
typedef enum SPVACTIONS
{
    SPVA_Speak,
    SPVA_Silence,
    SPVA_Pronounce,
    SPVA_Bookmark,
    SPVA_SpellOut,
    SPVA_Section,
    SPVA_ParseUnknownTag
} SPVACTIONS;
```

Elements

SPVA_Speak

The default value - the associated text fragment should be processed and spoken.

SPVA_Silence

There is no associated text string - the associated fragment was the result of a <Silence> XML tag, and a silence (of length specified in the associated SPVSTATE) should be inserted into the output stream.

SPVA_Pronounce

The associated text (possibly empty) is associated with a <Pron> XML tag, and should be pronounced as specified in the associated SPVSTATE.

SPVA_Bookmark

The associated text fragment is the contents of a bookmark.

SPVA_SpellOut

Each character, other than white space, of the associated text fragment should be expanded as a word (e.g., "word!" would become "w o r d exclamation point").

SPVA_Section

Reserved for future use.

SPVA_ParseUnknownTag

The associated text fragment is an unknown XML tag that may be interpreted (or ignored) by the engine.



SPVALUETYPE

SPVALUETYPE lists flags indicating portions of a recognition result to be removed or eliminated once they are no longer needed.

```
typedef enum tagSPDISCARDTYPES
{
    SPDF_PROPERTY,
    SPDF_REPLACEMENT,
    SPDF_RULE,
    SPDF_DISPLAYTEXT,
    SPDF_LEXICALFORM,
    SPDF_PRONUNCIATION,
    SPDF_AUDIO,
    SPDF_ALTERNATES,
    SPDF_ALL
} SPVALUETYPE;
```

Elements

SPDF_PROPERTY

Removes the property tree.

SPDF_REPLACEMENT

Removes the phrase replacement text for inverse text normalization.

SPDF_RULE

Removes the non-top level rule tree information for a phrase.

SPDF_DISPLAYTEXT

Removes the display text.

SPDF_LEXICALFORM

Removes the lexicon from text.

SPDF_PRONUNCIATION

Removes the pronunciation text.

SPDF_AUDIO

Removes the audio data that is attached to a phrase.
However, the audio had to be both set and retained.

SPDF_ALTERNATES

Removes the alternate data that is attached to a phrase.
Discarding alternates loses the words permanently and may not be retrieved even with [ISpSRAlternates::GetAlternates](#) or [ISpRecoResult::GetAlternates](#).

SPDF_ALL

Removes all the elements above.



SPVESACTIONS

SPVESACTIONS lists values returned by the [ISpTTSEngineSite::GetActions](#) call. From these values, the TTS engine receives the real-time action requests that have been made by an application.

```
typedef enum SPVESACTIONS
{
    SPVES_CONTINUE,
    SPVES_ABORT,
    SPVES_SKIP,
    SPVES_RATE,
    SPVES_VOLUME
} SPVESACTIONS;
```

Elements

SPVES_CONTINUE

Default value - indicates SAPI has not received any new information for the engine, and it should continue the synthesis process.

SPVES_ABORT

Flag indicating the engine should stop the synthesis process, and return from the current speak call immediately.

SPVES_SKIP

Flag indicating the application has requested a real-time skip. The engine should call [ISpTTSEngineSite::GetSkipInfo](#).

SPVES_RATE

Flag indicating the application has requested a real-time rate change. The engine should call [ISpTTSEngineSite::GetRate](#).

SPVES_VOLUME

Flag indicating the application has requested a real-time volume change. The engine should call [ISpTTSEngineSite::GetVolume](#).



SPVFEATURE

SPVFEATURE lists information about the features of the phonemes and visemes.

```
typedef enum SPVFEATURE
{
    SPVFEATURE_STRESSED,
    SPVFEATURE_EMPHASIS
} SPVFEATURE;
```

Elements

SPVFEATURE_STRESSED

This flag indicates that the phoneme is stressed relative to the other phonemes within a word.

SPVFEATURE_EMPHASIS

This flag indicates that the word (of which the phoneme is a part) is emphasized relative to the other words within a sentence.



SPVISEMES

SPVISEMES lists the SAPI 5 Viseme set. This set is based on the Disney 13 Visemes. Examples given are for the SAPI 5 English Phoneme set.

```
typedef enum SPVISEMES
{
    // English examples
    //-----
    SP_VISEME_0,    // silence
    SP_VISEME_1,    // ae, ax, ah
    SP_VISEME_2,    // aa
    SP_VISEME_3,    // ao
    SP_VISEME_4,    // ey, eh, uh
    SP_VISEME_5,    // er
    SP_VISEME_6,    // y, iy, ih, ix
    SP_VISEME_7,    // w, uw
    SP_VISEME_8,    // ow
    SP_VISEME_9,    // aw
    SP_VISEME_10,   // oy
    SP_VISEME_11,   // ay
    SP_VISEME_12,   // h
    SP_VISEME_13,   // r
    SP_VISEME_14,   // l
    SP_VISEME_15,   // s, z
    SP_VISEME_16,   // sh, ch, jh, zh
    SP_VISEME_17,   // th, dh
    SP_VISEME_18,   // f, v
    SP_VISEME_19,   // d, t, n
    SP_VISEME_20,   // k, g, ng
    SP_VISEME_21,   // p, b, m
} SPVISEMES;
```

Elements

SP_VISEME_0
Silence

SP_VISEME_1
ae, ax, ah

SP_VISEME_2
aa

SP_VISEME_3
ao

SP_VISEME_4
ey, eh, uh

SP_VISEME_5
er

SP_VISEME_6
y, iy, ih, ix

SP_VISEME_7
w, uw

SP_VISEME_8
ow

SP_VISEME_9
aw

SP_VISEME_10
oy

SP_VISEME_11
ay

SP_VISEME_12
h

SP_VISEME_13
r

SP_VISEME_14
l

SP_VISEME_15
s, z

SP_VISEME_16
sh, ch, jh, zh

SP_VISEME_17
th, dh

SP_VISEME_18
f, v

SP_VISEME_19
d, t, n

SP_VISEME_20
k, g, ng

SP_VISEME_21

p, b, m



SPVLIMITS

SPVLIMITS lists the minimum and maximum values that SAPI 5 TTS engines are required to support for rate and volume adjustments.

```
typedef enum SPVLIMITS
{
    SPMIN_VOLUME,
    SPMAX_VOLUME,
    SPMIN_RATE,
    SPMAX_RATE
} SPVLIMITS;
```

Elements

SPMIN_VOLUME

Value specifying the minimum volume level.

SPMAX_VOLUME

Value specifying the maximum volume level.

SPMIN_RATE

Value specifying the minimum rate level.

SPMAX_RATE

Value specifying the maximum rate level.



SPVPRIORITY

SPVPRIORITY lists the priorities that voices can have. See [ISpVoice::SetPriority](#) and [ISpVoice::GetPriority](#) for more information.

```
typedef enum SPVPRIORITY
{
    SPVPRI_NORMAL,
    SPVPRI_ALERT,
    SPVPRI_OVER
} SPVPRIORITY;
```

Elements

SPVPRI_NORMAL

Normal priority.

SPVPRI_ALERT

Alert priority.

SPVPRI_OVER

Over priority - the voice should mix its audio with all other audio on the system with no synchronization. **SPVPRI_OVER** voices only mix their audio on Windows 2000.



SPVSKIPTYPE

SPVSKIPTYPE lists the type of item to skip in an [ISpVoice::Skip](#) call.

```
typedef enum SPVSKIPTYPE
{
    SPVST_SENTENCE
} SPVSKIPTYPE;
```

Elements

SPVST_SENTENCE

Specifies that the structure to be skipped is a sentence.



SPWAVEFORMATTYPE

SPWAVEFORMATTYPE is used in the [ISpRecognizer::GetFormat](#) method (as SPSTREAMFORMATTYPE) to request either the input format for the original audio source or, the format actually arriving at the speech engine. SAPI may be performing on the fly conversion using an [SpStreamFormatConverter](#) in-between which will cause the two formats to differ.

```
typedef enum SPWAVEFORMATTYPE
{
    SPWF_INPUT,
    SPWF_SRENGINE
} SPSTREAMFORMATTYPE;
```

Elements

SPWF_INPUT

Request for the original audio input source information.

SPWF_SRENGINE

Request for the SR engine input source information.



SPWORDINFOPT

SPWORDINFOPT lists the options for a grammar word.

```
typedef enum SPWORDINFOPT
{
    SPWIO_NONE,
    SPWIO_WANT_TEXT
} SPWORDINFOPT;
```

Elements

SPWIO_NONE

Flag specifying the SPWIO_NONE option.

SPWIO_WANT_TEXT

Flag specifying the SPWIO_WANT_TEXT option.



SPWORDPRONOUNCEABLE

See also [ISpRecoGrammar::IsPronounceable](#) and [ISpSREngine::IsPronounceable](#).

```
typedef enum SPWORDPRONOUNCEABLE
{
    SPWP_UNKNOWN_WORD_UNPRONOUNCEABLE,
    SPWP_UNKNOWN_WORD_PRONOUNCEABLE,
    SPWP_KNOWN_WORD_PRONOUNCEABLE
} SPWORDPRONOUNCEABLE;
```

Elements

SPWP_UNKNOWN_WORD_UNPRONOUNCEABLE

Specifies an unrecognized word that does not have an available pronunciation.

SPWP_UNKNOWN_WORD_PRONOUNCEABLE

Specifies an unrecognized word that has an available pronunciation.

SPWP_KNOWN_WORD_PRONOUNCEABLE

Specifies a recognized word that has an available pronunciation.



SPWORDTYPE

SPWORDTYPE lists the change state of a word/pronunciation combination in a lexicon. Using this enumeration, an engine can determine what word pronunciation changes have occurred since it last checked using the [ISpLexicon::GetGenerationChange](#) method.

```
typedef enum SPWORDTYPE
{
    eWORDTYPE_ADDED,
    eWORDTYPE_DELETED
} SPWORDTYPE;
```

Elements

eWORDTYPE_ADDED

The word has been added to the lexicon.

eWORDTYPE_DELETED

The word has been deleted from the lexicon.



Helper Functions

Helper functions are available as a convenience to programming. Since many of the procedures use the same few methods or calls in the same sequence each time, these functions are available to consolidate those standard sequences. In all cases, the functions represent nothing more than the individual steps combined into one call; no additional features have been added or removed. Programmers are free to either use these functions or include the original lines of code. The function name attempts to clearly identify the purpose of the function itself.

The following helper functions are used with SAPI 5.

Token Helpers

- [SpCreateBestObject](#)
- [SpCreateDefaultObjectFromCategoryId](#)
- [SpCreateNewToken \(by category ID\)](#)
- [SpCreateNewToken \(by token ID\)](#)
- [SpCreateNewTokenEx \(by category ID\)](#)
- [SpCreateNewTokenEx \(by token ID\)](#)
- [SpCreateObjectFromSubToken](#)
- [SpCreateObjectFromToken](#)
- [SpCreateObjectFromTokenId](#)
- [SpCreatePhoneConverter](#)
- [SpEnumTokens](#)
- [SpFindBestToken](#)
- [SpGetCategoryFromId](#)
- [SpGetDefaultTokenFromCategoryId](#)
- [SpGetDefaultTokenIdFromCategoryId](#)
- [SpGetDescription](#)
- [SpGetSubTokenFromToken](#)
- [SpGetTokenFromId](#)
- [SpGetUserDefaultUILanguage](#)
- [SpSetCommonTokenData](#)
- [SpSetDefaultTokenForCategoryId](#)
- [SpSetDefaultTokenIdForCategoryId](#)

- [SpSetDescription](#)

Other Helpers

- [SPFEI](#)
- [SPBindToFile](#)
- [SpClearEvent](#)
- [SpConvertStreamFormatEnum](#)
- [SpEventSerializeSize](#)
- [SpInitEvent](#)

Helper Classes

- [CSpDynamicString](#)
- [CSpStreamFormat](#)
- [CSpEvent](#)

UI Helper Functions

- [UI Helper Functions](#)



CSpStreamFormat

CSpStreamFormat Methods	Description
<u>CSpStreamFormat</u>	The class constructors.
<u>~CSpStreamFormat</u>	The class destructor.
<u>AssignFormat</u>	Assigns (or copies) the instance's current format to a new format (or a new stream).
<u>Clear</u>	Clears values from an instance.
<u>CopyTo</u>	Copies the instance's wave format to a new stream.
<u>Deserialize</u>	Deserializes a stream and passes back a new stream.
<u>DetachTo</u>	Makes a copy of the instance's stream and frees (or detaches) the instance's stream.
<u>FormatId</u>	Returns the instance's format ID.
<u>IsEqual</u>	Compares a specified stream format and format ID to the instance's format and format ID.
<u>ParamValidateAssignFormat</u>	Validates the format ID and wave format and creates the stream.
<u>ParamValidateCopyTo</u>	Validates the pointers for the format ID and wave format parameters.
<u>Serialize</u>	Serializes the stream.
<u>SerializeSize</u>	Determines the serialized size of the instance's stream.
<u>WaveFormatExPtr</u>	Returns the instance's wave format.



CSpStreamFormat::Constructor

The following methods may be used to construct the instance.

Initializes the class members to null values

```
CSpStreamFormat( void );
```

Parameters

None.

Formats the instance into a wave format structure. The new format and format ID are passed back from the class' public members, **m_pCoMemWaveFormatEx** and **m_guidFormatId** respectively.

```
CSpStreamFormat(  
    SPSTREAMFORMAT    eFormat,  
    HRESULT            *p hr  
);
```

Parameters

eFormat

[in] The requested stream format. Must be a valid SPSTREAMFORMAT value of SPSF_8kHz8BitMono or greater.

p hr

The return value for the method.

Return values

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.
E_INVALIDARG	Either class member <i>m_guidFormatId</i> or <i>m_pCoMemWaveFormatEx</i> is invalid or bad. Alternatively, the current format is not recognized.
FAILED(hr)	Appropriate error message.

Formats the instance according to the format structure specified.

```
CSpStreamFormat(
    const WAVEFORMATEX *pWaveFormatEx,
    HRESULT *p hr
);
```

Parameters

pWaveFormatEx

[in] Address of the WAVEFORMATEX structure containing the wave file format information.

p hr

The return value for the method.

Return values

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.



CSpStreamFormat::Destructor

CSpStreamFormat::Destructor is the class destructor. CoMemTaskFree() is used to deallocate the instance.

```
~CSpStreamFormat( void );
```

Parameters

None.

Return values

No value is returned.



CSpStreamFormat::AssignFormat

CSpStreamFormat::AssignFormat assigns (or copies) the instance's current format to a new format (or a new stream).

Converts the instance's stream format into a wave format structure.

```
HRESULT AssignFormat(  
    SPSTREAMFORMAT eFormat  
);
```

Parameters

eFormat

[in] The requested stream format. Must be a valid SPSTREAMFORMAT value of SPSF_8kHz8BitMono or greater.

Return values

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.
E_INVALIDARG	Either class member <i>m_guidFormatId</i> or <i>m_pCoMemWaveFormatEx</i> is invalid or bad. Alternatively, the current format is not recognized.
FAILED(hr)	Appropriate error message.

Converts instance's stream format into a cached format. Class

member *m_pCoMemWaveFormatEx* will be NULL if an error occurred.

```
HRESULT AssignFormat(  
    ISpStreamFormat *pStream  
);
```

Parameters

pStream
[in] An ISpStreamFormat object.

Return values

Value	Description
S_OK	Function completed successfully.
E_POINTER	Either class member <i>m_guidFormatId</i> or <i>m_pCoMemWaveFormatEx</i> is invalid or bad.

Converts instance's stream format into the specified wave format. Class member *m_guidFormatId* will be GUID_NULL if an error occurred.

```
HRESULT AssignFormat(  
    const WAVEFORMATEX *pWaveFormatEx  
);
```

Parameters

pWaveFormatEx
[in] Address of the WAVEFORMATEX structure containing the

wave file format information.

Return values

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.

Assigns the instance's format according to a reference GUID and a wave format.

```
HRESULT AssignFormat(  
    REFGUID                rguidFormatId,  
    const WAVEFORMATEX *pWaveFormatEx  
);
```

Parameters

rguidFormatId

[in] The reference ID. If specified as SPDFID_WaveFormatEx, *pWaveFormatEx* is a WAVEFORMATEX data structure. Otherwise this is set to GUID_NULL.

pWaveFormatEx

[in] If *rguidFormatId* is not set to SPDFID_WaveFormatEx, this is passed back as NULL.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	<i>pWaveFormatEx</i> is invalid or bad.

E_OUTOFMEMORY Exceeded available memory.

Assigns (or copies) the instance's stream to a specific stream.

```
HRESULT AssignFormat(  
    const CSpStreamFormat & Src  
);
```

Parameters

Src

[out] The stream to copy to.

Return values

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.



CSpStreamFormat::Clear

CSpStreamFormat::Clear clears values from an instance.

```
void Clear( void );
```

Parameters

None.

Return values

No value is returned.



CSpStreamFormat::CopyTo

CSpStreamFormat::CopyTo copies the instance's wave format to a new stream.

```
HRESULT CopyTo(  
    GUID          *pFormatId,  
    WAVEFORMATEX **ppCoMemWFEX  
);
```

Parameters

pFormatId

The new format ID based on the class member *m_guidFormatId*'s ID. If class member *m_pCoMemWaveFormatEx* is invalid, *pFormatId* is set to NULL.

ppCoMemWFEX

The new wave format. If *ppCoMemWFEX* could not be successfully created, *pFormatId* is set to zero.

This method copies the instance's stream to an existing stream.

```
HRESULT CopyTo(  
    wCSpStreamFormat &Other  
);
```

Parameters

Other

The existing stream to copy to. The stream is freed using `CoTaskMemFree()` first and the instance is then copied to it.

Return values

The return values are the same for both methods.

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.



CSpStreamFormat::Deserialize

CSpStreamFormat::Deserialize deserializes a stream and passes back a new stream.

```
HRESULT Deserialize(  
    const    BYTE    *pBuffer,  
    ULONG    *pcbUsed  
);
```

Parameters

pBuffer

[in out] Buffer containing the serialized stream. After successfully completing the methods, *pBuffer* will also be set to class member *m_pCoMemWaveFormatEx*. If unsuccessful, class member *m_guidFormatId* is set to GUID_NULL.

pcbUsed

The number of bytes used for the stream.

Return values

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.



CSpStreamFormat::DetachTo

CSpStreamFormat::DetachTo copies the instance's stream and frees (or detaches) the instance's stream. The instance's stream is set to NULL and the format ID is set to zero.

Copies the instance's stream to another existing stream.

```
void DetachTo(  
    CSpStreamFormat &Other  
);
```

Parameters

Other

The existing stream to copy to. The stream is freed using CoTaskMemFree() first and the instance is then copied to it.

Makes a copy of the instance's stream with the specified format ID and wave format.

```
void DetachTo(  
    GUID *pFormatId,  
    WAVEFORMATEX **ppCoMemWaveFormatEx  
);
```

Parameters

pFormatId

The new format ID based on the class member

m_guidFormatId.

ppCoMemWaveFormatEx

The new wave format based on the class member
m_pCoMemWaveFormatEx.

Return values

Neither method returns a value.



CSpStreamFormat::FormatId

CSpStreamFormat::FormatId returns the instance's format ID.

```
const GUID& FormatId( void );
```

Parameters

None.

Return values

The format ID of the current instance is returned by the class member *m_guidFormatId*.



CSpStreamFormat::IsEqual

CSpStreamFormat::IsEqual compares a specified stream format and format ID to the instance's format and format ID.

```
BOOL IsEqual(  
    REFGUID rguidFormatId,  
    const WAVEFORMATEX *pwfex  
)
```

Parameters

rguidFormatId

The format ID of the stream to compare with.

pwfex

The wave format of the stream to compare with.

Return values

Value	Description
TRUE	The two streams are identical.
FALSE	The reference IDs of the streams are not the same, or the streams have different contents.



CSpStreamFormat::ParamValidateAssign

CSpStreamFormat::ParamValidateAssignFormat validates the format ID and wave format. If successful, creates the stream using [AssignFormat](#).

```
HRESULT ParamValidateAssignFormat(  
    REFGUID          rguidFormatId,  
    const WAVEFORMATEX *pWaveFormatEx,  
    BOOL             fRequireWaveFormat = FALSE  
)
```

Parameters

rguidFormatId

The reference format ID of the requesting stream.

pWaveFormatEx

The wave format of the requesting stream.

fRequireWaveFormat

Optional Boolean indicating to create the stream only if the wave format is of a standard type. It is FALSE by default and may be omitted. TRUE, allows the creation of custom formats.

Return values

Value	Description
E_INVALIDARG	At least one of <i>pWaveFormatEx</i> is NULL, <i>pWaveFormatEx</i> is bad or invalid, <i>rguidFormatId</i> is not SPDFID_WaveFormatEx, or <i>pWaveFormatEx</i> is a non-standard

format and is disallowed by
fRequireWaveFormat.



CSpStreamFormat::ParamValidateCopyTo

CSpStreamFormat::ParamValidateCopyTo validates the pointers for the format ID and wave format parameters. If successful, copies the stream using [CopyTo](#).

```
HRESULT ParamValidateCopyTo(  
    GUID          *pFormatId,  
    WAVEFORMATEX **ppCoMemWFEX  
);
```

Parameters

pFormatId

The proposed pointer for the format ID.

ppCoMemWFEX

The proposed pointer for the stream.

Return values

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.
E_POINTER	Either <i>pFormatId</i> or <i>ppCoMemWFEX</i> is invalid or bad.



CSpStreamFormat::Serialize

CSpStreamFormat::Serialize serializes the stream.

```
ULONG Serialize(  
    BYTE    *pBuffer  
);
```

Parameters

pBuffer

The buffer accepting the serialized results.

Return values

The length, in bytes, of the serialized stream and stream contents. Class member *m_pCoMemWaveFormatEx* may be NULL and therefore contributes zero bytes to the total length.



CSpStreamFormat::SerializeSize

CSpStreamFormat::SerializeSize determines the serialized size of the instance's stream.

```
ULONG SerializeSize( void );
```

Parameters

None.

Return values

Returns the serialized size of the stream, in bytes.



CSpStreamFormat::WaveFormatExPtr

CSpStreamFormat::WaveFormatExPtr returns the instance's wave format.

```
const WAVEFORMATEX* WaveFormatExPtr( void );
```

Parameters

None.

Return values

The wave format of instance is returned by the class member *m_pCoMemWaveFormatEx*.



CSpDynamicString

The following methods are available.

ISpDataKey Methods	Description
CSpDynamicString	The class constructors.
~CSpDynamicString	The class destructor.
Append	Appends a string or strings to the current instance.
Attach	Attaches or assigns a string to the instance.
Clear	Clears the text from an instance.
ClearAndGrowTo	Clears the instance and reallocates it.
Compact	Compacts the instance by reallocating it.
Copy	Makes a copy of the instance.
CopyToBSTR	Allocates a system BSTR.
Detach	Detaches a string from the instance.
Length	Returns the length of the instance.
LTrim	Trims the white space starting from the left.
RTrim	Trims the white space starting from the right.
TrimBoth	Trims the white space starting from both the right and left sides.
TrimToSize	Truncates the instance to a specific size.



CSpDynamicString::Constructor

The following methods may be used to construct the instance.

Creates and sets the instance to NULL.

```
void CSpDynamicString( void )
```

Parameters

None. Initializes string to NULL.

Creates and allocates an instance of the specified number of WCHARs.

```
void CSpDynamicString(  
    ULONG    cchReserve  
);
```

Parameters

cchReserve

[in] The length of the string to allocate.

Creates and initializes and copies the source string into it.

```
void CSpDynamicString(  
    const WCHAR *pSrc  
)
```

Parameters

cchReserve

[in] Initializes the string to *pSrc*.

Creates and initializes and copies the source string into it.

```
void CSpDynamicString(  
    const char *pSrc  
)
```

Parameters

pSrc

[in] Initializes the string to *pSrc*.

Makes a copy of the CSpDynamicString class string. The current instance is CoMemtaskFree() first, if needed.

```
void CSpDynamicString(  
    const CSpDynamicString &src  
)
```

Parameters

src

[in] Initializes the string by copying the string of *src*.

Allocates an instance and copies the reference GUID into it.

```
void CSpDynamicString(  
    REFGUID rguid  
)
```

Parameters

rguid

[in] Initializes the string from the CLSID of *rguid*.



CSpDynamicString::~Destructor

CSpDynamicString::~Destructor is the class destructor. CoMemTaskFree() is used to deallocate the instance.

`~CSpDynamicString(void)`

Parameters

None



CSpDynamicString::Append

The following methods append null-terminated strings together.

Appends the null-terminated source string to the end of the instance and returns the resulting string.

```
WCHAR* Append(  
    const WCHAR *pszSrc  
)
```

Parameters

pszSrc

[in] The null-terminated source string to append.

Appends the null-terminated source string to the end of the instance and returns the resulting string. The size to append is specified.

```
WCHAR* Append(  
    const WCHAR *pszSrc,  
    const ULONG lenSrc  
)
```

Parameters

pszSrc

[in] The null-terminated source string to append.

lenSrc

[in] Size, in WCHARs, to append.

Appends up to two null-terminated source strings to the end of the instance and returns the resulting string. At least one of the strings must be non-NULL.

```
WCHAR* Append2(  
    const WCHAR *pszSrc1,  
    const WCHAR *pszSrc2  
)
```

Parameters

pszSrc1

[in] The first source string to append.

pszSrc2

[in] The second source string to append.



CSpDynamicString::Attach

CSpDynamicString::Attach attaches or assigns a string to the instance.

```
void Attach  
    WCHAR *pszSrc  
)
```

Parameters

pszSrc
[in] The source string to attach or assign.



CSpDynamicString::Clear

CSpDynamicString::Clear clears the text from an instance. The instance is set to NULL.

```
void Clear( void );
```

Parameters

None



CSpDynamicString::ClearAndGrowTo

CSpDynamicString::ClearAndGrowTo clears the existing instance and reallocates it to the specified size. The subsequent instance is returned.

```
WCHAR* ClearAndGrowTo(  
    ULONG    cch  
)
```

Parameters

cch
[in] The new size of the instance.



CSpDynamicString::Compact

CSpDynamicString::Compact compacts the instance by reallocating it to the actual number of characters it currently contains. The subsequent instance is returned.

```
WCHAR* Compact( void )
```

Parameters

None



CSpDynamicString::Copy

CSpDynamicString::Copy copies the instance.

```
WCHAR* Copy( void );
```

Parameters

None



CSpDynamicString::CopyToBSTR

CSpDynamicString::CopyToBSTR allocates a system BSTR.

```
HRESULT CopyToBSTR(  
    BSTR *pbstr  
)
```

Parameters

pbstr
[in] The source string to allocate.



CSpDynamicString::Detach

CSpDynamicString::Detach detaches and returns a string from the instance. The instance is set to NULL.

```
WCHAR* Detach( void );
```

Parameters

None



CSpDynamicString::Length

CSpDynamicString::Length returns the length of the string instance. If the instance is NULL, zero is returned.

`Length(void)`

Parameters

None



CSpDynamicString::LTrim

CSpDynamicString::LTrim trims the white space starting from the left. The subsequent instance is returned.

```
WCHAR* LTrim( void );
```

Parameters

None



CSpDynamicString::RTrim

CSpDynamicString::RTrim trims the white space starting from the right. The subsequent instance is returned.

```
WCHAR* RTrim( void );
```

Parameters

None



CSpDynamicString::TrimBoth

CSpDynamicString::TrimBoth trims the white space starting from both the right and left sides. The subsequent instance is returned.

```
WCHAR* TrimBoth( void );
```

Parameters

None



CSpDynamicString::TrimToSize

CSpDynamicString::TrimToSize truncates the instance to a specific size.

```
void TrimToSize(  
    ULONG *ulNumChars  
)
```

Parameters

ulNumChars

[in] The new size of the instance. The value must be less than less than or equal to the current size.



CSpEvent

CSpEvent Methods	Description
<u>CSpEvent</u>	The class constructor.
<u>~CSpEvent</u>	The class destructor.
<u>Clear</u>	Clears an event instance.
<u>AddrOf</u>	Returns the address of the event instance.
<u>CopyTo</u>	Copies the event instance and sets the <i>lparam</i> accordingly.
<u>GetFrom</u>	Clears the current instance and retrieves the next event from the event queue.
<u>CopyFrom</u>	Clears the current instance and copies the next event in the event queue to this instance.
<u>Detach</u>	Clears the current instance and optionally makes a copy.
<u>SerializeSize</u>	Calculates the required size of a buffer to serialize an event.
<u>Serialize</u>	Copies event instance and then serializes the new instance.
<u>Deserialize</u>	Sets the current instance to the deserialized version of the specified event.
<u>Phoneme</u>	Returns the event as a cast type of SPPHONEID.
<u>Viseme</u>	Returns the event as a cast type of SPEI_VISEME.
<u>InputWordPos</u>	Returns the event as a cast type of ULONG.
<u>InputWordLen</u>	Returns the event as a cast type of ULONG.

<u>InputSentPos</u>	Returns the event as a cast type of ULONG.
<u>InputSentLen</u>	Returns the event as a cast type of ULONG.
<u>ObjectToken</u>	Returns the event as a cast type of ISpObjectToken pointer.
<u>VoiceToken</u>	Returns the event as a cast type of SpObjectToken pointer.
<u>PersistVoiceChange</u>	Returns the state of the voice change.
<u>Object</u>	Returns the event as a cast type of an IUnknown pointer.
<u>RecoResult</u>	Returns the event as a cast type of an ISpRecoResult pointer.
<u>IsPaused</u>	Returns the pause state.
<u>IsEmulated</u>	Returns the emulation state.
<u>String</u>	Returns the string from the event's <i>IParam</i> .
<u>BookmarkName</u>	Returns the bookmark string from the event's <i>IParam</i> .
<u>RequestTypeOfUI</u>	Returns the IU type string from the event's <i>IParam</i> .
<u>RecoState</u>	Returns the event's recognition state as a cast type of SPRECOSTATE.
<u>PropertyName</u>	Returns the property name string from the event's <i>IParam</i> .
<u>PropertyNumValue</u>	Returns the property number value as cast to LONG.
<u>PropertyStringValue</u>	Returns the property string value from the event's <i>IParam</i> .
<u>Interference</u>	Returns the event's interference value from <i>IParam</i> and cast SPINTERFERENCE.
<u>EndStreamResult</u>	Returns the event's end stream

result from the event's *IParam*.

InputStreamReleased

Returns the state from releasing the stream.



CSpEvent::Constructor

CSpEvent::Constructor is used to construct the instance. The [SPEVENT](#) instance is cleared to zero.

```
CSpEvent( void );
```

Parameters

None.

Return values

None.



CSpEvent::Destructor

CSpEvent::Destructor is the class destructor and is used to clear an event instance. This method is identical to [CSpEvent Clear](#).

```
~CSpEvent( void );
```

Parameters

None.

Return values

None.

See Also

[SpClearEvent](#)



CSpEvent::AddrOf

CSpEvent::AddrOf returns the address of the event instance. No ASSERT is made in the case of an error. For additional validation, call [GetFrom](#).

```
CSpEvent* AddrOf (void)
```

Parameters

None.

Return values

Returns the address of the event instance.



CSpEvent::BookmarkName

CSpEvent::BookmarkName returns the bookmark string from the event's *IParam*. The caller must make sure the event ID is SPEI_TTS_BOOKMARK.

```
const WCHAR* BookmarkName( void ) const;
```

Parameters

None.

Return values

Returns the bookmark string from the event's *IParam*.



CSpEvent::Clear

CSpEvent::Clear clears an event instance. This method is identical to CSpEvent [~CSpEvent](#).

```
void Clear( void );
```

Parameters

None.

Return values

None.

See Also

[SpClearEvent](#)



CSpEvent::CopyFrom

CSpEvent::CopyFrom clears the current instance and copies the next event in the event queue to this instance.

```
HRESULT CopyFrom(  
    const SPEVENT *pSrcEvent  
)
```

Parameters

pSrcEvent
[in] The event of which to copy from.

Return values

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.



CSpEvent::CopyTo

CSpEvent::CopyTo copies the event instance and sets the *lparam* accordingly.

```
HRESULT CopyTo(  
    SPEVENT *pDestEvent  
);
```

Parameters

pDestEvent

The event to copy to. The member *eiParamType* is copied according to the follow parameter value:

SPET_LPARAM_IS_POINTER copies *wparam* into *lparam* if *lparam* is currently valid. On error, *pDestEvent->eEventId* is set to SPEI_UNDEFINED.

SPET_LPARAM_IS_STRING copies *lparam* into *lparam* if it is currently valid. On error, *pDestEvent->eEventId* is set to SPEI_UNDEFINED.

SPET_LPARAM_IS_TOKEN or SPET_LPARAM_IS_OBJECT calls *AddRef()* for *lparam*.

Return values

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.



CSpEvent::Deserialize

CSpEvent::Deserialize sets the current instance to the deserialized version of the specified event.

The current instance is cleared first. The original event is not modified.

```
template <class T>
HRESULT Deserialize(
    const T *pSerEvent,
    ULONG *pcbUsed = NULL
);
```

Parameters

pSerEvent

The serialized event.

pcbUsed

Optional parameter passing back the number of bytes of the instance. The default NULL does not pass back any value; TRUE, does.

Return values

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.



CSpEvent::Detach

CSpEvent::Detach clears the current instance and optionally makes a copy.

```
void Detach(  
    SPEVENT *pDestEvent = NULL  
);
```

Parameters

pDestEvent

The event structure to copy to. If NULL, no copy is made before clearing the instance.

Return values

None.



CSpEvent::EndStreamResult

CSpEvent::EndStreamResult returns the end stream result from the event's *IParam*. The caller must make sure the event ID is SPEI_END_SR_STREAM.

```
HRESULT EndStreamResult( void ) const;
```

Parameters

None.

Return values

Returns the end stream result from the event's *IParam*.



CSpEvent::GetFrom

CSpEvent::GetFrom clears the current instance and retrieves the next event from the event queue.

```
HRESULT GetFrom(  
    ISpEventSource *pEventSrc  
)
```

Parameters

pEventSrc

The event object from which to get the next event.

Return values

Value	Description
S_OK	Function completed successfully and all requested events were returned.
S_FALSE	Success, but less than the requested amount of events were returned.
E_POINTER	<i>pEventArray</i> is invalid.
FAILED(hr)	Appropriate error message.



CSpEvent::InputSentLen

CSpEvent::InputSentLen returns the event as a cast type of ULONG. The caller must make sure the event ID is SPEI_SENTENCE_BOUNDARY.

```
ULONG   InputSentLen( void ) const;
```

Parameters

None.

Return values

The event cast type of ULONG.



CSpEvent::InputSentPos

CSpEvent::InputSentPos returns the event as a cast type of ULONG. The caller must make sure the event ID is SPEI_SENTENCE_BOUNDARY.

```
ULONG   InputSentPos( void ) const;
```

Parameters

None.

Return values

The event cast type of ULONG.



CSpEvent::InputStreamReleased

CSpEvent::InputStreamReleased returns the state from releasing the stream. The caller must make sure the event ID is SPEI_END_SR_STREAM.

See also [SPENDSRSTREAMFLAGS](#) for more information.

```
BOOL InputStreamReleased( void ) const;
```

Parameters

None.

Return values

Returns the state from releasing the stream. TRUE indicates the stream was successfully released; FALSE otherwise.



CSpEvent::InputWordLen

CSpEvent::InputWordLen returns the event as a cast type of ULONG. The caller must make sure the event ID is SPEI_WORD_BOUNDARY.

```
ULONG InputWordLen( void ) const;
```

Parameters

None.

Return values

The event cast type of ULONG.



CSpEvent::InputWordPos

CSpEvent::InputWordPos returns the event as a cast type of ULONG. The caller must make sure the event ID is SPEI_WORD_BOUNDARY.

```
ULONG InputWordPos( void ) const;
```

Parameters

None.

Return values

The event cast type of ULONG.



CSpEvent::Interference

CSpEvent::Interference returns the event's interference value from *IParam* and cast SPINTERFERENCE. The caller must make sure the event ID is SPEI_INTERFERENCE.

[SPINTERFERENCE](#) Interference(void) const;

Parameters

None.

Return values

Returns the interference value as cast SPINTERFERENCE.



CSpEvent::IsEmulated

CSpEvent::IsEmulated returns the emulation state. The caller must make sure the event ID is SPEI_RECOGNITION.

```
BOOL IsEmulated( void );
```

Parameters

None.

Return values

The event cast type of SPPHONEID.



CSpEvent::IsPaused

CSpEvent::IsPaused returns the pause state. The caller must make sure the event ID is either SPEI_RECOGNITION or SPEI_SR_BOOKMARK.

```
BOOL IsPaused( void )
```

Parameters

None.

Return values

Returns the pause state.



CSpEvent::Object

CSpEvent::Object returns the event as a cast type of an IUnknown pointer. The caller must make sure the event ID is SPET_LPARAM_IS_OBJECT.

```
IUnknown* Object( void ) const;
```

Parameters

None.

Return values

The event cast type of an IUnknown pointer.



CSpEvent::ObjectToken

CSpEvent::ObjectToken returns the event as a cast type of [ISpObjectToken](#) pointer. The caller must make sure the event type is SPET_LPARAM_IS_TOKEN.

```
ISpObjectToken* ObjectToken( void ) const;
```

Parameters

None.

Return values

The event cast type of [ISpObjectToken](#) pointer.



CSpEvent::PersistVoiceChange

CSpEvent::PersistVoiceChange returns the state of the voice change. The caller must make sure the event ID is SPEI_VOICE_CHANGE.

```
BOOL PersistVoiceChange( void ) const;
```

Parameters

None.

Return values

The event cast type of BOOL.



CSpEvent::Phoneme

CSpEvent::Phoneme returns the event as a cast type of SPPHONEID. The caller must make sure the event ID is SPEI_PHONEME.

```
SPPHONEID Phoneme( void ) const;
```

Parameters

None.

Return values

The event cast type of SPPHONEID.



CSpEvent::PropertyName

CSpEvent::PropertyName returns the property name string from the event's *IParam*. The caller must make sure the event ID is either SPEI_PROPERTY_NUM_CHANGE and event type is SPET_LPARAM_IS_STRING, or event ID is SPEI_PROPERTY_STRING_CHANGE and event type is SPET_LPARAM_IS_POINTER.

```
const WCHAR* PropertyName( void ) const;
```

Parameters

None.

Return values

Returns the property name string from the event's *IParam*.



CSpEvent::PropertyNumValue

CSpEvent::PropertyNumValue returns the property number value as cast to LONG. The caller must make sure the event ID is SPEI_PROPERTY_NUM_CHANGE.

```
const LONG PropertyNumValue( void ) const;
```

Parameters

None.

Return values

Returns the property number value as cast to LONG.



CSpEvent::PropertyStringValue

CSpEvent::PropertyStringValue returns the property string value from the event's *IParam*. The caller must make sure the event ID is SPEI_PROPERTY_STRING_CHANGE.

```
const WCHAR* PropertyStringValue( void ) const;
```

Parameters

None.

Return values

Returns the property string value from the event's *IParam*.



CSpEvent::RecoResult

CSpEvent::RecoResult returns the event as a cast type of an ISpRecoResult pointer. The caller must make sure the event ID is SPEI_RECOGNITION, SPEI_FALSE_RECOGNITION, or SPEI_HYPOTHESIS.

```
ISpRecoResult* RecoResult( void ) const;
```

Parameters

None.

Return values

The event cast type of an ISpRecoResult pointer.



CSpEvent::RecoState

CSpEvent::RecoState returns the event's recognition state as a cast type of SPRECOSTATE. The caller must make sure the event ID is SPEI_RECO_STATE_CHANGE.

```
SPRECOSTATE RecoState( void ) const;
```

Parameters

None.

Return values

Returns the event's recognition state as a cast type of SPRECOSTATE.



CSpEvent::RequestTypeOfUI

CSpEvent::RequestTypeOfUI returns the UI type string from the event's *IParam*. The caller must make sure the event ID is SPEI_REQUEST_UI.

```
const WCHAR* RequestTypeOfUI( void ) const;
```

Parameters

None.

Return values

Returns the UI type string from the event's *IParam*.



CSpEvent::Serialize

CSpEvent::Serialize copies an event instance and then serializes the new instance. The new instance must be based on [SPSERIALIZEDEVENT](#) or [SPSERIALIZEDEVENT64](#)

```
void Serialize(  
    T      *pSerEvent  
);
```

Parameters

pSerEvent

[out] The event for the passed back serialization. Member *elParamType* must not be SPET_LPARAM_IS_OBJECT.

Return values

None, however member *SerializedIParam* is set to the size of the serialized structure.

Copies an existing event instance and then serializes the new instance.

```
HRESULT Serialize(  
    T      **ppCoMemSerEvent,  
    ULONG  *pcbSerEvent  
);
```

Parameters

ppCoMemSerEvent

[out] The event for the passed back serialization. It is allocated and serialized. When no longer required, it must be manually freed with CoMemTaskFree().

pcbSerEvent

[out] The number of bytes allocated for the serialization. On an error, it will be zero.

Return values

Value	Description
S_OK	Serialization completed successfully.
E_OUTOFMEMORY	Exceeded available memory.



CSpEvent::SerializeSize

CSpEvent::SerializeSize calculates the required size of a buffer to serialize an event. The instance must be either [SPSERIALIZEDEVENT](#) or [SPSERIALIZEDEVENT64](#).

```
ULONG SerializeSize( void );
```

Parameters

None.

Return values

Size, in bytes, required to serialize the event.

Remarks

Due to a compiler issue, CSpEvent::SerializeSize may not be used with Visual Studio 6. Use [SpEventSerializeSize](#) instead.



CSpEvent::String

CSpEvent::String returns the string from the event's *IParam*. The caller must make sure the event type is SPET_LPARAM_IS_STRING.

```
const WCHAR* String( void ) const;
```

Parameters

None.

Return values

Returns the string from the event's *IParam*.



CSpEvent::Viseme

CSpEvent::Viseme returns the event as a cast type of SPEI_VISEME. The caller must make sure the event ID is SPEI_VISEME.

```
SPVISEMES    Viseme( void ) const;
```

Parameters

None.

Return values

The event cast type of SPVISEMES.



CSpEvent::VoiceToken

CSpEvent::VoiceToken returns the event as a cast type of [ISpObjectToken](#) pointer. The caller must make sure the event ID is SPEI_VOICE_CHANGE. This is an additional check for the helper [ObjectToken](#).

```
ISpObjectToken* VoiceToken( void ) const;
```

Parameters

None.

Return values

The event cast type of [ISpObjectToken](#) pointer.



UI Helper Functions

UI Helper Function Name	Description
<u>SpAddTokenToComboBox</u>	Adds the description of a token to a combo box.
<u>SpAddTokenToListBox</u>	Adds the description of a token to a list box.
<u>SpDeleteCurSelComboBoxToken</u>	Deletes a token as specified by the index of the currently selected item in a combo box
<u>SpDeleteCurSelListBoxToken</u>	Deletes a token as specified by the index of the currently selected item in a list box.
<u>SpDestroyTokenComboBox</u>	Destroys the tokens in a combo box.
<u>SpDestroyTokenListBox</u>	Destroys the tokens in a list box.
<u>SpGetComboBoxToken</u>	Returns a pointer to a token as specified by the index in a combo box.
<u>SpGetCurSelComboBoxToken</u>	Returns a pointer to a token as specified by the index of the currently selected item in a combo box.
<u>SpGetCurSelListBoxToken</u>	Returns a pointer to a token as specified by the index of the currently selected item in a list box.
<u>SpGetListBoxToken</u>	Returns a pointer to a token as specified by the index in a list box.

<u>SpInitTokenComboBox</u>	Initializes a combo box with the description of tokens from a specified category.
<u>SpInitTokenListBox</u>	Initializes a list box with the description of tokens from a specified category.
<u>SpUpdateCurSelComboBoxToken</u>	Updates the corresponding token of the item as specified by the index of the currently selected item in a combo box.
<u>SpUpdateCurSelListBoxToken</u>	Updates the corresponding token of the item as specified by the index of the currently selected item in a list box.

Microsoft Speech SDK

SAPI 5.1



SpAddTokenToComboBox

SpAddTokenToComboBox adds the description of a token to a combo box.

Found in: spuihelp.h

```
HRESULT SpAddTokenToComboBox(  
    HWND hwnd  
    ISpObjectToken* pToken  
);
```

Parameters

hwnd

[in] The handle to the combo box.

pToken

[in] Pointer to the token to be added.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Microsoft Speech SDK

SAPI 5.1



SpAddTokenToListBox

SpAddTokenToListBox adds the description of a token to a list box.

Found in: spuihelp.h

```
HRESULT SpAddTokenToListBox(  
    HWND hwnd  
    ISpObjectToken* pToken  
);
```

Parameters

hwnd

[in] The handle to the list box.

pToken

[in] Pointer to the token to be added.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Microsoft Speech SDK

SAPI 5.1



SpDeleteCurSelComboBoxToken

SpDeleteCurSelComboBoxToken deletes a token specified by the index of the currently selected item in a combo box.

Found in: spuihelp.h

```
HRESULT SpDeleteCurSelComboBoxToken(  
    HWND hwnd  
);
```

Parameters

hwnd

[in] The handle to the combo box.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Microsoft Speech SDK

SAPI 5.1



SpDeleteCurSelListBoxToken

SpDeleteCurSelListBoxToken deletes a token specified by the index of the currently selected item in a list box.

Found in: spuihelp.h

```
HRESULT SpDeleteCurSelListBoxToken(  
    HWND hwnd  
);
```

Parameters

hwnd

[in] The handle to the list box.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Microsoft Speech SDK

SAPI 5.1



SpDestroyTokenComboBox

SpDestroyTokenComboBox destroys the tokens in a combo box.

Found in: spuihelp.h

```
void SpDestroyTokenComboBox(  
    HWND hwnd  
);
```

Parameters

hwnd

[in] The handle to the combo box that contains the tokens to be destroyed.

Return values

None

Microsoft Speech SDK

SAPI 5.1



SpDestroyTokenListBox

SpDestroyTokenListBox destroys the tokens in a list box.

Found in: spuihelp.h

```
void SpDestroyTokenListBox(  
    HWND hwnd  
);
```

Parameters

hwnd

[in] The handle to the list box that contains the tokens to be destroyed.

Return values

None

Microsoft Speech SDK

SAPI 5.1



SpGetComboBoxToken

SpGetComboBoxToken returns a pointer to a token specified by the index in a combo box.

Found in: spuihelp.h

```
ISpObjectToken* SpGetComboBoxToken(  
    HWND          hwnd,  
    WPARAM       index  
);
```

Parameters

hwnd

[in] The handle to the combo box.

index

[in] Specifies the index of the member to be returned.

Return values

Returns a pointer to an ISpObjectToken type.

Microsoft Speech SDK

SAPI 5.1



SpGetCurSelComboBoxToken

SpGetCurSelComboBoxToken returns a pointer to a token specified by the index of the currently selected item in a combo box.

Found in: spuihelp.h

```
ISpObjectToken* SpGetCurSelComboBoxToken(  
    HWND hwnd  
);
```

Parameters

hwnd

[in] The handle to the combo box.

Return values

Returns a pointer to an ISpObjectToken type.

Microsoft Speech SDK

SAPI 5.1



SpGetCurSelListBoxToken

SpGetCurSelListBoxToken returns a pointer to a token specified by the index of the currently selected item in a list box.

Found in: spuihelp.h

```
ISpObjectToken* SpGetCurSelListBoxToken(  
    HWND hwnd  
);
```

Parameters

hwnd

[in] The handle to the list box.

Return values

Returns a pointer to an ISpObjectToken type.

Microsoft Speech SDK

SAPI 5.1



SpGetListBoxToken

SpGetListBoxToken returns a pointer to a token as specified by the index in a list box.

Found in: spuihelp.h

```
ISpObjectToken* SpGetListBoxToken(  
    HWND          hwnd,  
    WPARAM       index  
);
```

Parameters

hwnd

[in] The handle to the list box.

index

[in] Specifies the index of the member to be returned.

Return values

Returns a pointer to an ISpObjectToken type.

Microsoft Speech SDK

SAPI 5.1



SpInitTokenComboBox

SpInitTokenComboBox initializes a combo box with the description of tokens from a specified category.

Found in: spuihelp.h

```
HRESULT SpInitTokenComboBox(  
    HWND          hwnd,  
    const WCHAR*  pszCatName,  
    const WCHAR*  pszRequiredAttrib = NULL,  
    const WCHAR*  pszOptionalAttrib = NULL  
);
```

Parameters

hwnd

[in] The handle of the combo box that is to be initialized with tokens.

pszCatName

[in] The category of the tokens to initialize the combo box.

pszRequiredAttrib

[in] Required attributes.

pszOptionalAttrib

[in] Optional attributes.

Return values

Value	Description
S_OK	Function completed successfully.

FAILED (hr)

Appropriate error message.

Microsoft Speech SDK

SAPI 5.1



SpInitTokenListBox

SpInitTokenListBox initializes a list box with the description of tokens from a specified category.

Found in: spuihelp.h

```
HRESULT SpInitTokenListBox(  
    HWND          hwnd,  
    const WCHAR*  pszCatName,  
    const WCHAR*  pszRequiredAttrib = NULL,  
    const WCHAR*  pszOptionalAttrib = NULL  
);
```

Parameters

hwnd

[in] The handle of the list box that is to be initialized with tokens.

pszCatName

[in] The category of the tokens to initialize the list box.

pszRequiredAttrib

[in] Required attributes.

pszOptionalAttrib

[in] Optional attributes.

Return values

Value	Description
S_OK	Function completed successfully.

FAILED (hr)

Appropriate error message.

Microsoft Speech SDK

SAPI 5.1



SpUpdateCurSelComboBoxToken

SpUpdateCurSelComboBoxToken updates the corresponding token of the item specified by the index of the currently selected item in a combo box.

Found in: spuihelp.h

```
HRESULT SpUpdateCurSelComboBoxToken(  
    HWND hwnd  
);
```

Parameters

hwnd

[in] The handle to the combo box.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Microsoft Speech SDK

SAPI 5.1



SpUpdateCurSelListBoxToken

SpUpdateCurSelListBoxToken updates the corresponding token of the item specified by the index of the currently selected item in a list box.

Found in: spuihelp.h

```
HRESULT SpUpdateCurSelListBoxToken(  
    HWND hwnd  
);
```

Parameters

hwnd

[in] The handle to the list box.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpCreateBestObject

SpCreateBestObject creates an object from tokens that best match a set of criteria from required and optional attributes.

Found in: sphelper.h

```
SpCreateBestObject(  
    const WCHAR    *pszCategoryId,  
    const WCHAR    *pszReqAttribs,  
    const WCHAR    *pszOptAttribs,  
    T              **ppObject,  
    IUnknown       *IUnknown = NULL,  
    DWORD          dwClsCtx = CLSCTX_ALL,  
);
```

Parameters

pszCategoryId

[in] The null-terminated string category ID on which to base the new token.

pszReqAttribs

[in] The null-terminated string of required attributes for the token.

pszOptAttribs

[in] The null-terminated string of optional attributes for the token.

ppObject

[out, iid_is(riid)] Address of pointer variable that receives the interface pointer requested in riid. Upon successful return, *ppObject* contains the requested interface pointer. If the object does not support the interface specified in riid, the

implementation must set *ppObject* to NULL.

IUnknown

[in] Optional parameter used for creating aggregate objects. *pUnkOuter* is the data for the object. If not specified, the value defaults to NULL.

dwClsCtxt

[in] Context in which the code that manages the newly created object will run. It should be one of the following values. If not specified, the value defaults to CLSCTX_ALL.

- CLSCTX_INPROC_SERVER
- CLSCTX_INPROC_HANDLER
- CLSCTX_LOCAL_SERVER
- CLSCTX_REMOTE_SERVER

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpCreateDefaultObjectFromCategoryId

SpCreateDefaultObjectFromCategoryId creates the object instance from the default object token of a specified category.

Found in: sphelper.h

```
SpCreateDefaultObjectFromCategoryId(  
    const WCHAR    *pszCategoryId,  
    T              **ppObject,  
    IUnknown       *pUnkOuter = NULL,  
    DWORD          dwClsCtx    = CLSCTX_ALL  
);
```

Parameters

pszCategoryId

[in] The type of object token to create.

ppObject

[out] The object being created.

pUnkOuter

[in] Optional parameter used for creating aggregate objects. *pUnkOuter* is the data for the object. If not specified, the value defaults to NULL.

dwClsCtx

[in] The type of aggregate object being created. If *pUnkOuter* is not NULL, this must be supplied. If not specified otherwise, the value defaults to all object types.

Return values

Value	Description
-------	-------------

S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Example

```
CComPtr<ISpAudio> cpAudio;
```

```
hr = SpCreateDefaultObjectFromCategoryId(SPCAT_AUDIOIN, &cpA
```



SpCreateNewToken (by CategoryId)

SpCreateNewToken creates a token coercively. The token is created with the specified name, if provided. Otherwise it will automatically generate both a key name and name.

Found in: sphelper.h

```
SpCreateNewToken(  
    const WCHAR          *pszCategoryId,  
    const WCHAR          *pszTokenKeyName,  
    ISpObjectToken      **ppToken  
);
```

Parameters

pszCategoryId

[in] The null-terminated string indicating the category ID.

pszTokenKeyName

[in out] The token name being created. If NULL, a unique token key name will be generated. If a name is provided it will append Tokens before it.

ppToken

[out] The newly created token. The token will be created, if one does not currently exist.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.

Example

The following code snippet illustrates the use of [SpCreateNewToken \(by Category Id\)](#) with the [SPCAT_RECOPROFILE](#) category.

```
HRESULT hr = S_OK;

// create a new recognition profile
hr = SpCreateNewToken(SPCAT_RECOPROFILES, NULL, &cpObjec
// Check hr
```



SpCreateNewToken (by TokenId)

SpCreateNewToken creates a token forcefully.

Found in: sphelper.h

```
inline HRESULT SpCreateNewToken(  
    const WCHAR          *pszTokenId,  
    ISpObjectToken      **ppToken  
);
```

Parameters

pszTokenId

[in] The null-terminated string indicating the token IDw.

ppToken

[out] The token being created. It is created with CoCreateInstance and must be manually freed when no longer required.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpCreateNewTokenEx (by CategoryId)

SpCreateNewTokenEx creates a new token using the category ID and a token name. It subsequently completes the information with the helper function [SpSetCommonTokenData](#).

Found in: sphelper.h

```
SpCreateNewTokenEx(  
    const WCHAR      *pszCategoryId,  
    const WCHAR      *pszTokenKeyName,  
    const CLSID      *pclsid,  
    const WCHAR      *pszLangIndependentName,  
    LANGID           langid,  
    const WCHAR      *pszLangDependentName,  
    ISpObjectToken **ppToken,  
    ISpDataKey      **ppDataKeyAttribs  
);
```

Parameters

pszCategoryId

[in] The null-terminated string indicating the category ID.

pszTokenKeyName

[in out] The token name being created. If NULL, a unique token key name will be generated. If a name is provided it will append "Tokens" before it.

pclsid

[in] Sets the token's CLSID, if specified.

pszLangIndependentName

[in] Sets the null-terminated token language dependent name.

langid

[in] The language ID of the word. May be zero to indicate the word can be of any LANGID.

pszLangDependentName

[in] Sets the null-terminated token language dependent name.

ppToken

[out] The newly created token. The token is created if it currently does not exist.

ppDataKeyAttribs

[in] Opens the attributes key. The key is created if it does not currently exist. May be NULL if not needed.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpCreateNewTokenEx (by TokenId)

SpCreateNewTokenEx creates a new token using the category ID and a token name. It subsequently completes the information with the helper function [SpSetCommonTokenData](#).

Found in: sphelper.h

```
SpCreateNewTokenEx(  
    const WCHAR      *pszTokenId,  
    const CLSID      *pclsid,  
    const WCHAR      *pszLangIndependentName,  
    LANGID           langid,  
    const WCHAR      *pszLangDependentName,  
    ISpObjectToken **ppToken,  
    ISpDataKey      **ppDataKeyAttribs  
);
```

Parameters

pszTokenId

[in] The null-terminated string indicating the token ID.

pclsid

[in] Sets the token's CLSID, if specified.

pszLangIndependentName

[in] Sets the null-terminated token language dependent name.

langid

[in] The language ID of the word. May be zero to indicate that the word can be of any LANGID.

pszLangDependentName

[in] Sets the null-terminated token language dependent name.

ppToken

[out] The newly created token. The token is created if it does not currently exist.

ppDataKeyAttribs

[in] Opens the attributes key. The key is created if it does not currently exist. May be NULL if not needed.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpCreateObjectFromSubToken

SpCreateObjectFromSubToken creates an object from a specified subtoken.

Found in: spddkhlp.h

```
SpCreateObjectFromSubToken(  
    ISpObjectToken *pToken,  
    const WCHAR *pszSubKeyName,  
    T **ppObject,  
    IUnknown pUnkOuter = NULL,  
    DWORD *dwClsCtxt = CLSCTX_ALL  
);
```

Parameters

pToken

[in] Address of a pointer to an ISpObjectToken object containing the information associated with the tokens being added.

pszSubKeyName

[in] Address of a null-terminated string specifying the name of the subkey of the *pToken*'s corresponding datakey to open.

ppObject

[out, iid_is(riid)] Address of pointer variable that receives the interface pointer requested in riid. Upon successful return, *ppObject* contains the requested interface pointer. If the object does not support the interface specified in riid, the implementation must set *ppObject* to NULL.

IUnknown

[in] Optional parameter used for creating aggregate objects.

pUnkOuter is the data for the object. If not specified, the value defaults to NULL.

dwClsCtxt

[in] Context in which the code that manages the newly created object will run. It should be one of the following values. If not specified, the value defaults to CLSCTX_ALL.

- CLSCTX_INPROC_SERVER
- CLSCTX_INPROC_HANDLER
- CLSCTX_LOCAL_SERVER
- CLSCTX_REMOTE_SERVER

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpCreateObjectFromToken

SpCreateObjectFromToken creates an object instance from a specified object token.

Found in: sphelper.h

```
SpCreateObjectFromToken(  
    ISpObjectToken *pToken,  
    T **ppObject,  
    IUnknown *pUnkOuter = NULL,  
    DWORD dwClsCtx = CLSCTX_ALL  
);
```

Parameters

pToken

[in] The type of object token to create.

ppObject

[out] the object instance being created.

pUnkOuter

[in] Optional parameter used for creating aggregate objects. *pUnkOuter* is the data for the object. If not specified, the value defaults to NULL.

dwClsCtx

[in] The type of aggregate object being created. If *pUnkOuter* is not NULL, this must be supplied. If not specified otherwise, the value defaults to all object types.

Return values

Value	Description
-------	-------------

S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpCreateObjectFromTokenId

SpCreateObjectFromTokenId creates an object instance from a specified object token ID.

Found in: sphelper.h

```
SpCreateObjectFromTokenId(  
    const WCHAR    *pszTokenId,  
    T              **ppObject,  
    IUnknown       *pUnkOuter = NULL,  
    DWORD          dwClsCtxt = CLSCTX_ALL  
);
```

Parameters

pszTokenId

[in] The type of object token to create.

ppObject

[out] The object being created.

pUnkOuter

[in] Optional parameter used for creating aggregate objects. *pUnkOuter* is the data for the object. If not specified, the value defaults to NULL.

dwClsCtxt

[in] The type of aggregate object being created. If *pUnkOuter* is not NULL, this must be supplied. If not specified otherwise, the value defaults to all object types.

Return values

Value	Description
-------	-------------

S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpCreatePhoneConverter

SpCreatePhoneConverter creates a directly converted phone. Calls the helper function [SpCreateBestObject](#) with the category of SPCAT_PHONECONVERTERS.

Found in: sphelper.h

```
SpCreatePhoneConverter(  
    LANGID          *langid,  
    const WCHAR     *pszReqAttribs,  
    const WCHAR     *pszOptAttribs,  
    ISpPhoneConverter ppPhoneConverter  
);
```

Parameters

langid

[in] The language ID of the word. May not be NULL or zero.

pszReqAttribs

[in] The null-terminated string of required attributes for the token.

pszOptAttribs

[in] The null-terminated string of optional attributes for the token.

ppPhoneConverter

[out] The converted phone interface.

Return values

Value	Description
S_OK	Function completed

	successfully.
E_INVALIDARG	<i>langid</i> equals zero.
FAILED (hr)	Appropriate error message.



SpEnumTokens

SpEnumTokens enumerates the tokens for the specified category.

Found in: sphelper.h

```
inline HRESULT SpEnumTokens(  
    const WCHAR          *pszCategoryId,  
    const WCHAR          *pszReqAttribs,  
    const WCHAR          *pszOptAttribs,  
    IEnumSpObjectTokens **ppEnum  
);
```

Parameters

pszCategoryId

[in] The null-terminated string category ID on which to base the enumerations.

pszReqAttribs

[in] The null-terminated string of the required attributes for the token.

pszOptAttribs

[in] The null-terminated string of the optional attributes for the token. The order in which the tokens are listed in *ppEnum* is based on the order that they match *pszOptAttribs*.

ppEnum

[out] The enumerated list of tokens found.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpFindBestToken

SpFindBestToken enumerates the token category and finds the single best match (if any) based on the required and optional attributes.

Found in: sphelper.h

```
SpFindBestToken(  
    const WCHAR          *pszCategoryId,  
    const WCHAR          *pszReqAttribs,  
    const WCHAR          *pszOptAttribs,  
    ISpObjectToken      **ppObjectToken  
);
```

Parameters

pszCategoryId

[in] The null-terminated string category ID on which to base the enumerations.

pszReqAttribs

[in] The null-terminated string of the required attributes for the token.

pszOptAttribs

[in] The null-terminated string of the optional attributes for the token. The order in which the tokens are listed in *ppObjectToken* is based on the order they match *pszOptAttribs*.

ppObjectToken

[out] The single best matched token found.

Return values

Value	Description
S_OK	Function completed successfully.
SPERR_NOT_FOUND	No items match the given attributes.
FAILED (hr)	Appropriate error message.



SpGetCategoryFromId

SpGetCategoryFromId creates an object of CLSID_SpObjectTokenCategory. This function assists in locating and creating an object token category without having to search or change the registry directly.

Found in: sphelper.h

```
SpGetCategoryFromId(  
    const WCHAR          *pszCategoryId,  
    ISpObjectToken      **ppCategory,  
    BOOL                 *fCreateIfNotExist = FALSE  
);
```

Parameters

pszCategoryId

[in] The string indicating the CategoryId.

ppCategory

[out] The token being created. It is created with CoCreateInstance and must be manually freed when no longer required.

fCreateIfNotExist

[in] An optional parameter allowing the object to be created if one does not currently exist. The default is FALSE unless otherwise specified.

Return values

Value	Description
S_OK	Function completed successfully.

FAILED (hr)

Appropriate error message.



SpGetDefaultTokenFromCategoryId

SpGetDefaultTokenFromCategoryId gets the default token for the specified category ID.

Found in: sphelper.h

```
inline HRESULT SpGetDefaultTokenFromCategoryId(  
    const WCHAR          *pszCategoryId,  
    ISpObjectToken      **ppToken,  
    BOOL                 *fCreateCategoryIfNotExist = TRUE  
);
```

Parameters

pszCategoryId

[in] The null-terminated string for the category ID.

ppToken

[out] The default token for the category ID.

fCreateCategoryIfNotExist

[in] An optional parameter allowing the category to be created if one does not currently exist. By default, the value is TRUE unless otherwise indicated.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpGetDefaultTokenIdFromCategoryId

SpGetDefaultTokenIdFromCategoryId gets the default token ID for the specified category.

Found in: sphelper.h

```
inline HRESULT SpGetDefaultTokenIdFromCategoryId(  
    const WCHAR *pszCategoryId,  
    WCHAR **ppszTokenId  
);
```

Parameters

pszCategoryId

[in] The null-terminated string indicating the category ID.

ppszTokenId

[out] The null-terminated string ID of the owning default token.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpGetDescription

SpGetDescription passes back the textual description associated with the specified token.

Found in: sphelper.h

```
SpGetDescription(  
    ISpObjectToken *pObjToken,  
    WCHAR **ppszDescription,  
    LANGID *Language = SpGetUserDefaultUILanguage()  
);
```

Parameters

pObjToken

[in] The object token of the target resource.

ppszDescription

[out] A null-terminated string containing the resource description.

Language

[in] The language ID for the resource. Language is optional and if omitted the default language will be used.

Return values

This helper function calls [ISpDataKey::GetStringValue](#). See those return values.



SpGetSubTokenFromToken

SpGetSubTokenFromToken creates a subtoken from a token.

Found in: spddkhlp.h

```
SpGetSubTokenFromToken(  
    ISpObjectToken *pToken,  
    const WCHAR *pszSubKeyName,  
    ISpObjectToken **ppToken,  
    BOOL fCreateIfNotExist = FALSE  
);
```

Parameters

pToken

[in] The object token from which to create the subtoken. If not present, this token will be created if *fCreateIfNotExist* is TRUE.

pszSubKeyName

[out] The name of the subtoken to use.

ppToken

[in] The newly created subtoken.

fCreateIfNotExist

[in] Optional Boolean indicating that the token is to be created if one does not currently exist. TRUE allows the creation. The default value FALSE does not.

Return values

Value	Description
S_OK	Function completed successfully.

E_POINTER	At least one of the pointers <i>pToken</i> , <i>pszSubKeyName</i> , or <i>ppToken</i> is invalid or bad.
FAILED (hr)	Appropriate error message.



SpGetTokenFromId

SpGetTokenFromId creates an object token of CLSID_SpObjectToken. This function assists in locating and creating an object token without having to search or change the registry directly.

Found in: sphelper.h

```
inline HRESULT SpGetTokenFromId(  
    const WCHAR          *pszTokenId,  
    ISpObjectToken      **ppToken,  
    BOOL                 *fCreateIfNotExist = FALSE  
);
```

Parameters

pszTokenId

[in] The null-terminated string indicating the token ID.

ppToken

[out] The token being created. It is created with CoCreateInstance and must be manually freed when no longer required.

fCreateIfNotExist

[in] An optional parameter allowing the object to be created if one does not currently exist. The default is FALSE unless otherwise specified.

Return values

Value	Description
S_OK	Function completed successfully.

FAILED (hr)

Appropriate error message.



SpGetUserDefaultUILanguage

SpGetUserDefaultUILanguage returns the default user interface language.

Found in: sphelper.h

```
inline LANGID SpGetUserDefaultUILanguage ( void );
```

Parameters

None.

Return values

Returns the default language. If the attempt fails to find the default language from the specific operating system, the default language from SAPI is returned instead.



SpSetCommonTokenData

SpSetCommonTokenData fills in the token data with the information provided by the parameters. Unused parameters must be NULL.

Found in: sphelper.h

```
SpSetCommonTokenData(  
    ISpObjectToken *pToken,  
    const CLSID *pclsid,  
    const WCHAR *pszLangIndependentName,  
    LANGID langid,  
    const WCHAR *pszLangDependentName,  
    ISpDataKey **ppDataKeyAttribs  
);
```

Parameters

pToken

[in] Address of a pointer to an ISpObjectToken object containing the information associated with the tokens being added.

pclsid

[in] Sets the token's CLSID, if specified.

pszLangIndependentName

[in] Sets the token's language independent name.

langid

[in] The language ID of the word. May be zero to indicate the word can be of any LANGID.

pszLangDependentName

[in] Sets the token's language dependent name.

ppDataKeyAttribs

[in] Opens the attributes key. The key is created if it does not currently exist. May be NULL if not needed.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpSetDefaultTokenForCategoryId

SpSetDefaultTokenForCategoryId sets the default token for the specified category ID.

Found in: sphelper.h

```
inline HRESULT SpSetDefaultTokenForCategoryId(  
    const WCHAR      *pszCategoryId,  
    ISpObjectToken *pToken  
);
```

Parameters

pszCategoryId

[in] The null-terminated string for the category ID.

pToken

[in] The token to be set as the default for the category ID.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SpSetDefaultTokenIdForCategoryId

SpSetDefaultTokenIdForCategoryId sets a specific token ID as the default for the specified category ID.

Found in: sphelper.h

```
inline HRESULT SpSetDefaultTokenIdForCategoryId(  
    const WCHAR    *pszCategoryId,  
    const WCHAR    *pszTokenId  
);
```

Parameters

pszCategoryId

[in] The null-terminated token ID string.

pszTokenId

[in] The null-terminated token ID that will be set as default token ID for this specific category.

Return values

Value	Description
S_OK	Function completed successfully.
FAILED (hr)	Appropriate error message.



SPFEI

SPFEI casts a specified value into a 64-bit type.

For a list of the supported SPEI_ord event types, see [SPEVENTENUM](#)

Found in: sapi.idl

```
SPFEI(  
    void    *SPEI_ord  
);
```

Parameters

SPEI_ord

[in, out] The value to re-cast. The new value is passed back.

Return values

No error code is returned.

Related Helper Macros

Macro	Description
SPFEI_FLAGCHECK	Retrieves reserved flags for the event interest enum.
SPEI_ALL_EVENTS	Retrieves all possible events flags.
SPFEI_ALL_TTS_EVENTS	Retrieves all possible TTS event flags.
SPFEI_ALL_SR_EVENTS	Retrieves all possible SR event flags.

Remarks

The SPFEI_FLAGCHECK macro retrieves the flags that must always be included in any event interest. SAPI uses the SPFEI_FLAGCHECK macro to help developers avoid mistakes by

using event interest related methods (see [ISpEventSource::SetInterest](#) and [ISpRecoContext::SetVoicePurgeEvent](#)) with the actual enumeration instead of the SPFEI() macro.

Example

Here is an example of the SPFEI macro used to set SR event interest

```
hr = g_cpRecoCtxt->SetInterest(SPFEI(SPEI_RECOGNITION), SPFEI_
```

Here is an example of the SPFEI_ALL_SR_EVENTS macro used to set SR event interest in all events. Note this is an example and not recommended practice.

```
hr = g_cpRecoCtxt->SetInterest(SPFEI_ALL_SR_EVENTS, SPFEI_AL
```



SPBindToFile

SPBindToFile binds the audio stream to the specified file.

Found in: sphelper.h

```
SPBindToFile(  
    LPCWSTR                pFileName,  
    SPFILEMODE             eMode,  
    ISpStream              **ppStream,  
    const GUID             *pFormatId = NULL,  
    const WAVEFORMATEX     *pWaveFormatEx = NULL,  
    ULONGLONG              *ullEventInterest = SPFEI_ALL_EVENTS  
);
```

Parameters

pFileName

[in] Address of a null-terminated string containing the file name of the file to bind the stream to.

eMode

[in] Flag of the type [SPFILEMODE](#) to define the file opening mode. When opening an audio wave file, this must be SPFM_OPEN_READONLY or SPFM_CREATE_ALWAYS, otherwise the call will fail.

ppStream

[in, out] The address of an [ISpStream](#) pointer. If the function succeeds, this value is filled in with the newly created [ISpStream](#) interface.

pFormatId

[in] The data format identifier associated with the stream. This can be NULL (default) and the format will be determined

from the supplied wave file, if the file has the '.wav' extension. If it doesn't, the file is assumed to be a text file.

pWaveFormatEx

[in] Address of the [WAVEFORMATEX](#) structure that contains the wave file format information. If guidFormatId is SPDFID_WaveFormatEx, this must point to a valid [WAVEFORMATEX](#) structure. For other formats, it should be NULL.

ullEventInterest

[in] Flags of type [SPEVENTENUM](#) for the events wanted.

Return values

Value	Description
S_OK	Function completed successfully.
E_INVALIDARG	One or more parameters are invalid.
SPERR_ALREADY_INITIALIZED	The object has already been initialized.



SpClearEvent

SpClearEvent clears an event structure. May be used by clients not using CSpEvent class.

Found in: sphelper.h

```
inline void SpClearEvent(  
    SPEVENT *pe  
);
```

Parameters

pe

[in] The event to clear. Events of types SPET_LPARAM_IS_POINTER, SPET_LPARAM_IS_STRING, SPET_LPARAM_IS_TOKEN, or SPET_LPARAM_IS_OBJECT have the associated data in *pe->IParam* deallocated first.

Return values

No error code is returned.



SpConvertStreamFormatEnum

SpConvertStreamFormatEnum converts the specified stream format into a fully populated wave format structure.

Found in: sphelper.h

```
SpConvertStreamFormatEnum(  
    SPSTREAMFORMAT    eFormat,  
    GUID                *pFormatId,  
    WAVEFORMATEX       **ppCoMemWaveFormatEx  
);
```

Parameters

eFormat

[in] The requested stream format. Must be a valid SPSTREAMFORMAT value of SPSF_8kHz8BitMono or greater.

pFormatId

[in, out] The GUID of the new format. May be GUID_NULL if an error occurred.

ppCoMemWaveFormatEx

[out] The populated WAVEFORMATEX structure specified by the supplied SPSTREAMFORMAT.

Return values

Value	Description
S_OK	Function completed successfully.
E_OUTOFMEMORY	Exceeded available memory.
E_INVALIDARG	Either <i>pFormatId</i> or <i>ppCoMemWaveFormatEx</i> is invalid or bad. Alternatively, the specified

format is not recognized.

FAILED(hr)

Appropriate error message.



SpEventSerializeSize

SpEventSerializeSize calculates the required size of a buffer to serialize an event. The call must specify which type of serialized event.

Found in: sphelper.h

```
template <class T>
inline ULONG SpEventSerializeSize(
    const SPEVENT *pEvent
);
```

Parameters

pEvent

[in] The event structure to calculate the size of. *pEvent* must be either [SPSERIALIZEDEVENT](#) or [SPSERIALIZEDEVENT64](#).

Return values

Size, in bytes, required to serialize the event.



SpInitEvent

SpInitEvent clears the event structure.

Found in: sphelper.h

```
inline void SpInitEvent(  
    SPEVENT *pe  
);
```

Parameters

pe
[in] The event to clear.

Return values

No error code is returned.



SpSetDescription

SpSetDescription sets the string value for the specified token.

Found in: sphelper.h

```
inline HRESULT SpSetDescription(  
    ISpObjectToken *pObjToken,  
    const WCHAR *pszDescription,  
    LANGID Language = SpGetUserDefaultUILanguage(),  
    BOOL fSetLangIndependentId = TRUE  
);
```

Parameters

pObjToken

[in] The object token of the target resource.

pszDescription

[in] A null-terminated string containing the resource description.

Language

[in] The language ID for the resource. Language is optional and if omitted the default language will be used.

fSetLangIndependentId

[in] Boolean indicating whether the language independent ID is also changed. TRUE, changes it; FALSE does not. If *fSetLangIndependentId* omitted the default TRUE will be used.

Return values

Value	Description
-------	-------------

S_OK	Function completed successfully.
E_INVALIDARG	Either <i>pszDescription</i> or the language description is invalid or bad.
FAILED(hr)	Appropriate error message.



SAPI Object classes

This section covers the following topics:

- [SAPI Application Object Classes](#)
- [SAPI DDK Object Classes](#)

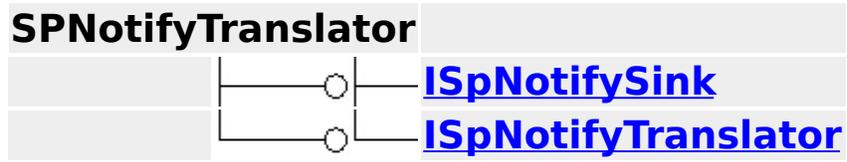


SAPI Application Object Classes

Object class	Related interfaces
<u>SpNotifyTranslator</u>	<u>ISpNotifySink</u> <u>ISpNotifyTransl</u>
<u>SpObjectTokenCategory</u>	<u>ISpObjectTokenCategory</u>
<u>SpObjectTokenEnum</u>	<u>IEnumSpObjectTokens</u>
<u>SpObjectToken</u>	<u>ISpObjectToken</u> <u>ISpDataKey</u>
<u>SpDataKey</u>	<u>ISpDataKey</u>
<u>SpResourceManager</u>	<u>IServiceProvider</u> <u>ISpResource</u>
<u>SpStreamFormatConverter</u>	<u>IStream</u> <u>ISpStreamFormat</u> <u>IS</u>
<u>SpMMAudioEnum</u>	<u>IEnumSpObjectTokens</u>
<u>SpMMAudioIn</u>	<u>ISpAudio</u> <u>IStream</u> <u>ISpStreamI</u> <u>ISpEventSink</u> <u>ISpObjectWithT</u>
<u>SpMMAudioOut</u>	<u>ISpAudio</u> <u>IStream</u> <u>ISpStreamI</u> <u>ISpEventSink</u> <u>ISpObjectWithT</u>
<u>SpRecPlayAudio</u>	<u>ISpAudio</u> <u>IStream</u> <u>ISpStreamI</u> <u>ISpEventSink</u> <u>ISpObjectWithT</u>
<u>SpStream</u>	<u>IStream</u> <u>ISpStreamFormat</u> <u>IS</u>
<u>SpVoice</u>	<u>ISpVoice</u> <u>ISpEventSource</u> <u>ISp</u>
<u>SpSharedRecognizer</u>	<u>ISpRecognizer</u>
<u>SpInprocRecognizer</u>	<u>ISpRecognizer</u>
<u>SpRecoContext</u>	<u>ISpRecoContext</u> <u>ISpEventSou</u>
<u>SpSharedRecoContext</u>	<u>ISpRecoContext</u> <u>ISpEventSou</u>
<u>SpRecoGrammar</u>	<u>ISpRecoGrammar</u> <u>ISpGramma</u>
<u>SpRecoResult</u>	<u>ISpRecoResult</u> <u>ISpPhrase</u>
<u>SpPhraseAlt</u>	<u>ISpPhraseAlt</u> <u>ISpPhrase</u>
<u>SpLexicon</u>	<u>ISpLexicon</u>
<u>SpUnCompressedLexicon</u>	<u>ISpLexicon</u> <u>ISpObjectWithTok</u>
<u>SpCompressedLexicon</u>	<u>ISpLexicon</u> <u>ISpObjectWithTok</u>
<u>SpPhoneConverter</u>	<u>ISpPhoneConverter</u> <u>ISpObject</u>

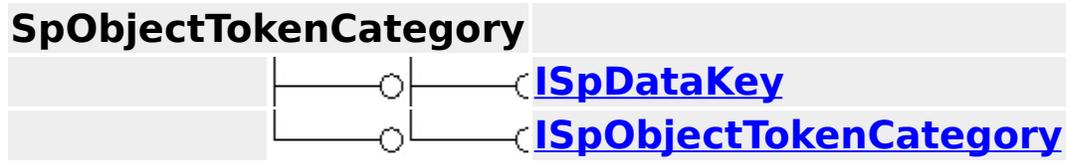


SPNotifyTranslator





SpObjectTokenCategory





SpObjectTokenEnum

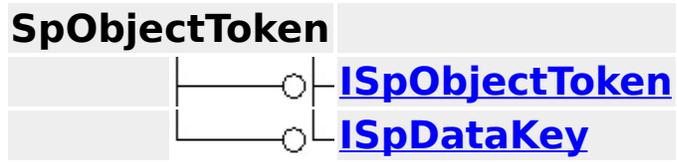
SpObjectTokenEnum



[IEnumSpObjectTokens](#)

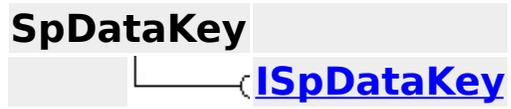


SpObjectToken



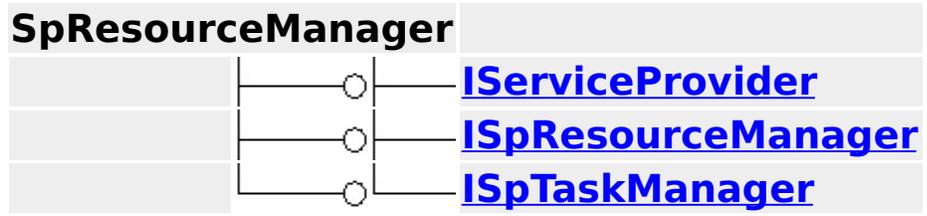


SpDataKey



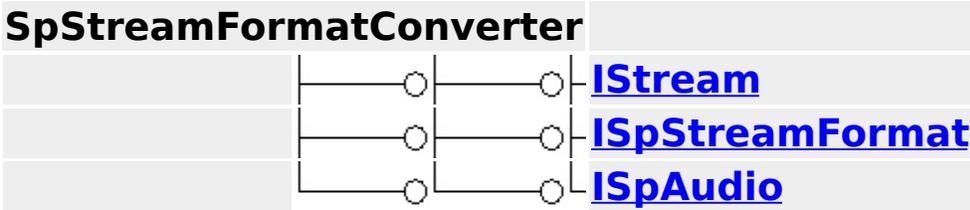


SpResourceManager





SpStreamFormatConverter





SpMMAudioEnum

SpMMAudioEnum



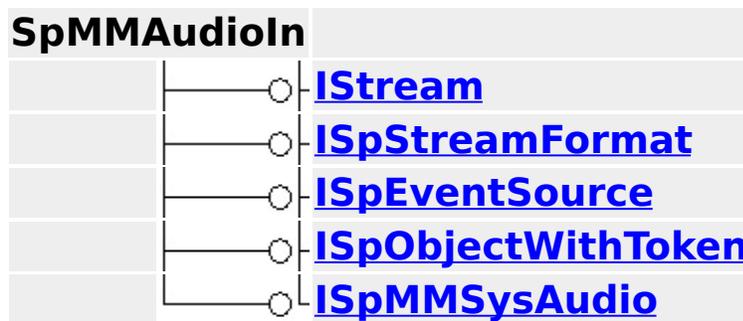
[IEnumSpObjectTokens](#)



SpMMAudiIn

The SAPI implementation of the SpMMAudiIn object supports the following UI through the [ISpTokenUI](#) interface if the object has a Windows mixer associated with it:

- [SPDUI_AudioProperties](#) - Displays advanced UI allowing user selection of the input line.
- [SPDUI_AudioVolume](#) - Displays the Windows mixer allowing user adjustment of audio volume.

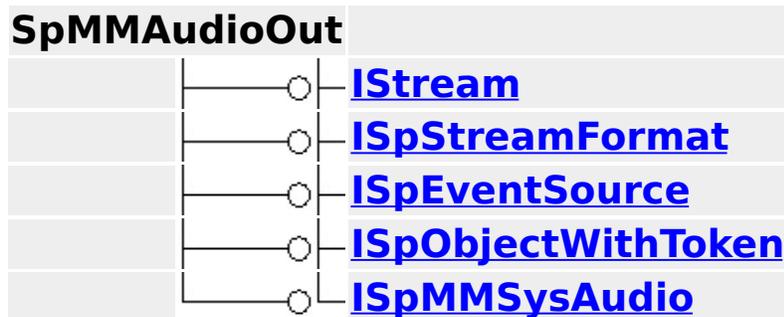




SpMMAudioOut

The SAPI implementation of the SpMMAudioOut object supports the following UI through the [ISpTokenUI](#) interface if the object has a Windows mixer associated with it:

- [SPDUI_AudioVolume](#) - Displays the Windows mixer allowing user adjustment of audio volume.



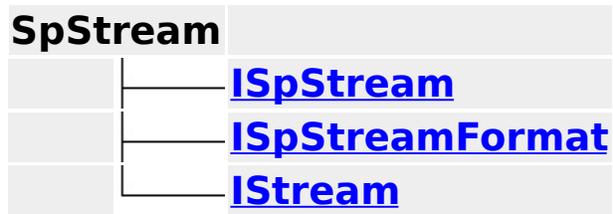


SpRecPlayAudio





SpStream





SpVoice



An application creates the SpVoice object and uses the [ISpVoice](#) interface to submit and control speech synthesis. Applications can speak text strings, text files, and audio files. Although this object is named the "SpVoice," it is actually a much higher-level object than a single voice. Conceptually, it is an object which accepts input data streams that are then rendered to the specified output, potentially using multiple speech synthesis voices in the process. Each SpVoice instance contains its own queue of input streams (usually just text) and its own output stream (usually an audio device). When an application calls [ISpVoice::Speak](#), another item is added to the end of the SpVoice queue.

Basic Synthesis

The main speech synthesis method is [ISpVoice::Speak](#). Almost everything having to do with controlling synthesis (for example, rate, pitch, and volume) is performed by this single function. This function can speak plain text, or the application can mark up the text using [synthesis markup tags](#). The speak method enables the application to specify whether the call should be synchronous or asynchronous. If the call is synchronous, the Speak method will not return until all of the text has been rendered. Speak returns immediately for asynchronous Speak calls, and the text is rendered on a background thread.

[ISpVoice::SpeakStream](#) is similar to the Speak method, but by using SpeakStream, streams of text or audio data can be added to the rendering queue.

Overriding Defaults

SAPI will automatically use the default voice and default audio output device if the application does not specify otherwise. The output can be controlled by the application through [ISpVoice::SetOutput](#). The default voice can be overridden in one of two ways: The application can call [ISpVoice::SetVoice](#) or it could speak a <VOICE> [synthesis markup tag](#).

Audio Device Sharing

When an SpVoice object is rendering to an audio device (as opposed to a stream), it will attempt to cooperate with other SpVoice objects that are sharing the same device based on the priority of the SpVoice. By default, a voice is set to SPVPRI_NORMAL which means that it will wait until other voices in the system have completed before it will begin rendering its input queue. A voice set to SPVPRI_ALERT will interrupt a normal priority voice by stopping the normal voice, rendering its own queue, and then restarting the normal priority voice. An SpVoice with a priority of SPVPRI_OVER will simply render its data immediately even if another voice is currently speaking (they would both speak at the same time).

Applications can control the priority of a voice by calling [ISpVoice::SetPriority](#).

Rendering to Streams

The SpVoice can render data to any object that implements ISpStreamFormat, which is a simple derivative of the COM standard IStream. The [SpStream](#) object is provided to allow easy conversion of existing IStreams to support ISpStreamFormat or to read or write wav or other files. Applications can [ISpVoice::SetOutput](#) to force the SpVoice to render to a stream. When rendering to a stream, the voice will render the data as quickly as possible.

Synthesis Events

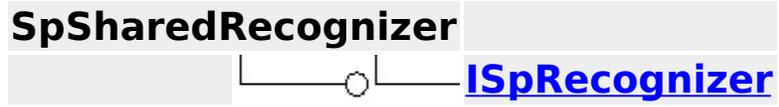
The SpVoice implements the [ISpEventSource](#) interface. It forwards events back to the application when the corresponding audio data has been rendered to the output device. Examples of events are reaching a word boundary, speaking a phoneme, reaching a bookmark, etc. Some applications can simply be notified when events occur and then call [ISpVoice::GetStatus](#) to determine the current stat of the SpVoice object. More complex applications may need to queue events. See the documentation for [ISpEventSource](#) for information on setting interest in events.

How Created

Create the SpVoice object by calling `::CoCreateInstance` with `CLSID_SpVoice`.



SpSharedRecognizer



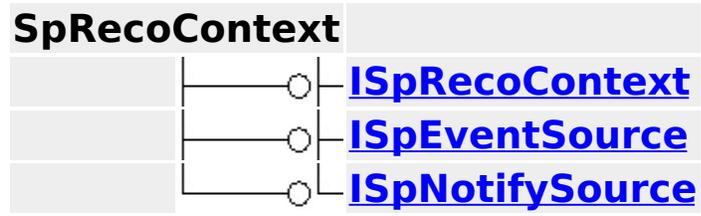


SpInprocRecognizer



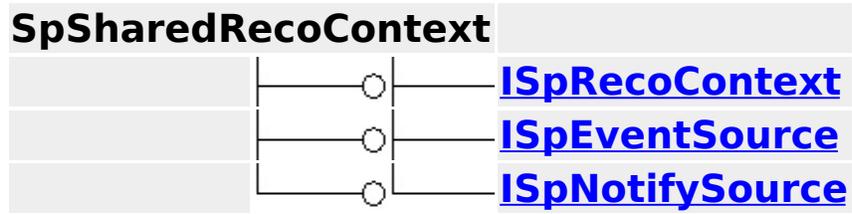


SpRecoContext



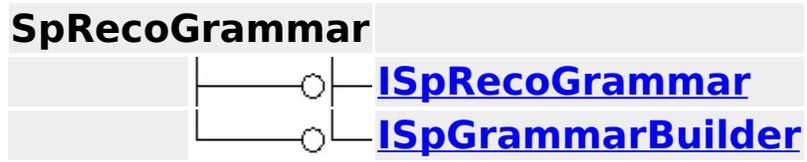


SpSharedRecoContext



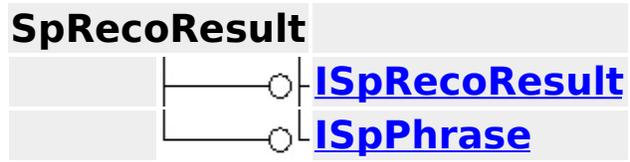


SpRecoGrammar



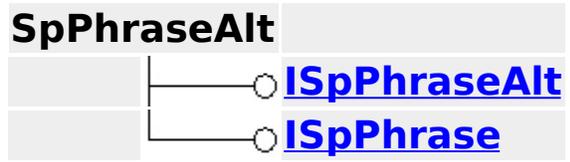


SpRecoResult



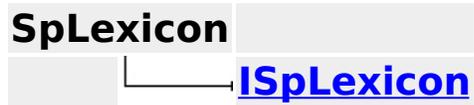


SpPhraseAlt





SpLexicon



The Lexicon database is a repository of words and word-related information such as pronunciations and parts of speech. The SAPI lexicon interface provides application, CSR engine, and TTS engine developers a standard method with which to create, access, modify, and synchronize with lexicons.

Types of Lexicons

There are two types of custom lexicons supported by lexicon interface: user and application. The user lexicon stores words specific to a user. It is a read/write lexicon and is shared among all applications. The application lexicon is supplied by the application and stores words specific to the application. The application supplied lexicons are read-only. Application lexicons ensure that the vocabulary used by the application is well represented in the lexicon.

Apart from custom lexicons, the lexicon interface provides access to vendor, morph, and letter-to-sound lexicons that Microsoft ships with SAPI. Vendor lexicons are large vocabulary lexicons holding words and their pronunciations and parts of speech. The morph lexicons derive pronunciations using the data in the vendor lexicon. The letter-to-sound lexicon computes the pronunciation of a word from its spelling.

User lexicons override application lexicons and engine private lexicons. You cannot change application lexicons from the [SpLexicon](#) object.

Modifying and Viewing the Contents of a Lexicon

An application can modify the user lexicon using the calls [ISpLexicon::AddPronunciation](#) and [ISpLexicon::RemovePronunciation](#). The function [ISpLexicon::GetWords](#) enables the caller to see what words are in the user or application lexicon. To obtain the pronunciation of a given word, the client would call [ISpLexicon::GetPronunciations](#). There is not a standard method for applications to access the lexicons that are supplied by the engine.

Synchronizing Changes to a Lexicon

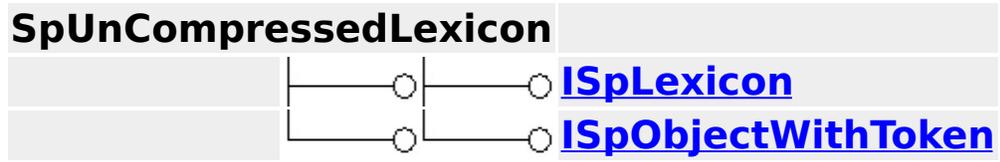
The lexicon interface provides methods to synchronize changes in lexicons using a lexicon generation ID, which is a sort of time-stamp on the lexicon. These changes in the lexicon are a result of modifications to user lexicons or for the installation or uninstallation of application lexicons. The client can get the current generation by calling [ISpLexicon::GetGeneration](#) and can see the change history since a given generation by calling [ISpLexicon::GetGenerationChange](#). A speech recognition engine might want to use the synchronization to update its private stores with the changes made to the custom lexicons while the client has been offline. For example, SR engines can update their language models with changes made to the custom lexicons while the SR engine had been offline.

How Created

An SpLexicon can be created by calling `::CoCreateInstance` with `CLSID_SpLexicon`.

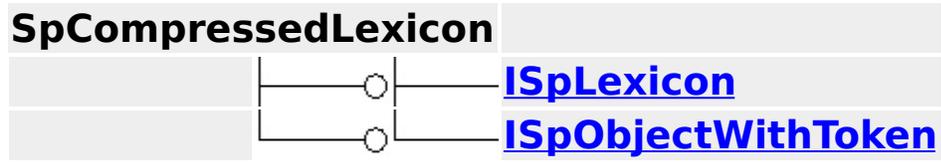


SpUnCompressedLexicon



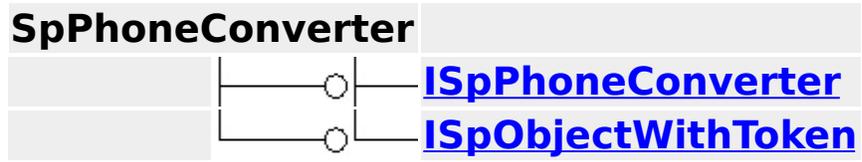


SpCompressedLexicon





SpPhoneConverter





SAPI DDK Object classes

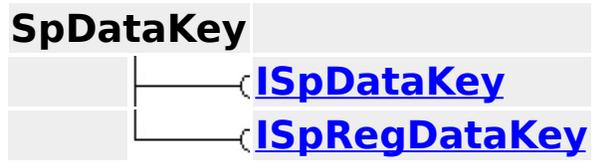
Object class	Related interfaces
<u>SpDataKey</u>	<u>ISpDataKey</u> <u>ISpRegDataKey</u>
<u>SpObjectTokenEnum</u>	<u>ISpObjectTokenEnumBuilder</u> <u>IEnum</u>
<u>SpPhraseBuilder</u>	<u>ISpPhraseBuilder</u> <u>ISpPhrase</u>
<u>SpITNProcessor</u>	<u>ISpITNProcessor</u>
<u>SpGrammarCompiler</u>	<u>ISpGrammarCompiler</u>
<u>SpGramCompBackend</u>	<u>ISpGramCompBackend</u> <u>ISpGrammar</u>
<u>SpSREngineSite</u>	<u>ISpSREngineSite</u>
<u>SpTTSEngineSite</u>	<u>ISpTTSEngineSite</u>

Abstract objects for engine developers

Object class	Related interfaces
<u>SpSREngine</u>	<u>ISpSREngine</u> <u>ISpObjectWithToken</u>
<u>SpTTSEngine</u>	<u>ISpTTSEngine</u> <u>ISpObjectWithToken</u>
<u>SpSRAAlternates</u>	<u>ISpSRAAlternates</u>
<u>SpRecoExtension</u>	<u>_ISpPrivateEngineCall</u> Implements extended interface(s)
<u>SpTokenUI</u>	<u>ISpTokenUI</u>



SpDataKey (DDK)



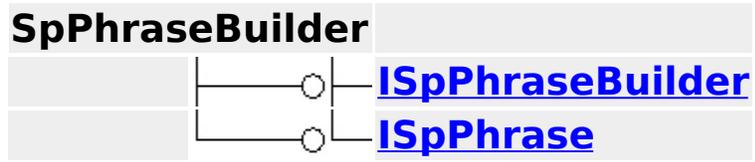


SpObjectTokenEnum (DDK)





SpPhraseBuilder (DDK)



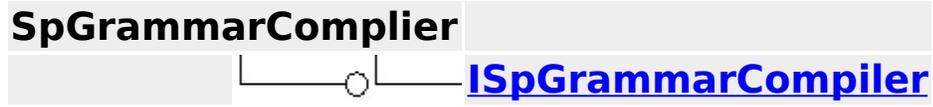


SpITNProcessor (DDK)



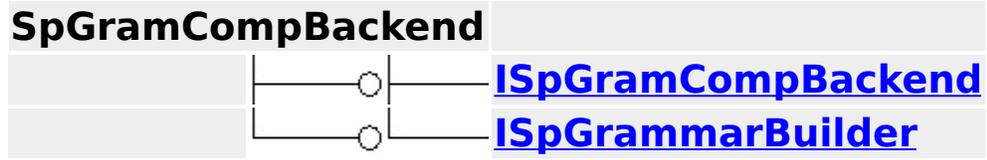


SpGrammarComplier (DDK)





SpGramCompBackend (DDK)





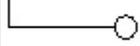
SpSREngineSite (DDK)





SpTTSEngineSite (DDK)

SpTTSEngineSite



ISpTTSEngineSite



SpSREngine (DDK)





SpTTSEngine (DDK)





SpSRAlternates (DDK)





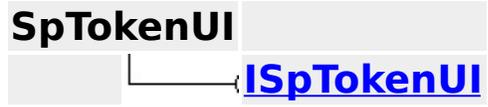
SpRecoExtension (DDK)

SpRecoExtension





SpTokenUI (DDK)





Error Codes Introduction

The following section includes:

- [Complete List of Error Codes](#)

Error codes listed here include the symbolic error name, and the numeric equivalent displayed as both decimal and hexadecimal. The following list may be searched using Ctrl+F.



Error Codes

The following table lists error codes returned by SAPI.

Error Name	Hexadecimal	Decimal	Description
SPERR_UNINITIALIZED	0x80045001	-2147201023	The object has not been properly initialized.
SPERR_ALREADY_INITIALIZED	0x80045002	-2147201022	The object has already been initialized.
SPERR_UNSUPPORTED_FORMAT	0x80045003	-2147201021	The caller has specified an unsupported format.
SPERR_INVALID_FLAGS	0x80045004	-2147201020	The caller has specified invalid flags for this operation.
SP_END_OF_STREAM	0x00045005	282629	The operation has reached the end of stream.
SPERR_DEVICE_BUSY	0x80045006	-2147201018	The wave device is busy.
SPERR_DEVICE_NOT_SUPPORTED	0x80045007	-2147201017	The wave device is not supported.
SPERR_DEVICE_NOT_ENABLED	0x80045008	-2147201016	The wave device is not enabled.
SPERR_NO_DRIVER	0x80045009	-2147201015	

There is no wave driver installed.

SPERR_FILEMUSTBEUNICODE | 0x8004500a | -2147201014

The file must be Unicode.

SP_INSUFFICIENTDATA | 0x0004500b | 282635

SPERR_INVALID_PHRASE_ID | 0x8004500c | -2147201012

The phrase ID specified does not exist or is out of range.

SPERR_BUFFER_TOO_SMALL | 0x8004500d | -2147201011

The caller provided a buffer too small to return a result.

SPERR_FORMAT_NOT_SPECIFIED | 0x8004500e | -2147201010

Caller did not specify a format prior to opening a stream.

SPERR_AUDIO_STOPPED | 0x8004500f | -2147201009

The stream I/O was stopped by setting the audio object to the stopped state. This will be returned for both read and write streams.

SP_AUDIO_PAUSED | 0x00045010 | 282640

This will be returned only on input (read) streams when the stream is paused. Reads on paused streams will not block, and this return code indicates that all of the data has been removed from the stream.

SPERR_RULE_NOT_FOUND | 0x80045011 | -2147201007

Invalid rule name passed to ActivateGrammar.

SPERR_TTS_ENGINE_EXCEPTION | 0x80045012 | -2147201006

An exception was raised during a call to the current TTS driver.

SPERR_TTS_NLP_EXCEPTION	0x80045013	-2147201005
-------------------------	------------	-------------

An exception was raised during a call to an application sentence filter.

SPERR_ENGINE_BUSY	0x80045014	-2147201004
-------------------	------------	-------------

In speech recognition, the current method cannot be performed while a grammar rule is active.

SP_AUDIO_CONVERSION_ENABLED	0x00045015	282645
-----------------------------	------------	--------

The operation was successful, but only with automatic stream format conversion.

SP_NO_HYPOTHESIS_AVAILABLE	0x00045016	282646
----------------------------	------------	--------

There is currently no hypothesis recognition available.

SPERR_CANT_CREATE	0x80045017	-2147201001
-------------------	------------	-------------

Cannot create a new object instance for the specified object category.

SP_ALREADY_IN_LEX	0x00045018	282648
-------------------	------------	--------

The word, pronunciation, or POS pair being added is already in lexicon.

SPERR_NOT_IN_LEX	0x80045019	-2147200999
------------------	------------	-------------

The word does not exist in the lexicon.

SP_LEX_NOTHING_TO_SYNC	0x0004501a	282650
------------------------	------------	--------

The client is currently synced with the lexicon.

SPERR_LEX_VERY_OUT_OF_SYNC	0x8004501b	-2147200997
----------------------------	------------	-------------

The client is excessively out of sync with the lexicon. Mismatches may not sync incrementally.

SPERR_UNDEFINED_FORWARD_RULE_REF	0x8004501c	-2147200996
----------------------------------	------------	-------------

A rule reference in a grammar was made to a named rule that was never defined.

SPERR_EMPTY_RULE	0x8004501d	-2147200995
------------------	------------	-------------

A non-dynamic grammar rule that has no body.

SPERR_GRAMMAR_COMPILER_INTERNAL_ERROR	0x8004501e	-2147200994
---------------------------------------	------------	-------------

The grammar compiler failed due to an internal state error.

SPERR_RULE_NOT_DYNAMIC	0x8004501f	-2147200993
------------------------	------------	-------------

An attempt was made to modify a non-dynamic rule.

SPERR_DUPLICATE_RULE_NAME	0x80045020	-2147200992
---------------------------	------------	-------------

A rule name was duplicated.

SPERR_DUPLICATE_RESOURCE_NAME	0x80045021	-2147200991
-------------------------------	------------	-------------

A resource name was duplicated for a given rule.

SPERR_TOO_MANY_GRAMMARS	0x80045022	-2147200990
-------------------------	------------	-------------

Too many grammars have been loaded.

SPERR_CIRCULAR_REFERENCE	0x80045023	-2147200989
--------------------------	------------	-------------

Circular reference in import rules of grammars.

SPERR_INVALID_IMPORT	0x80045024	-2147200988
----------------------	------------	-------------

A rule reference to an imported grammar that could not be resolved.

SPERR_INVALID_WAV_FILE	0x80045025	-2147200987
------------------------	------------	-------------

The format of the WAV file is not supported.

SP_REQUEST_PENDING	0x00045026	282662
<p>This success code indicates that an SR method called with the SPRIF_ASYNC flag is being processed. When it has finished processing, an SPFEI_ASYNC_COMPLETED event will be generated.</p>		

SPERR_ALL_WORDS_OPTIONAL	0x80045027	-2147200985
<p>A grammar rule was defined with a null path through the rule. That is, it is possible to satisfy the rule conditions with no words.</p>		

SPERR_INSTANCE_CHANGE_INVALID	0x80045028	-2147200984
<p>It is not possible to change the current engine or input. This occurs in the following cases: 1) SelectEngine called while a recognition context exists, or 2) SetInput called in the shared instance case.</p>		

SPERR_RULE_NAME_ID_CONFLICT	0x80045029	-2147200983
<p>A rule exists with matching IDs (names) but different names (IDs).</p>		

SPERR_NO_RULES	0x8004502a	-2147200982
<p>A grammar contains no top-level, dynamic, or exported rules. There is no possible way to activate or otherwise use any rule in this grammar.</p>		

SPERR_CIRCULAR_RULE_REF	0x8004502b	-2147200981
<p>Rule 'A' refers to a second rule 'B' which, in turn, refers to rule 'A'.</p>		

SP_NO_PARSE_FOUND	0x0004502c	282668
<p>Parse path cannot be parsed given the currently active rules.</p>		

--	--	--

SPERR_NO_PARSE_FOUND	0x8004502d	-2147200979
----------------------	------------	-------------

Parse path cannot be parsed given the currently active rules.

SPERR_REMOTE_CALL_TIMED_OUT	0x8004502e	-2147200978
-----------------------------	------------	-------------

A marshaled remote call failed to respond.

SPERR_AUDIO_BUFFER_OVERFLOW	0x8004502f	-2147200977
-----------------------------	------------	-------------

This will only be returned on input (read) streams when the stream is paused because the SR driver has not retrieved data recently.

SPERR_NO_AUDIO_DATA	0x80045030	-2147200976
---------------------	------------	-------------

The result does not contain any audio, nor does the portion of the element chain of the result contain any audio.

SPERR_DEAD_ALTERNATE	0x80045031	-2147200975
----------------------	------------	-------------

This alternate is no longer a valid alternate to the result it was obtained from. Returned from ISpPhraseAlt methods.

SPERR_HIGH_LOW_CONFIDENCE	0x80045032	-2147200974
---------------------------	------------	-------------

The result does not contain any audio, nor does the portion of the element chain of the result contain any audio. Returned from ISpResult::GetAudio and ISpResult::SpeakAudio.

SPERR_INVALID_FORMAT_STRING	0x80045033	-2147200973
-----------------------------	------------	-------------

The XML format string for this RULEREF is invalid, e.g. not a GUID or REFCLSID.

SP_UNSUPPORTED_ON_STREAM_INPUT	0x00045034	282676
--------------------------------	------------	--------

The operation is not supported for stream input.

SPERR_APPLEX_READ_ONLY	0x80045035	-2147200971
------------------------	------------	-------------

The operation is invalid for all but newly created application

lexicons.

SPERR_NO_TERMINATING_RULE_PATH 0x80045036 -2147200970

SP_WORD_EXISTS_WITHOUT_PRONUNCIATION 0x00045037 2826

The word exists but without pronunciation.

SPERR_STREAM_CLOSED 0x80045038 -2147200968

An operation was attempted on a stream object that has been closed.

SPERR_NO_MORE_ITEMS 0x80045039 -2147200967

When enumerating items, the requested index is greater than the count of items.

SPERR_NOT_FOUND 0x8004503a -2147200966

The requested data item (data key, value, etc.) was not found.

SPERR_INVALID_AUDIO_STATE 0x8004503b -2147200965

Audio state passed to SetState() is invalid.

SPERR_GENERIC_MMSYS_ERROR 0x8004503c -2147200964

A generic MMSYS error not caught by MMRESULT_TO_HRESULT.

SPERR_MARSHALER_EXCEPTION 0x8004503d -2147200963

An exception was raised during a call to the marshaling code.

SPERR_NOT_DYNAMIC_GRAMMAR 0x8004503e -2147200962

Attempt was made to manipulate a non-dynamic grammar.

SPERR_AMBIGUOUS_PROPERTY 0x8004503f -2147200961

Cannot add ambiguous property.

SPERR_INVALID_REGISTRY_KEY 0x80045040 -2147200960

The key specified is invalid.

SPERR_INVALID_TOKEN_ID 0x80045041 -2147200959

The token specified is invalid.

SPERR_XML_BAD_SYNTAX 0x80045042 -2147200958

The xml parser failed due to bad syntax.

SPERR_XML_RESOURCE_NOT_FOUND 0x80045043 -2147200957

The xml parser failed to load a required resource (e.g., voice, phoneconverter, etc.).

SPERR_TOKEN_IN_USE 0x80045044 -2147200956

Attempted to remove registry data from a token that is already in use elsewhere.

SPERR_TOKEN_DELETED 0x80045045 -2147200955

Attempted to perform an action on an object token that has had associated registry key deleted.

SPERR_MULTI_LINGUAL_NOT_SUPPORTED 0x80045046 -2147200954

The selected voice was registered as multi-lingual. SAPI does not support multi-lingual registration.

SPERR_EXPORT_DYNAMIC_RULE 0x80045047 -2147200953

Exported rules cannot refer directly or indirectly to a dynamic rule.

SPERR_STGF_ERROR 0x80045048 -2147200952

Error parsing the SAPI Text Grammar Format (XML grammar).

SPERR_WORDFORMAT_ERROR 0x80045049 -2147200951

Incorrect word format, probably due to incorrect pronunciation string.

SPERR_STREAM_NOT_ACTIVE 0x8004504a -2147200950

Methods associated with active audio stream cannot be called unless stream is active.

SPERR_ENGINE_RESPONSE_INVALID 0x8004504b -2147200949

Arguments or data supplied by the engine are in an invalid format or are inconsistent.

SPERR_SR_ENGINE_EXCEPTION 0x8004504c -2147200948

An exception was raised during a call to the current SR engine.

SPERR_STREAM_POS_INVALID 0x8004504d -2147200947

Stream position information supplied from engine is inconsistent.

SP_RECOGNIZER_INACTIVE 0x0004504e 282702

Operation could not be completed because the recognizer is inactive. It is inactive either because the recognition state is currently inactive or because no rules are active.

SPERR_REMOTE_CALL_ON_WRONG_THREAD 0x8004504f -2147200946

When making a remote call to the server, the call was made on wrong thread.

SPERR_REMOTE_PROCESS_TERMINATED 0x80045050 -2147200945

The remote process terminated unexpectedly.

SPERR_REMOTE_PROCESS_ALREADY_RUNNING	0x80045051	-2147200941
The remote process is already running; it cannot be started a second time.		

SPERR_LANGID_MISMATCH	0x80045052	-2147200942
An attempt to load a CFG grammar with a LANGID different than other loaded grammars.		

SP_PARTIAL_PARSE_FOUND	0x00045053	282707
A grammar-ending parse has been found that does not use all available words.		

SPERR_NOT_TOPLEVEL_RULE	0x80045054	-2147200940
An attempt to deactivate or activate a non top-level rule.		

SP_NO_RULE_ACTIVE	0x00045055	282709
An attempt to parse when no rule was active.		

SPERR_LEX_REQUIRES_COOKIE	0x80045056	-2147200938
An attempt to ask a container lexicon for all words at once.		

SP_STREAM_UNINITIALIZED	0x00045057	282711
An attempt to activate a rule/dictation/etc without calling SetInput first in the InProc case.		

SPERR_UNSUPPORTED_LANG	0x80045059	-2147200935
The requested language is not supported.		

SPERR_VOICE_PAUSED	0x8004505a	-2147200934
The operation cannot be performed because the voice is currently paused.		

--	--	--

SPERR_AUDIO_BUFFER_UNDERFLOW	0x8004505b	-2147200933
------------------------------	------------	-------------

This will only be returned on input (read) streams when the real time audio device stops returning data for a long period of time.

SPERR_AUDIO_STOPPED_UNEXPECTEDLY	0x8004505c	-2147200
----------------------------------	------------	----------

An audio device stopped returning data from the Read() method even though it was in the run state. This error is only returned in the END_SR_STREAM event.

SPERR_NO_WORD_PRONUNCIATION	0x8004505d	-2147200931
-----------------------------	------------	-------------

The SR engine is unable to add this word to a grammar. The application may need to supply an explicit pronunciation for this word.

SPERR_ALTERNATES_WOULD_BE_INCONSISTENT	0x8004505e	-2
--	------------	----

An attempt to call ScaleAudio on a recognition result having pre called GetAlternates. Allowing the call to succeed would result in previously created alternates located in incorrect audio stream p

SPERR_NOT_SUPPORTED_FOR_SHARED_RECOGNIZER	0x800450	
---	----------	--

The method called is not supported for the shared recognizer Fo ISpRecognizer::GetInputStream().

SPERR_TIMEOUT	0x80045060	-2147200928
---------------	------------	-------------

A task could not complete because the SR engine had timed out.

SPERR_REENTER_SYNCHRONIZE	0x80045061	-2147200927
---------------------------	------------	-------------

An SR engine called synchronize while inside of a synchronize call.

SPERR_STATE_WITH_NO_ARCS	0x80045062	-2147200926
--------------------------	------------	-------------

The grammar contains a node no arcs.

SPERR_NOT_ACTIVE_SESSION 0x80045063 -2147200925

Neither audio output nor input is supported for non-active console sessions.

SPERR_ALREADY_DELETED 0x80045064 -2147200924

The object is a stale reference and is invalid to use. For example, having an ISpeechGrammarRule object reference and then calling ISpeechRecoGrammar::Reset() will cause the rule object to be invalidated. Calling any methods after this will result in this error.



Miscellanea

The follow sections cover supporting speech functions:

- [Global Constants](#)
- [User Interfaces](#)
- [COM Class ID List](#)
- [Token Category IDs](#)
- [COM Interface IUnknown](#)
- [American English Phoneme Representation](#)
- [International Phoneme Representation](#)
- [Further Reading](#)

Microsoft Speech SDK SAPI 5.1



The following constants are used in SAPI. They are provided with the C/C++ name and Automation name although each pair uses the same numeric value.

C/C++ Name	Automation Name	Value
Description		

SP_MAX_WORD_LENGTH	Speech_Max_Word_Length	128
The maximum length of a word. Functions with input word strings matching or exceeding this limit will return E_INVALIDARG.		

SP_MAX_PRON_LENGTH	Speech_Max_Pron_Length	384
The maximum length of a phoneme pronunciation. This limit applies to zero-terminated lists of SPPHONEID elements. Functions with SPPHONEID parameters matching or exceeding this limit will return E_INVALIDARG.		

SP_STREAMPOS_ASAP	Speech_StreamPos_Asap	0
Indicates the Bookmark event will be fired as soon as possible after the speech recognition (SR) engine reaches a synchronization point. This allows the Bookmark to be sent at the first available opportunity.		

SP_STREAMPOS_REALTIME	Speech_StreamPos_RealTime	-1
Speech_StreamPos_RealTime indicates Bookmark event will occur when the SR engine reaches the current audio device position.		



User Interface

The following user interfaces (UI) are available. Each UI listed provides the C/C++ and Automation symbols although each pair refers to the same UI. The UI provided is for SAPI 5 speech recognition (SR) or text-to-speech engines (TTS).

Though not required, engines may employ a process improvement procedure and request additional information from the user. For example, if the recognition attempts are consistently poor or if the engine detects a consistent and interfering background noise, the SR engine could request that the user run the training or microphone wizard. This event is a suggestion by the SR engine to run the particular UI. The application may choose to initiate the UI or may ignore the suggestion.

Each manufacturer's engine may provide a different set of UI, so check documentation with that particular engine for additional details.

C/C++ Name	UI description.
Automation Name	

SPDUI_AddRemoveWord	Displays the Add/Remove word dialog box.
SpeechAddRemoveWord	

SPDUI_UserTraining	Displays the training wizard. Also available using Speech properties in Control Panel.
SpeechUserTraining	

SPDUI_MicTraining	Displays the microphone wizard. Also available using Speech properties in Control Panel.
SpeechMicTraining	

--	--

SPDUI_RecoProfileProperties	Displays the user profile wizard. Also available using Speech properties in Control Panel.
SpeechRecoProfileProperties	

SPDUI_AudioProperties	Displays the SR engine's audio properties.
SpeechAudioProperties	

SPDUI_AudioVolume	Displays the SR engine's audio level.
SpeechAudioVolume	

SPDUI_EngineProperties	Displays the engine's properties. This UI is available only in SAPI 5.0 engines; not in SAPI 5.1 or later versions.
SpeechEngineProperties	



SPDUI_EngineProperties (C/C++)

SpeechEngineProperties (Automation)

SPDUI_EngineProperties defines the string for displaying the UI for changing text-to-speech (TTS) or speech recognition (SR) engine properties on a per-user basis.

It is not a SAPI 5 compliance requirement for a speech engine to implement this UI for SPDUI_EngineProperties.

The Microsoft SR engine that ships in the SAPI 5 SDK does not support SPDUI_EngineProperties.

When to Implement

When writing a speech engine for the desktop or a graphical environment, users can change settings that should affect all of their recognition profiles, but be specific to each user. For example, the Microsoft TTS engine exposes some inverse-text-normalization (ITN) rules (e.g., comma versus period number delimiter, date format) in their engine properties.

Use Speech properties in Control Panel to change settings for all installed SAPI 5-compliant TTS and SR engines. Click **Settings** to change settings on a per-user/per-engine basis. Use Settings to directly access each Engine's Settings UI using SPDUI_EngineProperties. If the engine does not support the Engine Properties UI (see [ISpTokenUI::IsUISupported](#)), Settings will be unavailable.

When to Access

For advanced engine properties, the application could display a button or menu item that accessed SPDUI_EngineProperties (see [ISpVoice::DisplayUI](#)). Changes made within the engine properties UI will affect only one engine, and will not affect

other users.

```
#define SPDUI_EngineProperties          L"EngineProperties"
```

Example

The following code snippet illustrates the use of [ISpTokenUI::IsUISupported](#) using SPDUI_EngineProperties.

```
HRESULT hr = S_OK;  
  
// get the default text-to-speech engine token  
hr = SpGetDefaultTokenFromCategoryId(SPCAT_VOICES, &cpOb.  
// Check hr  
  
// get the object token's UI  
hr = cpObjectToken->QueryInterface(&cpTokenUI);  
// Check hr  
  
// check if the default text-to-speech engine has UI for  
hr = cpTokenUI->IsUISupported(SPDUI_EngineProperties, NU  
// Check hr  
  
// if fSupported == TRUE, then default speech text-to-sp
```

The following code snippet illustrates the use of [ISpVoice::DisplayUI](#) using SPDUI_EngineProperties.

```
HRESULT hr = S_OK;  
  
// display engine properties UI for the current TTS engi  
hr = cpVoice->DisplayUI(MY_HWND, MY_APP_VOICE_PROPERTIES  
// Check hr
```



SPDUI_AddRemoveWord (C/C++)

SpeechAddRemoveWord (Automation)

SPDUI_AddRemoveWord defines the string for displaying UI to modify the lexicon.

It is not a SAPI 5 compliance requirement for a speech recognition engine to implement this UI for SPDUI_AddRemoveWord.

If a speech engine is being written for the desktop or a graphical environment, users can modify the set of words that will be recognized or synthesized (see [ISpLexicon](#)).

Using the SDK sample application [Dictation Pad](#), the user can modify the lexicon using the Voice menu-->Add/Delete Words. When the user selects this menu item, Dictation Pad directly accesses the default SR engine's graphical lexicon editor through SPDUI_AddRemoveWord. If the speech recognition (SR) engine does not support the Training UI (see [ISpTokenUI::IsUISupported](#)), the menu item will be unavailable.

When to Access

The application could monitor the user's speech experience (correction type and frequency). If the user corrects a recognition by typing a word that is missing from the lexicon (see [ISpLexicon](#)), the application could prompt the user to add it to the lexicon using [ISpRecognizer::DisplayUI](#) and SPDUI_AddRemoveWord.

```
#define SPDUI_AddRemoveWord          L"AddRemoveWord"
```

Example

The following code snippet illustrates the use of [ISpTokenUI::IsUISupported](#) using SPDUI_AddRemoveWord.

```
HRESULT hr = S_OK;

// get the default speech recognizer token
hr = SpGetDefaultTokenFromCategoryId(SPCAT_RECOGNIZERS, &cpObjectToken);
// Check hr

// get the object token's UI
hr = cpObjectToken->QueryInterface(&cpTokenUI);
// Check hr

// check if the default speech recognizer has UI for editing
hr = cpTokenUI->IsUISupported(SPDUI_AddRemoveWord, NULL, &fSupported);
// Check hr

// if fSupported == TRUE, then default speech recognizer
```

The following code snippet illustrates the use of [ISpRecognizer::DisplayUI](#) using SPDUI_AddRemoveWord.

```
HRESULT hr = S_OK;

// display lexicon editing UI for the current recognizer
hr = cpRecognizer->DisplayUI(MY_HWND, MY_APP_MIC_TRAINING);
// Check hr
```



SPDUI_UserTraining (C/C++)

SpeechUserTraining (Automation)

SPDUI_UserTraining defines the string for displaying a speech recognition (SR) engine's User Training UI.

It is not a SAPI 5 compliance requirement for an SR engine to implement this UI for SPDUI_UserTraining.

When to Implement

When writing a speech engine for the desktop or a graphical environment, users can train the engine either before the first use, or when recognition accuracy is poor.

Use the SR tab of Speech properties in Control Panel to change settings for all installed SAPI 5-compliant SR engines. Click **Train Profile** to change settings on a per-user/per-engine basis. Use Train Profile to directly accesses each engine's Settings UI using SPDUI_UserTraining. If the SR engine does not support the training UI (see [ISpTokenUI::IsUISupported](#)), Train Profile will be unavailable.

When to Access

The application could monitor the user's speech experience by recognition accuracy. If a user encounters too many false recognitions or recognition corrections, the application could recommend that the user perform more recognizer training.

Also, an SR engine can send a [SPEI_REQUEST_UI](#) event to the application if it determines that the user needs to perform additional recognizer training. (see also [ISpRecognizer::IsUISupported](#) and [ISpRecognizer::DisplayUI](#))

```
#define SPDUI_UserTraining          L"UserTraining"
```

Example

The following code snippet illustrates the use of [ISpTokenUI::IsUISupported](#) using SPDUI_UserTraining.

```
HRESULT hr = S_OK;  
  
// get the default speech recognizer token  
hr = SpGetDefaultTokenFromCategoryId(SPCAT_RECOGNIZERS, &cpToken);  
// Check hr  
  
// get the object token's UI  
hr = cpObjectToken->QueryInterface(&cpTokenUI);  
// Check hr  
  
// check if the default speech recognizer has UI for per  
hr = cpTokenUI->IsUISupported(SPDUI_UserTraining, NULL, &fSupported);  
// Check hr  
  
// if fSupported == TRUE, then default speech recognizer
```

The following code snippet illustrates the use of [ISpRecognizer::DisplayUI](#) using SPDUI_UserTraining.

```
HRESULT hr = S_OK;  
  
// display user training UI for the current recognizer  
hr = cpRecognizer->DisplayUI(MY_HWND, MY_APP_USER_TRAINING_UI);  
// Check hr
```



SPDUI_MicTraining (C/C++)

SpeechMicTraining (Automation)

SPDUI_MicTraining defines the string for displaying a speech recognition (SR)engine's microphone training UI.

It is not a SAPI 5 compliance requirement for an SR engine to implement this UI for SPDUI_MicTraining.

When to Implement

When writing an SR engine for the desktop or a graphical environment, users might want to adapt the engine either before the first use, or when recognition accuracy is poor. Microphone quality can greatly benefit recognition accuracy, even for a speaker-independent engine.

Use Speech properties in Control Panel to change settings for all installed SAPI 5-compliant SR engines. Click **Configure Microphone** on the SR tab to adapt the microphone and directly access each SR engine's Microphone Training UI using SPDUI_MicTraining. If the SR engine does not support Microphone Training UI (see [ISpTokenUI::IsUISupported](#)), Configure Microphone will be unavailable.

When to Access

The application could monitor the user's speech experience by recognition accuracy. If a user encounters too many false recognitions or recognition corrections, the application could recommend that the user re-adapt the SR engine to their microphone (see [ISpRecognizer::DisplayUI](#)).

Also, an SR engine can send a [SPEI_REQUEST_UI](#) event to the application if it determines that the user needs to perform additional recognizer training. For example, if the input is too

quiet or too loud. Typically the UI type will be [SPDUI_MicTraining](#) to ensure that the SR engine is adapted to the current input audio settings.

```
#define SPDUI_MicTraining          L"MicTraining"
```

Example

The following code snippet illustrates the use of [ISpTokenUI::IsUISupported](#) using SPDUI_MicTraining.

```
HRESULT hr = S_OK;

// get the default speech recognizer token
hr = SpGetDefaultTokenFromCategoryId(SPCAT_RECOGNIZERS, &cpToken);
// Check hr

// get the object token's UI
hr = cpObjectToken->QueryInterface(&cpTokenUI);
// Check hr

// check if the default speech recognizer has UI for per
hr = cpTokenUI->IsUISupported(SPDUI_MicTraining, NULL, NULL);
// Check hr

// if fSupported == TRUE, then default speech recognizer
```

The following code snippet illustrates the use of [ISpRecognizer::DisplayUI](#) using SPDUI_MicTraining.

```
HRESULT hr = S_OK;

// display microphone training UI for the current recogn.
hr = cpRecognizer->DisplayUI(MY_HWND, MY_APP_MIC_TRAINING);
// Check hr
```



SPDUI_RecoProfileProperties (C/C++)

SpeechRecoProfileProperties (Automation)

SPDUI_RecoProfileProperties defines the string for displaying the properties associated with a specific recognition profile.

It is not a SAPI 5 compliance requirement for a speech recognition (SR) engine to implement this UI for SPDUI_RecoProfileProperties.

For more information about Recognition Profiles, see the [Object Tokens and Registry Settings White Paper](#).

When to Implement

When writing a speech engine for the desktop or a graphical environment, users can modify settings for specific run-time environments (e.g., mobile versus desktop, noisy versus quiet, fast versus slow computer, etc.).

Use Speech properties in Control Panel to change settings for all installed SAPI 5-compliant text-to-speech (TTS) and SR engines. Click **Settings** to change settings on a per-user/per-engine basis. Use SPDUI_RecoProfileProperties to directly access each Engine's Recognition Profile Settings UI. If the engine does not support the Recognition Profile Properties UI (see [ISpTokenUI::IsUISupported](#)), Settings will be unavailable.

When to Access

The application could display a button or menu item for environment profile settings that accessed SPDUI_RecoProfileProperties. If the application can be used in

either noisy or quiet environments, the user could be prompted to update the current recognition profile to reflect their specific environment (see [ISpRecognizer::DisplayUI](#)). Changes made within the engine properties UI will affect only one engine, and will not affect other users.

For information on creating new recognition profiles, see the helper function [SpCreateNewToken \(by Category Id\)](#).

```
#define SPDUI_RecoProfileProperties          L"RecoProfileProp
```

Example

The following code snippet illustrates the use of [ISpTokenUI::IsUISupported](#) using SPDUI_RecoProfileProperties.

```
HRESULT hr = S_OK;  
  
// get the default speech recognizer token  
hr = SpGetDefaultTokenFromCategoryId(SPCAT_RECOGNIZERS, &  
// Check hr  
  
// get the object token's UI  
hr = cpObjectToken->QueryInterface(&cpTokenUI);  
// Check hr  
  
// check if the default speech recognizer has UI for its  
hr = cpTokenUI->IsUISupported(SPDUI_RecoProfilePropertie:  
// Check hr  
  
// if fSupported == TRUE, then the default speech recogn.
```

The following code snippet illustrates the use of [ISpRecognizer::DisplayUI](#) using SPDUI_RecoProfileProperties.

```
HRESULT hr = S_OK;  
  
// display recognition profile properties UI for the cur  
hr = cpRecognizer->DisplayUI(MY_HWND, MY_APP_RECO_PROFI  
// Check hr
```



SPDUI_AudioProperties (C/C++)

SpeechAudioProperties (Automation)

SPDUI_AudioProperties defines the string for displaying an audio object's properties user interface (UI).

```
#define SPDUI_AudioProperties          L"AudioProperties"
```

When to Implement

An application can modify the object's implementation-specific properties for a custom audio object. For example, the SAPI implementation of the multimedia audio object has a UI so that the user can select the multimedia device line (e.g., microphone input, line input, etc.).

A custom audio object that performed noise reduction on the input stream could have a UI for selecting the audio input object to read data from initially. It could also display a UI that allows the user to adjust how much noise reduction is performed.

Speech properties in Control Panel allows the user to select the default audio input and output objects. Click **Properties** to directly access each audio object's properties UI using SPDUI_AudioProperties. If the audio object does not support the Properties UI (see [ISpTokenUI::IsUISupported](#)), Properties will be unavailable.

Example

The following code snippet illustrates the use of [ISpTokenUI::IsUISupported](#) using SPDUI_AudioProperties.

```
HRESULT hr = S_OK;
```

```
// get the default input audio object token
hr = SpGetDefaultTokenFromCategoryId(SPCAT_AUDIOIN, &cpObj
// Check hr

// get the object token's UI
hr = cpObjectToken->QueryInterface(&cpTokenUI);
// Check hr

// check if the default audio input object has UI for Pr
hr = cpTokenUI->IsUISupported(SPDUI_AudioProperties, NUL
// Check hr

// if fSupported == TRUE, then default audio input objec
```



SPDUI_AudioVolume (C/C++)

SpeechAudioVolume (Automation)

SPDUI_AudioVolume defines the string for displaying an audio object's volume UI.

When to Implement

An application can modify the object's volume settings for a custom audio object. For example, the SAPI implementation in the multimedia audio object displays the Windows Mixer associated with the audio device.

Using Speech properties in Control Panel, select the default audio input and output objects. Click **Volume** to directly access each audio object's volume UI using SPDUI_AudioVolume. If the audio object does not support the Volume UI (see [ISpTokenUI::IsUISupported](#)), Volume will be unavailable.

When to Access

The application accesses the audio object's volume UI if the output is too loud, or the input is too quiet.

Also, an SR engine can send an [SPEI_REQUEST_UI](#) event to the application if it determines that the user should perform microphone training. Typically the UI type will be [SPDUI_MicTraining](#) to ensure that the SR engine is adapted to the current input audio settings. For example, if the audio input volume is very low, it is preferable to raise the audio input volume, rather than request the SR engine to amplify a poor audio input signal.

SAPI will generally not recognize changes to the Windows Mixer settings. This Mixer is made available solely as a last resort in adjusting the sound system if the Microphone Training wizard

fails to set the volume suitably.

```
#define SPDUI_AudioVolume          L"AudioVolume"
```

Example

The following code snippet illustrates the use of [ISpTokenUI::IsUISupported](#) using SPDUI_AudioVolume.

```
HRESULT hr = S_OK;

// get the default input audio object token
hr = SpGetDefaultTokenFromCategoryId(SPCAT_AUDIOIN, &cpObj)
// Check hr

// get the object token's UI
hr = cpObj->QueryInterface(&cpTokenUI);
// Check hr

// check if the default audio input object has UI for Vo.
hr = cpTokenUI->IsUISupported(SPDUI_AudioVolume, NULL, NI)
// Check hr

// if fSupported == TRUE, then default audio input objec
```



COM Class ID List

The following Class IDs are used with SAPI 5.

CLSID	Interface on which CLSID can be co-created
CLSID_SpNotifyTranslator	ISpNotifyTranslator
CLSID_SpObjectTokenCategory	ISpObjectTokenCategory
CLSID_SpObjectToken	ISpObjectTokenInit
CLSID_SpResourceManager	ISpResourceManager
CLSID_SpStreamFormatConverter	ISpStreamFormatConverter
CLSID_SpStreamFormatConverter	ISpEventSource
CLSID_SpStreamFormatConverter	ISpEventSink
CLSID_SpStreamFormatConverter	ISpAudio
CLSID_SpMMAudioEnum	ISpObjectWithToken
CLSID_SpMMAudioEnum	IEnumSpObjectTokens
CLSID_SpMMAudioIn	ISpMMSysAudio
CLSID_SpMMAudioOut	ISpMMSysAudio
CLSID_SpRecPlayAudio	ISpAudio
CLSID_SpRecPlayAudio	ISpObjectWithToken
CLSID_SpStream	ISpStream
CLSID_SpStream	ISpEventSource
CLSID_SpStream	ISpEventSink
CLSID_SpStream	ISpTranscript
CLSID_SpVoice	ISpVoice
CLSID_SpVoice	ISpThreadTask
CLSID_SpSharedRecoContext	ISpRecoContext
CLSID_SpSharedRecoContext	_ISpPrivateEngineCall
CLSID_SpInprocRecognizer	ISpRecognizer
CLSID_SpSharedRecognizer	ISpRecognizer
CLSID_SpLexicon	ISpContainerLexicon
CLSID_SpUnCompressedLexicon	ISpLexicon

CLSID_SpUnCompressedLexicon	ISpObjectWithToken
CLSID_SpCompressedLexicon	ISpLexicon
CLSID_SpCompressedLexicon	ISpObjectWithToken
CLSID_SpPhoneConverter	ISpPhoneConverter
CLSID_SpNullPhoneConverter	ISpPhoneConverter
CLSID_SpResourceManager	ISpTaskManager



Token Category IDs

The following token category IDs are used with SAPI 5.

Category ID	Purpose
SPCAT_AUDIOOUT	Available audio output devices.
SPCAT_AUDIOIN	Available audio input devices.
SPCAT_VOICES	Available voices.
SPCAT_RECOGNIZERS	Available recognizers.
SPCAT_APPLEXICONS	Available application lexicons.
SPCAT_PHONECONVERTERS	Available phoneme converters
SPCAT_RECOPROFILES	Available recognition profiles.



COM Interface IUnknown

The IUnknown interface is a common interface supported by all COM objects and, therefore, by all speech objects. The IUnknown interface has the following member functions:

- [QueryInterface](#)
- [AddRef](#)
- [Release](#)

An application uses IUnknown to obtain pointers to other interfaces supported by an object and to manage the interface pointers after obtaining them.

QueryInterface

```
HRESULT QueryInterface(  
    REFIID riid,                //Identifier of the request  
    LPVOID FAR *ppvObj         //Address of output variable  
                                //interface pointer request  
);
```

Retrieves the address of a specified interface on a particular object so that an application can query an object to determine what interfaces it supports.

Returns NOERROR, if successful, or one of these error values:

- CO_E_OBJNOTCONNECTED
- E_NOINTERFACE
- E_OUTOFMEMORY
- E_INVALIDARG
- E_UNEXPECTED
- REGDB_E_IIDNOTREG

Parameter	Description
<i>riid</i>	[in] Interface identifier of the interface to be retrieved.
<i>ppvObj</i>	[out] Address of a variable that receives the address of the specified interface on the object. If the interface specified in <i>riid</i> is not supported by the object, the function returns E_NOINTERFACE. All errors set <i>ppvObj</i> to NULL.

AddRef

ULONG AddRef(void);

- Increments a reference count for every new copy of an interface pointer to a specified interface on a particular object.
- Returns the value of the reference count.

When an interface is fully released, the reference count is zero. This information should be used only for diagnostics and testing.

Release

ULONG Release(void);

- Decrements the reference count for the specified interface on a particular object.
- Returns the value of the reference count.

When an interface is fully released, the reference count is zero. This information should be used only for diagnostics and testing.

If the object reference count goes to zero as a result of calling Release, the object is freed from memory.

If the AddRef member function has been called on this object's interface n times and this is the $n+1$ th call to Release, the interface pointer frees itself. An object frees itself if the released pointer is the only pointer and if the object supports multiple interfaces through the QueryInterface member function.



American English Phoneme Representation

This is a brief introduction to the use and implementation of the SAPI phoneme representations.

Symbolic and Numerical Representation

Application developers can create pronunciations for words that are not currently in the lexicon by using the English phonemes represented in the following table. The phoneme set is composed of a symbolic phonetic representation (SYM).

The application developer will be able to enter the SYM representation to create the pronunciation using the XML PRON tag, or by creating a new lexicon entry. Each phoneme entry should be space delimited.

Tag	Description
PRON SYM	Tag used to insert a pronunciation using symbolic representation.

Example: pronunciation for "hello":

```
<PRON SYM = "h eh l ow"/>
```

For improved accuracy, the primary (1), secondary (2) stress markers, and the syllabic markers (-) can be added to the pronunciation.

Example: pronunciation for "hello" using the primary stress (1) and syllabic (-) markers:

```
<PRON SYM = "h eh - l ow 1"/>
```

American English Phoneme Table

SYM	Example	PhoneID
-	syllable boundary (hyphen)	1
!	Sentence terminator (exclamation mark)	2
&	word boundary	3
,	Sentence terminator (comma)	4
.	Sentence terminator (period)	5
?	Sentence terminator (question mark)	6
_	Silence (underscore)	7
1	Primary stress	8
2	Secondary stress	9
aa	f <u>a</u> ther	10
ae	ca <u>t</u>	11
ah	cu <u>t</u>	12
ao	do <u>g</u>	13
aw	fo <u>u</u> l	14
ax	ago	15
ay	bi <u>t</u> e	16
b	<u>b</u> ig	17
ch	<u>ch</u> in	18
d	<u>d</u> ig	19
dh	<u>th</u> en	20
eh	pe <u>t</u>	21
er	fu <u>r</u>	22
ey	<u>a</u> te	23
f	fo <u>r</u> k	24

g	g <u>ut</u>	25
h	h <u>elp</u>	26
ih	f <u>ill</u>	27
iy	f <u>ee</u> l	28
jh	jo <u>y</u>	29
k	c <u>ut</u>	30
l	l <u>id</u>	31
m	m <u>at</u>	32
n	n <u>o</u>	33
ng	s <u>ing</u>	34
ow	g <u>o</u>	35
oy	to <u>y</u>	36
p	p <u>ut</u>	37
r	r <u>ed</u>	38
s	s <u>it</u>	39
sh	sh <u>e</u>	40
t	t <u>alk</u>	41
th	th <u>in</u>	42
uh	bo <u>o</u> k	43
uw	to <u>o</u>	44
v	v <u>at</u>	45
w	w <u>ith</u>	46
y	y <u>ard</u>	47
z	z <u>ap</u>	48
zh	pleas <u>u</u> re	49

Please see [International Phonemes](#) for information on other phoneme sets.



International Phoneme Representation

You can create pronunciations for words that are not currently in the lexicon using the phonemes represented in the attached appendices. The proposed phoneme set is composed of a symbolic phonetic representation (SYM).

You can enter the SYM representation to create the pronunciation by using the XML PRON tag, or by creating a new lexicon entry. Each phoneme should be space delimited.

The engine is passed a USHORT structure called SPPHONEID (a number between 1 and n where n is the total number of phonemes for that language). The conversion from the SYM to SPPHONEID occurs in the SAPI PhoneConverter.

Mark Up Tag	Description
PRON SYM	Tag used to insert a pronunciation using symbolic representation

Example: pronunciation for "hello"

```
<PRON SYM = "h eh l ow"/>
```

For improved accuracy, the primary (1), secondary (2) stress markers, and the syllabic markers (-) can be added to the pronunciation.

Example: pronunciation for "hello" using the primary stress (1) and syllabic (-) markers:

```
<PRON SYM = "h eh - l ow 1"/>
```

SAPI-compliant engines are required to accept the PHONEID representation, and produce an articulation. The specific allophonic articulation is defined by the engine. There is no

provision for support of phonemes outside the SAPI phoneme set.

Main goals for defining the language dependent phoneme set:

- Provide an engine-independent architecture for application developers to create user and application lexicons.
- Make the English phonetic table simple enough to be used and understood by non-linguists who use the American English phoneme set.

International phoneme use

Using the international phoneme schema, you can create a phoneme set which can be used for each language independently. Using the numeric representation as opposed to the International Phonetic Alphabet (IPA) code will eliminate some of the problems regarding the possible differences in the IPA values for the same phonemes. Hence, an 'r' in English will correspond to a certain number (38) and an 'r' in French may correspond to a different number. It is up to the individual engine to provide the exact IPA value for the two 'r's.

Each language will be associated with a set of phonemes numbered from 1 to X. You can use either the symbolic representation or the number representation to enter the pronunciation. Since you are probably not a linguist, the IPA code will probably have little meaning.

Please note that consistent pronunciation is NOT a goal, while predictable pronunciation is. Using the phoneme set, an application developer can guarantee a minimal pronunciation, but not the exact allophonic expression. So, the word "first" will always be pronounced as "first", never as "fist" or "feast", etc, but the accent of the engine may be slightly different due to the fact that the internal allophone values may differ

For more information and definitions for international phoneme sets, please see:

- [Chinese Phonemes](#)
- [Japanese Phonemes](#)



Chinese Phonemes

The following table defines the Chinese language phoneme set.

Symbol	PhoneID	Example
-	1	Syllable boundary (hyphen)
!	2	Sentence terminator (exclamation mark)
&	3	word boundary
,	4	Sentence terminator (comma)
.	5	Sentence terminator (period)
?	6	Sentence terminator (question mark)
_	7	Silence (underscore)
+	8	primary stress
*	9	secondary stress
1	10	Tone 1
2	11	Tone 2
3	12	Tone 3
4	13	Tone 4
5	14	Tone 5
a	15	a 1 fei 1 (rogue)
ai	16	ai 4 ren 2 (lover)
an	17	an 1 quan 2 (safe)
ang	18	ang 1 zang 1 (dirty)
ao	19	jiao 1 ao 4 (proud)
ba	20	ba 4 ba 5 (dad)
bai	21	bai 2 se 4 (white)
ban	22	mu 4 ban 3 (board)

bang	23	bang 3 jia 4 (kidnap)
bao	24	yong 1 bao 4 (embrace)
bei	25	bei 3 fang 1 (north)
ben	26	ben 4 dan 4 (fool)
beng	27	beng 4 tiao 4 (jump)
bi	28	bi 2 zi 5 (nose)
bian	29	bian 4 hua 5 (change)
biao	30	biao 3 ge 2 (table)
bie	31	li 2 bie 2 (part with)
bin	32	bin 1 ke 4 (guest)
bing	33	shi 4 bing 1 (soldier)
bo	34	bo 2 dou 4 (wrestle)
bu	35	bu 4 xing 2 (walk)
ca	36	ca 1 gan 1 (wipe)
cai	37	cai 1 ce 4 (guess)
can	38	can 1 jia 1 (join)
cang	39	cang 1 ying 5 (fly)
cao	40	cao 1 chang 3 (playground)
ce	41	ce 4 liang 2 (measure)
cen	42	cen 1 ci 1 (uneven)
ceng	43	ceng 2 jing 1 (once)
cha	44	jian 3 cha 2 (check)
chai	45	chai 1 hui 3 (demolish)
chan	46	chan 3 ye 4 (industry)
chang	47	jing 1 chang 2 (often)
chao	48	chao 1 yue 4

		(exceed)
che	49	qi 4 che 1 (automobile)
chen	50	chen 2 jiu 4 (old)
cheng	51	cheng 2 che 1 (ride)
chi	52	chi 2 dao 4 (late for)
chong	53	chong 1 man 3 (full of)
chou	54	chou 2 hen 4 (hatred)
chu	55	chu 2 fa 3 (division)
chuai	56	chuai 3 ce 4 (guess)
chuan	57	lun 2 chuan 2 (ship)
chuang	58	chuang 4 ye 4 (carve out)
chui	59	tie 3 chui 2 (hammer)
chun	60	chun 1 tian 1 (spring)
chuo	61	wo 4 chuo 4 (dirty)
ci	62	ci 2 qi 4 (porcelain)
cong	63	cong 2 lin 2 (thicket)
cou	64	jin 3 cou 4 (compact)
cu	65	cu 4 xiao 1 (sales promotion)
cuan	66	tao 2 cuan 4 (run away)
cui	67	cui 1 cu 4 (press)
cun	68	xiang 1 cun 1 (country)
cuo	69	cuo 4 wu 4 (error)
da	70	da 4 (big)
dai	71	dai 4 biao 3

		(delegate)
dan	72	dan 4 shi 4 (but)
dang	73	dang 1 ran 2 (sure)
dao	74	dao 1 (knife)
de	75	de 5 (function word)
dei	76	dei 3 (must)
den	77	den 4 (yank)
deng	78	ban 3 deng 4 (wooden stool)
di	79	di 2 que 4 (certainly)
dia	80	dia 3 (coquetry voice)
dian	81	dian 3 (dot)
diao	82	diao 4 (hang)
die	83	die 1 dao 3 (tumble)
ding	84	jue 2 ding 4 (decide)
diu	85	diu 1 qi 4 (discard)
dong	86	dong 1 fang 1 (east)
dou	87	zhan 4 dou 4 (struggle)
du	88	du 4 jue 2 (stop)
duan	89	duan 3 (short)
dui	90	dui 4 (right)
dun	91	ting 2 dun 4 (halt)
duo	92	duo 1 yu 2 (unnecessary)
e	93	e 4 (hungry)
ei	94	ei 4 (yes)
en	95	en 1 hui 4 (favor)
er	96	er 3 (ear)
fa	97	fa 1 zhan 3 (development)
fan	98	fan 4 (rice)

fang	99	fang 2 jian 1 (room)
fei	100	fei 1 (fly)
fen	101	fen 1 bie 2 (separate)
feng	102	feng 1 (wind)
fo	103	fo 2 (buddha)
fou	104	fou 3 ding 4 (denial)
fu	105	fu 4 qin 1 (father)
ga	106	gan 1 ga 4 (awkward)
gai	107	ying 1 gai 1 (should)
gan	108	gan 1 jing 4 (neatness)
gang	109	gang 1 cai 2 (just)
gao	110	gao 1 (tall)
ge	111	pin 3 ge 2 (character)
gei	112	gei 3 yu 3 (give)
gen	113	gen 1 (root)
geng	114	geng 4 jia 1 (much more)
gong	115	gong 1 ren 2 (worker)
gou	116	zu 2 gou 4 (enough)
gu	117	gu 4 xiang 1 (hometown)
gua	118	xi 1 gua 1 (watermelon)
guai	119	qi 2 guai 4 (oddness)
guan	120	guan 1 bi 4 (close)
guang	121	guang 1 (light)
gui	122	gui 3 (ghost)
gun	123	gun 4 (stick)

guo	124	guo 4 qu 4 (past)
ha	125	ha 1 (sound of laugh)
hai	126	hai 2 zi 5 (child)
han	127	chu 1 han 4 (sweat)
hang	128	hang 2 xing 2 (sail)
hao	129	hao 3 (good)
he	130	he 2 (river)
hei	131	hei 1 (black)
hen	132	hen 3 (very)
heng	133	heng 2 xiang 4 (landscape orientation)
hong	134	hong 2 (red)
hou	135	hou 2 zi 5 (monkey)
hu	136	hu 2 xu 1 (beard)
hua	137	hua 4 (picture)
huai	138	huai 4 (bad)
huan	139	huan 2 jing 4 (environment)
huang	140	huang 2 se 4 (yellow)
hui	141	hui 1 huang 2 (refulgence)
hun	142	hun 2 zhuo 2 (muddy)
huo	143	huo 3 (fire)
ji	144	ji 2 shi 2 (in time)
jia	145	jia 4 qi 1 (holiday)
jian	146	jian 3 dan 1 (simple)
jiang	147	jiang 1 (river)
jiao	148	jiao 1 tong 1 (traffic)
jie	149	jie 2 ri 4 (feast)

jin	150	jin 3 zhang 1 (strain)
jing	151	gan 1 jing 4 (neatness)
jiong	152	jiong 3 po 4 (embarrassed)
jiu	153	jiu 3 (nine)
ju	154	ju 4 zi 5 (sentence)
juan	155	juan 3 qu 1 (curl)
jue	156	jue 2 ding 4 (decide)
jun	157	jun 1 dui 4 (army)
ka	158	ka 3 che 1 (truck)
kai	159	kai 3 xuan 2 (triumph)
kan	160	kan 4 (see)
kang	161	di 3 kang 4 (resist)
kao	162	kao 3 shi 4 (test)
ke	163	ke 3 (thirsty)
kei	164	kei 1 (scold)
ken	165	ken 3 ding 4 (affirm)
keng	166	keng 1 hai 4 (entrap)
kong	167	kong 1 qi 4 (air)
kou	168	kou 3 (mouth)
ku	169	jian 1 ku 3 (trial)
kua	170	kua 1 jiang 3 (praise)
kuai	171	kuai 4 (fast)
kuan	172	kuan 1 kuo 4 (openness)
kuang	173	kong 1 kuang 4 (void)
kui	174	kui 1 qian 4 (owe)
kun	175	kun 4 nan 2 (hard)
kuo	176	kuo 4 da 4 (enlarge)
la	177	la 4 jiao 1 (hot)

		pepper)
lai	178	lai 2 (come)
lan	179	lan 2 se 4 (blue)
lang	180	lang 2 (wolf)
lao	181	lao 3 (old)
le	182	kuai 4 le 4 (happy)
lei	183	lei 2 (thunder)
leng	184	leng 3 ku 4 (steeliness)
li	185	li 2 (pear)
lia	186	lia 2 (two)
lian	187	lian 3 (face)
liang	188	li 4 liang 4 (power)
liao	189	zuo 2 liao 4 (seasoning)
lie	190	lin 3 lie 4 (severe)
lin	191	lin 3 lie 4 (severe)
ling	192	ling 2 mu 4 (mausoleum)
liu	193	liu 2 dong 4 (flow)
lo	194	lo 5 (function word)
long	195	long 2 (dragon)
lou	196	lou 4 (leak)
lu	197	lu 4 di 4 (land)
luan	198	hun 4 luan 4 (chaos)
lue	199	ce 4 lue 4 (tactic)
lun	200	yi 4 lun 4 (discuss)
luo	201	xia 4 luo 4 (whereabouts)
lv	202	lv 4 se 4 (green)
ma	203	ma 3 (horse)
mai	204	mai 2 zang 4 (bury)

man	205	man 3 zu 2 (satisfy)
mang	206	cong 1 mang 2 (hurry)
mao	207	mao 1 (cat)
me	208	shen 2 me 5 (what)
mei	209	mei 2 you 3 (no)
men	210	wo 3 men 2 (we)
meng	211	meng 4 (dream)
mi	212	hun 1 mi 2 (coma)
mian	213	mian 4 ji 5 (area)
miao	214	miao 4 (temple)
mie	215	mie 4 jue 2 (annihilation)
min	216	ren 2 min 2 (people)
ming	217	ming 2 bai 2 (clearness)
miu	218	huang 1 miu 4 (absurd)
mo	219	mo 4 shui 3 (ink)
mou	220	mou 2 lue 4 (trick)
mu	221	mu 4 biao 1 (target)
na	222	na 4 li 3 (there)
nai	223	nai 3 nai 5 (grandmother)
nan	224	nan 2 fang 1 (south)
nang	225	nang 2 kuo 4 (include)
nao	226	re 4 nao 4 (liveliness)
ne	227	mu 4 ne 4 (numb)
nei	228	nei 4 bu 4 (inside)
nen	229	nen 4 lv 4 (light green)

neng	230	neng 2 gou 4 (be capable of)
ni	231	ni 3 (you)
nian	232	nian 2 (year)
niang	233	gu 1 niang 5 (girl)
niao	234	niao 3 (bird)
nie	235	nie 4 (bite)
nin	236	nin 2 (you)
ning	237	an 1 ning 2 (peace)
niu	238	niu 2 (bull)
nong	239	nong 2 min 2 (farmer)
nou	240	nou 4 (weeding)
nu	241	nu 3 li 4 (try hard)
nuan	242	wen 1 nuan 3 (warm)
nue	243	nue 4 dai 4 (abuse)
nuo	244	nuo 4 yan 2 (promise)
nv	245	nv 3 zi 3 (woman)
o	246	o 5 (function word)
ou	247	ou 1 yang 2 (Chinese first name)
pa	248	hai 4 pa 4 (scare)
pai	249	pai 4 qian 3 (send)
pan	250	pan 4 tu 2 (betrayed)
pang	251	pang 2 da 4 (huge)
pao	252	pao 3 (run)
pei	253	pei 2 ban 4 (accompany)
pen	254	pen 2 di 4 (pan)
peng	255	peng 2 pai 4 (surge)
pi	256	pi 2 fu 1 (skin)

pian	257	qi 1 pian 4 (cheat)
piao	258	piao 4 liang 4 (pretty)
pie	259	pie 1 kai 1 (put aside)
pin	260	pin 2 fan 2 (frequency)
ping	261	ping 2 zi 5 (bottle)
po	262	po 4 huai 4 (damage)
pou	263	pou 2 (hold something with cupped hand)
pu	264	pu 2 tao 5 (grape)
qi	265	qi 3 qiu 2 (beg)
qia	266	qia 4 dang 4 (proper)
qian	267	qian 1 xu 1 (humility)
qiang	268	qiang 2 da 4 (powerful)
qiao	269	qiao 3 miao 4 (artifice)
qie	270	xiu 1 qie 4 (shyness)
qin	271	qin 1 zi 4 (oneself)
qing	272	qing 1 song 1 (easy)
qiong	273	qiong 2 (poverty)
qiu	274	qiu 2 fan 4 (prisoner)
qu	275	qu 1 dong 4 (drive)
quan	276	quan 2 bu 4 (all)
que	277	que 4 ding 4 (ensure)
qun	278	qun 2 zhong 4 (crowd)

ran	279	ran 2 hou 4 (then)
rang	280	tu 3 rang 3 (soil)
rao	281	wei 2 rao 4 (surround)
re	282	re 4 (hot)
ren	283	ren 2 lei 4 (human being)
reng	284	reng 2 ran 2 (all the same)
ri	285	ri 4 chu 1 (sunrise)
rong	286	rong 2 yao 4 (glory)
rou	287	niu 2 rou 4 (beef)
ru	288	ru 2 guo 3 (if)
ruan	289	rou 2 ruan 3 (soft)
rui	290	rui 4 zhi 4 (smart)
run	291	shi 1 run 4 (wetness)
ruo	292	ruo 4 xiao 3 (puniness)
sa	293	sa 1 (three)
sai	294	bi 3 sai 4 (compete)
san	295	fen 1 san 4 (disperse)
sang	296	sang 4 (die)
sao	297	da 3 sao 3 (sweep)
se	298	yan 2 se 4 (color)
sen	299	sen 1 lin 2 (forest)
seng	300	seng 1 lv 3 (monk)
sha	301	sha 1 chang 3 (battlefield)
shai	302	ri 4 shai 4 (be exposed to the sun)
shan	303	shan 4 liang 2 (goodness)

shang	304	shang 4 mian 4 (top)
shao	305	shao 4 nian 2 (youth)
she	306	she 2 (snake)
shei	307	shei 2 (who)
shen	308	shen 2 me 5 (what)
sheng	309	sheng 1 yin 1 (voice)
shi	310	shi 4 fei 1 (dispute)
shou	311	shou 3 (hand)
shu	312	shu 4 mu 4 (tree)
shua	313	shua 1 zi 5 (brush)
shuai	314	shuai 1 da 3 (beat)
shuan	315	men 2 shuan 1 (latch)
shuang	316	shuang 1 (pair)
shui	317	shui 3 (water)
shun	318	shun 4 li 4 (all right)
shuo	319	shuo 1 (say)
si	320	si 4 (four)
song	321	song 4 bie 2 (send-off)
sou	322	sou 1 cha 2 (search)
su	323	su 4 du 4 (speed)
suan	324	suan 4 fa 3 (arithmetic)
sui	325	sui 1 ran 2 (though)
sun	326	sun 3 shi 1 (loss)
suo	327	suo 3 yi 3 (so)
ta	328	ta 1 men 2 (they)
tai	329	tai 4 (too)
tan	330	tan 2 hua 4 (talk)
tang	331	tang 2 (sugar)

tao	332	tao 2 pao 3 (flee)
te	333	te 4 bie 2 (special)
tei	334	tei 1 (very)
teng	335	ben 1 teng 2 (riot)
ti	336	ti 1 zi 5 (ladder)
tian	337	tian 1 (sky)
tiao	338	tiao 4 yue 4 (jump)
tie	339	tie 3 (iron)
ting	340	ting 1 (listen)
tong	341	tong 3 yi 1 (unify)
tou	342	tou 2 (head)
tu	343	tu 3 di 4 (earth)
tuan	344	tuan 2 jie 2 (solidify)
tui	345	tui 1 dong 4 (push)
tun	346	tun 2 ji 1 (store up)
tuo	347	tuo 1 yi 1 (undress)
wa	348	qing 1 wa 1 (frog)
wai	349	wai 4 mian 4 (outside)
wan	350	wan 4 (ten thousands)
wang	351	si 3 wang 2 (death)
wei	352	wei 3 da 4 (great)
wen	353	wen 2 hua 4 (culture)
weng	354	lao 3 weng 1 (oldman)
wo	355	wo 3 (I)
wu	356	wu 2 lun 4 (no matter what)
xi	357	xi 1 fang 1 (west)
xia	358	xia 4 mian 4 (bottom)

xian	359	xian 4 zai 4 (now)
xiang	360	xiang 4 (like)
xiao	361	xiao 3 (small)
xie	362	xie 3 (write)
xin	363	xin 1 (new)
xing	364	xing 2 dong 4 (act)
xiong	365	xiong 2 (bear)
xiu	366	xiu 1 xi 5 (break)
xu	367	xu 1 yao 4 (need)
xuan	368	xuan 1 bu 4 (declare)
xue	369	xue 3 (snow)
xun	370	xun 2 wen 4 (ask for)
ya	371	ya 4 zhou 1 (Asia)
yan	372	yan 4 zi 5 (swallow)
yang	373	tai 4 yang 2 (sun)
yao	374	yao 1 qiu 2 (require)
ye	375	ye 2 ye 5 (grandfather)
yi	376	yi 1 (one)
yin	377	yin 1 yue 4 (music)
ying	378	ying 1 gai 1 (should)
yo	379	yo 5 (function word)
yong	380	yong 3 yuan 3 (forever)
you	381	you 2 yu 2 (due to)
yu	382	yu 2 (fish)
yuan	383	yuan 2 lai 2 (formerly)
yue	384	yue 4 (month)
yun	385	yun 2 (cloud)
za	386	za 2 luan 4 (disorder)

zai	387	zai 4 jian 4 (farewell)
zan	388	zan 2 men 5 (our)
zang	389	ang 1 zang 1 (dirty)
zao	390	zao 3 chen 2 (morning)
ze	391	ze 2 ren 4 (response)
zei	392	zei 2 (thief)
zen	393	zan 4 yang 2 (praise)
zeng	394	zeng 1 jia 1 (add)
zha	395	zha 4 dan 4 (bomb)
zhai	396	qian 4 zhai 4 (owe)
zhan	397	zhan 4 li 4 (stand)
zhang	398	zhang 1 kai 1 (open)
zhao	399	zhao 1 huan 4 (summon)
zhe	400	zhe 4 li 3 (here)
zhei	401	zhei 5 (this)
zhen	402	zhen 1 zheng 4 (real)
zheng	403	zheng 4 yi 4 (justice)
zhi	404	yi 4 zhi 4 (restrain)
zhong	405	zhong 1 guo 2 (China)
zhou	406	si 4 zhou 1 (around)
zhu	407	zhu 1 (pig)
zhua	408	zhua 1 (grab)
zhuai	409	zhuai 4 (pull)
zhuan	410	zhuan 1 (brick)
zhuang	411	zhuang 1 (pretend)
zhui	412	zhui 1 (chase)
zhun	413	zhun 3 (precise)
zhuo	414	zhuo 1 (table)

zi	415	zi 4 (character)
zong	416	zong 3 (total)
zou	417	zou 3 (walk)
zu	418	zu 1 (rent)
zuan	419	zuan 4 (diamond)
zui	420	zui 4 (most)
zun	421	zun 1 (respect)
zuo	422	zuo 4 (sit)



Japanese Phonemes

SAPI Supported Japanese Phonemes (Katakana)

Symbol	Phonetic	Unicode Value	Remarks
'		0027	Accent position
+		002B	Accent boundary
		007C	Phrase boundary
.		002E	Sentence end (standard)
?		003F	Sentence end (interrogative)
!		0021	Sentence end (exclamation)
-		005F	One mora pause
		309C	Semi-voiced sound
	a	30A1	Modifier
	a	30A2	
	i	30A3	Modifier
	i	30A4	
	u	30A5	
	u	30A6	

	e	30A7	Modifier
	e	30A8	
	o	30A9	Modifier
	o	30AA	
	ka	30AB	
	ga	30AC	
	ki	30AD	
	gi	30AE	
	ku	30AF	
	gu	30B0	
	ke	30B1	
	ge	30B2	
	ko	30B3	
	go	30B4	
	sa	30B5	
	za	30B6	
	shi	30B7	
	ji	30B8	
	su	30B9	
	zu	30BA	
	se	30BB	

	ze	30BC	
	so	30BD	
	zo	30BE	
	ta	30BF	
	da	30C0	
	chi	30C1	
	di	30C2	Deprecated - use 30B8
	q	30C3	
	tsu	30C4	
	du	30C5	Deprecated - use 30BA
	te	30C6	
	de	30C7	
	to	30C8	
	do	30C9	
	na	30CA	
	ni	30CB	
	nu	30CC	
	ne	30CD	
	no	30CE	
	ha	30CF	

	ba	30D0	
	pa	30D1	
	hi	30D2	
	bi	30D3	
	pi	30D4	
	hu	30D5	
	bu	30D6	
	pu	30D7	
	he	30D8	
	be	30D9	
	pe	30DA	
	ho	30DB	
	bo	30DC	
	po	30DD	
	ma	30DE	
	mi	30DF	
	mu	30E0	
	me	30E1	
	mo	30E2	
	ya	30E3	Modifier
	ya	30E4	

	yu	30E5	Modifier
	yu	30E6	
	yo	30E7	Modifier
	yo	30E8	
	ra	30E9	
	ri	30EA	
	ru	30EB	
	re	30EC	
	ro	30ED	
	wa	30EE	Modifier
	wa	30EF	
	wi	30F0	Deprecated - use 30A4
	we	30F1	Deprecated - use 30A8
	wo	30F2	
	nn	30F3	
	vu	30F4	
	ka	30F5	
	ke	30F6	
	va	30F7	
	vi	30F8	

	ve	30F9	
	vo	30FA	
		30FB	Middle dot
		30FC	Prolonged sound
		30FD	Iteration
		30FE	Voiced iteration



Further Reading

The descriptions for the following items are contained in the Microsoft[®] [Platform Software Development Kit](#) (SDK). 🌐➡

IStream

IServiceProvider



SDK Samples, Tools, and Tutorials

The SDK provides examples of applications using SAPI 5 within a representative range of uses. These examples may contain only the executable files and are provided for illustrative purposes only. Most however, contain source code. You are free to model applications on these samples.

Since SDK samples are meant to demonstrate basic speech technologies, not all options or contingencies are checked. For instance, some examples the words may display incorrectly if the engine language is not the same as the system language. In such cases, this is the expected result and it is not an error of the speech system. Additionally, error conditions may not always be checked as thoroughly as robust applications should.

The following sections cover tools, samples, and tutorials supporting speech functions:

- [SDK Samples \(C/C++\)](#)
- [SDK Samples \(Automation\)](#). Includes Visual Basic, JScript and C# examples
- [SDK Samples \(Utilities\)](#)
- [SDK Tutorials](#)



SDK Samples for C/C++

The following topics are available:

Demonstration only (no source code)

- [Age of Empires II Speech Interface](#)
- [Reco \(a recognition test tool\)](#)

Samples with source code

- [CoffeeS0 Sample Application](#)
- [CoffeeS1 Sample Application](#)
- [CoffeeS2 Sample Application](#)
- [CoffeeS3 Sample Application](#)
- [CoffeeS4 Sample Application](#)
- [CoffeeS5 Sample Application](#)
- [CoffeeS6 Sample Application](#)
- [Dictation Pad](#)
- [Simple Dictation](#)
- [TTS Application](#)
- [Talkback](#)
- [Simple Telephony](#)



Age of Empires Speech

Introduction

Age of Empires® Speech is an application adding a speech interface to Microsoft's Age of Empires II: The Age of Kings (AOE II). You do not need a special version of AOE II; it is run normally. The speech interface passes keyboard commands to the game. That is, you can replace most keyboard tasks with speech commands. Some commands, such as scrolling and changing game speeds, are not currently supported by the speech interface. Also, you cannot implement certain mouse operations such as selecting and moving units, or placing buildings with speech commands.

Start Age of Empires Speech to enable speech operations. In the main Age of Empires Speech dialog window, select **Listening** to enable recognition, and then open AOE II. After selecting the options and a scenario, the game will begin. At that point, you may speak commands for game play. No action will take place if commands not recognized. You will not need additional speech and command training to play; however, speak the commands clearly, slowly, and deliberately. Your familiarity at playing the game using speech commands will increase over time and your proficiency will improve.

You will find that a complete list of commands and their syntax is available in Commands. Use speech in the same manner as the keyboard for issuing commands. Some commands are global and you may issue them at any time. These include chat commands and pause game, or menu items such as diplomacy, objectives, or display game time. Other commands are specific to units or buildings. With the target selected, issue a command. If the command is appropriate, it will be carried out. For example, select a barrack; you may then set a gather point, train specific units, or delete the barrack altogether. However, if you attempt to train a cavalry unit there, this command is

inappropriate and no action will be taken.

Options

Several options are presented in the dialog window. By default, all options are initially on except for Listening. Selecting various options will enable or disable entire categories of commands. Check each option for complete access to all commands.

Activate game commands

Activates a series of administrative commands and menus. These include diplomacy, chat and objective windows as well as game pause. It also displays the technology tree. You can initiate, forward, and send chat by speech commands, but you must still type the actual content.

Activate unit commands

Activates commands for villager units. You must activate the unit with the mouse, but once selected, you can issue speech commands. If a command is not applicable for the unit or the group of units, no action will be taken. Examples include setting gather points; attempting to select economic or military buildings; or the villager unit attempting to pack.

Military unit commands

Activates commands for military units. You must activate the unit with the mouse, but once selected, you can issue speech commands. If a command is not applicable for the unit or the group of units, no action will be taken. Examples include the military unit to take an aggressive, defensive, or stand ground posture, form a line, patrol an area, or guard a building.

Activate training commands

Activates the build, create and train commands. This command allows the creation of many items including types of buildings, villagers, and different military units such as swordsmen, knights, and archers. You can also use this command to activate weapons including fire ships, trebuchets, and battering rams.

Activate view commands

Activates the Go to and View commands. The commands may be simple such as "go to barracks," or "go to last notification." They may also be more complex such as "go to next idle villager unit" although you may omit the final "unit."

Activate idle commands

Activates the idle command for villagers or military units.

Activate villager build commands

Activates a series of villager/ship build commands. You must activate the unit with the mouse, but once selected, you can issue speech commands. You can issue the build order directly without selecting it from the build list. For example, a selected villager may be told to "build house." An image of the dimmed house will appear on the screen, then use the mouse to place it. Other examples of villager build commands include building the market, blacksmith, farm, fish trap, and military objects such as archery range, barracks and stone wall.

Automatically view before training

Displays the building prior to issuing the build or train command. This allows you to view the building centered on the screen. Unlike the other categories, there are no subsequent commands available.

Listening

Enables speech recognition. Once activated, Age of Empires Speech attempts to issue keystrokes if a command is recognized. AOE II is intended to be the recipient of the commands. However, if another active application capable of accepting keyboard input is in the Windows foreground, it is possible that the keystrokes will be passed along to that window instead. Take care to avoid unintended input to the foreground application. As mentioned above, open AOE II immediately after Age of Empires Speech.



Reco

Introduction

Reco is a speech tool that you can use to examine and test the speech process. You may speak using a microphone for either dictation or command and control. Alternatively, you may directly type a command and control order in an edit box and submit it to the speech recognition engine.

During the recognition process, events will be displayed in the top window as they occur. An event log is kept and to examine the details of each one, double-click the item. You may use this log to trace the sequence and number of events. You may also examine speech results. Double-click the text or phrase to display more detailed information about the text in the bottom window.

Options

The dialog box presents several options. By default, all options are initially on. The following options display detailed information suited for speech testing.

Create Recognition Context

Creates the recognition context for the selected engine. This must be off before selecting a different engine. Select this to activate the engine. You must select it before any speech is recognized.

Retain Reco Audio

Keeps the recorded sound of the actual spoken content in memory. Select this option and double-click the text in the display area of either the Events or Properties window to play back the spoken text. If it is not selected, the playback will be in

a synthesized text-to-speech voice.

Activate Microphone

Controls the microphone's input. Select this option to enable the microphone to accept sound input. If it is not selected, the microphone will be blocked and no sound will be registered.

Load Dictation

Controls loading of dictation grammar. Select it to load the grammar for dictation and enable the speech recognition engine to process free-formed speech. You must load this grammar before selecting Activate Speech. If you do not select Load Dictation, the grammar will be unloaded.

Activate Dictation

Controls the activation status of dictation features. If you have loaded the dictation grammar (see Load Dictation), select this to allow speech attempts to be recognized as free-formed dictation.

Load Command and Control

Controls loading of command and control grammar. Select this to load the grammar for dictation and enable the speech recognition engine to process command-oriented speech. You must load this grammar before selecting Activate command and control. If you do not select Load Command and Control, the grammar will be unloaded.

Activate Command and Control

Controls the activation status of command and control features. If you have loaded the command and control grammar, (see Load Command and Control), select this to allow speech attempts to be recognized as commands according to the rules

defined the in current grammar. By default, the Solitaire grammar is installed. However, you may change the grammar using the Command and Control->Load Grammar menu item.

Load Spelling

Enables or disables the letter-by-letter spelling feature. You must enable spelling before you select Activate Spelling. If Load Spelling is not selected, it will disable spelling.

Activate Spelling

Controls the activation status of the letter-by-letter spelling function. If you have loaded the spelling grammar, (see Load Spelling), select this to activate the spelling capability of the speech recognition engine. Using Activate Spelling, you can spell words in a letter-by-letter fashion while in dictation mode. For example, you can say, "Let's take the dogs on a w-a-l-k," and the engine will attempt to recognize the sentence as "Let's take the dogs on a walk."

Shared Recognizer

Selects the engine as a shared engine. Using Shared Recognizer, other speech-enabled applications running at the same time can use the microphone. To do this:

1. In Control Panel, double-click the **Speech** icon.
2. On the Speech Recognition tab, in the Speech Engine window, select an engine.

InProc

Selects an engine as an in-process (InProc) or non-shared engine. InProc restricts speech-enabled applications running at the same time from using the microphone. Because the engine will not be shared among other applications, using this option you can select different engines from the drop-down menu

rather than using the Speech icon. Selecting an InProc engine will not change the engine used by shared applications or the one selected in the Speech Engine window.

Event Window

Displays events and any associated results. The name of the event is displayed inside brackets. Certain events such as recognition or a hypothesis, have text associated with them and that text is displayed immediately under the event. Double-click the text to hear it spoken. If Retain Reco Audio is currently selected, the playback will be in your voice, recorded when it was spoken; otherwise, the text-to-speech voice will be played back. You may change the synthesized voice using the TTS tab in Speech properties. Double-click the event to display additional information about it. The Results window will update to the new information.

Results Window

Displays details of the properties and values associated with the recognition result. This information includes the applicable rule, the text of the recognition, the parsing of information, and the individual elements of each word.

Alternates

Command and Control Edit Box

Enables you to directly enter a command and control phrase. If any text appears in the box, **Submit** activates it. Click **Submit** to enter a command, bypassing the microphone. In this way, the microphone does not need to be active or even present to test or watch the recognition process.

Submit

Submits any text in the command and control edit box, see

Command and Control Edit Box.



CoffeeS0

Introduction

CoffeeS0 is the first sample application in a series called Coffee. It uses a consistent coffee shop motif. You will eventually be able to enter the shop, go to the order counter to order drinks, go to the gift store, or speak to management. The samples are intended to demonstrate adding speech recognition to an application. They are designed for the application-level (API) programmer and for those not familiar with speech technology. Writing engines such as speech recognition or text-to-speech, also called device driver programming, will be covered separately. Each sample will progressively add new features and increase in complexity. The tutorial chapters explain in detail the particulars of the code. You are encouraged to read each chapter.

As the introductory sample, CoffeeS0 is a simple application. There is one window and a limited vocabulary from which to speak. After opening CoffeeS0, you may speak any of the commands. If successfully recognized, the window displays the response, "Please order when ready!" To keep the application simple at this point, no other commands may be used. If the command was not successfully recognized, no action will take place. Due to the speed and processing capabilities of some computers, there might be a slight delay before the CoffeeS0 responds. If after a moment nothing happens, try the command again.

Commands may not be recognized for two reasons. First, the speech may not have been clear. Perhaps the words were not spoken clearly enough or distinctively enough. Speak the command again more slowly and clearly. Second, the words spoken may not have been in the command list. Look at the available commands and speak again. CoffeeS0 has a limited command list. If a word used is not in the list, the command will

not be recognized.

To quit CoffeeSO, click the **Close** button in the upper right of the window frame.

Commands

Choosing one word from each line of a category forms the command. Commands in parenthesis are optional and do need to be included. Words or phrases separated by slashes indicate that any of the choices listed may be used although only one may be selected. Sections marked RULEREf indicate words or phrases may be chosen from the corresponding rule ID. Rule names are the same as listed in the corresponding XML configuration file.

For example, you can say, "enter counter," "please enter counter," or "please enter the counter." All three are recognizable commands. Since CoffeeSO can only take you to the ordering counter, "please enter the shop," and "please enter the store," will have the same effect. Partial phrases or words not listed below may not be used. "Please enter the restaurant," or simply "counter" will not be recognized. The following commands are available:

Command List

XML rule ID: VID_Navigation

- (please)
- enter/go to
- RULEREf: VID_Place

XML rule ID: VID_Place

- (the)

- counter/shop/store



CoffeeS1

Introduction

CoffeeS1 is the second sample application in a tutorial series named Coffee. It uses a consistent coffee shop motif. You will eventually be able to enter the shop, go to the order counter to order drinks, go to the gift store, or speak to management. The samples are intended to demonstrate speech recognition capabilities within an application. They are designed for the application-level (API) programmer and for those not familiar with speech technology. Writing engines such as speech recognition or text-to-speech, also called device driver programming, will be covered separately. Each sample will progressively add new features and increase in complexity. The tutorial chapters explain in detail the particulars of the code. You are encouraged to read each chapter.

As the second example, CoffeeS1 builds on the framework of CoffeeS0. There is one window and although there is a limited vocabulary from which to speak, it is expanded from CoffeeS0. CoffeeS0 restricted you to moving to the counter only. In contrast, CoffeeS1 now lets you order one of a variety of coffee drinks. After opening CoffeeS1, you may speak any of the commands. If successfully recognized, the window displays the response, "Please order when ready!" If the command was not successfully recognized, no action will take place. Due to the speed and processing capabilities of some computers, there might be a slight delay before CoffeeS1 responds. If after a moment nothing happens, try the command again.

Commands may not be recognized for two reasons. First, the speech may not have been clear. Perhaps the words were not spoken clearly enough or distinctively enough. Speak the command again more slowly and clearly. Second, the words spoken may not have been in the command list. Look at the available commands and speak again. CoffeeS0 has a limited

command list. If a word used is not in the list, the command will not be recognized.

To quit CoffeeS1, click the **Close** button in the upper right of the window frame.

Commands

Command sequences are built using at least one word from among the various categories. Commands are grouped into two major (or top-level) categories: navigation and drink ordering. Navigation allows you to move around the coffee shop to places such as the counter, shop, or store. In CoffeeS1, the effects of going to any location are minimal and any navigation command takes you to the counter. The distinction allows the code sample to demonstrate additional SR features. Subsequent Coffee examples will expand on this. Drink ordering allows you to place drink requests.

In either case, words or phrases may be selected from each category. Some are marked as Required and others are marked as Optional. You must use at least one word or phrase from the list marked Required to successfully initiate the command. Optional words, though not required, provide a more natural speech. As a special case, the drink orders may include words from up to seven of the categories and may be spoken in any order. However, at least one category is required to successfully match command requirements.

For example, at the minimum you can say, “get me mocha.” This satisfies the command rules of the required phrase (“get me”) from the VID_EspressoDrinks required list and at least one of the seven other drink categories, in this case VID_DrinkType. Likewise, you could also say, “I would like a tall hazelnut two percent latte.” Words not appearing in the list may not be used. “Give me a latte,” would fail since “give me” is not on the list. “Please get me a” will also fail. Although the required top-level phrase is correctly used, a subsequent drink was not ordered. The following commands are available:

Command List

Choosing one word from each line of a category forms the command. Commands in parenthesis are optional and do need to be included. Words or phrases separated by slashes indicate any of the listed choices may be used although only one may be selected. Sections marked RULEREf indicate words or phrases may be chosen from the corresponding rule ID. Rule names are the same as listed in the corresponding XML configuration file.

Navigation

Requires a match from each VID_Navigation and VID_Place.

XML rule ID: VID_Navigation

- Optional: Please
- Enter
- Go to

XML rule ID: VID_Place

- Optional: The
- Counter
- Shop
- Store

Drink Ordering

Requires a match from VID_EspressoDrinks and at least one (but

no more than seven) of the subsequent categories.

XML rule ID: VID_EspressoDrinks

- May I have
- Can I have
- Can I get
- Please get me
- Get me
- I'd like
- I would like
- Optional: a

XML rule ID: VID_Iced

- Iced

XML rule ID: VID_Decaf

- Decaf
- Decaffeinated

XML rule ID: VID_Shots

- Single
- Double
- Triple
- Quad

XML rule ID: VID_Size

- Short
- Tall
- Grande

XML rule ID: VID_Syrup

- Hazelnut
- Irish cream
- Almond
- Peppermint

XML rule ID: VID_Milk

- Nonfat
- Two percent
- Whole

XML rule ID: VID_DrinkType

- Latte/Mocha/Espresso/Americana/Cappuccino



CoffeeS2

Introduction

CoffeeS2 is the third sample application in a tutorial series named Coffee. It uses a consistent coffee shop motif. Customers enter the shop, go to the service counter, speak to order drinks or to enter the front office.

The samples are intended to demonstrate speech recognition capabilities within an application. They are designed for the application-level (API) programmer and for those not familiar with speech technology. Each sample will progressively add new features and increase in complexity. The tutorial chapters explain in detail particulars of the code. You are encouraged to read each chapter. Writing engines such as speech recognition or text-to-speech, also called device driver programming, will be covered separately. The samples can use engines provided by the SAPI SDK or third party SAPI-compliant engines.

Using CoffeeS2

As the third example, CoffeeS2 builds on the framework of its predecessors. There is one window and an expanded vocabulary from which to order drinks and move around the business. In addition to placing an order, you can move to both the counter and the office. However, drinks may not be requested while in the office.

CoffeeS2 allows you to speak any of the CoffeeS1 commands. If successful, the window displays the action such as "please order when ready!" If the command was not successfully recognized, no action will take place.

Due to the speed and processing capabilities of some computers, there might be a slight delay before CoffeeS1 responds. If after a moment nothing happens, try the command again. Commands may not be recognized for two common reasons. One, the speech may not have been clear. Perhaps the words were not spoken clearly enough or distinctively enough. Speak the command again more slowly and clearly. Second, the words spoken may not have been in the command list. Look at the commands available and speak again. The Coffee examples have a limited command list and if a word is used but is not from those commands, it will not be recognized.

To quit CoffeeS2, click the **Close** button in the upper right of the window frame.

New Features

See CoffeeS1 users note for a detailed description about using the application.

A new navigation command has been added allowing you to visit the office. You may request "go to the office." The screen will change to "Welcome to the SAPI coffee shop office." Drinks may not be ordered while in the office. As a result, drink commands are unavailable while in the office. To leave, navigate to another part of the business. Like CoffeeS1, navigation commands involving the counter, store, and shop take you to the same place.

A second new feature asks you to repeat the order if it is not understood. If a recognition takes place but the word or phrase is not in the command list, CoffeeS2 will display "Sorry, I didn't get that order. Please try again."

New Commands

Choosing one word from each line of a category forms the command. Commands in parenthesis are optional and do need to be included. Words or phrases separated by slashes indicate any of the listed choices may be used although only one may be selected. Sections marked RULEREF indicate words or phrases may be chosen from the corresponding rule ID. Rule names are the same as listed in the corresponding XML configuration file.

XML rule ID: VID_Place

- office



CoffeeS3

Introduction

CoffeeS3 is the fourth sample application in a tutorial series named Coffee. It uses a consistent coffee shop motif. Customers enter the shop, go to the service counter, speak to order drinks or to enter the front office.

The samples are intended to demonstrate speech recognition capabilities within an application. They are designed for the application-level (API) programmer and for those not familiar with speech technology. Each sample will progressively add new features and increase in complexity. The tutorial chapters explain in detail particulars of the code. You are encouraged to read each chapter. Writing engines such as speech recognition or text-to-speech, also called device driver programming, will be covered separately. The samples can use engines provided by the SAPI SDK or third party SAPI-compliant engines.

Using CoffeeS3

CoffeeS3 is identical to the previous CoffeeS2 sample. See the CoffeeS2 guide for detailed instructions for using this.

However, a synthesized voice has been used for this module. At certain times, the text-to-speech engine will speak the command, an order, or a greeting. For example, the initial screen will not only display the greeting, but will also say, "Welcome to the SAPI coffee shop. Speak for service."

New Commands List

There are no new commands for CoffeeS3.



CoffeeS4

Introduction

CoffeeS4 is the fifth sample application in a tutorial series named Coffee. It uses a consistent coffee shop motif. Customers enter the shop, go to the service counter, speak to order drinks or to enter the front office.

The samples are intended to demonstrate speech recognition capabilities within an application. They are designed for the application-level (API) programmer and for those not familiar with speech technology. Each sample will progressively add new features and increase in complexity. The tutorial chapters explain in detail particulars of the code. You are encouraged to read each chapter. Writing engines such as speech recognition or text-to-speech, also called device driver programming, will be covered separately. The samples can use engines provided by the SAPI SDK or third party SAPI-compliant engines.

Using CoffeeS4

CoffeeS4 introduces the concepts of resources and resource management. SAPI stores information in the form of tokens. These tokens are used later to instantiate features such as voices and recognizers. However, programmers can query SAPI for the presence of tokens to learn more about available features. For example, each available voice is kept as a token.

CoffeeS4 displays available voices and may even speak using the currently active voice. Three new commands are used. To do so, enter the office by saying, “go to the office” or “enter office.” Once there, display the voice list by saying, “manage the employees.” A list of available voices will display on the right side of the screen. The active voice will be indicated in red.

To have the employee speak, say, “hear them speak.” The statement “I will be the best employee you've ever had. Let me work.” will be spoken in the current voice. The voice may be changed using Speech properties in Control Panel.

New Commands List

Choosing one word from each line of a category forms the command. Commands in parenthesis are optional and do not need to be included. Words or phrases separated by slashes indicate any of the listed choices may be used although only one may be selected. Sections marked RULEREf indicate words or phrases may be chosen from the corresponding rule ID. Rule names are the same as listed in the corresponding XML configuration file.

XML rule ID: VID_Manage

- (please)
- manage
- (the)
- RULEREf: VID_ThingsToManage

XML rule ID: VID_ThingsToManage

- employees

XML rule ID: VID_HearTheVoice

- hear them speak



CoffeeS5

Introduction

CoffeeS5 is the sixth sample application in a tutorial series named Coffee. It uses a consistent coffee shop motif. Customers enter the shop, go to the service counter, speak to order drinks or to enter the front office.

The samples are intended to demonstrate speech recognition capabilities within an application. They are designed for the application-level (API) programmer and for those not familiar with speech technology. Each sample will progressively add new features and increase in complexity. The tutorial chapters explain in detail particulars of the code. You are encouraged to read each chapter. Writing engines such as speech recognition or text-to-speech, also called device driver programming, will be covered separately. The samples can use engines provided by the SAPI SDK or third party SAPI-compliant engines.

Using CoffeeS5

CoffeeS5 expands the concepts of resources and resource management introduced in CoffeeS4. Using information that was learned by polling tokens about available voices, CoffeeS5 allows users to change the active voice. In doing so, a dynamic grammar is used. In the previous Coffee samples, all the speech commands were determined ahead of time and could not be changed. For example, the drinks were limited to five basic types and a new one could not be added. A dynamic grammar allows adding or removing commands during the program execution.

To change the voices, enter the office by saying, "go to the office" or "enter office." Once there, display the voice list by saying, "manage the employees." A list of available voices will display on the right side of the screen. The active voice will be indicated in red. To hear the employee speak, say, "hear them speak." The statement "I will be the best employee you've ever had. Let me work." will be spoken in the current voice.

To change the voice, say the voice name as it appears on the screen. For example, if "Microsoft Mary" is displayed, say, "Microsoft Mary." The highlighting will change to the selected voice. Having the employee speak will do so in the voice. Additionally, the list of available voices may be filtered by gender. The left side of the screen displays available commands for this. For example, "Show males only," will display only the male voices.

Some voices may not be applicable to this example. For instance, Sample TTS Voice is a composite voice for use with the SDK application MkVoice. The voice contains only seven words with an eighth word being the default for all other words. As a result, it will say "blah" most of the time. In the same way, the MS Simplifying Chinese Voice will spell the content rather than speak it.

New Commands List

Choose one word from each line of a category forms the command. Commands in parenthesis are optional and do need to be included. Words or phrases separated by slashes indicate any of the listed choices may be used although only one may be selected. Sections marked RULEREF indicate words or phrases may be chosen from the corresponding rule ID. Rule names are the same as listed in the corresponding XML configuration file.

XML rule ID: VID_OtherRules

- (show) males only / (show) females only / (show) both genders

Rule ID: DYN_TTSVOICERULE

This is dynamic rule generated during run time. No XML code is present. After generation, it displays the names for all the available voices. The contents of the rule is displayed on the right side of the screen in the CoffeeS5 office after issuing the "manage the employees" command. The rule is generated at the time that command is issued and is destroyed after leaving the office afterward.



CoffeeS6

Introduction

CoffeeS6 is the seventh and final sample application in a tutorial series named Coffee. It uses a consistent coffee shop motif. Customers enter the shop, go to the service counter, speak to order drinks or to enter the front office.

The samples are intended to demonstrate speech recognition capabilities within an application. They are designed for the application-level (API) programmer and for those not familiar with speech technology. Each sample will progressively add new features and increase in complexity. The tutorial chapters explain in detail particulars of the code. You are encouraged to read each chapter. Writing engines such as speech recognition or text-to-speech, also called device driver programming, will be covered separately. The samples can use engines provided by the SAPI SDK or third party SAPI-compliant engines.

Using CoffeeS6

CoffeeS6 expands the concepts of grammars and dictations as they have been presented in the previous examples. The early Coffee examples used a fixed grammar to include a select set of words. CoffeeS5 introduced dynamic grammars so you could add new words to an existing word list. CoffeeS6 goes one additional step and uses dictation. For dictation, you are no longer limited to an explicit list but may now use almost any word or words. However, instead of demonstrating this as a free-formed dictation application, CoffeeS6 uses it in association with existing grammars.

To showcase this ability, CoffeeS6 enables you to rename the coffee shop. From the office, a new option is presented: Manage Store Name. Speak this command and the screen changes again. A new order is displayed on the screen allowing you to rename the shop. Say "Rename the coffee shop to" and speak a name. CoffeeS6 then speaks the new name of the store. For example, by saying, "Rename the coffee shop to My Coffee Emporium," the new name will be "My Coffee Emporium." The name will also be displayed throughout CoffeeS6. After saying the command, you may then navigate to another location or continue to rename the store any number of times.

New Commands List

Choosing one word from each line of a category forms the command. Commands in parenthesis are optional and do not need to be included. Words or phrases separated by slashes indicate any of the choices listed may be used although only one may be selected. Sections marked RULEREf indicate words or phrases may be chosen from the corresponding rule ID. Rule names are the same as those listed in the corresponding XML configuration file.

The following rule is used to rename the coffee shop. However, because the shop name is not limited to a particular element from list, the asterisk acts as a wildcard. Any word or words are permitted. Additionally, the plus sign forces a greater level of confidence. See VID_ThingsToManage for a more detailed explanation of the sign. Requiring greater confidence forces the speech recognition engine to spend additional time processing the word. Greater confidence results in either better recognition of the word, or a higher confidence of the word returned by the speech recognition engine.

XML rule ID: VID_Rename

- Rename the coffee shop to *+

The last two items of the next rule are new to CoffeeS6. As expected, you can say "shop" and "store." However, the command allows for a change of emphasis on words. The plus or minus sign changes the required confidences of the word recognition. By increasing the required confidence level, the speech recognizer demands a higher quality of the word being recognized. In a similar way, decreasing the confidence allows for greater latitude of the word's recognized quality. This way, certain words can be emphasized. For instance, the following

rule ensures that the word "name" is recognized. Because regional accents or background noise may detract from the quality of the word, the new emphasis makes certain that the sound heard was actually "name." In turn, the rule places less emphasis on the word "shop" or "store." Although these words are still required, it is not as important to be certain.

XML rule ID: VID_ThingsToManage

- employees
- -shop +name
- -store +name

Additionally, two existing rules have new list elements. The ellipsis disregards any words spoken for the current element. For example, using VID_EspressoDrinks you can say "May I have a coffee," or "Get me coffee," as expected. However, by including the ellipsis in the rule, you may also say "I dunno, how about a coffee." The statement will be recognized since everything before the drink name can essentially be ignored. Including the ellipsis as a list element significantly reduces the constraint of listing all words individually.

XML rule ID: VID_EspressoDrinks

XML rule ID: VID_OrderList

- ...



Dictation Pad

Dictation Pad is an example of a speech-enabled word processor. This sample application is intended to demonstrate many of the features for SAPI 5 in a single coherent application. It is not a full featured speech-enabled application, although the foundations of many of options are present.

Using Dictation Pad you can speak into a microphone and, following successful speech recognition (SR), Dictation Pad will display the sentence on the screen as text. The words can also be spoken back in a text-to-speech (TTS) voice, highlighting words as they are spoken. Features include the following:

- Dictation - Recognizes words in any context.
- Command and control - Recognizes a limited selection of words and applies them to control the flow of Dictation Pad. This includes using speech to select items from menus and changing the SR mode from dictation to command and control.
- Playback - Plays back words appearing on the screen in a TTS voice.
- Speakback - Keeps an audio record of the actual spoken content. You can play it back to confirm or verify speech recognition.
- Phrase tracking - Maintains a list of phrase element information. This can locate the parts of an SR phrase even if the dictation becomes broken or disjointed. It also demonstrates text replacement such as inverse text normalization. This is the process of converting text to numbers such as "one two three" into "1-2-3" or "first" into "1st."

- Word Alternates - Displays a list of alternates for the recognized text. From this list you can select a replacement for the original text.
- Adding words to a grammar - Demonstrates the SR engine's capability to add words or phrases to an existing grammar or word database. Adding a word allows it to be recognized on subsequent occurrences.
- Document management - Saves documents and opens them retaining the data associated with the recognized SR results.

The complete code base for Dictation Pad is included with the SDK and you are encouraged to look at and examine the code. It is intended to be a training aid and to demonstrate as many features as possible.

Note About SR/TTS Engines

Dictation Pad supports common SAPI features such as various user interface calls. However, SR or TTS engines are not required to provide all the features. The current engine will be queried for features that Dictation Pad supports. If available, Dictation Pad will use a feature; otherwise, the feature will not be available or the menu item will be inactive.

For example, Dictation Pad uses the SAPI feature of the Add/Remove Words interface. The Microsoft ASR Version 5 engine supports this feature and it is available to Dictation Pad. The SAPI 5 Sample Engine from the SAPI SDK does not support it; hence, Dictation Pad deactivates the Add/Remove dialog menu item.

Dictation Pad Menu/Toolbar

The main window of the Dictation Pad contains both a toolbar and a menu bar you can use to control all the application's functions. The toolbar is a convenience feature and you may access many of its functions through the menu.



File Menu



File menu items control the documents that are used in the application.

New

Creates a new document. Multiple documents cannot be open at the same time so the existing document must be saved and closed, or discarded before creating the new one.

Open

Opens a previously saved document. Similar to creating a new document, any current one must be saved first.

Save

Saves the current file. Files are saved as a proprietary *.dpd format.

Save As

Saves the current file under a new or different name.

Exit

Quits Dictation Pad. Unsaved files may be saved before exiting.

Edit Menu



Edit menu items control the copy and pasting of text for the current document.

Cut

Copies the text to the clipboard and removes the selected text from the document.

Copy

Copies the text to the clipboard.

Paste

Copies the text from the clipboard to the document. The text is either placed starting with the cursor insertion point or, if text is selected, replaces the selection with the new text.

Voice Menu



Voice menu items control the speech enabling aspect of the application.

Listen for Dictation

Enables Dictation Pad to receive speech in dictation form rather than for command and control. This means you can speak any words or combination of words and they will



be recognized. This option is mutually exclusive from Listen for Commands.

Listen for Commands

Enables Dictation Pad to receive speech in command and control form rather than as dictation. You are limited to words defined for application control only. This includes a word equivalent for most menu items or buttons. This option is mutually exclusive from Listen for Dictation.



Playback

Reads back the text. To have a portion of the text read back, select the text you want. If you do not select any text, reading begins at the cursor insertion point. If the insertion point is at the end of the document, the entire document will be read.

If the text was originally dictated, the playback will be the recorded audio of your voice. If you did not dictate the text, but rather typed or pasted it into the document, the TTS voice will read the text.



Grammar Activation

Turns grammars on or off. Words or phrases added through the Add/Delete Word(s) option are neither available nor recognized if the grammar is turned off. By default, this option is turned off.



Add/Delete Word(s)

Brings up the SR engine-specific user interface so you can add words to or delete words from the lexicon (or dictionary).

Select Whole Words

Sets the selection state for word highlighting. If selected, the entire word will be automatically highlighted during the selection process. Otherwise, only the words and letters actually selected will remain selected.

Shared recognition engine

Sets resource sharing. By default, Dictation Pad uses the "In process" (also referred to in SAPI 5 as InProc) resource model that causes the SR engine to exist in the same process as Dictation Pad and restricts other applications from using resources required by this application. Other resources include the microphone, so that all audio input is given to Dictation Pad rather than another application currently running. If selected, the SR engine may reside in a separate process.

Voice Training

This brings up Speech Training Recognition Wizard for training or additional training. This wizard is accessed through Speech properties in Control Panel. On the SR tab, click **Train Profile** to bring up the voice training wizard.

Microphone Setup

This brings up the Microphone Wizard for adjusting the microphone set up. This wizard is accessed through Speech properties in Control Panel. On the SR tab, click **Configure Microphone** to bring up the microphone wizard.

Using Dictation Pad

Speech Recognition

By default, Dictation Pad starts in dictation mode with the microphone off. To start speech recognition, click **Microphone**, select the **Voice->Microphone** menu item or use control-m. Begin speaking. To indicate processing, ellipses ("...") display in the window. During the recognition process as SAPI starts returning words or phrases, the text appears dimmed until a final recognition is made. When a final recognition is determined, the text will darken and the insertion point will advance.

Below the insertion point is a small box. Click this box to display a list of alternate words. SAPI 5 places the final result of its word search on the screen, but you can choose another word by selecting it from the alternate list. This new choice replaces the existing word.

Text-to-Speech

Text may be read back using the TTS voice. Select **Voice->Playback** or click **Play** to hear the text.

If no text is selected, Dictation Pad will begin reading from the insertion point to the end of the document. If you select specific text, only the highlighted portion will be read. In either case, the word currently being read will be highlighted. The words will continue to be highlighted until the selection or portion is read. Any text selected prior to being read will remain selected afterward. If the insertion point is at the end of the document, the entire document will be read.

To stop or interrupt playback, click **Play** again from either the toolbar, menu, or use control-p. You may also click anywhere in the edit window or press Esc to stop playback.

You may change voices characteristics from the TTS tab of Speech properties. For example, you may change voices or change the speaking rate of the voice. The newly selected voice will automatically be previewed so you can confirm the choice. You can change the speaking rate with the Speed slider bar.

Command and Control

You may use speech to control program flow rather than using it as dictation. In this way, the menus, menu items, and the cursor may be controlled by speech and are collectively referred to as command and control. This is fundamentally different from dictation both programmatically and functionally. One difference is that the word selection is severely limited. Words are restricted to essentially coincide with the menu items, buttons, or serve as logical cursor commands. Other words will not be recognized.

You can switch to command mode in one of three ways. The first is through the **Voice->Listen for Commands** menu item. The second is to click **Command** on the toolbar. The third way to switch is during dictation by saying "command." Regardless, **Command** on the toolbar will automatically depress as visual confirmation of the current mode.

Once in command mode, you can speak the commands. If it is recognized successfully, the action takes place. A menu command will drop down the appropriate menu. The action for a menu item will be directly applied. A cursor command will move the cursor. If the command was not understood, no action will

take place. A command may not be recognized for several reasons. The word itself may not be on the command list, or the word may not have been spoken clearly, or background noise may have obscured it. Similarly, the menu item may not be applicable to the situation. If the reason is unclear, you should repeat the command clearly and perhaps more slowly. There might also be a slight lag time in response depending on the computer system's capability.



Simple Dictation

Introduction

Simple Dictation is a rudimentary application showcasing speech recognition dictation processes.

Using Simple Dictation

As the name implies it is a simple application that only displays the text of speech recognition. To use Simple Dictation, begin speaking into the microphone after the application launches. The results of the speech recognition display in the text box. There are no other controls for Simple Dictation. Dictation cannot be turned off through the application and different grammars may not be used. In addition, text normalization is not supported so that saying "five dollars" will produce the phrase "five dollars" rather than "\$5." Other substitutions are not supported either such as *new line* or *paragraph*.

Speak at a normal rate and volume for best results. Do not pause unnecessarily or excessively between words. Speech recognition yields the best results from natural speech patterns.

Options

Exit

Quits Simple Dictation. Simple Dictation may also be exited by clicking on the close button in the title bar.



TTSApp

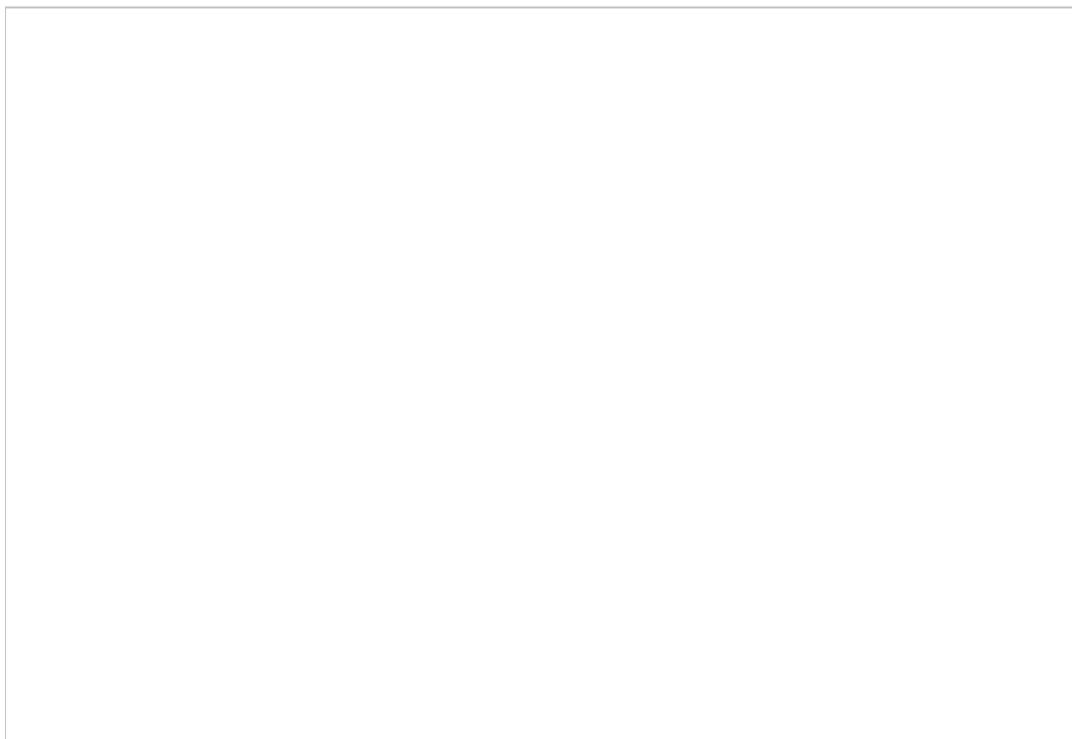
TTSApp is an example of a text-to-speech (TTS) enabled application. This sample application is intended to demonstrate many of the features for SAPI 5 in a single coherent application. It is not a full featured TTS-enabled application although the foundations of many of the options are present.

TTSApp allows you to hear the resulting audio output from the TTS process for text entered in the main window. Alternatively, you can open a file and TTSApp will speak the contents of that file.

Each word is highlighted in the text window to indicate the current TTS processing position. Features include:

SAPI5	The main display window of the TTSApp sample application.
TTSApp	
Text window	TTSApp speaks the text contained in this window using TTS.
Speak	Initiates the TTS process.
Voices	Selects the voice for the audio output.
Rate	Selects the rate of speech.
Volume	Selects the volume level of the audio output stream.
Open File	Enables TTSApp to open and speak the contents of a stored text file.
Pause	Pauses the TTSApp text phrase speaking process.
Resume	Resumes the TTSApp text phrase speaking process.
Stop	Stops the TTSApp text phrase speaking process.
About	Displays the About TTSApp information dialog box.
Format	Selects the audio format.

Skip	Specifies the number of sentences to skip in the phrase speaking process.
Speak wav	Speaks the contents of a stored wav file.
Reset	Resets TTSApp to its original configuration setting.
Save to wav	Saves the contents of the TTSApp audio output stream to a wav file.
Show all events	Displays all TTSApp SAPI events.
Process XML	Specifies that the TTS voice will speak the XML tags and their contents in the TTS process.
Mouth Position	Displays mouth shapes for phrase elements as they are spoken.



SAPI5

TTSApp main window.

Use the main TTSApp window to select the configuration

settings that affect the TTS process. The elements of TTSApp are listed above. Click the text in the left column for additional information.

Text window

The text content of this window is spoken by TTSApp. All text entered in this window is processed and spoken by TTSApp voice.

By default, the text content of this window is, "*Enter the text you wish spoken here.*"

Speak

Click **Speak** to initiate the text-to-speech process.

Voices

Select a voice using the drop-down list. TTSApp uses the selected voice when speaking a wav file or the contents of the text window.

Rate

Move the slide control to the right to increase the speech rate, and to the left to decrease the speech rate. The Rate level determines the number of text units spoken per minute.



Volume

Move the slide control to the right to increase the volume level, and to the left to decrease the volume level.



Open File

Click **Open File** to access the Windows **Open** dialog box. Select the file, and then click **Open**.



Pause

Click **Pause** to interrupt the TTS process.



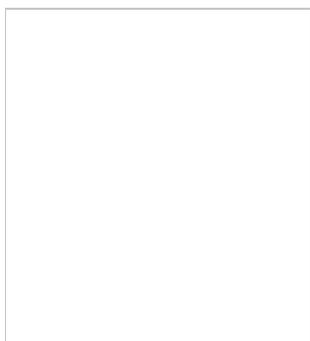
Resume

Click **Resume** to continue the TTS process.



Stop

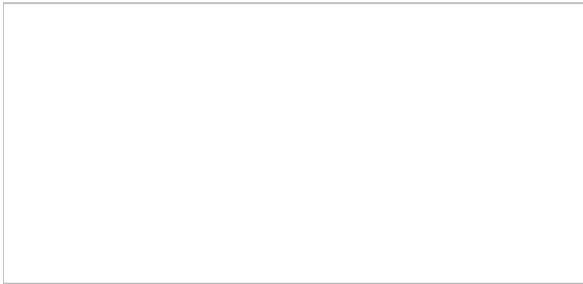
Click **Stop** to stop the TTS process.



About

The **About** window displays information related to TTSApp.

Click **OK** to close the **About** window.



Format

Use the drop-down list in **Format** to select one of the following format rates.

Selectable format rates				
8kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
11kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
12kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
16kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
22kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
24kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
32kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
44kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
48kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo



Skip

Use the spin box to select the number of skipped sentences.
Skip functions only while text is being spoken.



Speak wav

Speak wav enables TTSApp to speak the contents of a wav file. Click **Speak wav** to access the Windows **Open** dialog box. Select a wav file from the dialog box, and then click

Open.

Reset

Click **Reset** to reset TTSApp to its original configuration state.

Save to wav

Click **Save to wav** to save the TTSApp audio output stream to a wav file.

Show all events

Select **Show all events** to display SAPI related events in the event display window as the input text is processed by TTSApp.

Process XML

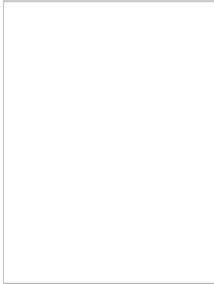
Select **Process XML** to include the XML tags and their contents in the audio output stream from TTSApp. When this option is selected, the application will parse and interpret the XML tags literally.

For example, if the Process XML option is selected, the application could be paused for the specified number of milliseconds in the SILENCE tag.

Process XML	XML tag	Result
<input type="checkbox"/>	<SILENCE MSEC = "3000"/>	The application would speak 3000 milliseconds of silence.
<input checked="" type="checkbox"/> Process XML	<SILENCE MSEC =	The application will speak

```
"3000"/>
```

the phrase, "less than
silence msec equals quote
three thousand quote slash
greater than."



Mouth Position

The mouth position displays the various mouth shapes and positions as TTSApp processes the input text stream.



Talkback

Introduction

TalkBack demonstrates speech recognition (SR) and text-to-speech (TTS) capabilities. You can speak any word or phrase and TalkBack will attempt to recognize it, display it on the screen and even play back the actual spoken phrase, allowing you to confirm the transcription.

Its unsophisticated interface belies the richness of the underlying tool suite. Talkback does everything that a major SR application needs to do and does it in only a few lines. It initializes the SR and TTS engines, accepts input from the microphone, and provides audio playback through speakers. It also recognizes words, displays them on the screen, and speaks them back with the computer's TTS voice.

You are encouraged to look at the code. To keep it simple, a DOS interface is used. Therefore, it needs very little code for maintaining the user interface. The few messages that exist are posted to the screen using the `printf()` command. The Coffee Tutorial samples provide a more detailed explanation for writing speech-enabled applications.

Using TalkBack

TalkBack is a console application but may be treated as a Windows one. Double-clicking its application icon opens it and it will be run in a new console window. You may also open it from within a console window by typing "Talkback" along with any command line parameters. The application will then run in the current console window.

TalkBack accepts up to two optional parameters:

```
Talkback -noTTS -noReplay
```

-noTTS disables the TTS voice from speaking the recognized result.

-noReplay disables the playback of your actual spoken word. By default, neither function is disabled.

Once started, you will be prompted on the screen with:

```
I will repeat everything you say.  
Say "Stop" to exit.
```

You may talk into the microphone. Any word or words may be spoken, although it works best with only a few words at a time. This recognition may take a few moments depending the system's capabilities. The SR engine processes the phrase and displays it on the screen:

```
I heard:  though
```

If replay is not disabled, it plays back the recorded word by speaking "...when you said:" and the actual word. This may be repeated any number of times. As part of the demonstration, the SR engine will force a match for the word even if the match is an unlikely one. For example, a nonsense word will return a match. A more robust application would likely question or mark

the word to warn you that no reasonable match was found.

To exit the application, you may say, "stop." Upon a successful recognition, TalkBack will quit. You may also use the close box on the window or, if running from the command line, use **CTRL+C**.



Simple Telephony

Introduction

Simple Telephony is a speech recognition (SR) and text-to-speech (TTS) engine that uses telephony (TAPI) interfaces. It is intended for use with a phone system, allowing you to dial in and talk to Simple Telephony.

Using Simple Telephony

The functionality of Simple Telephony is very similar to the SDK sample, TalkBack. When prompted, you may speak a word, phrase, or a short sentence and Simple Telephony will attempt recognition. It then speaks the result using a synthesized voice. It will also play back your recorded voice so that you can confirm the accuracy of the recognition. Please see the TalkBack User's Guide for more information.

Setting up Simple Telephony

Simple Telephony has two requirements. First, it runs only on the Windows 2000 operating system. Second, you must have a modem, since it is a telephony application. There is no restriction on the type or speed of modem, although it must be properly connected and able to process data. Follow the instructions provided by the modem manufacturer for specific loading and installation details.

Using Simple Telephony

The application's interface is simple. The phone line or connection displays the status. Commonly it will display "Waiting for a call..." or "Answering..." or "Connected." Several error messages may also be displayed.

Simple Telephony will not automatically open if a phone call is made to it. It must be already running prior to answering calls. If Auto Answer is selected, Simple Telephony will pick up the call; otherwise, you must click the Answer button. Simple Telephony will greet you and ask you to speak something. After the recognition and subsequent playbacks, the call will automatically terminate. Only one recognition is allowed per session.

Options

Auto Answer

Directs Simple Telephony to automatically answer incoming calls. Select this to have calls picked up on the second ring. If Auto Answer is not selected, you must click **Answer** to answer all calls manually.

Answer

Answers incoming calls manually. If the Auto Answer feature is not selected, you must click **Answer** to answer all calls manually.

Exit

Quits Simple Telephony. Any current call is disconnected and the application exits cleanly.

Compiling Simple Telephony

The Platform SDK is required to compile Simple Telephony. See [Microsoft Platform SDK](#) section in this SDK for those requirements and the download location.



SDK Samples for Automation

The SDK provides examples of applications using automation. Each sample is marked according to the language in which it is written. All samples contain source code. You are free to model applications on these samples. Notes inside each sample provide information compilation environment setup and settings, and additional information needed to run the sample.

Since SDK samples are meant to demonstrate basic speech technologies, not all options or contingencies are addressed. The samples are not intended to be complete or robust applications. In many situations, the samples do not provide return codes or extensive error checking.

The following topics are available:

Visual Basic Samples

- [Speech List Box for Visual Basic](#)
- [Simple Dictation for Visual Basic](#)
- [Simple TTS for Visual Basic](#)
- [RecoVB for Visual Basic](#)
- [AudioApp for Visual Basic](#)
- [TTSApp for Visual Basic](#)
- [VB Outgoing Call](#)
- [VB TAPI With Internet](#)

JScript Sample

- [Simple TTS for JScript](#)

C-Sharp (C#) Samples

- [Speech List Box for C#](#)
- [SimpleTTS for C#](#)

Microsoft Speech SDK

Speech Automation 5.1



Speech List Box for Visual Basic

Introduction

Speech List Box for Visual Basic is an elementary application showcasing speech recognition (SR) and dynamic grammars.

In general, a grammar is the set of words that the engine can recognize. In the case of dictation, all words are in the grammar and no limitation is imposed on the user. In the case of command and control applications, the list of words is much more restricted. For example, a grammar containing commands to operate a menu system, might include less than a dozen words, each word corresponding to a menu or menu item. Often, this list of words is generated ahead of time and represents a static or fixed grammar. However, dynamic grammars allow users to add words while running the application. This permits users to customize the grammar according to their needs. The list box sample demonstrated making and maintaining a dynamic grammar.

Using Speech List Box for Visual Basic

Speech List Box for Visual Basic opens with no words in the dynamic grammar. You can add words or phrases by typing the text in the edit box and clicking Add. After adding at least one item, individual items may be highlighted by saying, "select" followed by the text of the item. For example, if one of the items a list of cities were Seattle, saying "Select Seattle" will choose the Seattle item.

Add

Use Add to move the text in the edit box to the display area. The text can be either a single word or a phrase.

Remove

Use Remove to remove the selected item in the display area from the dynamic grammar. Highlight the item to be removed.

Speech Enabled

Use Speech Enabled to enable the speech recognizer. By default it is on and the speech engine will attempt to recognize voice commands. Disabling this option prohibits recognition attempts. Regardless of the recognizer state, if no items have been added to the display area, no recognition will occur.

Exit

Click the close box in the title bar to exit Speech List Box for Visual Basic.

Compile

Speech List Box for Visual Basic is actually two projects running at the same time. To compile them, open ListboxSample.vbg

instead of a vbp file. One project is the application itself. However, the display area is actually a control component. By running ListboxSample.vbg, the control component is actually loaded and registered for the computer.

Speech List Box for Visual Basic is a standard Visual Basic application and does not require additional special support. However, Visual Basic 6 Professional, Enterprise or later must be used to compile this example. Visual Basic 6 Learning Edition does not supply all the needed ActiveX Controls for the sample.

To compile correctly, the Speech reference must be active; see [Creating a Speech-Enabled Visual Basic Project in Using the Visual Basic Code Examples](#) for details to speech enable Visual Basic applications. Additionally, the samples are installed as Locked files. To modify them, they must be unlocked. To unlock, right-click the file or files, select Properties, and clear the Read-Only check box.

Microsoft Speech SDK

Speech Automation 5.1



Simple Dictation for Visual Basic

Introduction

Simple Dictation is an elementary application showcasing speech recognition (SR). Specifically, it uses a dictation model, so you can speak anything and Simple Dictation attempts to recognize it. Using Simple Dictation, speak into a microphone and following successful speech recognition, the text appears on the screen.

Options

Simple Dictation has two controls: Start and Stop.

Start

Click Start to start dictation. Once the SR engine is on, any recognized speech is displayed on the screen.

Stop

Click Stop to end dictation. The SR engine will stop and no further speech recognition takes place.

Exit

Click the close box in the title bar to exit Simple Dictation.

Compile

Simple Dictation is a standard Visual Basic application and does not require special support. However, the Speech reference must be active; see [Creating a Speech-Enabled Visual Basic Project in Using the Visual Basic Code Examples](#) for details on how to speech enable Visual Basic applications. Additionally, the samples are installed as Locked files. To modify them, they must

be unlocked. To unlock, right-click the file or files, select Properties, and clear the Read-Only check box.

Programming Notes

If Simple Dictation is run from the Visual Basic development environment, the debugger's Immediate window displays information not available from the executable version. See the code for exact details.

Microsoft Speech SDK

Speech Automation 5.1



Simple TTS for Visual Basic

Introduction

Simple TTS is an elementary application showcasing text-to-speech (TTS). The application speaks the text in the text box using a default voice. Simple TTS can also save the speech to a wav file.

Options

Simple TTS has two options: Speak the text (Speak It) or save the speech to a file (Save to wav).

Speak It

Click Speak It to hear the text in the text box spoken. You can change the voice using Speech properties in Control Panel. You can change or edit the text allowing different words or phrases to be spoken, although the text reverts to a default phrase each time you open the application.

Save to wav

Select Save to wav to save the output to a wav file. This way, the text will not be spoken audibly. After selecting Speak It, a dialog box appears requesting a file name and location to save the file. Double-click the wav file to play it back. As a wav file, it may also be used for other speech purposes such as an input source for either off-line dictation or custom engine voices.

Exit

Click the close box in the title bar to exit Simple TTS.

Compile

Simple TTS is a standard Visual Basic application and does not require special support. However, the Speech reference must be active; see [Creating a Speech-Enabled Visual Basic Project in Using the Visual Basic Code Examples](#) for details to speech enable Visual Basic applications. Additionally, the samples are installed as Locked files. To modify them, they must be unlocked. To unlock, right-click the file or files, select Properties, and clear the Read-Only check box.

Microsoft Speech SDK

Speech Automation 5.1



RecoVB for Visual Basic

Introduction

RecoVB is an application demonstrating basic speech recognition (SR) techniques. It displays the following information associated with the SR process:

- Text of the recognition
- Associated phrase elements and information
- Events for the recognition recorded as they are initiated and processed
- Event interests for the SR attempt
- Grammars to be used
- Control of SR engine

You can use RecoVB to test SR processes and see the results of the attempts.

To use RecoVB, select the characteristics of the recognition context. This includes the recognition type (command and control or dictation), the engine type to create (shared or InProc). By default, this configuration is set to command and control (C and C) in a shared environment. The default grammar is sol.xml. Once these parameters are set, click Start Recognition and speak into the microphone. The text appears on the screen as the speech is processed. If you selected any events or stream information, these will appear in the events display at the bottom of the main window. Once a recognition occurs, the larger Recognition window displays the results. Each recognition will be a new line in a tree structure. The result will display as Recognition and the actual word or phrase recognition.

To see information associated with a specific recognition, open

the tree view for that item by clicking on the line of the recognition or the small box to the left of the recognition. Like other tree view displays, if the box has a small plus sign ("+") in it, there is additional information to display. The expanded display lists the recognition result of that recognition.

Options

There are numerous options for RecoVB.

Start Recognition

Starts the SR process. Use Activate Mic to turn on the microphone, then you may speak into it. After starting SR, the button label will change to Stop Recognition. While active, the Recognition Type and Engine Creation radio buttons will be inactive since they cannot change during an SR session.

Activate Mic

Controls the microphone status. If selected, the microphone is active and receives sound for processing. If not selected, no sound will be processed through the microphone. While an inactive microphone does not process sound for SR, this is not the preferred method to turn SR on or off. Activate mic will be active only during speech recognition. The microphone may be turned off for brief periods but the SAPI engine is still active and consumes computer resources. To turn SR off, click Stop Recognition.

Show Stream Info

Displays the stream information associated with the SR session in the events list window.

Recognition Type

Controls the type of the recognition grammar. Select one of the following two types.

C&C;

A command and control (C and C) grammar recognizes specific

words. By default, the sol.xml grammar is used as an example, although a different grammar may be selected using the Recognition menu->Load Grammar. A C and C grammar is intended to restrict the user to a set of words often associated with specific tasks such as selecting menu items or, in the case of the default grammar, playing a game of solitaire. The limited grammar results in a better quality of recognition for the words that the application needs to process. It also filters out unnecessary words.

Dictation

A dictation grammar imposes no restrictions on the words that may be recognized. Unlike a C and C grammar, you can say any word or phrase and the SR process will attempt to recognize it. This enables you to dictate a letter or memo, for instance.

Engine Creation

Controls how the engine is instantiated for the session. Select one of the following two types, shared or InProc.

Shared

A shared environment (also called context) allows the SR engine to be used by other applications concurrently. This is the more common of the two environments. See [ISpeechRecognizer](#) for additional details.

Inproc

An in-process or InProc environment restricts the SR engine to only one application. No other application may use that engine concurrently. See [ISpeechRecognizer](#) for additional details.

Engine

This drop-down box lists the engines available. Only one engine

may used at a time and all SR instances must be of the same engine type.

Emulate Recognition

This option allows typed text to be processed by the SR engine. There are instances when users may wish to see the results and events associated with an SR attempt and will need method to replicate the speech attempt each time. Enter the text in the edit box and click Emulate to start the process.

Emulate

Click Emulate to start the emulated speech process. See Emulate Recognition for more details.

Current C&C; Grammar

Displays the current C and C grammar. The grammar may be changed through the Recognition menu Load Grammar item.

Event Interests

Use Event Interests to set which event interests to display and process. An event interest is a flag allowing SAPI or the SR engine to return information back to the application. When an event occurs (such as a recognition or the start of a new stream, for example), the SR engine can send a message back to the application. For example, for an application to display the text of a successful recognition, the application must receive the Recognition event interest. The application uses that message as a key before extracting the contents of the recognition. However, not all events are useful to the application at any one time. It is possible to prevent the application from receiving these events interests by turning off Event Interests. Likewise, they may be reactivated at anytime.

To receive certain event interests, select the ones you want from the list box. By default, all are active except for Audio Level. To

suppress receiving an event interest, clear the check box. See [ISpeechRecoContextEvents](#) for details about shared or InProc event interests.

Clear Event List

Clears the event interest window.

Clear Tree View

Clears the recognition results window.

Play Audio

Plays back the audio portion of the last recognition. This audio is the actual audio spoken by the user and is the sound sent to the SR engine. This is helpful in attempting to understand the results of a particular recognition. You must select Retain Audio in order to use this option.

Retain Audio

Keeps, or retains, the audio from the recognition attempt. See Play Audio for complete details.

Exit

Exits RecoVB. The application may also be exited by clicking the close box in the title bar or by the File menu Exit item.

File Menu: Exit

Exits RecoVB. The application may also be exited by clicking the close box in the title bar or by the File menu Exit item.

Help Menu: About

Displays the About box for the RecoVB.

Compile

RecoVB is a standard Visual Basic application and does not require special support. However, the Speech reference must be active; see [Creating a Speech-Enabled Visual Basic Project in Using the Visual Basic Code Examples](#) for details to speech enable Visual Basic applications. Additionally, the samples are installed as Locked files. To modify them, they must be unlocked. To unlock, right-click the file or files, select Properties, and clear the Read-Only check box.

Microsoft Speech SDK

Speech Automation 5.1



AudioApp for Visual Basic

Introduction

AudioApp is a rudimentary application showcasing custom audio objects.

A custom audio object uses audio in a non-standard way, and is designed to handle specialized audio needs. Most applications or programmers, however, will be able to use the standard audio devices or the default devices connected to the computer.

AudioApp performs speech recognition using a text-to-speech (TTS) voice. However, conventional output devices (speakers) are not used. A custom audio device is used instead. The computer does not need to have speakers installed since the speaking functions are essentially emulated.

Using AudioApp

To run AudioApp, enter the text to be spoken in the first text box. Click the Reco From TTS button to initiate the recognition attempt. The status bar at the bottom of the dialog box displays the process that is currently performing. The results of a successful recognition display in the second text box.

The custom audio device does not play the spoken text audibly so there will be no voice to hear. The recognition attempt is based on the TTS voice speaking the text. The quality of the spoken text changes slightly each time the application is run. As a result, the recognized text may also be different each time. This is unlike *ISpeechRecognizer.EmulateRecognition* in which the recognitions will always be the same for all attempts. Final recognition is identical to the original text.

Options

AudioApp has one control: Reco From TTS.

Reco From TTS

Starts the recognition attempt. See [Using AudioApp](#) for complete details.

Exit

Click the close box in the title bar to exit AudioApp.

Compiling

AudioApp is a standard Visual Basic application and does not require special support. However, the Speech reference must be active; see [Creating a Speech-Enabled Visual Basic Project in Using the Visual Basic Code Examples](#) for details to speech enable Visual Basic applications. Additionally, the samples are installed as Locked files. To modify them, they must be unlocked. To unlock, right-click the file or files, select Properties, and clear the Read-Only check box.

Programming Notes

If AudioApp is run from the Visual Basic development environment, the debugger's Immediate window displays information not available from the executable version. See the code for exact details.

See [Using a Custom Audio Object](#) for details writing and implementing them. The custom audio object in this sample is of type SpAudioPlug, which is specific and unique to the application. It is a DLL (simpleaudio.dll) loaded during the SAPI install and the Reference object is named SimpleAudio 1.0 Type Lib and display in Visual Basic's Object Browser as SimpleAudioLib.

Microsoft Speech SDK

Speech Automation 5.1



TTSApp for Visual Basic

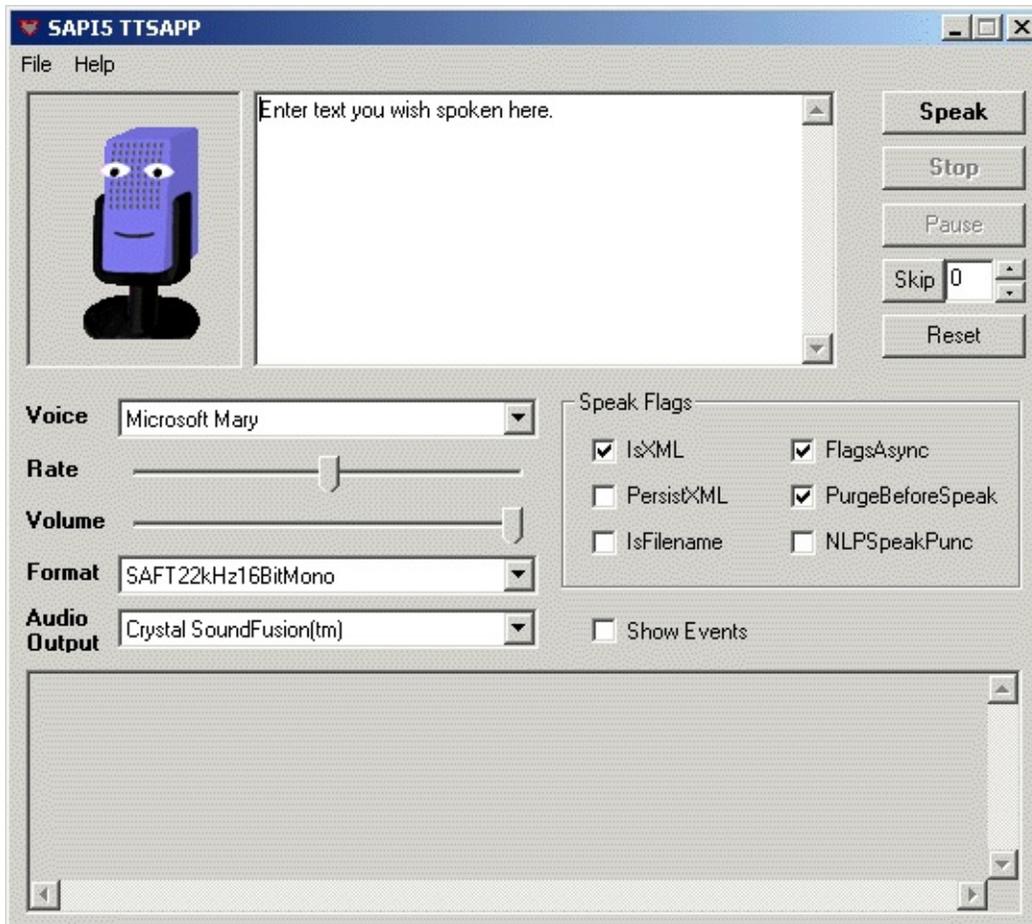
TTSApp is an example of a text-to-speech (TTS) enabled application. This sample application is intended to demonstrate many of the features for SAPI 5 in a single coherent application. It is not a full featured TTS-enabled application although the foundations of many of the options are present.

Using TTSApp you can hear the resulting audio output from the TTS process for text entered in the main window. Alternatively, you can open a file and TTSApp will speak the contents of that file.

Each word is highlighted in the text window to indicate the current TTS processing position. Features include:

SAPI5 TTSApp	The main display window of the TTSApp sample application.
Text window	TTSApp speaks the text contained in this window using TTS.
Speak	Initiates the TTS process.
Voices	Selects the voice for the audio output.
Rate	Selects the rate of speech.
Volume	Selects the volume level of the audio output stream.
Pause	Pauses the TTSApp text phrase speaking process.
Stop	Stops the TTSApp text phrase speaking process.
Format	Selects the audio format.
Audio Output	Selects the output device.
Skip	Specifies the number of sentences to skip in the phrase speaking process.

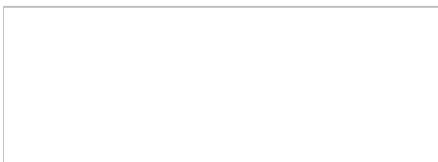
Reset	Resets TTSApp to its original configuration setting.
Show Events	Displays all TTSApp SAPI events.
IsXML	Specifies that the TTS voice will speak the XML tags and their contents in the TTS process.
PersistXML	Retains XML changes from one speaking attempt to another.
IsFileName	Interprets the text as a file name or file path rather than as text.
FlagsAsync	Speaks the text asynchronously. Asynchronous speaking allows SAPI to process other events at the same time of the speak.
PurgeBeforeSpeak	Deletes a voice before it is completed speaking. This allows a new voice to be created and used by the same object.
NLPSpeakPunc	Speak punctuation as text rather than as grammatical entities.
Mouth Position	Displays mouth shapes for phrase elements as they are spoken.
Open Text File	Opens a text file for display in the text box.
Speak Wave File	Opens a wav file to speak.
Save To Wave File	Saves the spoken content to a wave file.



SAPI5

TTSApp main window.

Use the main TTSApp window to select the configuration settings that affect the TTS process. The elements of TTSApp are listed above. Click the text in the left column for additional information.



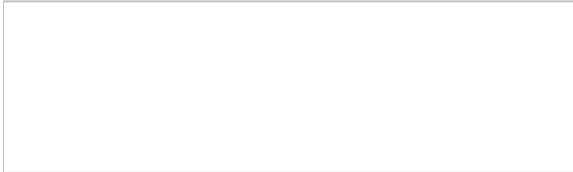
Text window

TTSApp speaks the text content of this window is spoken. All text entered in this window is processed and spoken by a TTSApp voice. By default, the text content of this window is, "Enter the text you wish spoken here."



Speak

Click **Speak** to initiate the text-to-speech process.



Voice

Select a voice using the drop-down list. TTSApp uses the selected voice when speaking a wav file or the contents of the text window.



Rate

Move the slide control to the right to increase the speech rate, and to the left to decrease the speech rate. The Rate level determines the number of text units spoken per minute.



Volume

Move the slide control to the right to increase the volume level, and to the left to decrease the volume level.



Pause

Click **Pause** to interrupt the TTS process.



Stop

Click **Stop** to stop the TTS process.

Format

Use the drop-down list to select one of the following format rates.

Selectable format rates				
8kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
11kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
12kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
16kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
22kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
24kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
32kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
44kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo
48kHz	8 Bit Mono	8 Bit Stereo	16 Bit Mono	16 Bit Stereo

Audio Output

Audio Output

Use the drop-down list to select the output device. In most cases, only one device will be available and represents the sound card for the computer.

Skip

Use the spin box to select the number of skipped sentences. **Skip** functions only while text is being spoken.

Reset

Click **Reset** to reset TTSApp to its original configuration state.

Show Events

Show Events

Select **Show Events** to display SAPI related events in the event display window as the input text is processed by TTSApp.

IsXML

IsXML

Select **IsXML** to include the XML tags and their contents in the audio output stream from TTSApp. When this option is selected, the application will parse and interpret the XML tags literally.

For example, if the IsXML option is selected, the application could be paused for the specified number of milliseconds in the SILENCE tag.

IsXML selected?	XML tag	Result
Yes	<SILENCE MSEC = "3000"/>	The application would speak 3000 milliseconds of silence.
No	<SILENCE MSEC = "3000"/>	The application will speak the phrase, "less than silence msec equals quote three thousand quote slash greater than."

PersistXML

PersistXML

Select **PersistXML** to retain XML changes from one speaking attempt to another. By default, this option is not selected.

This means that XML changes are not retained and each speaking attempt will begin using the default values for the engine. However, insert the following line in the text box: "Enter text <rate speed = "7"/> you wish spoken here" and select the PersistXML and IsXML boxes. The first speaking attempt will be as predicted in that the last part of the sentence will be read more quickly than the first part. The difference is that the second speaking attempt will begin at the same rate of the previous sentence ended. In addition, the sentence will get progressively faster each time the XML rate tag is encountered. Clearing the box after the second speaking attempt will not revert the rate back the default since the engine has already been changed for that session.

IsFileName

IsFileName

Select **IsFileName** to interpret the text as a file name or file path rather than as text. For example, in the case of a standard SAPI install, select IsFileName and paste the following line into the text box: C:\Program Files\Microsoft Speech SDK

5.1\Samples\CPP\Engines\TTS\MkVoice\enter.wav. Click Speak and the content of wav file is played. In this example, the wav file speaks "enter. If IsFileName is clear, the application will speak the contents of the edit box as "c colon backslash program files..."

FlagsAsync

FlagsAsync

Select **FlagsAsync** to speak the text asynchronously. Asynchronous speaking allows SAPI to process other events at the same time as speech. In contrast, synchronous speaking does not. For example, with FlagsAsync selected, the speaking attempt displays each word as it is being spoken. If the FlagsAsync is not selected, the text will still be spoken; however, the words will not highlight until the text

has been spoken. At that time, each word will highlight in turn. Highlighting may occur quickly due to the fact that events were being queued but not processed until the speech had finished.

PurgeBeforeSpeak

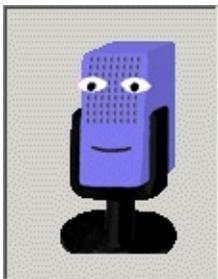
PurgeBeforeSpeak

Select **PurgeBeforeSpeak** to interrupt the speech attempt. With **FlagsAsync** selected, **PurgeBeforeSpeak** releases the current voice and speech, and allows a new voice to be queued. For instance, select both **PurgeBeforeSpeak** and **FlagsAsync** and speak the text. However before the sentence is complete, click **Speak** again. The voice stops and the sentence is restarted from the beginning. The previous voice has been deleted and a new one created.

NLPSpeakPunc

NLPSpeakPunc

Select **NLPSpeakPunc** to speak punctuation as text rather than as grammatical entities. Paste the following sentence into the text box: I like coffee! With **NLPSpeakPunc** selected, speak the sentence. Rather than ending after the word *coffee*, the exclamation point is read as the phrase "exclamation point."



Mouth Position

The mouth position displays the various mouth shapes and positions as TTSApp processes the input text stream.

Open Text File Ctrl+O

Open Text File

From the File menu, select **Open Text File** to open a text file to display in the text box rather than typing or pasting the content in manually. XML files may also be opened and displayed. Other file types can be opened, but since the text box only supports plain text, the contents may not display or speak in a predictably.

Speak Wave File Ctrl+W

Speak Wave File

From the File menu, select **Speak Wave File** to speak the contents of a wav file. Use the standard file dialog box to select the wav file. Once chosen, the file speaks automatically.

Save To Wave File Ctrl+S

Save To Wave File

From the File menu, select **Save To Wave File** to save the output of the spoken content to a wave file. Use the standard file dialog box to select the file. Once chosen, the contents of the text box is spoken automatically and the file is saved. The spoken portion is sent directly to the file and no audible speech will be heard. Of course, the file may be played back using Speak Wave File.

Microsoft Speech SDK

Speech Automation 5.1



VB Outgoing Call

Introduction

You can use SAPI one of two ways: on the desktop or with communication devices, such as a modem or the Internet. On the desktop, SAPI is restricted to running on one computer and applications must use that particular device. Using SAPI on the Internet, speech or voice information may be transmitted over the lines of communication.

Two sample applications are provided to demonstrate these different means of communication: VB TAPI with Internet and VB Outgoing Call. VB TAPI with Internet allows a computer to call another computer using a simple answering machine format. You can leave or retrieve messages using voice navigation. VB Outgoing Call also calls another computer but is restricted to sending a text-to-speech (TTS) voice message.

Using VB Outgoing Call

VB Outgoing Call is an example of a one-way telephony message. It is intended to represent an automatic reminder or notification system.

After a connection has been successfully made from VB Outgoing Call, your computer should notify you of an in-coming call. This notification is usually a series of beeps. Click Take Call to accept the call. You will hear the text-to-speech voice of the intended message.

Setting Up VB Outgoing Call

Two computers are required to use VB TAPI with Internet. The first, or source computer, will be the one you talk from. The second, or target computer, acts in the role of a server and operates VB TAPI with Internet. To use this application you need the following operating systems and programs:

- Both computers must have Windows 2000 or a later version, although the operating systems do not need to be the same.
- Both computers must be connected to each other using the Internet. Part of this connection can use a conventional telephone line as long as there is a valid Internet connection.
- The source computer (the one initiating the call) must have SAPI 5.1 loaded. The target computer does not require SAPI.

Now that both computers are set up with the necessary programs and connections, you can begin sending and receiving messages.

1. On the target computer, run Windows Phone Dialer. Click Start->Programs->Accessories->Communications->Phone Dialer.
2. To find the computer name, right-click the My Computer icon on the target computer's desktop and select the Network Identification. The name appears in the Full computer name line, although often only the first part of the name is needed.
3. On the source computer, start VB Outgoing Call and enter the name of the computer in the Internet Call combo box.
4. In the text box, enter the text that you wish spoken to the target computer, and then click Dial.
5. On the target computer, you will receive notification of

an incoming call. Click Take Call to accept the call.

Microsoft Speech SDK

Speech Automation 5.1



VB TAPI with Internet

Introduction

You can use SAPI one of two ways: on the desktop or with communication devices, such as a modem or the Internet. On the desktop, SAPI is restricted to running on one computer and applications must use that particular device. Using SAPI on the Internet, speech or voice information may be transmitted over the lines of communication.

Two sample applications are provided to demonstrate these different means of communication: VB TAPI with Internet and VB Outgoing Call. VB TAPI with Internet allows a computer to call another computer using a simple answering machine format. You can leave or retrieve messages using voice navigation. VB Outgoing Call also calls another computer but is restricted to sending a text-to-speech (TTS) voice message.

Using VB TAPI with Internet

Upon connecting to the Internet, VB TAPI with Internet will speak an introduction and present two options: Leave Message or Check Message. For example, to leave a message, simply say, "leave message" when prompted.

Leave message

You may leave a brief message. After speaking the message, the application will wait a few seconds and then play back the message. When the message has finished playing, the call will be disconnected. Although VB TAPI with Internet may be used many times to leave a message, only the last message will be retained.

Check message

You may retrieve a message. The last message recorded will be played back. When the message has finished playing back, the call will be disconnected. Only the most recent message will be played back.

Setting up VB TAPI with Internet

Two computers are required to use VB TAPI with Internet. The first, or source computer, will be the one you talk from. The second, or target computer, acts in the role of a server and operates VB TAPI with Internet. To use this application you need the following operating systems and programs:

- Both computers must have Windows 2000 or a later version, although the operating systems do not need to be the same.
- Both computers must be connected to each other using the Internet. Part of this connection can use a conventional telephone line as long as there is a valid Internet connection.
- The target computer must have SAPI 5.1 loaded. The source computer (the one initiating the call) does not require SAPI.

Now that both computers are set up with the necessary programs and connections, you can begin sending and receiving messages.

1. On the target computer, start VB TAPI with Internet. The Answer button will be disabled since no incoming call can be answered.
2. On the source computer, run Windows Phone Dialer. Click Start->Programs->Accessories->Communications->Phone Dialer.
3. To find the computer name, right-click the My Computer icon on the target computer's desktop and select the Network Identification. The name appears in the Full computer name line, although often only the first part of the name is needed.
4. Click the Dial icon on tool bar and enter the name of the target computer in the Dial dialog box. Or, from the Phone menu, click Dial and enter the name of the

computer. Make sure the Internet Call option is selected.

5. Click Place Call to initiate the connection.
6. On the target computer, the Answer button becomes active when the connection is made. Click Answer to accept the call.

The source computer is ready to use.

Microsoft Speech SDK

Speech Automation 5.1



Simple TTS for JScript

Introduction

Simple TTS for JScript® is a fundamental text-to-speech application using dynamic HTML (DHTML). DHTML allows the use of SAPI automation directly from a web page. To use Simple TTS for JScript, click Speak Text to synthesize the text in the text edit box. You may change the text anytime that the application is not speaking.

Requirements

Microsoft Internet Explorer 5.0 or later is required. If needed, download the latest version of [Microsoft Internet Explorer](#).

An ActiveX® control is used for SAPI automation and some browsers may display a warning about possible interactions between the SAPI control and standard systems. For the sample to run properly this interaction is required. Click Yes to allow this interaction.

Options

There are several options for Simple TTS for JScript.

Text Edit Box

Enter the text to be spoken in this box. Click Speak Text to hear the text spoken. Although the text may be changed while the web page is open, Simple TTS for JScript defaults to the standard phrase upon initial opening.

Speak Text

Speaks the text in the text edit box.

Rate

Controls the rate of the voice. Click the plus sign ("+") to speed up the spoken rate, or click the subtraction sign ("-") to slow down the rate. The rate will be changed for the next speak attempt. The changes in rate apply only for the current session and are not saved as the current speaking rate for other applications using SAPI.

Volume

Controls the volume of the voice. Click the plus sign ("+") to increase the volume, or click the subtraction sign ("-") to reduce the volume. The volume will be changed for the next speak attempt. The changes in volume apply only for the current session and are not saved as the current speaking rate for other applications using SAPI.

Voice

Selects the voice to use. Select a voice from the drop-down list of available voices. The next speech attempt will use that voice.

Audio Output

Selects the audio output device to use. This device contains the hardware sound card that processes the output. Select a device from the drop-down list of available devices; in many cases, there will only be one device to select from. The next speech attempt will use that output device.

Compile

Simple TTS for JScript and HTML application requires no compiling to run it. Instead, the code may be modified by any text editor or application capable of editing HTML or plain text. The sample may be installed as Locked files. To modify them, they must be unlocked. To unlock, right-click the file or files, select Properties, and clear the Read-Only check box.

Microsoft Speech SDK

Speech Automation 5.1



Speech List Box for C#

Introduction

Speech List Box for C# is an elementary application showcasing speech recognition (SR) and dynamic grammars.

In general, a grammar is the set of words that the engine can recognize. In the case of dictation, all words are in the grammar and no limitation is imposed on the user. In the case of command and control applications, the list of words is much more restricted. For example, a grammar containing commands to operate a menu system, might include less than a dozen words, each word corresponding to a menu or menu item. Often, this list of words is generated ahead of time and represents a static or fixed grammar. However, dynamic grammars allow users to add words while running the application. This permits users to customize the grammar according to their needs. The list box sample demonstrated making and maintaining a dynamic grammar.

Compile

To compile Speech List Box for C#, Visual Studio.NET and specifically Visual Studio C#.NET is required. Speech List Box for C# is actually two projects compiled at the same time. The one file is the application itself and the other is a control component for the display area. By running ListboxSample.sln, the control component is actually loaded and registered in the computer as one step.

The easiest way to compile them is a batch build. Open ListboxSample.sln. For a standard SAPI SDK 5.1 installation this is C:\Program Files\Microsoft Speech SDK 5.1\Samples\CSharp\Listbox\ListBoxCSharp.sln. Once loaded, compile the the solution by selecting Build->Batch Build. In the subsequent dialog select the Build checkbox for Debug if you need the debugging symbols in the project or the Release checkbox if not. Then click build to compile. Both projects build together and then display the list box dialog.

The samples are installed as Locked files. To modify them, they must be unlocked. To unlock, right-click the file or files, select Properties, and clear the Read-Only check box.

Using Speech List Box for C#

Speech List Box for C# opens with no words in the dynamic grammar. You can add words or phrases by typing the text in the edit box and clicking Add. After adding at least one item, individual items may be highlighted by saying, "select" followed by the text of the item. For example, if one of the items a list of cities were "Seattle," saying "Select Seattle" will choose the Seattle item.

Add

Use Add to move the text in the edit box to the display area. The text can be either a single word or a phrase.

Remove

Use Remove to remove the selected item(s) in the display area from the dynamic grammar. Highlight the item to be removed.

Speech Enabled

Use Speech Enabled to enable the speech recognizer. By default it is on and the speech engine will attempt to recognize voice commands. Disabling this option prohibits recognition attempts. Regardless of the recognizer state, if no items have been added to the display area, no recognition will occur.

Exit

Click the close box in the title bar to exit Speech List Box for C#.

Microsoft Speech SDK

Speech Automation 5.1



SimpleTTS for C#

Introduction

SimpleTTS for C# is an elementary application showcasing text-to-speech (TTS). The application speaks the text in the text box using a default voice. SimpleTTS for C# can also save the speech to a wav file.

Compile

To compile SimpleTTS for C#, Visual Studio.NET and specifically Visual Studio C#.NET is required.

To run the application open ListboxSample.sln. For a standard SAPI SDK 5.1 installation this C:\Program Files\Microsoft Speech SDK 5.1\Samples\CSharp\SimpleTTS\SimpleTTS.sln. Once loaded, compile and run SimpleTTS by selection Debug>Run. The application displays the main dialog.

The samples are installed as Locked files. To modify them, they must be unlocked. To unlock, right-click the file or files, select Properties, and clear the Read-Only check box.

Using Speech List Box

Simple TTS has two options: Speak the text (Speak) or save the speech to a file (Save to .wav).

Speak

Click Speak to hear the text in the text box spoken. You can change or edit the text allowing different words or phrases to be spoken, although the text reverts to a default phrase each time you open the application. You can change the voice using Speech properties in Control Panel.

Save to .wav

Select Save to .wav to save the output to a wav file. This way, the text will not be spoken audibly. After selecting Speak, a dialog box appears requesting a file name and location to save the file. Double-click the wav file to play it back. As a wav file, it may also be used for other speech purposes such as an input source for either off-line dictation or custom engine voices.

Exit

Click the close box in the title bar to exit Simple TTS.



SDK Utilities

The SDK provides utilities. These tools assist with the testing or development of speech-enabled applications.

The following topics are available:

Tools

- [MKVoice](#)
- [GC \(gc.exe\) grammar compiler for SAPI 5 XML conversion](#)

Engine Samples with source code

- [Speech Recognition Engine](#)
- [Text-to-Speech Engine](#)
- [Compliance Testing Tool](#)



MkVoice

Introduction

MkVoice creates text-to-speech (TTS) voice fonts for sample use. It combines a series of individually spoken words into a single file. The resulting file is automatically loaded and you may use it in any TTS application recognizing SAPI 5 voices.

MkVoice is intended to demonstrate making sample voice fonts. The limited scope is not ideal to create larger, more robust voice samples. Additionally, extensive error checking or error prevention routines are not included so that the resulting file may contain conditions not optimal to superior performance.

Running MkVoice

MkVoice is command line application that accepts three parameters:

```
mkvoice WordListFile VoiceFile VoiceName
```

WordListFile

This is a document list of words to concatenate and form the output file. The file needs to be saved as text only or created using a simple editor such as NotePad. No character formatting is allowed. The list requires only one word per line and that the line terminate with a carriage return.

The list can be of any size but each word must have a corresponding wav file with the same name. That is, if you use the word "enter," you need a file named enter.wav. The first entry in the list is used as the default word. If a word is encountered that is not otherwise in the list, use this default word instead.

For example, if using the SDK example TTSApp with the "Sample TTS Voice," the text initially displayed will be spoken: enter text to be spoken here. If you change the text by adding a word not in the file list such as "enter text now," it will be spoken as "enter text blah." "Now" is not a part of word list.txt. However, since "blah" is the first entry, it will be used for all unknown words.

VoiceFile

This is the resulting output file. By SAPI 5 convention, the vce name is the recommended suffix, although it is not required. If successfully generated, MkVoice automatically loads the sample. The new voice will be displayed with the name "Sample TTS Voice." This replaces any previous voice fonts. The voice will be defined as an English-speaking male.

Finally, to run MkVoice, the application must be in the same folder as the wav files and word list. If run successfully, the application creates an output file; otherwise, it will display appropriate error messages.

VoiceName

This is the name associated with the voice by the object token.
Creating voice fonts

Creating Voice Fonts

Voice fonts are a collection of words spoken by a person and assembled into a phonetic dictionary. When SAPI encounters a word, this database looks it up. A successful match plays a portion of the sound file of the word. By contrast, a synthesized voice uses mathematical algorithms to produce the word. The voice fonts produce what is often considered better and more natural prosody--the way the word sounds.

Voice fonts require two components. First, a word list must be generated. This is the same list as the first parameter described above. In the SDK, the MkVoice example uses wordlist.txt.

Second, individual wav files are needed for each word. The first part of the file name must correspond to an entry in the word list. The name suffix must be wav. Additionally, the wav file must contain the following characteristics:

Value	Description
Audio format	PCM
Sample rate	11.25 kHz
Audio sample size	16 bit
Channels	1 (mono)

Files should contain only one word each. Silence leading or trailing a word should be minimal to provide the best playback. You may not use punctuation marks. Replace any marks with underscores ("_") in both the file name and word list. For instance, MkVoice provides a file named computer_s.wav and this matches the corresponding entry in wordlist.txt. A simple way to generate the wav files is to use Sound Recorder provided by the Windows operating system. The file characteristics may have to be changed to the above requirements.



Grammar Compiler

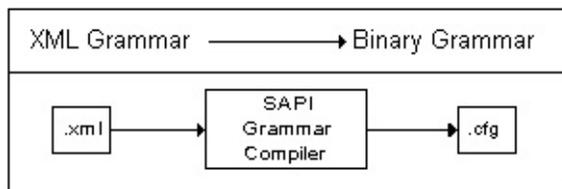
The SAPI grammar compiler (gc.exe) creates binary grammars from extensible markup language (XML) defined grammars.

The following topics are discussed in this section:

- [Introduction](#)
- [Using the Grammar Compiler](#)
- [Grammar compiler custom build settings](#)

Introduction

The SAPI grammar compiler is divided into two parts, the front-end section and the back-end section. The front end parses the grammars described in XML and optimizes the XML text formatted grammar if requested by the application. For example, the front end can remove the left recursion. The front end then calls the back-end compiler to convert the internal representation into the SAPI binary format.



[Back to top](#)

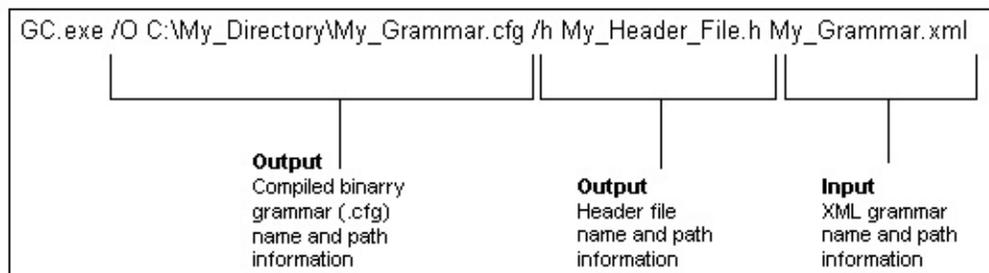
Using the grammar compiler

The SAPI 5 grammar compiler (gc.exe) can be used from the command line, or added to the Microsoft Visual C++ custom build environment.

Using GC from the command line

When compiling XML grammars from the command line, the following options are available:

Command line argument	Definition
/O	The file name and path information associated with the compiled grammar output file. (CFG) For example: My_CFG_Grammar.cfg
/H	The file name and path information associated with the header file that receives the # define information. For example: My_Header.h See specifying the grammar compiler /h option for more information.
file_name.xml	The file name and path information associated with the grammar file. (XML) For example: My_XML_Grammar.xml



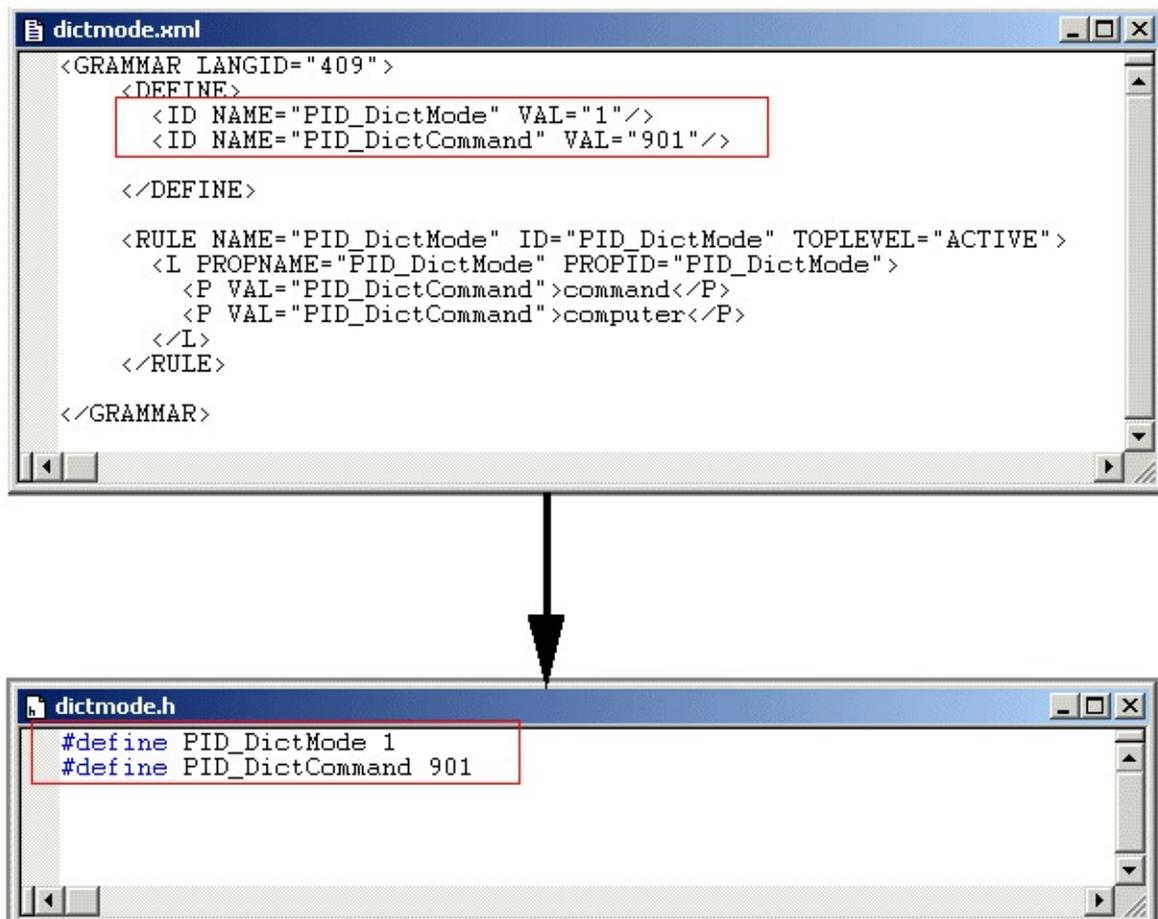
Grammar Compiler (GC.EXE) command line arguments

[Back to top](#)

Specifying the grammar compiler /h option

The SAPI grammar compiler can add the XML ID tag NAME and VAL contents to a specified header file. By choosing the **/h** compile option, the contents of the XML ID tags are added to the specified header file in the standard C-style **#define** format.

The following illustration shows the result of the **/h** command line argument while compiling the dictmode.xml file. Notice the contents of the XML ID tags NAME and VAL are added to the dictmode.h file as c-style **#define** statements.



[Back to top](#)

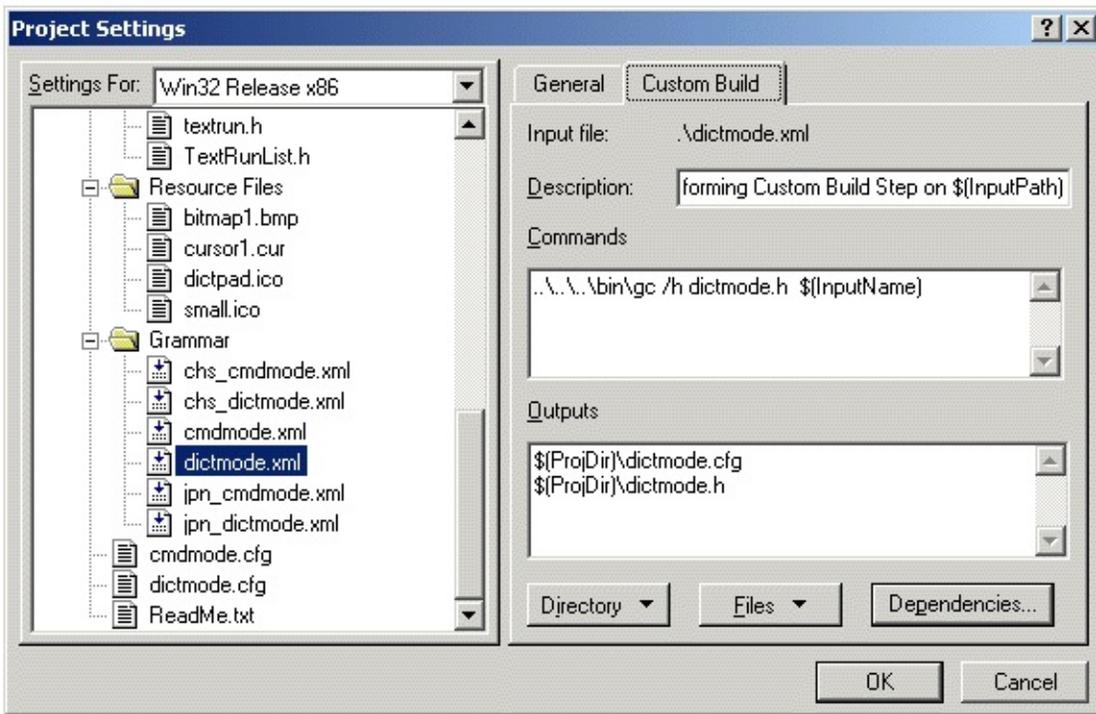
Grammar compiler custom build settings

The SAPI 5 grammar compiler (gc.exe) can be added to the Microsoft Visual C++ custom build environment. The XML grammar compile settings can be pre-configured in the project custom build settings. Each time the XML grammar is compiled, the grammar compiler uses the settings that are specified in the custom build settings.

The Speech SDK 5.1 sample application Dictation Pad illustrates the use of the grammar compiler in the project custom build settings. Compile the dictpad.dsp project and review the Dictation Pad sample application project settings to learn more about how the grammar compiler can be added to the Microsoft Visual C++ custom build environment.

The Dictation Pad SDK sample application can be found in the following location: C:\Program Files\Microsoft Speech SDK5.0\Samples\CPP\DictPad\dictpad.dsp.

Custom build settings



[Back to top](#)



Compliance Testing Tool

Introduction

The Compliance Testing Tool allows engine vendors to test their speech engines for SAPI compliance and port these speech engines to SAPI 5. The tests also help vendors to support various SAPI features that are not required for compliance. These tests do not evaluate the speech or performance quality of the engines. All compliance tests assume that SAPI will validate parameters; hence, vendors do not check the engine's ability to handle invalid parameters such as null, bad pointers, or values out of range.

Compliance Testing Tool

For a complete discussion of the Compliance Testing Tool see [Using the compliance testing tool](#).

For a complete discussion of Compliance testing in general see the [Compliance Tests](#) white paper.



Sample Speech Recognition Engine

Introduction

The sample speech recognition (SR) engine demonstrates the design, compilation, installation, and testing for engines. The following is basic information you should know about the SR sample engine:

1. It is not necessary to compile the sample SR engine project (sreng.dsp) to install the sample SR engine (sreng.dll). It is installed with the Speech SDK 5.1.
2. The path name is: C:\Program Files\Microsoft Speech SDK 5.1\bin\sreng.dll (if you install the SDK under C:\Program Files).
3. The C++ project file C:\Program Files\Microsoft Speech SDK 5.1\Samples\CPP\Engines\SR\sreng.dsp. To compile this project, the Platform SDK needs to be also installed. See [Microsoft Platform SDK](#) for additional information.

How to Register the Sample Engine dll from the Command Line:

Although applicable to different engines, this demonstration uses the sample sreng.dll. You will need to register the engine with the operating system. At the command line type:

```
regsvr32 C:\Program Files\Microsoft Speech SDK5.0\bin\sreng.dll
```

After the .dll registration process, the sample SR engine is available on the current computer.

How to Set the Sample Engine to the Default Engine

1. In Control Panel, double-click the Speech icon.
2. On the Speech Recognition tab, select the SAPI Developer Sample Engine from the list of available engines.
3. Click OK.

Sample Engine Expected Results

The sample engine randomly generates context-free grammar (CFG) recognition results based on the CFG grammar you select. The sample engine will also generate other events, such as interference and requestui etc.

Sample SR engine notes

1. The engine does not perform the recognition based on an acoustic or language model. Instead, it retrieves the CFG information from SAPI and constructs random results.
2. The sample engine provides a basic idea about how to develop an SR engine to interact with SAPI.
3. The sample engine does not pass the compliance tests.
4. If you experience unexpected results for the real SR activity, make sure that the sample engine is not in use and that the sample engine has not been set as the default engine.



Sample Engines

Introduction

Sample text-to-speech (TTS) engines are provided to demonstrate the design, compilation, installation, and testing of engines.

Installing TTS Engines

There are two ways to install the engines: **command line** and **compiler**.

Command Line Installation

Although applicable to different engines, this demonstration uses the sample ttseng.dll. You will need to register the engine with the operating system. At the command line type:

```
regsvr32 ttseng.dll  
mkvoice wordlist.txt voiceFile voicename
```

The second command registers the accompanying voice file **samplevoice**. The voice must be compiled in the same directory as the sample engine. From the SAPI5SDK directory:

```
Copy bin\mkvoice.exe samples\CPP\Engines\TTS\MkVoice  
cd samples\cpp\engines\tts\mkvoice  
mkvoice wordlist.txt samplevoice.vce samplevoice
```

The SDK tool [MkVoice](#) creates this voice file (these are also called voice fonts) and it is documented separately. Speech SDK 5.0 provides the sample file **samplevoice** although you may create your own by following the MkVoice directions.

Compiler Installation

Engines will automatically be registered after compiling if you use the SDK's project workspace. Installation occurs as part of the post-build commands. Likewise, the associated voice fonts will also be registered at the same time. To verify installation, you may examine the registry at:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Voices\T
```

If successfully loaded, you will see an entry with the file name such as *SampleVoice*. Please note it is important that you do not manually change or edit any of these registry entries. Only SAPI may modify speech registries and improperly changed ones could irrevocably damage the system. This is provided only for verification reasons.

Setting a Default Engine

Once installed, you can set the TTS engine as the default engine.

1. In Control Panel, double-click the Speech icon.
2. On the Text-to-Speech tab, from the list box, select the engine you want. In this case, select the Sample TTS Voice.
3. A green check mark will appear in the check box, indicating that the Sample TTS Voice has been selected as the voice for the default engine.

All speech applications will now use this voice. As a confirmation of the selection, you will hear the voice speak, "You have selected the Sample TTS voice as the computer's default voice."

Confirming the Default Engine

To test the TTS, select any application that uses a voice. SDK provides a convenient application--TTSApp.exe.



SDK Tutorials

This section provides an SDK Tutorial overview. Two topics are available. The Coffee series a complete example with documentation. Code projects are available for each Coffee topic and are installed in the SDK. The TTS section has a two brief tutorial overviews for using TTS. While code examples are listed directly in the documentation, no separate project is available.

The following topics are available:

Coffee Examples

- [Setting Up SAPI 5](#)
- [CoffeeS0 Tutorial](#)
- [CoffeeS1 Tutorial](#)
- [CoffeeS2 Tutorial](#)
- [CoffeeS3 Tutorial](#)
- [CoffeeS4 Tutorial](#)
- [CoffeeS5 Tutorial](#)
- [CoffeeS6 Tutorial](#)

TTS Examples

- [TTS Tutorial Example](#)
- [Using Events with TTS](#)
- [TTS Events Explanation](#)



Setting Up SAPI 5

Introduction

The SAPI 5 Reference API is an excellent guide for programming speech applications. With the large number of methods, interfaces, structures, and enumerations SAPI 5 offers, the reference API is a required document. However, those who are new to speech applications may be a little lost at first, as the reference API makes little attempt to weave all the parts together.

The goal of this book is to help you write properly structured SAPI 5 applications using a series of examples called Coffee. The application uses a coffee shop motif designed so that you may place orders, talk to management, or buy items at the store. Since each application builds upon the previous one, you need to understand each step before moving on to the next one.

What you need to know

There are two prerequisites for using these examples. First, that you generally understand graphical interface programming and specifically understand native Windows programming. Although you will concentrate on SAPI 5 topics, the Coffee examples are Windows applications. The majority of the code in the samples is a framework and is used only for running the application. It handles keyboard, mouse, screen updates, and other processing messages. Unless there is a relevance to SAPI 5, much of the code is will not be discussed.

Second, you should have some experience with C/C++ programming. The intent is to keep programming simple and consistent. There are other models and languages for programming Windows including Visual Basic, JAVA and Microsoft Foundation Class (MFC). MFC adds a layer of complexity that you do not need at the moment. After you gain proficiency with SAPI 5 code, theory, and implementation, feel free to change approaches.

Additionally, SAPI 5 is component object model (COM) based. Although COM proficiency is not required, some understanding of it is. To further simplify COM, SAPI 5 uses active template library (ATL) functions to complement COM. You need a basic understanding of COM smart pointer CComPtr. Additional information about these topics may be found in MSDN or in the myriad of books available through popular bookstores.

Using this book

Each chapter describes the important concepts introduced. The discussion follows the code and provides examples from the Coffee code itself. The first Coffee example is the foundation of the entire process and is slightly longer than other chapters. The narrative is a brief overview and presents enough material to complete the example, though not enough to exhaust the subject.

Each Coffee chapter builds on the previous examples. Since no code is ever removed, changes from chapter to chapter may be found by comparing files or entire folders using a difference engine. However, in general, any application may be used.

What you need to get started

You will need a copy of Visual C 6.0 with Service Pack 3 or later version. In general, any 32-bit C compiler will work. However, the samples assume Microsoft Visual Studio 6.0 SP3 or later.

SAPI 5 Installation source. This may be a SAPI developer's CD or the installer through another source (if available) such as the Microsoft Web site. Regardless, you are encouraged to install SAPI 5 only through an installation package. The SAPI CD installs all the required components including the required dynamic link libraries (DLL), headers, registry entries, and other resources.

SAPI 5.1 SDK

In addition to having up-to-date source files and examples, you need the reference API to look up interfaces and methods. Even though the examples are well commented and the tutorials contain narratives about the calls and approaches, the reference API explains each call in detail and to a greater extent than is possible in other sources.

Setting up

To configure your system for speech recognition, go to Speech properties in Control Panel and click the Speech Recognition tab. Speak into your microphone and observe the volume meter in the microphone window; if the meter registers the volume level the microphone works. Then click the Text-to-Speech tab. To test the audio output, click Preview Voice. The text in this section will be spoken, highlighting the words as they are spoken. If this is the case, then audio output also works. If neither works, see the Troubleshooting section.

It is also recommended that you use the microphone training wizard. From the Speech Recognition tab, click Train Profile. The training wizard instructs you in microphone placement and input level adjustment so that SAPI is able to recognize your commands. For the Coffee examples, the list of commands is quite limited and this training is not required. Speak clearly and deliberately into the microphone and Coffee should be able to recognize your words.



CoffeeS0

Introduction

This first example represents a basic speech recognition (SR) application. The foundation is presented and over the next series of examples, progressively complex features will be added.

The example will focus on setting the foundation of speech-enabled applications in general. The following topics will be discussed:

Initialization: Setting up engines and grammars

Events: Definition of events, notifications, interests, expanding events

Phrases: Definition of phrases, grammar rules, accepting phrases

Running the Example

Run the application by executing CoffeeS0.exe. A window appears and along the top you are greeted: Welcome to the SAPI coffee shop. Speak for service! Since this is a simple example, your only option is to say, "Please go to the counter." Variations are also accepted such as, "Please go to counter," or "enter counter." For this example, even "go to shop" and "enter the store" are recognized for reasons examined later. Speak clearly to activate the voice recognizer and the screen will

change to “Please order when ready!”

Perhaps limited in functionality (you cannot actually order coffee in this coffee shop) it does represent the fundamentals of speech recognition. That is, commands are spoken, recognized, and executed. Look at the Coffee code in Coffee.cpp. It looks very much like a simple Windows application, because, in fact, that is what it is. The application brings up one window, and draws two strings to it. To keep the code to a minimum, it does not support keyboard operations, a mouse, or a menu. To exit the application, click the Close button in the upper right of the screen.

The speech API commands are interspersed with the Windows API. Chances are, if you do not recognize a command as belonging to the Windows API, it does not. It belongs either to SAPI 5 or it is part of the structure needed to process speech. For instance, there is a new message, WM_RECOEVENT, which defines the application. It is not a part of SAPI but the application structure needs this message to process speech. In the same manner, several routines are also defined by the application (InitSAPI(), CleanupSAPI(), ProcessRecoEvent(), and ExecuteCommand()). Inside these Coffee-defined functions are the SAPI 5 method calls. There are also a few #defines at the top of the file for including SAPI 5-specific headers.

Header Files

Before starting out with any coding, the proper headers need to be introduced.

```
// Contains definitions of SAPI functions
#include <sp-helper.h>

// Contains common defines
#include "common.h"

// Forward declarations and constants
#include "coffee.h"
```

```
// This header is created by our grammar compiler and has the rule i
#include "cofgram.h"
```

Of the five headers, two are application-specific to the Coffee series. `Coffee.h` contains the function prototypes and global variables for the application and `common.h` lists `#defines` for the application's windows and other features. As the samples change and expand, these two files are updated. However, this has no effect on the speech aspects of the program. A third file, `stdafx.h`, is related to COM programming and it is maintained by the compiler. This file can also change and have no impact on speech-related issues.

The other two files are speech related. `Sphelper.h` is provided by SAPI 5 and is the header file for the helper functions. For the programmer's convenience, there is a list of functions consolidating a series of SAPI API methods into a single call. You are not required to use helper functions for SAPI programming, although doing so may simplify programming. The Coffee samples use helper functions whenever possible. Finally, the grammar compiler uses `cofgram.h`. A grammar is a list of words available to the application. The topics of grammars and the compiler in general will be discussed in later chapters.

For SAPI 5 to work, it needs to be initialized. This is done in four basic steps with each step depending on the success of the previous one. You should always check the return values just as in Windows programming.

Initialization

Step one: COM

First, initialize COM making sure it is present and active. Use the COM command `CoInitialize()` and later `CoUninitialize()`. To ensure COM is active throughout the application's session, these commands are usually nested around the main message loop.

```

// Only continue if COM is successfully initialized
if ( SUCCEEDED( CoInitialize( NULL ) ) )
{
    // Main message loop:
    while ( GetMessage(&msg, NULL, 0, 0) )
    {

        //Code here

    }

    CoUninitialize();
}

```

Step two: Recognizer Object

Second, create the recognizer object. This object provides access to the recognition engine.

```

//Global Definition

CComPtr<ISpRecognizer> g_cpEngine;

// create a recognition engine
hr = g_cpEngine.CoCreateInstance(CLSID_SpSharedRecognizer);
if ( FAILED( hr ) ) // Leave application

```

A single instance of a recognizer object is created. There are two options for setting up this object: shared and in-process (InProc). Use the following class identifiers (CLSIDs) to set up the instance:

CLSID_SpSharedRecoContext	Creates a shared resource instance
CLSID_SpInprocRecognizer	Creates an InProc or non-shared resource instance.

Shared Instance

Shared instances allow resources such as recognition engines, audio input (microphones), and output devices to be used by several applications at the same time. This is the preferred option for most desktop applications. It is common for a desktop

system to have only one microphone and by using shared instances, different applications such as a browser, word processor, and a game can use the microphone. Any application using a shared instance will start the SAPI server process. This is an executable program running in the background. It delivers events to the owning application.

InProc Instance

InProc instances, however, allow one and only one application to control the resources. This includes the microphone and speech recognition engine. Using an InProc procedure is very restrictive and you should use it only in special circumstances. For example, you would use InProc if you wanted the entire microphone input to be channeled through one application. Telephony applications are a good example of the need to restrict use to one microphone or audio input source.

Step three: Recognition Context

Third, create a recognition context for the engine.

```
//Global Definition
CComPtr<ISpRecoContext> g_cpRecoCtxt;

// create the command recognition context
hr = g_cpEngine->CreateRecoContext(&g_cpRecoCtxt );
if ( FAILED( hr ) ) //Leave application
```

A context is any single area of the application needing to process speech. A simple case (like CoffeeS0) assigns the entire application to only one recognition context. No matter where you are in the application, all speech events and messages are handled by the same procedure. Alternatively, each part of the application may have a different context. Individual windows, dialog boxes, menu bars, or even menu items (such as the Open or Print menu items) may have their own context. Events or messages generated from these areas are processed by their

own procedures. This is similar to the way individual windows process events and messages in standard Win32 applications. That is, each window is assigned a window procedure that handles all the events and messages. In the same way, each recognition context is assigned a procedure as well. This way, you have greater control over the program and the handling of speech events. Contexts are created dynamically the moment they are needed and destroyed afterward. Alternatively, you may create them once and retain them throughout the application's life. However, Coffee is a simple example and uses only one context.

IspRecoContext is an important interface and will be the primary means for recognition. From the interface, the application can load and unload grammars as well as get and respond to events.

Step four: Loading Grammars and Rules

The last major part of the startup sequence is loading the grammar. A grammar specifies what the speech recognizer will recognize.

```
// Load our grammar

// user defined
("SRGRAMMAR") resource type.

hr =
g_cpRecoCtxt->CreateGrammar(GRAMMARID1, &g_cpCmdGrammar);

if ( FAILED( hr ) ) //Leave application

hr = g_cpCmdGrammar->LoadCmdFromResource(
    NULL,
    MAKEINTRESOURCEW(IDR_CMD_CFG),
    L"SRGRAMMAR",
    MAKELANGID( LANG_NEUTRAL, SUBLANG_NEUTRAL), TRUE);

if ( FAILED( hr ) ) //Leave application
```

Essentially there are two types of grammars. One is for dictation

and the other is command and control. The dictation grammar is a more free-formed approach to speech. You are able to draw on a very large portion of the body of words for the language. Command and control is a much more limited list of words. For Coffee examples, you only need certain words and only then to move around the store and to order drinks. It makes no sense for Coffee to know about the word “opisthognathous” so why even attempt to find it? Besides, you are going to sip your coffee anyway.

The Coffee list is a pregenerated set of commands stored as a resource internal to the application. Coffee.xml saves this in a human-readable format. Extensible Markup Language (XML) is the markup language used to generate the grammar and the format that the file uses as defined by SAPI. You need to compile this file into a binary version so SAPI 5 can use it. You can do this ahead of time or on the fly. Here it has been precompiled it so that the grammar is delivered inside the application. The SAPI 5.1 SDK has a grammar compiler called GramComp, delivered in the tool suite. Grammars are covered in more detail in the CoffeeS1 example.

IspRecoContext, as mentioned in Step 3, creates the grammar. Once made, you populate the grammar with words from command list. Use `::LoadCmdFromResource`, since it is stored as an application resource. Another way you could load it is from other sources, including an external file, memory, or an existing object. After you have retrieved the grammar, you need to set the rules. As a convenience, the XML itself activates the initial set of rules. Specifically the `TOPLEVEL="ACTIVE"` tag in coffee.xml does this. The following is an example of an explicit application setting:

```
// Set rules to
active, we are now listening for commands
hr = g_cpCmdGrammar->SetRuleState( NULL, NULL, SPRS_ACTIVE );
```

The method explicitly sets any rules it encounters to become

active. Because the two NULL parameters values did not exclude any rules, all of them were activated. You can also deactivate rules using this method. If the call fails, the application posts a message giving the most likely reason.

Events

CoffeeS0 is now an application that can take speech input. It is processing speech in the background. When SAPI has information, it returns it back to the application. SAPI will notify you when an event happens. In short, an event is a condition of special interest to SAPI. Examples of events include, when a sound is first detected on the microphone (SPEI_SOUND_START), when it ends (SPEI_SOUND_END), or when it successfully completes a word recognition (SPEI_RECOGNITION). SAPI maintains several types of events, of which the enumerated type SPEVENTENUM, maintains a complete list. Two important concepts tie events into the application: Notifications and Interests.

Notifications

A notification indicates that a SAPI event has occurred and the application might want to react. It does not relay exactly what happened – for that you will have to dig a little deeper.

To react to notifications, the application has to associate them with specific procedures. There are several ways to do this. The ISpNotifySource interface has four methods: SetNotifyCallbackFunction, SetNotifyCallbackInterface, SetNotifyWin32Event, and SetNotifyWindowMessage. An additional one, ISpNotifySink::Notify, provides a generic method allowing for special or unusual conditions. You can use any or all of these methods depending on your needs. For instance, it might be easier for an application to handle a notification by directly calling a function (::SetNotifyCallbackFunction) such as bringing up a new dialog box or automatically logging the

activity in a file. Three of the methods, `::SetNotifyCallbackFunction`, `::SetNotifyCallbackInterface`, `::SetNotifyWindowMessage`, require a message loop and therefore may only be used by Windows applications. You can use the other two, `::SetNotifyWin32Event` and `::ISpNotifySink::Notify`, without a message loop to provide additional flexibility.

CoffeeS0 sends the notification through a Window procedure. Since SAPI messages are not system level messages, you have to tell the application explicitly about them.

```
hr = g_cpRecoCtxt->SetNotifyWindowMessage( hWnd, WM_RECOEVENT, 0, 0
if ( FAILED( hr ) ) //Leave application
```

This method associates a message to a specific window. Afterward, any events SAPI passes back will be received by the application in the singular form of a `WM_RECOEVENT` message, and then sent to the window pointed to by `hWnd`. You have the option of directing the `wParam` and `lParam` window parameters as well. Because CoffeeS0 is not concerned about them here, the application has set them to zero or NULL.

Interests

An interest is a flag allowing or restricting the kind of events SAPI passes back. By default, SAPI sends all events back to the application. So far that is more than 30 different kinds of events. You cannot be concerned about all of them. In reality, CoffeeS0 truly cares about one type: the successful word recognition or `SPEI_RECOGNITION` event. You can tell SAPI to pass back only this one event. To filter these events, use `::SetInterest`.

```
hr =
g_cpRecoCtxt->SetInterest( SPFEI(SPEI_RECOGNITION), SPFEI(SPEI_RECOG

if ( FAILED( hr ) ) //Leave application
```

This sets the interest to just one message, `SPEI_RECOGNITION`.

That is, only a successful recognition event generates a notification. SAPI will not notify the application on any other event.

You can define multiple interests using the exclusive OR operator. Two values are set. The first parameter lists the interests in general. That is, it defines all the events you are, or could be, concerned with for the time being. The second parameter lists the events to be queued so that the application can handle them in due time. In this tutorial, you are interested in every occurrence of SPEI_RECOGNITION, even if they come so quickly that the application cannot handle them at one time. Often, these two parameters will be identical but, obviously, they don't have to be. SPFEI() is a helper function used to reformat the enumerated events into a ULONGLONG number.

Is the application now fully functional? Again, almost. It is true, the application can initialize SAPI, accept speech from a microphone, attempt to recognize it, and send back an event to the application if a word is matched. You still need to put the message in the event loop in order for the application to handle the event. Because it is already defined and known by application, WM_RECOEVENT can be put in like any other message. The following code fragment is from main window procedure WndProc:

```
// This is our application defined window message to let us know tha
// speech recognition event has occurred.

case WM_RECOEVENT:
    ProcessRecoEvent( hwnd );
    break;
```

Each time SAPI is ready to return a word, it sends out an event. The application receives this message as WM_RECOEVENT. The main message loop picks it up and, in this case, sends it to the ProcessRecoEvent() for routine processing.

For speech recognition, this event is handled in a slightly

different manner than the way Windows approaches it. Normally, Windows sends the exact message based on the event. In this way you do not have to determine if it were a mouse or keyboard event; the message provides this information. SAPI, on the other hand, does not. You still need to know the exact nature of the event as well as the number of events waiting in the queue.

For this reason, SAPI introduces its own event description system. See the following code for an example:

```
void ProcessRecoEvent( HWND hWnd )
{
// Event helper class

CSpEvent event;

// Loop processing events while there are any in the queue
while (event.GetFrom(g_cpRecoCtxt) == S_OK)
    {
        // Look at recognition event only

        switch (event.eEventId)
        {
            case SPEI_RECOGNITION:
                ExecuteCommand(event.RecoResult(), h
                break;
        }
    }
}
```

Three items are needed to fully describe the event. The first item is CspEvent. This is a helper class function that contains several useful functions and is an SPEVENT structure. One method, ::GetFrom, does two things at the same time. It retrieves the next event from the queue and loads the corresponding information into the SPEVENT structure making it ready for your inspection.

The second item is to determine which SAPI event actually took place. At this point you only have to look at the member *eEventId*. If it is an event you are not interested in, skip it and

keeping looking or waiting for other events.

The last item is to match the event with your needs. In this example you are interested only in SPEI_RECOGNITION. If there is a match, you are closer to your goal of finding out which word was spoken. The switch statement handles that for you.

Phrases

Determining the actual phrase is the last part of the process. Having initialized SAPI, received a notice that SAPI has identified a word, and had the application process the message, you now need to isolate the word.

SAPI returns the word information in a list or a series of lists. These lists contain not only the word, but also additional information about the word, words or the entire phrase. Examine the following code:

```
void ExecuteCommand(ISpPhrase *pPhrase, HWND hWnd)
{
    SPPHRASE *pElements;

    // Get the phrase elements, one of which is the rule id we s
    // the grammar. Switch on it to figure out which command was
    if (SUCCEEDED(pPhrase->GetPhrase(&pElements)))
    {
        switch ( pElements->Rule.ulId )
        {
            case VID_Navigation:
            {
                switch( pElements->pProperties->vVal
                {
                    case VID_Counter:
                        PostMessage( hWnd, W
                        break;

                }
            }
            break;
        }
    }
    // Free the pElements memory which was allocated for us
    ::CoTaskMemFree(pElements);
}
```

```
}  
}
```

Without knowing too much about phrase structures, it is obvious that the code drills down into it. This function takes an `IspPhrase` interface, extracts an exact phrase element (in the form of *pElements*) and determines which rule has been invoked. Other examples will go one step further and get the exact words spoken. Phrases and rules will be discussed in languishing detail [CoffeeS2].

To understand rules better, look at `coffee.xml`. Notice that there are two rules defined. RULE ID tags delineate both. The main rule is `VID_Navigation`. The second one is `VID_Place`, but this is of lesser importance because it is subservient to `VID_Navigation`. In essence, `coffee.xml` defines two sets of phrases. The first set uses the commands “Enter” and “Go To,” and the second set uses the words “counter,” “shop,” and “store.” The recognizer mixes and matches words, selecting one from the first set and one from the second set. Therefore, sentences such as “enter store” and “go to counter” invoke a SAPI rule. There are also optional words that may be used or not used. “Please enter the store” is not only more polite but it is also a valid match. However, the recognizer ignores “please” and “the.” This way, you can speak more naturally and at the same time not encumber SAPI.

However, this example does not go quite that far. The code stops at the rule level in the switch statement with “case `VID_Counter`.” Once you have said, “Please enter the store,” SAPI invokes the appropriate rule (`VID_Navigation`). That is why you could have said “counter” or “shop” and still have gone to the same place. In contrast, had you said, “Please enter the restaurant,” no rule would have been invoked because no definition includes “restaurant.” In later examples, the exact word will be of more interest.

At this point, a rule has been invoked and caught by the

application. Now you can process it as you see fit. CoffeeS0 is interested only in providing you with textual feedback and so it passes a `PostMessage()` back to the owning window with instructions to change the text. "Please order when ready!" appears on the screen.

Conclusions

Hopefully you are not overwhelmed at this point. SAPI 5, like all other systems, requires a certain amount of overhead programming. The intent was to minimize this overhead so that you can concentrate on the function of the application rather than working with the SAPI code. Most of the code introduced here is initialization and should only happen once during the application launch. With that done, you are free to add additional features.



CoffeeS1

Introduction

With the basic structure for speech recognition firmly in place, CoffeeS1 expands word recognition capabilities. You navigate around the coffee shop, and you can place orders for different kinds of coffee drinks.

The example will focus on grammars generally and, on command and control grammar specifically. The following topics will be discussed

- Grammars: Command and Control, dictation.
- Phrase: Phrase structure, word recognition.
- Grammar Files: XML tagging.

Grammar Types

Command and Control

The last example (CoffeeS0) was not robust. You were limited to about five words. However, they are five words of special interest and using them, you could move around the application. Using grammar in the command and control function limits the use of words and bestows upon them specialized meanings. This is convenient for some uses in applications. In the last chapter, you learned that grammars could have limited recognition contexts such as for menus. As an example, you would want the application to respond, or even

to attempt to respond, to certain words relating to menu items or to the menu bar such as “file,” “open,” or “print.”

These words are provided in an exclusive list. If the word is not found, it is not recognized. Also, word order matters in some cases. In CoffeeS0, “Go to counter” was understood and “Counter go to” was not. Using a list approach is also called rule-based or context-free grammar. Words are evaluated according to a fixed set of rules. In short, the word is either in the list or it is not. There is no attempt to figure out the intent of the word based on the words that came before or after it. That is, there is no context for the words.

SAPI 5 uses extensible markup language (XML) to create this list. The file may be generated ahead of time or compiled during program execution. Because command and control deals mostly with lists, words can be added dynamically and can accommodate new situations easily.

CoffeeS1 addresses word order and the ability to sequence words.

Dictation

Command and control has obvious shortcomings. As mentioned, it is limited in the words used. Someone has to spend the time to manually define the command set. Often, you will want to speak any word and have it recognized. This is what a traditional speech recognition (SR) program does. That is, you can dictate any word, no matter how esoteric, into a word processor and have that word translated into text. For this use of a speech recognition engine, you must move from command and control to a dictation grammar. Instead of an XML-based vocabulary, dictation grammar uses a much more extensive range of words and determines each word based on context. The words immediately before and after it are studied and dictation grammar chooses the most likely outcome. For this reason, this is also called a statistical language model (SLM).

SR engines have wide latitude of vocabularies. The Microsoft SR engine that SAPI 5 includes 60,000 English words and provides an adequate engine for most people. Other engines are specialized for the legal and medical professions, for example. These can be massive databases generated by commercial firms. In addition, different languages including Japanese, Chinese, German, and Russian are also available.

For as widely disparate as these languages and usages seem, SAPI 5 handles them in the same way. The programming approach is very similar. Two other samples provide a dictation approach to speech recognition: Simple Dictation and Dictation Pad. These may be found on the SAPI 5.1 SDK and are documented separately. Coffee on the other hand, limits itself solely to command and control usage.

Phrases

SAPI returns the actual recognized words through a series of structures collectively called phrases. You have seen evidence of this with SPEI_PHRASE_START event indicating the start of the recognition process. For command and control uses, it is a two-step process: Determine the activated rule, and then inspect the elements (or words) within that phrase.

CoffeeS0 briefly introduced the first step. While processing a recognition event, you discovered which rule was activated but stopped there. CoffeeS1 takes the next logical step to recognize the exact words used so patrons can get their drinks.

This examination takes place in the ExecuteCommand() routine. As in CoffeeS0, one of the parameters is the phrase. Remember, this phrase is the final result, rather than a hypothesis, so assume SAPI is savvy enough to translate exactly what you said. You will be depending on this phrase for navigation. At this point, you are only interested in which grammar rule was activated. That means the patrons still cannot go to different

places in the shop even though they might request to do so. All navigation statements always lead to the counter.

However, CoffeeS1 introduces a new grammar rule: VID_EspressoDrinks. Defined in coffee.xml, this rule lists all drinks available to the customers. Actually, it is several rules bound together that will be discussed later. Again, you are only concerned in the top-level rule of VID_EspressoDrinks. If you place an order that matches this rule, the rule activates and the result is passed back from SAPI. In typical demanding coffee shop fashion, this could be “Get me an iced decaf single tall peppermint whole espresso.” Orders could even include “single triple short tall grande,” and still be valid SAPI grammar although it might raise an eyebrow (if that were possible).

With the order placed and recognition successful, CoffeeS1 now gets to the task of dissecting the phrase. From the original phrase, an IspPhrase interface contains the method GetPhrase() to construct the elements (or word) list.

```
SPPHRASE *pElements;  
if (SUCCEEDED(pPhrase->GetPhrase(&pElements)))
```

If successful, *pElements* contains all the information required to construct the sentence. To determine which rule activated and then to learn more about it, look at the Member Rule. This is a structure (SPPHRASERULE) but one that fully describes the rule. The rule ID is found in its member ulld. CoffeeS1 numerically defines the rule VID_EspressoDrinks in the XML file, so that matching becomes easy. Use a simple switch statement in the code to determine the more specific handling routines.

Two things need to be pointed out about the upcoming word list. First, the words are represented numerically rather than by a string. Associating the value of the word to the string itself uses a look-up table. In this case, CoffeeS1 stores the words as a resource in the application.

Second, the actual words are formed by a link list with each

word represented by a member in the sequence. The first element is a structure (of type SPPHRASEPROPERTY) pointed to by *pElements->pProperties* and each subsequent structure uses the SPPHRASEPROPERTY's *pNextSibling* member. Traveling this chain is standard link list operation.

```
case VID_EspressoDrinks:
    // This memory will be freed when the WM_ESPRESSOORDER
    ULONG *pulIds = new ULONG[MAX_ID_ARRAY];

    const SPPHRASEPROPERTY *pProp = NULL;
    int iCnt = 0;
    if ( pulIds )
    {
        ZeroMemory( pulIds, MAX_ID_ARRAY * sizeof(ULONG) );
        pProp = pElements->pProperties;
        // Fill in an array with the drink properties received
        while ( pProp && iCnt < MAX_ID_ARRAY )
        {
            pulIds[iCnt] = static_cast< ULONG >(pProp->v);
            pProp = pProp->pNextSibling;
            iCnt++;
        }
        PostMessage(hwnd, WM_ESPRESSOORDER, NULL, (LPARAM) pulIds );
    }
}
```

To inspect the elements, the code steps through the links one node at a time until the next node is NULL (meaning there are no more nodes to transverse) or it has already visited at least `MAX_ID_ARRAY` number of nodes. `CoffeeS1` imposes this `MAX_ID_ARRAY` limitation.

Besides stepping through the link list, this code also stores the words in an internal array for later processing. This not only keeps a record of the words but also helps with sorting. Remember, don't worry about word order. Customers can say "get me a mocha two percent tall," and still end up with a tall two percent mocha. However, if you do change the word order then you need the ability to sort internally. To indicate empty array elements, flag them with a zero, hence the `Win32 ZeroMemory()` call. You can use other methods; this one was just convenient for this example.

After going through the list, CoffeeS1 is ready to display the newly derived information. A message is passed to the owning window (WM_ESPRESSOORDER) indicating the application has additional processing. At this point, SAPI is no longer involved. SAPI will even free the objects it created although CoffeeS1 must manually free *pElements* since it manually created it. Even so, COM is smart enough to delete any nodes in the link list associated with the list. The rest of the processing is on CoffeeS1's part and mostly to update the screen. When you speak again, the whole process above is repeated.

Grammar Files

As mentioned, the Coffee examples use command and control grammar. This is a discrete list of words associated with certain rules. Coffee keeps this list in two forms. An XML-based file allows you to maintain this list. Ultimately, SAPI can only read a binary or compiled version of that file. This is a grammar configuration that is saved with the .cfg file suffix. It was by clever design that CFG not only means "configuration," but also "context-free grammar." Approbation aside, grammar files may be generated dynamically during the application's run time. If it is provided with only an xml file, SAPI will compile the file automatically and use the resulting grammar. On the other hand, the grammar may also be compiled ahead of time by the programming team. This restricts access to the vocabulary so users cannot change grammars unexpectedly. This method is also faster for applications since no compiling time is required during operation. The SAPI SDK application provides a compiler called GramComp. Grammar compilation using this tool is documented separately.

SAPI defines the XML tags and their uses and lists them in Reference API. For a more complete discussion, see [Text grammar format](#). As a brief overview of the structure, look at coffee.xml in the CoffeeS1 project. There are several rules defined but only two are considered top level:

VID_EspressoDrinks and VID_Navigation. These are the significant rules for SAPI. When a rule match is made, it is one of these IDs that is passed back to the application. Also, look at ExecuteCommand(). The two case statements coincide with the top-level rule names.

The TOPLEVEL tag within the RULE statement gives these rules their special status. Not only does this identify the rule as being top level, but it also sets the activation state. Only top-level rules may be activated or deactivated. SAPI recognizes active rules and conversely does not recognize deactivated ones. The application may change the state of the rules during execution. If a rule is no longer needed, it may be deactivated. This allows you to turn rules on and off based on the current recognition context. For example, if you have a menu or menu item deactivated, SAPI will not need to attempt to recognize the words associated with it. When the menu is active again, the rule will likewise be activated.

The words or phrases are listed inside the rule. The words or phrases may be optional or required. As the name implies, optional words are not required for a successful rules match. SAPI adds them as a convenience to the speaker. "Please enter the shop" is natural and pleasant sounding as opposed the demanding version of the statement. Required words are, of course, required. However, you can present an alternative word list from which any one word can be used to complete the match. In the case of VID_Navigation, you can say either "enter," or "go to," but not both.

In the same manner, you may reference other rules but not other top-level rules. Continuing the VID_Navigation example, the last portion of the requirement is that the rule VID_Place must be successfully matched. The three alternatives, "counter," "shop," and "store" are defined as VID_Place. If you say one of these three words, the rule is successfully matched. Upon successful completion of all the requirements, the top-level rule, VID_Navigation matches, and an SPEI_RECOGNITION

event passes back to the application.

Additional study of coffee.xml helps you understand how complex rules are constructed. The other rules are basically the same format and follow the same structure. Look up unfamiliar tags in the “Text grammar format” section of the reference API. Curiously enough, the program is case sensitive. “The” and “the” may be duplicated as entries. They may even have the same ID such as `<ID NAME="The" VAL="1" />` and `<ID NAME="the" VAL="1" />`. While this case has the same pronunciation, consider other words such as “Polish” and “polish.” This applies equally to rule names. There is no requirement for engines to recognize the words as different; however, engine vendors may want to do so. By making the word case sensitive, newer engines can take advantage of these differences.

The first portion of the file assigned numeric values to the individual elements. SAPI does not require this, although in the CoffeeS1 example, you can sort the words. The sorting is based on the “VAL=” tag. Remember to keep the words actually found in the array *pullds* for this purpose.

Activating the rules is the same as in CoffeeS0.



CoffeeS2

Introduction

This chapter introduces two new concepts: navigation and politeness. Not only will you be able to go to the counter (this is the same as in CoffeeS1) but the manager will also open the office to you. However, you will not be able to order drinks from the office. The second feature will ask the patron to repeat the order if it was not clearly understood.

The following topics will be discussed:

Grammars: Rules activation and deactivation.

Events: Expanding events, SPEI_FALSE_RECOGNITION.

Grammar Rule Activation

CoffeeS0 and CoffeeS1 introduced the concept of rule activation/deactivation in only the broadest sense. In fact, those rules made two assumptions. The first is that the grammar rule itself wanted the rules on by default and the second assumption was that the application wanted them on by default.

The command and control grammar sets the initial (or default) state of a particular rule to either active or inactive. In the XML file, the top-level rule is defined as:

```
<RULE ID="VID_Navigation" TOPLEVEL="ACTIVE">
```

If nothing changes this status, it remains in that state for the

duration of the application's lifespan. Setting "INACTIVE" will set the state to "off" by default.

However, you will need to react to the changes made by the users. In some situations, the grammar rule may no longer be appropriate. In CoffeeS1, you ordered drinks from any location. That is, from the counter, store or office. CoffeeS2 is more discriminating and limits drink order placement to the counter. As another example (and perhaps a more practical one), if you dictated to a speech-enabled word processor, the command and control words would no longer require special actions. If you spoke the word "menu," it would confuse you to suddenly see a menu drop down.

In either case, you need a way to disable the grammar. A drastic method would be to simply unload the entire grammar when it was no longer required. However, in CoffeeS2's instance, both the navigation and drink order rules are in the same file. If you unloaded the file, you would lose both rules altogether. Rather, selectively activate or deactivate individual top-level rules.

In the past, Coffee has simply accepted the rules as they were.

```
hr = g_cpCmdGrammar->SetRuleState( NULL, NULL, SPRS_ACTIVE );
```

Passing NULL for both the rule name and rule value, you accept all the rules in the file as they stand. CoffeeS2 initially places caffeine-seeking patrons outside the shop. Their first command has to be a navigational one. Because drinks cannot be ordered at any place other than the counter, you must deactivate the rule. From InitSAPI() after LoadCmdFromResource():

```
hr = g_cpCmdGrammar->SetRuleIdState( VID_EspressoDrinks, SPRS_INACTI  
hr = g_cpCmdGrammar->SetRuleIdState( VID_Navigation, SPRS_ACTIVE );
```

The second line is included for completeness. If not included, the state would still be activated since that particular top-level rule is active. In the same vein, you could use the CoffeeS1 method and activate all the rules at one time and then

selectively change rule states. CoffeeS2 has only two rules so either way you have the same number of lines of code.

If you activate a rule, you can also deactivate it. Deactivation is set in the CounterPaneProc() procedure, WM_INITPANE message. The placement of the call will be discussed in a moment. When the time comes, the rule deactivates with:

```
g_cpCmdGrammar->SetRuleIdState( VID_EspressoDrinks, SPRS_INACTIVE );
```

In order to choose which of the two locations (the counter or office), CoffeeS2 has to check its ExecuteCommand(). The matched grammar rule holds this information. By contrast, CoffeeS1 only wants to know if any navigation rule was invoked, not necessarily which one. For this new information, you can determine the rule almost identically to the way in which the case statement VID_EspressoDrinks determines drink orders. Given an lspPhrase interface with ExecuteCommand(), use GetPhrase() to retrieve the phrase returned by SAPI.

The next step between CoffeeS1 and CoffeeS2 appears to be different but fundamentally it is the same. Look at the same value structure that CoffeeS1 used to determine the rule :

```
switch( pElements->pProperties->vValue.ulVal )
```

Remember that in the XML grammar file (coffee.xml), you not only numerically defined each drink characteristic individually, but you also defined each rule and sub-expression. That is, although “Decaffeinated” is defined (VID_Decaf) as a drink characteristic. You also have the OrderList rule (VID_OrderList) allowing you to use multiple characteristics and a top-level navigation rule (VID_Navigation). This convenience allows you to look in one location rather than drilling down into structures. The difference in the two switch statements for ExecuteCommand() is that case VID_EspressoDrinks goes one step further and moves along a possible link list where case VID_Navigation needs to look at this information only once.

Expanding Events

Up until now you have been interested in only one event: SPEI_RECOGNITION. SAPI currently handles more than 30 different kinds of events. To better understand the process, suppose you wanted to add two new events to your repertoire, a sound start (SPEI_SOUND_START) and a sound end (SPEI_SOUND_END). The speech recognition engine triggers these events when the microphone detects or stops detecting sound. In the best case scenario, it is your voice triggering this event. However, it may be another sound such as a phone, a cough, or any one of a myriad of cacophonous noises.

It is the event SPEI_SOUND_START that initiates SAPI's recognition attempts. Once detected, SAPI will start processing on the audio stream. It will listen to and process the stream concurrently. That is, you don't have to stop speaking before it attempts to recognize your voice. After SAPI detects a sound start, it begins processing and sends an SPEI_PHRASE_START event. During the recognition process, interim SPEI_HYPOTHESIS events indicate attempts are being made. In brief, a hypothesis is the current best guess about what you have spoken up to that moment. Having spoken "Please go to the counter," SAPI might return with five SPEI_HYPOTHESIS events (one for each new word spoken added) along with a phrase structure with the actual words in it.

An SPEI_SOUND_END occurs after a pre-determined amount of time passes during which SAPI detects no useful sounds. After enough silence, SAPI assumes you have stopped, or paused between phrases or sentences. SAPI then finishes the recognition process and returns the final decision about what was spoken. Rather than sending back one last SPEI_HYPOTHESIS, it returns one (and only one) of three values.

An SPEI_RECOGNITION event indicates a word match (from one of the available grammars) and with a sufficiently high

confidence value to consider recognition successful. As an example, in the CoffeeS0 application you might have said, "Please go to the counter." This matches a grammar rule and SAPI returns SPEI_RECOGNITION.

An SPEI_FALSEERECOGNITION indicates that you probably spoke words (as opposed to a sound of coughing) but SAPI could not find a close enough match to either existing words or grammar rules. For a CoffeeS2 example, you could have said, "please go to the veranda," or have mumbled inaudibly as morning coffee drinkers are prone to do. Since the former case does not match an existing rule and the latter case implies no word could be recognized, SAPI returns SPEI_FALSEERECOGNITION.

An SPEI_RECO_OTHER_CONTEXT indicates a successful recognition was made but that another other application currently running claims it. This is a useful event if there are multiple shared instances running at the same time. For example, if you had said, "please go to the veranda." CoffeeS2 does not have a rule covering this but suppose another application did. The second application, even if not currently the active one, receives an SPEI_RECOGNITION event and CoffeeS2 gets SPEI_RECO_OTHER_CONTEXT. In a way it offers closure for CoffeeS2.

Since there is a range of events possible and not all of them are relevant to all applications at all times, SAPI can filter them. Using ISpEventSource::SetInterest, you can determine which events you want to see. Those events not included in the call will not get generated. By default, that is if you never even call ::SetInterest, only SPEI_RECOGNITION events are generated. To allow other events, you would have to modify the ::SetInterest call and explicitly include them for SAPI. For instance, in InitSAPI():

```
hr = g_cpRecoCtxt->SetInterest(  
    SPFEI(SPEI_RECOGNITION) | SPFEI(SPEI_SOUND_START) | SPFEI(SPEI_S  
    SPFEI(SPEI_RECOGNITION) | SPFEI(SPEI_SOUND_START) | SPFEI(SPEI_S  
);
```

```
//Check return value
```

As expected, `SPEI_SOUND_START` and `SPEI_SOUND_END` are added. Notice if `::SetInterest` is called, you have to explicitly add `SPEI_RECOGNITION` along with any other events. The second parameter is the list of events you want queued. Events can happen faster than even the processor can handle. If this is the case, rather than losing the events, they are put in a queue where they wait until they can be processed. Most of the time, you would want the two parameters to be identical. If you were interested in an event in the first place, you would also be interested in doing something with it. However, this is not always the case and SAPI lets you decide.

If you see the events, you will also want to handle them. You can add the two events to the recognition event loop in `ProcessRecoEvent()`.

```
case SPEI_SOUND_START:
    PostMessage( hWnd, WM_STARTEDTALKING, 0, 0 );
    break;

case SPEI_SOUND_END:
    PostMessage( hWnd, WM_STOPPEDTALKING, 0, 0 );
    break;
```

These post messages back to the `hWnd` window. `CoffeeS1` does not handle these messages past this point but it does demonstrate how it could be done.

Event `SPEI_FALSE_RECOGNITION`

Moving from the hypothetical to the practical, `CoffeeS2` introduces a new event, `SPEI_FALSE_RECOGNITION`. As explained earlier, the event indicates that you spoke a word but that word was not found in the command and control list. If you simply made a noise or if the microphone picked up spurious noises, `SPEI_FALSE_RECOGNITION` will not be returned. In fact, in those two cases, no recognition takes place because SAPI is

smart enough to tell the difference between words and noises. Only sounds close enough to real or acceptable words will trigger an SPEI_RECOGNITION or SPEI_FALSE_RECOGNITION event. At least that's the intent. It is the responsibility of the speech recognition engine to attempt to detect the difference. However, due to the wide latitude of sounds possible and differences among vendors, an occasional SPEI_FALSE_RECOGNITION may be returned instead.

In addition to SPEI_FALSE_RECOGNITION, CoffeeS2 also introduces a time element. You can determine if the patron had spoken a legitimate command and also if that command was spoken over a predetermined amount of time. You can also determine if the utterance was intentional. The temperamental barristas that CoffeeS2 employs ignore a false recognition if spoken too quickly. Otherwise, they will politely ask for the order to be repeated.

First you need to set the interest. In CounterPaneProc(), the WM_INITPANE message actually controls this.

```
hr = g_cpRecoCtxt->SetInterest(  
    SPFEI(SPEI_RECOGNITION) | SPFEI(SPEI_FALSE_RECOGNITION),  
    SPFEI(SPEI_RECOGNITION) | SPFEI(SPEI_FALSE_RECOGNITION)  
);
```

Notice that the interests are chained together with bitwise (rather than logical) OR statements. This way you can add multiple events at the same time. However, the method is in an odd place located in CounterPaneProc() instead of the initialization routine of InitSAPI(). Remember, you can order drinks only in the counter area. Therefore, you want this event available only after the patron has entered the counter area. Although there are several ways to approach the programming logic of this problem, CoffeeS2 sets the events as the patron enters different areas. In contrast, the WM_GOTOOFFICE of the same CounterPaneProc() routine sets the interests again.

```
hr = g_cpRecoCtxt->SetInterest( SPFEI(SPEI_RECOGNITION), SPFEI(SPEI_R
```

The same logic is used for rule activation and deactivation described above.

When you set the interests at the time of entry to a room, you can now have one message handling routine for all speech events. This is the same process as in the previous Coffee examples, that is, `ProcessRecoEvent()` handles the messages at that point, and each event is assigned to an action in the subsequent switch statement.

Use `HandleFalseReco()` in the case of `SPEI_FALSE_RECOGNITION`. You may use `IspRecoResult::GetResultTimes` to determine the time element. This passes back an `SPRECORESULTTIMES` structure containing different timing information for the event. Specifically, *dwTickCount* keeps the time from the start of the event. Subtracting this from the system's time yields the duration of the recognition. If the false recognition took longer than the arbitrarily determined value of `MIN_ORDER_INTERVAL`, the patron is asked to repeat the request.



CoffeeS3

Introduction

TTS! You're finally going to use text-to-speech. Up until now, the Coffee examples have limited themselves to simply accepting speech. You could talk to your CoffeeS3 minions and expect to get drinks. Now you can add the second of the two major components of SAPI – that of text-to-speech.

With all the excitement from the first two examples, CoffeeS3 slows the pace down for the moment. You will be pleasantly surprised at how easy it is to add this feature. The design stage placed emphasis on making things simple. In contrast, SAPI 4 required 200 lines to make a so-called simple “hello world” speak. SAPI 5 requires as few as two. This remarkable reduction in code was possible due to consolidation of overhead. SAPI marshals the required elements for you so your programs have less material to access directly. Also, SAPI uses intelligent defaults whenever possible. You may set many of the elements using existing defaults. The Speech Recognition tab in Speech properties accesses most of these elements, such as voice and speaking rate. Therefore, at the simplest, it is very simple. Naturally, you may override any of these assumptions or defaults. However, for CoffeeS3, you will start with simple tasks. Don't worry; additional features will be addressed in the next few chapters.

The following topic will be discussed

- Text-to-speech: Initialization, implementation and speaking

text

Initialization

The initialization routine is almost anticlimactic. It is essentially two lines. This is very similar to setting up a speech recognition (SR) engine as you did in CoffeeS0: declare the interface and create the instance from the class ID.

```
COMPtr<ISpVoice> cpVoice;  
  
hr = cpVoice.CoCreateInstance(CLSID_SpVoice);
```

To actually speak something takes one more line of code.

```
hr = cpVoice->Speak(L" Hello, world ", 0, NULL);
```

Any application may initialize TTS this way and this is often the preferred method. However, CoffeeS3 takes a slightly different approach. SAPI realizes that because SR and TTS are commonly used together, the initialization routine takes a short cut. Basically, the SR engine provides this capability for you. The following is in CoffeeS3's InitSAPI():

```
COMPtr<ISpVoice> g_cpVoice;  
hr = g_cpRecoCtxt->GetVoice(&g_cpVoice);
```

Although it takes the same number of lines of code, TTS is available through the `ISpRecoContext` interface. In fact, this makes the same call to `CoCreateInstance(CLSID_SpVoice)`. The difference is that this method automatically provides the ability to interrupt TTS whenever you start speaking again. This is appropriately known as "bargen in." Without this capability, the TTS voice would continue to speak in the background even if you were talking. This might cause audio feedback. Of course, you can still write your own TTS interrupt routine, but the barge in service is provided as a convenience.

Defaults are usually found in Speech properties. That is, when SAPI is properly installed, Speech properties will have defaults

for all parameters and will use those defaults. These defaults include the voice, speaking rate, and the language used. This is how TTS can get away with using only two lines of code.

However, it is possible that the defaults may not be available or valid, and the application must always check the return value.

That's about it for TTS. You have seen how to initiate a voice and how to speak something. The rest of the code implements these instances. For example, CoffeeS3 talks on five occasions and you need `::Speak` at those times.



CoffeeS4

Introduction

With the completion of the first four Coffee samples, you covered the basics. You were shown how to place orders by speaking and you were able to hear the request spoken back. In both cases, SAPI used its defaults. Specifically, the barrista talked back to you with the voice set using Speech properties in Control Panel. In truth, SAPI usually offers much more than a single voice. Other voices may be available as well as different languages, engines, or other resources. Many of these can be changed either dynamically (always through Speech properties and sometimes by the application itself) and programmatically (through the application's code).

CoffeeS4 introduces the fundamentals of resource management. As in CoffeeS3, you can enter the manager's office. In the next two tutorials, you are going to manage employees. CoffeeS4 displays the voices available. Like any good coffeehouse help, you will not be able to manage them too much, but you will be able to hear one of them speak in the currently selected voice. CoffeeS5 lets you change the voice. In doing this simple task, you prepare for bigger things.

The following topic will be discussed

- Resources
- Managing Resources

Resources

To provide the robust range of options, such as many different voices, recognizers, languages, and user interface dialog boxes, SAPI needs to store this information for later use. This stored information is collectively referred as resources. Resource management, therefore, is the ability to query SAPI for the availability of resources, gather information about the resource, initiate, instantiate, or remove resources as needed. CoffeeS4 finds and displays voices stored as resources, and determines which one is active. Using the same techniques, you can filter the voices for language, gender, age, or some other criterion and display only those.

SAPI relies on two closely associated terms and concepts: objects and tokens. COM and object oriented programmers will recognize objects. They are the functional implementation of a class. That is, an implemented object will have memory allocated to it and have its members and methods initialized. Once implemented or instantiated, the object may be used by the application.

However, SAPI only instantiates objects when they are needed. It is a waste of the computer's memory to have unneeded objects allocated. Therefore, SAPI stores the information needed to create those objects including the voice's name, language, and GUID. This stored information is referred to as a token. A token is the textual representation of the parameters needed to fully implement that resource. Stated another way, tokens are the parameters for the object. SAPI needs only to read a token at the right time and instantiate an object based on that information.

In addition, there can be many tokens and SAPI organizes them into related groups called categories. Currently SAPI maintains about eight general categories. Of particular interest to CoffeeS4 is SPCAT_VOICES. This category contains all the voice-related tokens.

Each type of token is different and is documented separately.

That is, a voice token will have different values and key entries from a recognizer token because each has different functions. Tokens of the same type (such as voice tokens) will contain the same required entries (such as language, gender, age) although vendors may provide additional and non-standard information for their resource.

The following is a sample representation of a voice token. Not all entries are listed.

Token	ValueName	Sample Value	Co
MSMary			Required entry. Th the token name.
	(Default)	MS Mary	Required entry. Th the langu independ name.
	409	MS Mary	There ma multiple entries; c for each language voice supports least one entry is required. numeric is the standard Windows language code and hexadeci
	809	MS Mari	

			with no leading notation as "0x."
	CLSID	{65DBDDEF-0725-11D3-B50C-00C04F797396}	Required entry. This is the class (CLSID) for the object.
MSMary/Attributes			Required entry. Additional information is available through the attribute, although all attributes are required.
	Language	409;809	Required is the language voice supports
	Age	Adult	Required is the age of the voice
	Gender	Female	Required is the gender for the voice

Methods

Resources alone would be only marginally useful if you could not work with them. Often you will need to know what resources are

available. CoffeeS4, for instance, searches for all the voices it can use. Once found, you may need to use that resource either to instantiate an object or to know more about the resource. There is a rich set of interfaces and helper functions available to do just that. Many of these interfaces are found in the Resource Manager set in Application Level Interfaces of the reference API document. In the same manner, the helper functions provide convenient ways to perform a task without having to know all the underlying details at this point.

For instance, the heart of the matter for CoffeeS4 is getting a list of all the available voices. This task can be done by one helper function `SpEnumTokens`. At the simplest, the call would look like this:

```
//Pointer to token enumerator
CComPtr<IEnumSpObjectTokens> cpEnum;

// Get a token enumerator for tts voices available
HRESULT hr = SpEnumTokens(SPCAT_VOICES, NULL, NULL, &cpEnum);
// check hr result
```

`SPCAT_VOICES` restricts the search to the voice category. You can filter the resources using the middle two parameters (both of which are set to `NULL`). In this case, you want all voice resources so no criteria were set. Just as easily, you could have searched for female voices or narrowed it down even more with just the adult, English-speaking female voices.

cpEnum is an interface pointer to `IEnumSpObjectTokens`. `SpEnumTokens` does all the initialization work and returns a complete list for you. In this case, the list is a complete set of tokens in `SPCAT_VOICES`. You can think of `IEnumSpObjectTokens` as a link list with built-in support functions. Using `IEnumSpObjectTokens`, you can find the next item in the list, skip several items, make a copy of the list, or go back to the beginning of it, among other things. Although some of the methods will be described here, see the reference API section for additional methods and details.

You are on your way to finding and listing all voices. CoffeeS4 uses the following algorithm in the code:

- Finds the voices.
- Searches the list one by one and retrieves the name of each voice.
- Stores the display names of the voices. CoffeeS4 needs this indexed array to refresh the screen during updates.
- Makes one extra step and displays the current voice in red.

Like the display name, CoffeeS4 stores the token name in an indexed array for later use. In both cases, storing the names is not a requirement, rather a convenience. For screen updates, CoffeeS4 could also poll resources again but that seems like a waste of time. For both steps 3 and 4, this involves looping through the list, extracting the appropriate name and assigning it to the array.

All the work is done in the CoffeeS4 ManageEmployeesPaneProc() procedure and specifically the WM_INITPANE case. This initialization is logically placed here because the information must be present at the time the window is rendered.

In one sense, the hard part is done for you. Finding the available voices is accomplished in the one-line SpEnumTokens that passes back a list of all the voices and even provides the means to navigate that list. It also provides a method to determine how many items were found using ::GetCount.

```
static ULONG ulNumTokens;  
hr = cpEnum->GetCount( &ulNumTokens );
```

Knowing the total number of items, CoffeeS4 now allocates the two indexed arrays.

```
static CSpDynamicString*  
ppcDescriptionString;  
ppcDescriptionString = new CSpDynamicString [ulNumTokens];  
//Check hr result
```

```

static WCHAR** ppszTokenIds;
ppszTokenIds = new WCHAR* [ulNumTokens];

//Check hr result
ZeroMemory( ppszTokenIds, ulNumTokens*sizeof( WCHAR* ) );

```

CspDynamicString is a helper function for handling string arrays. It is a string class similar to other object oriented string classes. The subsequent allocation and release of each of its elements is automatic. You do not have to remember to do it manually. On the other hand, *ppszTokenIds* is simpler array of pointers for storing GUIDs of the token. CoffeeS4 manually allocates it because it is needed throughout the application. As a result, it must also be manually freed when no longer required. This is done in `ManageEmployeesPaneCleanup()`. The `ZeroMemory()` confirms all the values are initialized to zero. No valid GUID will be zero.

The next step of looping through the array is equally easy. CoffeeS4 navigates the list item by item to find the voice's name. As mentioned earlier, `IEnumSpObjectTokens` has such a method named `::Next`.

```

IspObjectToken *pToken = NULL
while (cpEnum->Next(1, &pToken, NULL) == S_OK)
{
    //Code here
}

```

The list represented by *cpEnum* is traversed one item at a time as the first parameter indicates. The information is passed back in *pToken*, which is an interface to `IspObjectToken`. You might correctly guess that you will be looking at this interface in a moment. The last parameter is the number of items actually read. Since it is possible to read more than one at a time, it is also possible that not many items are left to read. If that were the case, it would return the number of items it could read. If this parameter simply cannot read any more items, it returns an error. In this case, CoffeeS4 stops looping through the "while"

statement.

As CoffeeS4 steps through the list one at a time, it retrieves the names of the resources. There are two names for the particular token: the token name (also called the token ID) and the display name. The two could be the same but not necessarily. Also the display name can vary by language. It is the display name CoffeeS4 shows in the management window. Again, a helper function is available to simplify this task. SpGetDescription retrieves the display name and assumes the current language. In the case of the sample token, that name would be MSMary/409 value of “MS Mary.”

At the same time, the token ID is also retrieved. This token Id is needed since CoffeeS4 also determines which voice is currently in use and it will need it shortly. No helper function is provided, as this is a straightforward call to get token ID. In the sample token, this would be “MSMary.” In both cases, the information is stored in indexed arrays for later use.

```
while (cpEnum->Next(1, &pToken, NULL) == S_OK)
{
    // Get a string which describes the token, in our case, the
    hr = SpGetDescription( pToken, &ppcDescriptionString[ulIndex]

    // Get the token id, for a low overhead way to retrieve the
    // without holding on to the object itself
    hr = pToken->GetId( &ppszTokenIds[ulIndex] );
    ulIndex++;

    // Release the token itself
    pToken->Release();
    pToken = NULL;
}
```

When no longer needed, the token must be explicitly released. CoffeeS4 resets the pointer to NULL and is ready for the next loop. With all the information stored, the last task is to determine which is the currently active voice. This too, is a simple task. CoffeeS4 loops through the token ID array (ppszTokenIds) and compares each location of the array with the

system voice. If there is a match, it breaks out of the loop and stores the index position in ulCurToken.

```
// Get the token representing the current voice
HRESULT hr = g_cpVoice->GetVoice( &pToken );

if ( SUCCEEDED( hr ) )
{
    // Get the current token ID, and compare it against others t
    // which description string is the one currently selected.
    hr = pToken->GetId( &pszCurTokenId);
    if ( SUCCEEDED( hr ) )
    {
        ulIndex = 0;
        while ( ulIndex < ulNumTokens && 0 != _wcsicmp(pszCu
        {
            ulIndex++;
        }

        // We found it, so set the current index to that of
        if ( ulIndex < ulNumTokens )
        {
            ulCurToken = ulIndex;
        }

        CoTaskMemFree( pszCurTokenId);
    }
    pToken->Release();
}
```

The key to this is the ::GetVoice call. This passes back the current voice. Compare this value against the current voice. The current voice, stored in UICurToken, is used in ManageEmployeesPanePaint() to highlight the active voice in a different color.

The rest of the codes for CoffeeS4 are relatively simple and are basically modifications of previously discussed techniques. Two new rules have been added: Please Manage the Employees and Hear Them Speak. The former displays the list of voices once you are in the office and the latter speaks the current voice. Arrogant perhaps, but the employee states confidently "I will be the best employee you've ever had. Let me work." In order to hear those words every employer loves, a new case must be

added to ProcessRecoEvent(), that of VID_Manage.

The rest of the new code essentially handles events to the screen. OfficePaneProc(), ManageEmployeesPaneProc() and ManageEmployeesPaneCleanup() do the rest of the work.



CoffeeS5

Introduction

The concepts of resource management were introduced in CoffeeS4. There, the sources of information that SAPI needed (such as voices and recognizers) were stored as tokens waiting to be used. When needed, the tokens were read and used as parameters to create objects. In turn, the objects were the actual implementation of those resources in working form. As objects, they have methods, validated data and communication paths to other parts of the speech system.

CoffeeS4 displayed the names of the available voices. It could even speak something in the current voice. However, as far as managing employees, there was little else it could do. CoffeeS5 addresses managing employees a little bit further. In addition to displaying voice names, you can pick a different voice for each employee. You can also choose the type of voice you want displayed and spoken. You can choose feminine, masculine, or all voices.

The following topic will be discussed

- Dynamic Grammars

Dynamic Grammars

Up until now Coffee has only worked with static grammars. That is, for command and control issues, the word list has been both explicit and static. Explicit in that a list of exact words has been

defined and therefore no words outside of this list are recognized. For instance, the grammar rule VID_DrinkType lists five types of drinks. Latte is included and may be ordered, but cola is not included and therefore cannot be ordered. Any order including cola would not be recognized. Static means this list is defined ahead of time and is not subject to change. Although the .xml file may be edited independently of the application, it may not be changed during execution.

As the name implies, a dynamic grammar is the opposite of a static one. First, words may be added and deleted during run time and the list does not need to be predetermined. This gives applications much greater latitude. CoffeeS5 demonstrates this by dynamically adding voices. Because the list of available voices may change not only from one computer to another, but also by user profile, there is no way to predetermine the available voices. CoffeeS4 started the process by polling the system for voice tokens-- the definitive list of available voices. CoffeeS5 expands now to construct and use a grammar.

The logic to initializing the grammar is straightforward although there are a few subtleties. First, the rule is retrieved or created. CoffeeS5 creates a new rule but other applications could modify existing ones. Second, unwanted words are removed. Since CoffeeS5 adds all the words itself, for simplicity, the entire existing rule is erased. Again, another application could examine each word first before removing it. Regardless, SAPI needs to know explicitly to make the changes. As a last step, the new commands are added and the changes committed.

The magic happens in the ManageEmployeesPaneProc() WM_InitPane case. Specifically, because of the ::GetRule call.

```
SPSTATEHANDLE hDynamicRuleHandle;
```

```
hr = g_cpCmdGrammar->GetRule(NULL, DYN_TTSVOICERULE, SPRAF_TopLevel
```

Words may be added and deleted freely now. That is, up to the point of removing all words from a grammar. Conversely, and in

this case, a completely empty grammar may be used and words added in; there is nothing wrong with an empty grammar. As an example, CoffeeS5 can actually add a cola drink to the order list or remove latte if the milk runs out. However, CoffeeS5 starts the rule from scratch. `::GetRule` not only retrieves existing rules but also creates new ones. The `fCreateIfNotExist` member (TRUE in this case) creates the rule.

The first two parameters are often mutually exclusive. The first one, requests a rule by name and the second one searches by ID. In practice, knowing either the name or the ID is enough and you can leave the unneeded one NULL or zero respectively. Since you are creating a new rule, it will not have an ID to search for (hence the NULL value) and you must specify the name (DYN_TTSVOICERULE). The attributes may also be set here. Some attributes may (and will be) set later. This includes activating and deactivating grammars. Other attributes must be set at the time of creation. The rule is being made to be top-level, dynamic and active. The values may be strung together with logical operators to get the exact nuance you want. And last, a handle (`hDynamicRuleHandle`) is passed back.

It is possible that there was already a rule existing for DYN_TTSVOICERULE. It might not have been properly destroyed before or you just forgot to remove it. `::ClearRule` removes the state information for the rule. In essence, all the words are removed and it effectively guarantees a clean start.

```
// Clear the rule first
hr = g_cpCmdGrammar->ClearRule( hDynamicRuleHandle );

// Commit the changes
hr = g_cpCmdGrammar->Commit(0);
```

After making any changes to the grammar, SAPI must be notified explicitly. `::Commit` submits the changes to SAPI and thereafter the grammar is considered in the new state, with the new words. The parameter must be set to zero or it returns an error. In the example above, CoffeeS5 simply clears out the

words and commits the changes. The grammar is now in a pristine state.

Words are added to the grammar using `::AddWordTransition`.



CoffeeS6

Introduction

In CoffeeS6, the last of the tutorial chapters, no new programming code is introduced, per se. Rather, CoffeeS6 makes a variation on an existing theme. The past several Coffee tutorials demonstrated working with context-free grammars, also called grammar rules. In short, they were predetermined lists of words that needed to be matched exactly. Even dynamic grammars, though more flexible, still had to match exact words once the word list was determined. For all the promise of speech recognition, using the models presented so far, you have not been able to dictate or use free-formed speech. With CoffeeS6, you can use unrestrained speech in your applications. It uses the simple case of renaming the coffee shop to anything you want.

The following topics will be discussed:

- Embedded Dictation
- Grammar Modifiers

Also see [Grammar Format Tags: Special Characters](#) for more about grammar modifiers.

Embedded Dictation

As mentioned in the introduction, it would be limiting if you could not speak anything; that is, dictate to your application. Remember, the Coffee samples showcase a command and

control model. As users, you are ordering the application to perform the following: get coffee, hire and fire employees, and make them talk to you. There is little room to get chatty with the other , customers or employees. In fact, the Dictation model is better covered in both width and depth in the SDK sample applications Dictation Pad and Simple Dictation. By the same token, CoffeeS6 would be remiss in omitting this entirely. For that reason, the coffee shop manager now allows the user to change the name of the shop.

For instance, go to office and give the order “manage store name.” A new screen displays and if you follow the instructions, you can say, “Rename the coffee shop to” and provide any name you want. CoffeeS6 will echo back the new name as “Welcome to the X coffee shop,” X, of course, being the moniker. The name will even display in all in the subsequent windows.

This is called embedded dictation because it combines the two forms. CoffeeS6 is still in command and control mode and using the same grammar rules as before. The advantages are that Coffee users can easily expand their drink offerings with only a minor change to the grammar file. Embedded dictation is not dynamic grammar. A rule is not created right before use and seeded with the words you want. In fact, embedded dictation would not handle this case appropriately. In renaming the coffee shop, you as the programmer, will have no idea what the user may choose for a name. The only real limitation is if the new name is in the dictation or not. Even this can be worked around.

Keep in mind, embedded dictation does not create new rules. CoffeeS6 has explicit support for renaming the store because a rule provides for that case. Instead, embedded dictation lessens restrictions on existing grammars, whether they are static or dynamic. You need not even change the code. Your application might want to handle the words elements differently, but that change is not really SAPI-related.

Grammar Modifiers

Programmatically, there is no new code to support modification. CoffeeS6 simply modifies existing grammar rules. Technically, these modifiers are not XML tags in the same way that <P> and <I> are, for example. They appear inside the rule and are associated to other text or the modifier appears as the element itself. Each is explained below.

Wildcard: ...

The ellipsis is a wildcard symbol indicating that any word or words may be accepted in this position. Informally it is also known as a garbage collector because it is used to accept words that the application may not explicitly care about. In this role, four things happen. First, the user may say anything, including a series of words. As expected, the engine attempts to recognize the words, although not much is done with the words afterward. Coffee needs to know that a word was actually spoken and that the user was not just coughing or sneezing. Second, the rule will still be matched and activated. If the rule uses several parts (such as VID_EspressoDrinks), the wildcard words successfully match the rule requirements. Third, the parsed phrase returns only one element for the ellipsis regardless of the number of words actually spoken. For the most part, Coffee's only interest is that legitimate words were spoken but not what the words actually were. And fourth, the element in the returned phrase itself will be the ellipsis rather than any useful word. Again, since Coffee is interested only that something was spoken, it makes sense not to return the actual word. If you are interested in the word, then the wildcard marker is not the right one for the rule; see Dictation below. In short, it really is a wildcard because any word may be used and still activate the rule.

The Coffee.xml (for CoffeeS6) snippet uses it in the following role in VID_EspressoDrinks:

```
<L>
    <P>May I have</P>
    <P>Can I have</P>
    <P>Can I get</P>
    <P>Please get me</P>
    <P>Get me</P>
    <P>I'd like</P>
    <P>I would like</P>
    <P>...</P>
</L>
```

In previous Coffee examples, the wording of the request was limited to one of the first seven requests. That is, the user had to begin by saying, “please get me,” or “can I get,” a drink. Because of the wildcard, the CoffeeS6 user may say almost anything and still get the drink. “Gimme a mocha,” will work (mimicking real life to boot). Even “gee-I-dunno-I-suppose-I’d-like a mocha” will also work provided the customer slurs the words together enough. Remember, the rules are phrased-based and a sufficiently long pause between words never activates rules in the same way that the indecisive customer will never gets drinks by saying “may (pause) I (pause) have (pause) a (pause) mocha.”

So why even have the other phrases if the ellipsis is present? There are some subtleties to that answer. First, astute programmers may notice that they do not care what is spoken here. In the code, nothing is ever actually dependent on the fact that the customer said “please” or not. Yet, a rule consisting of only “coffee” may fire inappropriately. For example, a customer simply saying, “coffee is good,” might fire a too-simple rule. In this case, some sort of introductory clause is needed. Second, the additional words speed up the recognition process. The engine is much more likely to recognize “please” or “may” because it is described exactly in the rule. It also increases the confidence rating for the rule overall. Though both of the following phrases would activate the drink rule, “please get me a mocha,” returns a much higher confidence rating than would “aardvark a mocha.”

Dictation: *

The asterisk is a dictation indicator. Like the wildcard ellipsis, any word (or possibly words, see next entry) will validate the rule. In the same manner, the engine will attempt to recognize the word. The difference is that the actual word is returned back to the user in the phrase element. This is the key to renaming the coffee store.

The new rule VID_Rename is defined:

```
<RULE ID="VID_Rename" TOPLEVEL="ACTIVE">  
    <P PROPNAME="Named SAPI Coffee Shop to"> Rename the coffee s  
</RULE>
```

Ignoring the plus sign for the moment, VID_Rename is activated on upon successfully matching “rename the coffee shop to,” followed by any word in the engine’s dictionary. The parsed phrase returns an element containing the actual word. With the actual word available, CoffeeS6 can use it to display the new name as it would with any stored variable.

Multiple dictation: +

The dictation entry in VID_Rename has a plus sign after the asterisk. This indicates that multiple words may be accepted in the rule. This way you can dictate longer phrases. Over-zealous customers may now rename the coffee store to virtually any name they want. By saying, “rename the coffee shop to Billy Bob Joe’s and Sally Jean Ann’s Coffee Emporium on the Highway,” they have successfully changed the name.

Confidence Increase: +

Confidence Decrease: -

One of these two signs placed in front of words respectively increases or decreases the required confidence for a successful

recognition. Obviously increasing the required confidence means that the speech recognition engine will have to be much more certain that the word it hears really is the expected word. For example, if the user is responding to an important question such as “Reformat hard disk?” you want to make an extra effort that what is recognized as “yes” really is “yes.” To make sure, the rule is noted as “+yes”.

Likewise, the minus sign decreases the required confidence for the word. That is, you can de-emphasize some words. Although the word is required for the rule, it is not important to verify that the user actually said it. It is a case of “close enough is good enough.”

In CoffeeS6 this is seen in the VID_ThingsToManage rule:

```
<RULE ID="VID_ThingsToManage" >
  <L PROPID="VID_ThingsToManage">
    <P VAL="VID_Employees">employees</P>
    <P VAL="VID_ShopName">-shop +name</P>
    <P VAL="VID_ShopName">-store +name</P>
  </L>
</RULE>
```

The last two phrase elements allow the store name to be changed. However, Coffee is de-emphasizing the words “shop” and “store.” It is not important that users speak this word precisely; it just needs to be reasonably close. However, “name” has to be recognized clearly.

Code Modifications

In terms of code support for embedded dictation, there are very few changes. The case of handling a new rule is added, of course. The real work of CoffeeS6 is in the case of VID_Rename of ExecuteCommand(). Notice there is no extra effort required to implement the dictation itself.

```
if ( 5 <= pElements->Rule.ulCountOfElements )
{
```

```

if ( SUCCEEDED( pPhrase->GetText( 5, pElements->Rule.ulCount
{
    int ilen = wcslen(
    pElements->pProperties->pszName );
    ilen = (ilen + wcslen(wszCoMemNameText ) + 2) * size
    wszCoMemValueText = (WCHAR *)CoTaskMemAlloc( ilen );

    if ( wszCoMemValueText )
    {
        wcscpy( wszCoMemValueText, pElements->pProper
        wcscat( wszCoMemValueText, L" " );
        wcscat( wszCoMemValueText, wszCoMemNameText )

        // Copy new shop name to global shop name
        _tcsncpy( g_szShopName, W2T(wszCoMemNameText
        PostMessage( hWnd, WM_RENAMEWINDOW, 0, (LPAR

        CoTaskMemFree( wszCoMemNameText );
    }
}
}
}

```

This code filters through the phrase elements to retrieve the dictated text. The phrase elements are examined and retrieved. In practice, it is usually better not to assume that the sixth element is always the one you want but, in this case, CoffeeS6 does. Since the rename rule allows multiple words, CoffeeS6 starts at the sixth element (since it knows the first five elements have to be “rename the coffee shop to”) and strings together the rest of the words for the new name.



Text-to-Speech Tutorial

This tutorial covers a very basic text-to-speech (TTS) example. The console application is one of the simplest demonstrations of speech. It is the "Hello World" equivalent for TTS. An equivalent sample for a Windows application using a graphical interface (and event pump) is available in [Using Events with TTS](#).

The sample builds up from the simplest (though nonfunctional) COM framework to speaking a sentence. Steps are provided for each new function. The sample even goes one step beyond demonstrating the use XML tags to modify speech. The [Complete Sample Application](#) is at the bottom of the page.

[Step 1: Setting Up The Project](#)

[Step 2: Initialize COM](#)

[Step 3: Setting Up Voices](#)

[Step 4: Speak!](#)

[Step 5: Modifying Speech](#)

Step 1: Setting up the project

While it is possible to write an application from scratch, it is easier to start from an existing project. In this case, use Visual Studio's application wizard to create a Win32 console application. Choose "Hello, world" as the sample when asked during the wizard set up. After generating it, open the STDAfx.h file and paste the following code after "#include <stdio.h>" but before the "#endif" statement. This sets up the additional dependencies SAPI requires.

```
#define _ATL_APARTMENT_THREADED

#include <atlbase.h>
//You may derive a class from CComModule and use it if you want to c
//but do not change the name of _Module
extern CComModule _Module;
#include <atlcom.h>
```

Code Listing 1

Next add the paths to SAPI.h and SAPI.lib files. The paths shown are for a standard SAPI SDK install. If the compiler is unable to locate either file, or if a nonstandard install was performed, use the new path to the files. Change the project settings to reflect the paths. Using the Project->Settings. menu item, set the SAPI.h path. Click the C/C++ tab and select Preprocessor from the Category drop-down list. Enter the following in the "Additional include directories": C:\Program Files\Microsoft Speech SDK 5.1\Include.

To set the SAPI.lib path:

1. Select the Link tab from the Same Settings dialog box.
2. Choose Input from the Category drop-down list.
3. Add the following path to the "Additional library path":

C:\Program Files\Microsoft Speech SDK 5.1\Lib\i386.

4. Also add "sapi.lib" to the "Object/library modules" line. Be sure that the name is separated by a space.

Step 2: Initialize COM

SAPI is a COM-based application, and COM must be initialized both before use and during the time SAPI is active. In most cases, this is for the lifetime of the host application. The following code (from Listing 2) initializes COM. Of course, the application does not do anything beyond initialization, but it does ensure that COM is successfully started.

```
#include <stdafx.h>
#include <sapi.h>

int main(int argc, char* argv[])
{
    if (FAILED(::CoInitialize(NULL)))
        return FALSE;

    ::CoUninitialize();
    return TRUE;
}
```

Code Listing 2

Step 3: Setting up voices

Once COM is running, the next step is to create the voice. A voice is simply a COM object. Additionally, SAPI uses intelligent defaults. During initialization of the object, SAPI assigns most values automatically so that the object may be used immediately afterward. This represents an important improvement from earlier versions. The defaults are retrieved from Speech properties in Control Panel and include such information as the voice (if more than one is available on your system), and the language (English, Japanese, etc.). While some defaults are obvious, others are not (speaking rate, pitch, etc.). Nevertheless, all defaults may be changed either programmatically or in Speech properties in Control Panel.

Setting the *pVoice* pointer to NULL is not required but is useful for checking errors; this ensures an invalid pointer is not reused, or as a reminder that the pointer has already been allocated or deallocated

```
#include <stdafx.h>
#include <sapi.h>
```

```
int main(int argc, char* argv[])
{
```

```
    ISpVoice * pVoice = NULL;
```

```
    if (FAILED(::CoInitialize(NULL)))
        return FALSE;
```

```
    HRESULT hr = CoCreateInstance(CLSID_SpVoice, NULL, CLSCTX_SVC, IID_ISpVoice, (void**)&pVoice);
    if( SUCCEEDED( hr ) )
    {
        pVoice->Release();
        pVoice = NULL;
    }
```

```
    }  
  
    ::CoUninitialize();  
    return TRUE;  
}
```

Code Listing 3. Bold text represents new code for this example.

Step 4: Speak!

The actual speaking of the phrase is an equally simple task: one line calling the Speak function. When the instance of the voice is no longer needed, you can release the object.

```
#include <stdafx.h>
#include <sapi.h>

int main(int argc, char* argv[])
{
    ISpVoice * pVoice = NULL;

    if (FAILED(::CoInitialize(NULL)))
        return FALSE;

    HRESULT hr = CoCreateInstance(CLSID_SpVoice, NULL, CLSCTX_INPROC_SERVER,
    if( SUCCEEDED( hr ) )
    {
        hr = pVoice->Speak(L"Hello world", 0, NULL);
        pVoice->Release();
        pVoice = NULL;
    }

    ::CoUninitialize();
    return TRUE;
}
```

Code Listing 4. Bold text represents new code for this example.

Step 5: Modifying Speech

Voices may be modified using a variety of methods. The most direct way is to apply XML commands directly to the stream. The commands are outlined in [XML Schema](#). In this case, a relative rating of 10 will lower the pitch of the voice.

```
#include <stdafx.h>
#include <sapi.h>

int main(int argc, char* argv[])
{
    ISpVoice * pVoice = NULL;

    if (FAILED(::CoInitialize(NULL)))
        return FALSE;

    HRESULT hr = CoCreateInstance(CLSID_SpVoice, NULL, CLSCTX_
    if( SUCCEEDED( hr ) )
    {
        hr = pVoice->Speak(L"Hello world", 0, NULL);

        // Change pitch
        hr = pVoice->Speak(L "This sounds normal <pitch middle = '-
        pVoice->Release();
        pVoice = NULL;
    }
    ::CoUninitialize();
    return TRUE;
}
```

Code Listing 5. Bold text represents new code for this example.

This is the complete code sample.



Using Events with TTS

This tutorial covers a basic text-to-speech example but uses a Windows application with a graphical interface.

Setting up the project

Create the project

The code is generated from Visual C++ 6.0 and uses the "Hello, World" example. To make the sample base, create a new project as a Windows 32 application and call it "Test." In the subsequent wizard, select "a typical 'Hello World!' application." The resulting project is lengthier than the command line version. Most of the new complexity has little to do with SAPI however, since graphical interfaces require more code to function.

Set SAPI paths

The SAPI paths need to be declared. Add Sapi.h to the path:

1. On the **File** menu, select **Tools**, and then click **Options**.
2. Click the **Directories** tab.
3. Select the **Include Files** drop-down menu.
4. Add the path by clicking in the first unused line in the paths list and enter
"C:\Program Files\Microsoft Speech SDK 5.1\Include".
5. Add a path to the SAPI library file by selecting the Library Files drop-down menu and adding "C:\Program Files\Microsoft Speech SDK 5.1\Lib\i386". Click **OK**.

Create speak menu item

To be able to speak on demand, one modification is required; this is a mechanism to initiate speech. To use the current example in Visual C++, the user should add a File menu item called Speak with a resource ID of IDM_SPEAK. The code handling the event from this menu item will be addressed later in this example. Compile and run the application to make sure

everything works. The application does not display anything other than "Hello, World" along the top of the screen. Even so, it's a good start.

Using the sample

This sample is not a practical one since it speaks only one sentence. The sentence is hard coded, something few applications would do in a practical situation. A more complete or robust application would retrieve the text from a dialog box, resource, or file. However, the sample does represent the foundation of text-to-speech and showcases many of those mechanisms.

More importantly, it demonstrates the interaction between SAPI and the application. Text-to-speech would be marginally useful if that is all it did. However, using this interaction, the application determines words being spoken. In two separate examples using this information, the application displays the words on the screen and highlights them in real time. In doing so, the application also demonstrates the eventing model for SAPI. This includes a brief explanation about speech messages and a related feature, interests. Interests are unique to SAPI.

Furthermore, the interaction is not limited to determining words spoken. A multitude of activities involving SAPI or speech engines could interest the application. [SPEVENTENUM](#) lists these possible activities. For instance, if your application is animating a character for speech, you would be interested each time a new viseme is encountered. The viseme essentially represents a change in the mouth position during speech. Accordingly, the character's mouth would move, or even close. In the same way, starting and stopping of the speech audio stream could interest the application. In general, these activities are called interests.

[Step 1: Initialize COM](#)

[Step 2: Setting up voices](#)

[Step 3: Speak!](#)

[Step 4: Setting events](#)

Step 5: Determining events

Step 6: Reacting to events

Step 1: Initialize COM

As with any SAPI application, COM must be successfully initialized. This is done in a simple manner illustrated below in a snippet from WinMain(). The only restriction is that COM must be available before any SAPI-specific code is implemented and it must be active during the time SAPI is used. Since SAPI is implemented in InitInstance(), the COM statements come before InitInstance() and after the event loop, essentially enclosing the entire initialization and message loop.

```
if( FAILED( CoInitialize(NULL) ) )
{
    return FALSE;
}
```

```
// Perform application initialization:
if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}
```

```
hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_GUIA
```

```
// Main message loop:
while (GetMessage(&msg;, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg;))
    {
        TranslateMessage(&msg;);
        DispatchMessage(&msg;);
    }
}
```

CoUninitialize();

Code Listing 1. Bold text represents new code for this example.

Step 2: Setting up voices

Once COM is running, the next step is to create the voice. Simply declare the instance and use `CoCreateInstance()`. As mentioned in the command line example, SAPI uses intelligent defaults. This requires a minimal amount of initialization and you can use the voice immediately. The defaults are located in Speech properties in Control Panel and include a selection of voices (if more than one is available on your system), and languages (English, Japanese, etc.). While some defaults are obvious, others are not (speaking rate, pitch, etc.). Nevertheless, you can change all defaults either through Speech properties or programmatically.

This example makes several exceptions for the sake of brevity and convenience. First, it uses `InitInstance()` to initialize the voice. `InitInstance()` is the least intrusive call to be placed for this demonstration. Applications, especially those using speech recognition (SR) instances, may have their own procedures explicitly for this so that the speech code is more isolated. Second, the voice is defined globally. Depending on your application's design and requirements, you may not need a global declaration. Third, the instance is immediately released and the memory freed. Obviously, if the voice is to be used, it cannot be released beforehand. In fact, even this application is not going to keep those statements for long. And last, if the initialization fails, this application stops. A more robust application would check errors more extensively and report more detailed information.

ISpVoice *pVoice; //SAPI voice

·
·
·

`BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)`

```

{
    HWND hWnd;

    hInst = hInstance; // Store instance handle in our global variable

    hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPED
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hI

    if (!hWnd)
    {
        return FALSE;
    }

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    //Initialize SAPI
HRESULT hr = CoCreateInstance(CLSID_SpVoice, NULL, CLS
if( SUCCEEDED( hr ) )
{
    pVoice->Release();
    pVoice = NULL;
}
else
    return FALSE;

    return TRUE;
}

```

Code Listing 2. Bold text represents new code for this example.

Step 3: Speak!

Fortunately, the most interesting part of the task is also the simplest. Speaking a sentence involves calling one line. The text to be spoken is provided as a parameter. The source of that text depends on the application. As mentioned previously, the string is usually from a dialog box or a file. Alternatively, the string can also be from a stream but that is handled by another call, [ISpVoice::SpeakStream](#). This example uses a simple, hard-coded sentence. While `::Speak` could have used an inline string such as:

```
Speak( L"I am glad to speak.", SPF_ASYNC, NULL);
```

The string will be used several times during the application. The application retrieves each word and parses it accordingly. For that reason, it is copied to a global string before being used.

The code is placed inside the window messaging area within `WndProc()`. Selecting the Speak from the File menu will produce the following message: "I am glad to speak."

```
WCHAR theString[30];
```

```
.  
. .  
.
```

```
case IDM_SPEAK:
```

```
    wcscpy( theString, L"I am glad to speak." );
```

```
    pVoice->Speak( theString, SPF_ASYNC, NULL);
```

```
    break;
```

Code Listing 3. Bold text represents new code for this example.

Step 4: Setting events

Like most Windows applications, there are interactions among the components and messages are sent to indicate these. SAPI is no different. As information is processed by either the TTS or SR engine, certain activities are initiated or completed. Many times these activities by SAPI or SAPI engines are of interest to the application. For example, the application could be informed when a recognition process is started, so that the user can subsequently be informed. Likewise, the application may be interested in knowing when there is no more information to process, perhaps to inform the user of this condition, or even to shut down either the engine or application itself when it is safe to do so.

An application processes the information of these activities in a two step operation. First, it receives a general message from SAPI or a SAPI engine. This message is similar to other messages, such window events, mouse clicks or a myriad of other messages used by the operating system. Since the message is not defined by the operating system, the application must define it. However, all activities from SAPI use the same message. To determine the exact activity taking place, additional information is provided by SAPI and is called an interest. A complete list of interests is found in [SPEVENTENUM](#).

The second step comes after trapping the message. The application examines an event structure completed by SAPI and retrieves the relevant information.

Setting interests

During initialization, SAPI can be informed of which interests to pass back to the application. This is done using [iSpEventSource::SetInterest](#). By default, TTS does not set any interests and SR uses only recognition (SPEI_RECOGNITION). That is, if the SetInterest call were omitted entirely, TTS would

not pass back any interest information to the application and SR would report only successful recognitions. Values can be combined with logical OR statements. Using this combination, two or more interests can be specifically set, while excluding others at the same time. Using the first parameter, the application can be notified when a specific interest occurs. The second parameter queues the interest for later retrieval. For the moment, keep the two parameters of SetInterest identical since the application will need to store information later. Interests can be changed at anytime in the application as the user's requirements change.

Setting messages

Regardless of the interests set, the application has to associate a message to SAPI. This is done with [ISpNotifySource::SetNotifyWindowMessage](#). If this call is not included, no message could be sent back to the application. There are three types of message notifications and at least one must be included to receive messages. A fourth type is for multithreaded applications and is not used here. All four are explained in the [ISpNotifySource](#) interface section. The actual message name and ID is determined by the application. This example uses the standard WM_USER for private messages.

```
//Initialize SAPI
HRESULT hr = CoCreateInstance(CLSID_SpVoice, NULL, CLSCTX_INPROC_SERVER, IID_ISpVoice, (void**)&pVoice);
if( SUCCEEDED( hr ) )
{
    pVoice->SetInterest( SPFEI(SPEI_WORD_BOUNDARY),SPFEI(SPEI_SENTENCE_BOUNDARY));
    pVoice->SetNotifyWindowMessage( hWnd, WM_USER, 0, 0 );
}
else
    return FALSE;
```

Code Listing 4. Bold text represents new code for this example.

Step 5: Determining events

As mentioned previously, working with events is a two step process. The first is a simple and standard approach to Windows events. A message (however generated) is sent back to the application and the message loop dispatches it accordingly. In this example, `WndProc()` receives the `WM_USER` message. Once the message is trapped, the rest relies on SAPI.

The second step is to determine which interest occurred. Since the `SetInterest` method responds only to `SPEI_WORD_BOUNDARY`, it is likely that it is an `SPEI_WORD_BOUNDARY` interest. However, in larger applications or if several interests were set, the application must be able to determine the exact one. SAPI determines this using the event structure, [SPEVENT](#) and the [GetEvents](#) method. Used together, you can retrieve specific information about the SAPI event, including the type of interest. This value in member `eEventId` coincides with parameters used by `SetInterest`. The `SPEVENT` structure must be initialized before first use and cleared before reuse. It is possible for information to persist from call to call. The helper function [SpClearEvent](#) clears the event.

It is possible for events and interests to occur faster than the application can process them. This is a common situation especially if a viseme interest is set, because it generates an event for each sound encountered. `GetEvents` can retrieve more than one event at time. This allows for batch processing of events should a more specialized application need to do so. Another way to handle this situation is to use a while loop. This retrieves each event one at a time. Regardless of the design, once a valid `SPEVENT` is available, the application has only to compare the interest type from `eEventId` with an action. Again for simplicity, a switch statement filters interests and subsequent code completes the action.

case WM_USER:

```
SPEVENT eventItem;  
memset( &eventItem;, 0,sizeof(SPEVENT));  
while( pVoice->GetEvents(1, &eventItem;, NULL ) == S_OK  
{  
    switch(eventItem.eEventId )  
    {  
        case SPEI_WORD_BOUNDARY :  
            .  
            .  
            .  
            break;  
  
        default:  
            break;  
    }  
  
    SpClearEvent( eventItem );
```

Code Listing 5. Bold text represents new code for this example.

Step 6: Reacting to events

Once the event and interest is determined, the programming becomes more standard. How an actual interest is handled is the application's own design and implementation. In this example, the application identifies individual words using the SPEI_WORD_BOUNDARY interest. Whenever this interest is returned, the SAPI engine has found a distinct word, usually offset by white spaces or certain punctuation. Also in this case, relevant information is passed back from a [Voice::GetStatus](#) call using [SPVOICESTATUS](#) structure.

The individual words are noted as offsets from the complete string, marking the positions of the first letter and last letters of the sequence. For demonstration, the words are then displayed in a Win32 message box on the screen. One subtlety to notice is that each word is displayed as soon as possible. That is, the screen is updated during the actual speaking of the text. This characteristic is controlled during by the SPF_ASYNC flag of the Voice::Speak method:

```
pVoice->Speak( theString, SPF_ASYNC, NULL);
```

The alternative is to wait until all the speech is complete and then process the events and interests. For example, if the second parameter was replaced with NULL, the message boxes would still display but would wait until the speaking is complete. The difference in timing may be important to applications depending on needs.

```
case SPEI_WORD_BOUNDARY :  
    SPVOICESTATUS eventStatus;  
    pVoice->GetStatus( &eventStatus;, NULL );  
  
    ULONG start, end;  
    start = eventStatus.ulInputWordPos;
```

```
end = eventStatus.ulInputWordLen;  
wcsncpy( tempString, theString + start , end );  
tempString[ end ] = '\0';  
  
MessageBoxW( hWnd, tempString, L"GUIApp", MB_C  
break;
```

Code Listing 6. Bold text represents new code for this example.

Complete code listing

```
// GUIApp.cpp : Defines the entry point for the application.
#include "stdafx.h"
#include <sapi.h>
#include "string.h"
#include "resource.h"

#include "sphelper.h"

#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst;                // current instance
TCHAR szTitle[MAX_LOADSTRING] = _T("Speak Hello world Ap
TCHAR szWindowClass[MAX_LOADSTRING] = _T("SpeakWinCl

//For SAPI
WCHAR theString[30];
ISpVoice *pVoice;              //SAPI voice

// Forward declarations of functions included in this code module:
ATOM      MyRegisterClass(HINSTANCE hInstance);
BOOL      InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow)
{
```

```

    // TODO: Place code here.
    MSG msg;
// HACCEL hAccelTable;

    // Initialize global strings
// LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRT
//LoadString(hInstance, IDC_GUIAPP, szWindowClass, MAX_LOA
MyRegisterClass(hInstance);

if( FAILED( CoInitialize(NULL) ) )
{
    return FALSE;
}

// Perform application initialization:
if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}

//hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_GUL

// Main message loop:
while (GetMessage(&msg;, NULL, 0, 0))
{
    //if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg;))
    {
        TranslateMessage(&msg;);
        DispatchMessage(&msg;);
    }
}

CoUninitialize();

```

```

    return msg.wParam;
}

//
// FUNCTION: MyRegisterClass()
//
// PURPOSE: Registers the window class.
//
// COMMENTS:
//
// This function and its usage is only necessary if you want this code
// to be compatible with Win32 systems prior to the 'RegisterClassEx
// function that was added to Windows 95. It is important to call this
// so that the application will get 'well formed' small icons associated
// with it.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEX wcex;

    wcex.cbSize = sizeof(WNDCLASSEX);

    wcex.style      = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc = (WNDPROC)WndProc;
    wcex.cbClsExtra  = 0;
    wcex.cbWndExtra  = 0;
    wcex.hInstance   = hInstance;
    wcex.hIcon       = NULL; //LoadIcon(hInstance, (LPCTSTR)IDI_G
    wcex.hCursor     = NULL; //LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);

```

```

wcecx.lpszMenuName = NULL; //MAKEINTRESOURCE(IDC_TE
wcecx.lpszClassName = szWindowClass;
wcecx.hIconSm = NULL; //LoadIcon(wcecx.hInstance, (LPCTSTR

return RegisterClassEx(&wcecx);
}

//
// FUNCTION: InitInstance(HANDLE, int)
//
// PURPOSE: Saves instance handle and creates main window
//
// COMMENTS:
//
// In this function, we save the instance handle in a global variable
// create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;

    hInst = hInstance; // Store instance handle in our global variable

    hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPE
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hI

    if (!hWnd)
    {
        return FALSE;
    }

    // Instead of using IDC_TEST, use the identifier of menu resource
    // of the current application.

```

```

SetMenu(hWnd, LoadMenu(hInstance, MAKEINTRESOURCE(IDC

ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);

//Initialize SAPI
HRESULT hr = CoCreateInstance(CLSID_SpVoice, NULL, CLS
if( SUCCEEDED( hr ) )
{
    pVoice->SetInterest( SPFEI(SPEI_WORD_BOUNDARY),SPF
    pVoice->SetNotifyWindowMessage( hWnd, WM_USER, 0, 0 );
}
    else
        return FALSE;

return TRUE;
}

//
// FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
//
// PURPOSE: Processes messages for the main window.
//
// WM_COMMAND - process the application menu
// WM_PAINT - Paint the main window
// WM_DESTROY - post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WI
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

```

```

TCHAR szHello[MAX_LOADSTRING];
LoadString(hInst, IDS_HELLO, szHello, MAX_LOADSTRING);
WCHAR tempString[30];

switch (message)
{
case WM_COMMAND:
    wmId  = LOWORD(wParam);
    wmEvent = HIWORD(wParam);
    // Parse the menu selections:
    switch (wmId)
    {
case IDM_ABOUT:
        DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd, (D
        break;

case IDM_EXIT:
        DestroyWindow(hWnd);
        break;

case IDM_SPEAK:
        wcscpy( theString, L"I am glad to speak." );
        pVoice->Speak( theString, SPF_ASYNC, NULL);
        break;

default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    break;

case WM_USER:
    SPEVENT eventItem;
    memset( &eventItem, 0, sizeof(SPEVENT));

```

```

while( pVoice->GetEvents(1, &eventItem;, NULL ) == S_OK
{
switch(eventItem.eEventId )
{
case SPEI_WORD_BOUNDARY :
    SPVOICESTATUS eventStatus;
    pVoice->GetStatus( &eventStatus;, NULL );

    ULONG start, end;
    start = eventStatus.ulInputWordPos;
    end = eventStatus.ulInputWordLen;
    wcsncpy( tempString, theString + start , end );
    tempString[ end ] = '\0';

    MessageBoxW( hWnd, tempString, L"GUIApp", MB_C
break;

default:
    break;
}

SpClearEvent( &eventItem; );
}
break;

case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps;);
    // TODO: Add any drawing code here...
    RECT rt;
    GetClientRect(hWnd, &rt;);
    DrawText(hdc, szHello, strlen(szHello), &rt;, DT_CENTER);
    EndPaint(hWnd, &ps;);
    break;

```

```

case WM_DESTROY:
    if (pVoice)
    {
        pVoice->Release();
        pVoice = NULL;
    }
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, message, wParam, lParam);
}

return 0;
}

// Message handler for about box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM
{
    switch (message)
    {
        case WM_INITDIALOG:
            return TRUE;

        case WM_COMMAND:
            if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDC
            {
                EndDialog(hDlg, LOWORD(wParam));
                return TRUE;
            }
            break;
    }
}

```

```
    return FALSE;  
}  
  
}
```

Complete code listing. Lines in bold are SAPI-related.



TTS Events Explanation

Events are structures that pass information from the TTS engine back to the application. When the audio data is output, SAPI fires corresponding events. Applications react to audio output as it occurs. Examples of reactions include animating a face appropriately as viseme events are received, or highlighting text as it is spoken. See the sample application, TTSApp, for an example of each.

Applications call [ISpEventSource::SetInterest](#) to inform SAPI about the types of events that they are interested in receiving. Applications can also call this through [ISpVoice](#), because it inherits from [ISpEventSource](#). Applications can then call [ISpEventSource::GetEvents](#) to retrieve fired events from SAPI.

The following is a set of event types generated by TTS engines (this is a subset of the [SPEVENTENUM](#) enumeration):

```
typedef enum SPEVENTENUM
{
    //--- TTS engine
    SPEI_START_INPUT_STREAM      = 1,
    SPEI_END_INPUT_STREAM        = 2,
    SPEI_VOICE_CHANGE            = 3,    // LPARAM_IS_TOKEN
    SPEI_TTS_BOOKMARK            = 4,    // LPARAM_IS_STRING
    SPEI_WORD_BOUNDARY           = 5,
    SPEI_PHONEME                 = 6,
    SPEI_SENTENCE_BOUNDARY       = 7,
    SPEI_VISEME                  = 8,
    SPEI_TTS_AUDIO_LEVEL         = 9
} SPEVENTENUM;
```

The [SPEVENT](#) structure contains varying information depending on which of these event types it represents.

```
typedef struct SPEVENT
{
    WORD          eEventId;
```

```

WORD           eParamType;
ULONG          ulStreamNum;
ULONGLONG      ullAudioStreamOffset;
WPARAM         wParam;
LPARAM         lParam;
} SPEVENT;

```

You can analyze the various fields of the SPEVENT structure for the event types they correspond to. For all event types, *ulStreamNum* corresponds to the stream number returned using [ISpVoice::Speak](#) or [ISpVoice::SpeakStream](#).

The SPEI_START_INPUT_STREAM event indicates that the output object has begun receiving output for a specific stream number. The rest of the fields are not of interest to this event type.

The SPEI_END_INPUT_STREAM event indicates that the output object has finished receiving output for a specific stream number. The rest of the fields are not of interest to this event type.

The SPEI_VOICE_CHANGE event indicates that the voice responsible for speaking the input text (or stream) has changed because of a <Voice> XML tag. It is fired at the beginning of each Speak call. For more information on using object tokens, see the [Object Tokens and Registry Settings](#) white paper.

SPEVENT Field	Voice Change event
<i>eEventId</i>	SPEI_VOICE_CHANGE
<i>eParamType</i>	SPET_LPARAM_IS_TOKEN
<i>wParam</i>	
<i>lParam</i>	Object token of the new voice.

The SPEI_TTS_BOOKMARK event indicates that the speak stream has reached a bookmark. Bookmarks can be inserted into the input text using the <Bookmark> XML tag.

SPEVENT Field	Bookmark event
<i>eEventId</i>	SPEI_TTS_BOOKMARK
<i>eiParamType</i>	SPET_LPARAM_IS_STRING
<i>wParam</i>	Value of the bookmark string when converted to a long (_wtol(...)) can be used).
<i>lParam</i>	Null-terminated copy of the bookmark string.

The SPEI_WORD_BOUNDARY event indicates that it has reached the beginning of a word.

SPEVENT Field	Word Boundary event
<i>eEventId</i>	SPEI_WORD_BOUNDARY
<i>eiParamType</i>	SPET_LPARAM_IS_UNKNOWN
<i>wParam</i>	Character offset at the beginning of the word being synthesized.
<i>lParam</i>	Character length of the word in the current input stream being synthesized.

The SPEI_SENTENCE_BOUNDARY event indicates that the speak stream has reached the beginning of a sentence.

SPEVENT Field	Sentence Boundary event
<i>eEventId</i>	SPEI_SENTENCE_BOUNDARY
<i>elParamType</i>	SPET_LPARAM_IS_UNKNOWN
<i>wParam</i>	Character offset at the beginning of the sentence being synthesized.
<i>lParam</i>	Character length of the sentence in the current input stream being synthesized.

The SPEI_PHONEME event indicates that the speak stream has reached the phoneme.

SPEVENT Field	Phoneme event
<i>eEventId</i>	SPEI_PHONEME
<i>elParamType</i>	SPET_LPARAM_IS_UNKNOWN
<i>wParam</i>	The high word is the duration, in milliseconds, of the current phoneme. The low word is the PhoneID of the next phoneme.
<i>lParam</i>	The low word is the PhoneID of the current phoneme.

The high word is the SPVFEATURE value associated with the current phoneme.

The SAPI 5 American English phoneme set can be found [here](#). The SAPI 5 Chinese phoneme set can be found [here](#). The SAPI 5 Japanese phoneme set can be found [here](#).

SPVFEATURE contains two flags: SPVFEATURE_STRESSED and SPVFEATURE_EMPHASIS. SPVFEATURE_STRESSED means that the phoneme is stressed relative to the other phonemes of a word (stress is usually associated with the vowel of a stressed syllable). SPVFEATURE_EMPHASIS means that the phoneme is part of an emphasized word. That is, stress is a syllabic phenomenon within a word, and emphasis is a word-level phenomenon within a sentence.

The SPEI_VISEME event indicates that it has reached the viseme.

SPEVENT Field	Viseme event
<i>eEventId</i>	SPEI_VISEME
<i>eiParamType</i>	SPET_LPARAM_IS_UNKNOWN
<i>wParam</i>	The high word is the duration, in milliseconds, of the current viseme. The low word is the code for the next viseme.
<i>lParam</i>	The low word is the code of the current viseme. The high word is the SPVFEATURE value

associated with the current viseme (and phoneme).

See [SPVISEMES](#) for a listing of the SAPI 5 viseme set.

The SPEI_TTS_AUDIO_LEVEL event indicates the audio has reached the level of the synthesis at any given point.

SPEVENT Field	Audio Level event
<i>eEventId</i>	SPEI_TTS_AUDIO_LEVEL
<i>elParamType</i>	SPET_LPARAM_IS_UNDEFINED
<i>wParam</i>	TTS audio level (ULONG).
<i>lParam</i>	NULL

For an example of how to use TTS events in an application, see the [Text-to-Speech Tutorial](#).



White Papers

The following items are covered in this section:

- [SR Properties White Paper](#)
- [TTS Engine Vendor Porting Guide White Paper](#)
- [SR Engine Vendor Porting Guide White Paper](#)
- [Object Tokens and Registry Settings White Paper](#)
- [VendorPreferred Attribute](#)
- [Simple TTS Guide - Speak to a File and Speak a File](#)
- [SAPI 5.1 64-bit Issues](#)
- [Speech Telephony Application Guide](#)
- [Using Sample Audio Object \(SpAudioPlug\)](#)
- [Audio Object](#)
- [Compliance Tests White Paper](#)
- [Microsoft Speech SDK Setup 5.1](#)
- [XML Schema : Grammar](#)
- [XML Schema : SAPI](#)
- [XML TTS Tutorial](#)
- [Text Normalization](#)
- [Using Microsoft Foundation Class \(MFC\) to Automate SAPI](#)
- [Persisting Recognized Wav Audio from the Speech Recognition Engine](#)
- [Using Wav File Input with the Speech Recognition Engine](#)



SAPI 5.0 SR Properties White Paper

Introduction

This document describes the ISpProperties elements for SAPI 5 compliant SR engines. This spec will serve to define these attributes only for SR engines. Application developers hoping to build a SAPI 5 compliant engine should reference this document. For more information, developers should refer to the SAPI SDK help documents.

ISpProperties

ISpProperties is an interface that enables the SR and TTS engines to get or set various attributes for an object. The attributes are passed to the engine via the ISpProperties interface. ISpProperties are identified by a unique LONG value. SAPI defines certain attributes known as system attributes. The range of these attributes is from 0x0001 to **0xffff**. Vendor ISpProperties attributes are defined by a unique high word value (two ANSI Characters that identify the engine vendor).

Attributes may be LONGs, strings, or memory addresses.

SR Properties

The following table lists the SR properties that are set by the application and passed to the SR engine via SAPI. These attributes are not required for SAPI compliance. However, the ranges accompanied by the attributes are required values and the exact interpretation of the values is left to the SR engine. The different implementation is defined by each property. The SAPI ranges and defaults for each property are also shown.

NOTE: The attributes are associated with a user profile and written in the registry by SAPI. SAPI detects the correct settings. The application should not write attribute changes to the registry.

<i>dwAttrib Value</i>	<i>WCHAR Value</i>	<i>Meaning</i>
SPPROP_RESOURCE_USAGE	ResourceUsage	The ResourceUsage specifies the engine CPU consumption. As the resource usage increases, so does the required CPU power.
SPPROP_HIGH_CONFIDENCE_THRESHOLD	HighConfidenceThreshold	The threshold values are used to divide a confidence scale into four portions: rejected, low, medium, and high. The location of the low confidence, normal confidence, and high confidence markers control how the confidence of a word is labeled. The HighConfidenceThreshold (HCT) separates the high and medium confidence range. The NormalConfidenceThreshold (NCT) separates the medium and the low confidence thresholds. The LowConfidenceThreshold (LCT) separates the low and rejected confidence range.
SPPROP_NORMAL_CONFIDENCE_THRESHOLD	NormalConfidenceThreshold	
SPPROP_LOW_CONFIDENCE_THRESHOLD	LowConfidenceThreshold	
		If the all three confidences are equal to 0, then all wor

		will have high confidence. all three confidences are equal to 100, then all words will have low confidence.
SPPROP_RESPONSE_SPEED	ResponseSpeed	This indicates the amount of silence the engine looks for before completing a recognition. This attribute is used when the recognition is not ambiguous. For example, in the case of a context-free grammar (CFG) which has two sentences: 1) new game please and 2) new game, a non-ambiguous recognition would be "new game please."
SPPROP_COMPLEX_RESPONSE_SPEED	ComplexResponseSpeed	This indicates the amount of silence that the engine will look for before completing recognition. This attribute is used when the recognition is ambiguous. For example, in the case of a CFG which has two sentences: 1) new game please and 2) new game, an ambiguous recognition would be "new game." This property's value must be greater than the ResponseSpeed value.
SPPROP_ADAPTATION_ON	AdaptationOn	Indicates whether the recognition engine should adapt the acoustic model.



TTS Engine Vendor Porting Guide

Table of Contents

[Overview of SAPI 5.0 Architecture](#)

[SAPI Objects and Interfaces](#)

[Creating and Initializing the Engine - ISpObjectWithToken](#)

[Receiving Calls from SAPI - ISpTTSEngine](#)

[GetOutputFormat](#)

[Speak](#)

[Fragment List Example](#)

[Writing Data Back to SAPI - ISpTTSEngineSite](#)

[Getting Real-Time Action Requests](#)

[Volume](#)

[Rate](#)

[Skip](#)

[Queuing Events](#)

[Bookmarks](#)

[Word Boundaries](#)

[Sentence Boundaries](#)

[Phonemes](#)

[Visemes](#)

[Queuing Audio Data](#)

[Creating an Engine Properties UI - ISpTokenUI](#)

[Using SAPI Lexicons](#)

[Appendix A - SAPI 5 Phonemes](#)

[Appendix B - SAPI 5 Visemes](#)

Overview of SAPI 5.0 Architecture

The Microsoft Speech API (SAPI) is a layer of software which sits between applications and speech engines, allowing them to communicate in a standardized way. One of its main goals is enabling application developers to use speech technology in a simple and straightforward way. Another goal is solving some of the more basic complications of developing speech engines, such as audio device manipulation and threading issues, thus allowing engine developers to focus on speech.

From an engine vendor's point of view, there are a number of technical advantages to using SAPI 5 over SAPI 4:

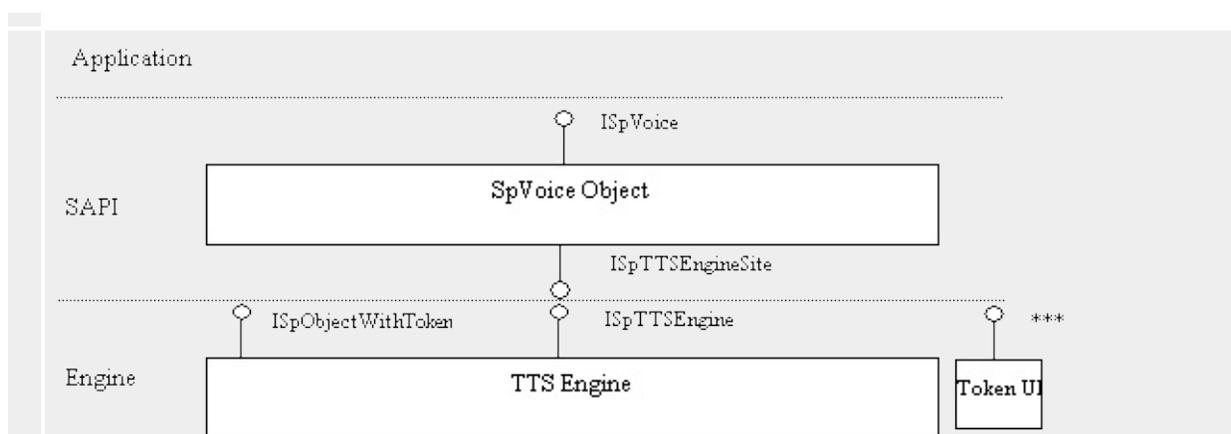
- § The SAPI 5 DDI has been greatly simplified.
- § SAPI 5 can handle all audio format conversion for the TTS engine.
- § SAPI 5 parses SAPI 5 XML for the TTS engine. Engine proprietary tags are passed to the engine untouched, allowing the engine to interpret them.
- § SAPI 5 performs parameter validation for the engine.
- § SAPI 5 has lexicon management features.

SAPI Objects and Interfaces

There are two main objects of interest to a TTS Engine developer: the SpVoice object (SAPI) and the TTS Engine object (refer to figure 2). The third object in the figure is a UI component which an engine may or may not implement.

The SpVoice object implements two interfaces which we will be concerned with - [ISpVoice](#), which is the interface which the application uses to access TTS functionality, and [ISpTTSEngineSite](#), which the engine uses to write audio data and queue events. The TTS Engine must implement two interfaces as well - [ISpTTSEngine](#), which is the interface through which SAPI will call the engine, and [ISpObjectWithToken](#), which is the interface through which SAPI will create and initialize the engine. The UI object, if it exists, must implement [ISpTokenUI](#), through which it will be accessed by the SAPI control panel (or, potentially, other applications).

For the most part this document is not concerned with ISpVoice, and so it won't be covered in any detail. Each of the other interfaces, however, will be discussed in depth.

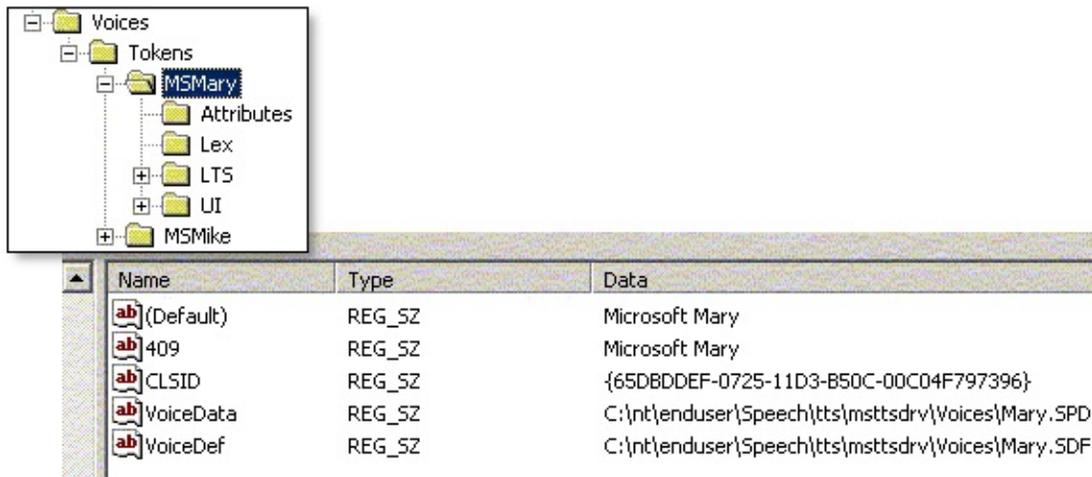


Creating and Initializing the Engine - ISpObjectWithToken

One important thing to realize about the SAPI 5 architecture is that while SAPI knows about TTS Engines, applications only know about TTS *voices*. The difference between these two is fairly obvious - one engine implementation can potentially support any number of different voices, with the only differences being data files, parameters, etc. What this means at the engine level is that an engine will be created *by* one of its voices, in a certain sense.

SAPI 5 uses tokens to represent resources available on a computer (see the [Object Tokens and Registry Settings White Paper](#) for more details), including TTS voices. These tokens contain the CLSID of the objects they represent, as well as various attributes of those objects. When an application wishes to use a TTS voice, SAPI will get that voice's token from the registry. Through the voice token, an engine will be cocreated using its CLSID. The SpVoice object then queries the engine for the ISpObjectWithToken interface, through which it calls SetObjectToken.

Here is an example of what a voice token might look like in the registry (voices are located under HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Speech\Voices\To



The SetObjectToken call gives the TTS Engine a pointer to the token (and thus the voice) from which it was created, which gives the Engine a chance to initialize itself based on information stored in the token. In the example token above, the VoiceData and VoiceDef keys in the token allow the TTS engine to load the appropriate voice data, once it has a pointer to the token. Similarly, the Lex and LTS subkeys allow the TTS engine to load the appropriate lexicon and letter-to-sound rules.

Again, for more details on registering a TTS engine, see the [Object Tokens and Registry Settings White Paper](#).

Receiving Calls from SAPI - ISpTTSEngine

Once an engine has been created SAPI will begin calling the engine using ISpTTSEngine. ISpTTSEngine has only two methods - GetOutputFormat and Speak.

GetOutputFormat is used to query the engine about a specific output format - the engine should examine the desired output format and return to the SpVoice object the closest format which it supports. This function may potentially be called many times during the life of the engine.

```
HRESULT GetOutputFormat(  
    [in] const GUID * pTargetFmtId,  
    [in] const WAVEFORMATEX * pTargetWaveFormatEx,  
    [out] GUID * pOutputFormatId,  
    [out] WAVEFORMATEX **  
    ppCoMemOutputWaveFormatEx  
);
```

In the normal case, pTargetFmtId will be SPDFID_WaveFormatEx, and pTargetWaveFormatEx will be a pointer to a WAVEFORMATEX structure describing the desired output format. In this case, the engine should set pOutputFormatId to SPDFID_WaveFormatEx, allocate space (using ppCoMemOutputWaveFormatEx) for a WAVEFORMATEX structure, and set it to the closest format to pTargetWaveFormatEx it supports.

If `pTargetFmtId` is `NULL`, the engine should simply return to SAPI its default format.

NOTE: If `pTargetFmtId` is `SPDFID_Text`, engines can do whatever they please. Essentially, this format type is provided for debugging purposes – it is *not* required that any engine support this for SAPI 5.0 compliance, nor is it required that engines do anything specific with this format if they do support it.

See the Sample TTS Engine's `GetOutputFormat` implementation for more details.

`Speak` is the main function of the interface – it passes the engine the text to be rendered, an output format to render it in, and an output site to which the engine should write audio data and events. A `Speak` call should return when either all of the input text has been rendered, or the engine has been told to abort the call by the `SpVoice` object. Let's look at the parameters in more detail.

```
HRESULT Speak(  
    [in]DWORD dwSpeakFlags,  
    [in]REFGUID rguidFormatId,  
    [in]const WAVEFORMATEX * pWaveFormatEx,  
    [in]const SPVTEXTFRAG* pTextFragList,  
    [in]ISpTTSEngineSite* pOutputSite,  
);
```

The first parameter of the Speak call, dwSpeakFlags, is a DWORD which will have one of two values - 0, or SPF_NLP_SPEAK_PUNC (all other flags in the SPEAKFLAGS enumeration are masked out, since they are handled by SAPI). If the value is SPF_NLP_SPEAK_PUNC, the engine should speak all punctuation (e.g. "This is a sentence." should be expanded to "This is a sentence period").

The second and third parameters of the Speak call will specify the output format which the engine should use for rendering the text passed in for this call. This format is guaranteed to be one which the engine told SAPI it supports using a previous GetOutputFormat call. Again, if this rguidFormatId is SPDFID_Text, it is not required that engines support this format, nor is it required that engines do anything specific with this format if it is supported.

The fourth parameter is the text to be rendered in the form of a linked list of SPVTEXTFRAGs. Let's look at this structure in more detail.

```
typedef struct SPVTEXTFRAG
{
    struct SPVTEXTFRAG *pNext;
    SPVSTATE State;
    LPCWSTR pTextStart;
    ULONG ulTextLen;
    ULONG ulTextSrcOffset;
} SPVTEXTFRAG;
```

pTextStart is a pointer to the beginning of the text associated

with the fragment. `ulTextLen` is the length of this text, in `WCHARs`. `ulTextSrcOffset` is the offset of the first character of the text associated with the fragment. Finally, `State` is the SAPI 5.0 XML state associated with this fragment (see the [XML Schema : SAPI](#) white paper for more details on SAPI 5 XML markup).

```
typedef [restricted] struct SPVSTATE
{
    SPVACTIONS eAction;
    LANGID LangID;
    WORD wReserved;
    long EmphAdj;
    long RateAdj;
    ULONG Volume;
    SPVPITCH PitchAdj;
    ULONG SilenceMSecs;
    SPPHONEID *pPhonelds;
    SPPARTOFSPEECH ePartOfSpeech;
    SPVCONTEXT Context;
} SPVSTATE;
```

`eActions` is an enumerated value which tells the engine what it should do with this fragment.

```
typedef enum SPVACTIONS
{
```

```
    SPVA_Speak = 0,  
    SPVA_Silence,  
    SPVA_Pronounce,  
    SPVA_Bookmark,  
    SPVA_SpellOut,  
    SPVA_Section,  
    SPVA_ParseUnknownTag  
} SPVACTIONS;
```

SPVA_Speak (the default value) means that the engine should process the text associated with the fragment and render it in the proper output format. SPVA_Silence means that SAPI was passed a <Silence> SAPI 5.0 XML tag, and that the engine should write SilenceMSecs (see structure SPVSTATE) milliseconds of silence. SPVA_Pronounce means that SAPI was passed a <Pron> SAPI 5.0 XML tag, and that the engine should use pPhonelds (see structure SPVSTATE) as the pronunciation of the associated text, or just insert the pronunciation if there is no associated text. SPVA_Bookmark means that SAPI was passed a <Bookmark> SAPI 5.0 XML tag, and that the engine should write a Bookmark event (see below for information on writing events). SPVA_SpellOut means that the engine should spell out the associated text letter by letter, including punctuation and miscellaneous characters (and render this expanded version of the text in the proper output format). SPVA_Section is currently unused. SPVA_ParseUnknownTag means that a non-SAPI 5.0 XML tag was passed to SAPI - if the engine supports additional tags, it should attempt to parse this tag. Otherwise, it should just ignore it.

LANGID will be zero, unless a language was specified to SAPI using a <Lang> SAPI 5.0 XML tag.

EmphAdj will be zero, unless SAPI was passed an <Emph> SAPI 5.0 XML tag.

RateAdj will be 0, unless SAPI was passed a <Rate> SAPI 5.0 XML tag. This gives the absolute rate which the engine should use to render the text associated with this fragment. **NOTE:** the engine should combine these values with values obtained through ISpTTSEngineSite::GetRate calls to arrive at a final value.

Volume will be 100, unless SAPI was passed a <Volume> SAPI 5.0 XML tag. This gives the absolute volume which the engine should use to render the text associated with this fragment. **NOTE:** the engine should combine these values with values obtained through ISpTTSEngineSite::GetVolume calls to arrive at a final value.

PitchAdj will have a MiddleAdj of zero and a RangeAdj of zero, unless SAPI was passed a <Pitch> SAPI 5.0 XML tag. This gives the absolute pitch middle and range which the engine should use to render the text associated with this fragment (the pitch middle is used to raise or lower the overall pitch of the voice, the pitch range is used to expand or contract the pitch range of the voice, making it more or less monotone).

```
typedef struct SPVPITCH
{
    long MiddleAdj;
    long RangeAdj;
} SPVPITCH;
```

ePartOfSpeech will be SPPS_Unknown (see SPPARTOFSPEECH) unless SAPI was passed a <PartOfSp> SAPI 5.0 XML tag. This part of speech should be used for the text associated with this fragment (e.g. to disambiguate a word with multiple pronunciations).

Finally, the pointers within Context will be NULL unless SAPI was passed a <Context> SAPI 5.0 XML tag.

```
typedef [restricted] struct SPVCONTEXT
{
    LPCWSTR pCategory;
    LPCWSTR pBefore;
    LPCWSTR pAfter;
} SPVCONTEXT;
```

This field can be used to disambiguate items in the text associated with this fragment (e.g. ambiguous date formats).

Let's look at an example of a fragment list.

Imagine this text is passed to SAPI:

"This is a <PITCH MIDDLE = '6'> sample piece of <PARTOFSPEECH PART = 'Noun'> text </PARTOFSPEECH> which will <BOOKMARK MARK = '1'> demonstrate <VOLUME LEVEL = '30'> what a <VOLUME LEVEL = '90'> fragment </VOLUME> list </VOLUME> looks like </PITCH> conceptually."

This will be the resulting linked list of SPVTEXTFRAGs passed to the TTS Engine:

SPVTEXTFRAGs		Element 1	Element 2
pNext		Element 2	Element 3
State	eAction	SPVA_Speak	SPVA_Speak
	LangId	0	0
	EmphAdj	0	0
	RateAdj	0	0
	Volume	100	100
	PitchAdj	MiddleAdj	0
		RangeAdj	0
	SilenceMSecs	0	0
	pPhonIds	NULL	NULL
	ePartOfSpeech	SPPS_Unknown	SPPS_Unknown
Context	pCategory	NULL	NULL
	pBefore	NULL	NULL
	pAfter	NULL	NULL

pTextStart	"This is a <PITCH ..."	"sample piece of <PART..."
ulTextLen	10	16
ulTextSrcOffset	0	31

SPVTEXTFRAGs	Element 3	Element 4	Element 5
pNext	Element 4	Element 5	Element 6
State eAction	SPVA_Speak	SPVA_Speak	SPVA_Speak
LangId	0	0	0
EmphAdj	0	0	0
RateAdj	0	0	0
Volume	100	100	100
PitchAdj MiddleAdj	6	6	6
RangeAdj	0	0	0
SilenceMSecs	0	0	0
pPhonIds	NULL	NULL	NULL
ePartOfSpeech	SPPS_Noun	SPPS_Unknown	SPPS_Unknown
Context pCategory	NULL	NULL	NULL

	pBefore	NULL	NULL	NULL
	pAfter	NULL	NULL	NULL
	pTextStart	“text </PART...”	“which will <B...”	“1’/> demo
	ulTextLen	5	11	1
	ulTextSrcOffset	72	89	100

SPVTEXTFRAGs		Element 6	Element 7	Element 8
	pNext	Element 7	Element 8	Element 9
State	eAction	SPVA_Speak	SPVA_Speak	SPVA_Speak
	LangId	0	0	0
	EmphAdj	0	0	0
	RateAdj	0	0	0
	Volume	100	30	90
	PitchAdj	MiddleAdj	6	6
		RangeAdj	0	0
	SilenceMSecs	0	0	0
	pPhonIds	NULL	NULL	NULL
	ePartOfSpeech	SPPS_Unknown	SPPS_Unknown	SPPS_Unknown

Context	pCategory	NULL	NULL	NU
	pBefore	NULL	NULL	NU
	pAfter	NULL	NULL	NU
pTextStart		“demonstrate <V...”	“what a <VOL...”	“fra </A
ulTextLen		12	7	9
ulTextSrcOffset		123	157	180

SPVTEXTFRAGs		Element 9	Element 10	Ele
pNext		Element 10	Element 11	Ele
State	eAction	SPVA_Speak	SPVA_Speak	SPV
	LangId	0	0	0
	EmphAdj	0	0	0
	RateAdj	0	0	0
	Volume	30	100	100
	PitchAdj	MiddleAdj	6	0
		RangeAdj	0	0
	SilenceMSecs	0	0	0
	pPhonIds	NULL	NULL	NU

ePartOfSpeech		SPPS_Unknown	SPPS_Unknown	SPI
Context	pCategory	NULL	NULL	NU
	pBefore	NULL	NULL	NU
	pAfter	NULL	NULL	NU
pTextStart		"list </VOL..."	"looks like </PIT..."	"cc
ulTextLen		5	11	14
ulTextSrcOffset		205	220	240

The last parameter of the Speak call is an ISpTTSEngineSite pointer - pOutputSite. This pointer should be stored by the engine, as it will be used to write audio data and events back to the SpVoice object, as well as to poll the SpVoice object for real-time action requests.

Writing Data Back to SAPI - ISpTTSEngineSite

Getting Real-Time Action Requests

Within a Speak call, an Engine should call ISpTTSEngineSite::GetActions as often as possible to ensure near real-time processing of SAPI actions. This is an inexpensive call - it simply returns a DWORD which will contain one or more values from the SPVES_ACTIONS enumeration.

```
DWORD GetActions( void );
```

```
typedef enum SPVES_ACTIONS  
{  
    SPVES_CONTINUE = 0,  
    SPVES_ABORT = ( 1L << 0 ),  
    SPVES_SKIP = ( 1L << 1 ),  
    SPVES_RATE = ( 1L << 2 ),  
    SPVES_VOLUME = ( 1L << 3 )  
} SPVES_ACTIONS;
```

SPVES_CONTINUE is the default case (no actions) - it means to continue processing normally. SPVES_ABORT means that the engine should abort the Speak call and return immediately. The other three cases require a bit more explanation.

SPVES_VOLUME - the engine should call ISpTTSEngineSite::GetVolume, which will return a new volume

level. The engine should adjust its volume level accordingly. Note that when no XML volume has been specified, the level returned by GetVolume should be exactly the level used by the engine, but if the volume is already affected by an XML tag, the final volume should be a combination of the two.

```
HRESULT GetVolume(  
    [out] USHORT *pusVolume  
);
```

SPVES_RATE - the engine should call ISpTTS_EngineSite::GetRate, which will return a new rate level. The engine should adjust its rate level accordingly. Note that, similarly to volume, XML rate levels and GetRate rate levels should be combined to produce the final rate.

```
HRESULT GetRate(  
    [out] long *pRateAdjust  
);
```

SPVES_SKIP - the engine should call ISpTTS_EngineSite::GetSkipInfo, which will return a type of unit to skip (currently only sentences are supported) and the number of such units to skip. This number can be positive (skip forward in the text), negative (skip backward in the text), or zero (skip to the beginning of the current item). The engine should stop writing data to SAPI, skip the appropriate number of units (or as many as it can) and then call ISpTTS_EngineSite::CompleteSkip to tell SAPI how many units it was able to successfully skip. If it was able to successfully skip the entire number returned by GetSkipInfo, the engine should then continue rendering text at

the appropriate point. Otherwise, it should abort the current Speak call and return immediately.

```
HRESULT GetSkipInfo(  
    [out] SPVSKIPTYPE *peType,  
    [out] long *plNumItems  
);
```

```
HRESULT CompleteSkip(  
    [in] long ulNumSkipped  
);
```

As an example, imagine an engine was passed this text:

“This is sentence one. This is sentence two. This is sentence three.”

Now suppose that the engine was currently rendering the second sentence when it discovered, using `GetActions` and `GetSkipInfo`, that it was being asked to skip +1 sentence. The engine should stop rendering the second sentence, skip forward to the third sentence, call `CompleteSkip` with a parameter of +1, and begin rendering the third sentence. Now imagine that the engine was asked to skip -2 sentences. The engine should again stop rendering the second sentence, and then skip backward until it discovers that it cannot skip the appropriate number. It would then call `CompleteSkip` with a parameter of -1 and abort its `Speak` call.

Queuing Events

Events are structures which are used to pass information from the engine back to the application. The engine is responsible for generating certain types of events, and then handing them to SAPI through the function `ISpTTSEngineSite::AddEvents`. SAPI will then take care of firing the events at the appropriate times.

```
HRESULT AddEvents(  
    [in] const SPEVENT* pEventArray,  
    [in] ULONG ulCount  
);
```

Engines should call the function `ISpTTSEngineSite::GetEventInterest`, which will tell them which events the application (and/or SAPI) is interested in receiving.

```
HRESULT GetEventInterest(  
    [out] ULONGLONG * pullEventInterest  
);
```

This function will return (using `pullEventInterest`) a `ULONGLONG` which will contain one or more values from the TTS subset of the `SPEVENTENUM` enumeration:

- § `SPEI_TTS_BOOKMARK`
- § `SPEI_WORD_BOUNDARY`
- § `SPEI_SENTENCE_BOUNDARY`
- § `SPEI_PHONEME`
- § `SPEI_VISEME`

The engine must then generate the appropriate types of events. Here is the structure of an SPEVENT:

```
typedef [restricted] struct SPEVENT
{
    WORD eEventId;
    WORD elParamType;
    ULONG ulStreamNum;
    ULONGLONG ullAudioStreamOffset;
    WPARAM wParam;
    LPARAM lParam;
} SPEVENT;
```

Note that SAPI is responsible for setting ulStreamNum – the engine need not worry about this field. ullAudioStreamOffset should in each case be the byte (*not* sample) offset in the audio stream at which the event should be fired. **NOTE:** this offset should correspond to a sample boundary.

Let's go through what the various fields of the SPEVENT structure correspond to for each event type.

The SPEI_TTS_BOOKMARK event indicates that the TTS engine has reached a bookmark. Here is the format for the fields of the Bookmark event:

SPEVENT Field	Bookmark event
eEventId	SPEI_TTS_BOOKMARK
eiParamType	SPET_LPARAM_IS_STRING
wParam	Value of the bookmark string when converted to a long (_wtol(...)) can be used)
lParam	Null terminated copy of the bookmark string

For example, if an engine was passed a bookmark corresponding to this XML marked up text:

“<BOOKMARK MARK=“this is a bookmark”/>”

The engine would need to generate an event whose lParam was “this is a bookmark”. If the engine was passed a bookmark corresponding to this XML marked up text:

“<BOOKMARK MARK='1'/>”

The engine would need to generate an event whose wParam was equal to the integer, one.

The SPEI_WORD_BOUNDARY event indicates that the TTS engine

has started synthesizing a word. Here is the format for the fields of the word boundary event:

SPEVENT Field	Word Boundary event
eEventId	SPEI_WORD_BOUNDARY
eParamType	SPET_LPARAM_IS_UNKNOWN
wParam	Character offset of the beginning of the word being synthesized.
lParam	Character length of the word in the current input stream being synthesized

The SPEI_SENTENCE_BOUNDARY event indicates that the TTS engine has started synthesizing a sentence. Here is the format for the fields of the sentence boundary event:

SPEVENT Field	Sentence Boundary event
eEventId	SPEI_SENTENCE_BOUNDARY
eParamType	SPET_LPARAM_IS_UNKNOWN
wParam	Character offset of the beginning of the sentence being synthesized.

IParam	Character length of the sentence in the current input stream being synthesized
--------	--

The SPEI_PHONEME event indicates that the TTS engine has synthesized a phoneme. Here is the format for the fields of the phoneme event:

SPEVENT Field	Phoneme event
eEventId	SPEI_PHONEME
eiParamType	SPET_LPARAM_IS_UNKNOWN
wParam	The high word is the duration in milliseconds of the current phoneme. The low word is the PhoneID of the next phoneme.
IParam	The low word is the PhoneID of the current phoneme. The high word is the SPVFEATURE value associated with the current phoneme.

See Appendix A for the SAPI 5.0 phoneme set.

SPVFEATURE contains two flags - SPVFEATURE_STRESSED, which means that the phoneme is stressed relative to the other phonemes of a word (stress is usually associated with the vowel of a stressed syllable), while SPVFEATURE_EMPHASIS means that the phoneme is part of an emphasized word. That is, stress is a syllabic phenomenon within a word, while emphasis is a word-level phenomenon within a sentence.

The SPEI_VISEME event indicates that the TTS engine has synthesized a viseme. Here is the format for the fields of the viseme event:

SPEVENT Field	Viseme event
eEventId	SPEI_VISEME
eiParamType	SPET_LPARAM_IS_UNKNOWN
wParam	The high word is the duration in milliseconds of the current viseme. The low word is the code for the next viseme
lParam	The low word is the code of the current viseme. The high word is the SPVFEATURE value associated with the current viseme (and phoneme).

The SAPI visemes are based off the Disney 13 Visemes and are

described in Appendix B for the SAPI American English phoneme set.

Queuing Audio Data

After an engine has queued events, it should write audio data to the output site in the appropriate format. **NOTE:** the order of these two events is important – events should not be queued after their associated audio data has already been written or they cannot be fired at the proper times. The function `ISpTTSEngineSite::Write` is used to write audio data.

```
HRESULT Write(  
    const void* pBuff,  
    ULONG cb,  
    ULONG *pcbWritten  
);
```

This function is straightforward – `pBuff` points to a buffer of audio data to be written to the output site, `cb` is the number of bytes (*not* samples) to be written, and `pcbWritten` will return the number of bytes actually written (which should be the same as `cb`, assuming nothing has gone wrong). **NOTE:** only complete samples should be written. If the `Write` function returns `SP_AUDIO_STOPPED` the audio device has been stopped and the `Speak` call should abort immediately.

It should be noted that if an engine (from the application's perspective, a *voice*) is paused (using [ISpVoice::Pause](#)), SAPI will block an `ISpTTSEngineSite::Write` call until the engine is to resume. The same thing will happen if an alert priority voice

interrupts a normal priority voice (see [ISpVoice::SetPriority](#) for more information on voice priorities).

Creating an Engine Properties UI - ISpTokenUI

TTS Engines may wish to supply various UI components - one example is an Engine Properties component which users can access through the SAPI 5.0 control panel. SAPI provides mechanisms for engines to describe what UI components they have, and for applications to request the display of these components.

The UI components that an engine supports should be contained within the engine voice's object tokens (refer to the Object Tokens and Registry Settings White Paper for more discussion of tokens) within a UI subkey. Within this key should be subkeys for each UI component the engine implements. For example, an engine properties component would be in HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Voices\Token\{Voice Name}\UI\EngineProperties. The EngineProperties key would then contain the CLSID of the class to be created when this UI component is displayed. The engine setup should install and register this class, and the class must implement the interface ISpTokenUI.

An application can then see if a particular UI component is supported by an engine by calling ISpTokenUI::IsSupportedUI on the engine's object token.

```
[local] HRESULT IsUISupported(  
    [in] const WCHAR *pszTypeOfUI,  
    [in] void *pvExtraData,  
    [in] ULONG cbExtraData,
```

```
[in] IUnknown *punkObject,  
[out] BOOL *pfSupported  
);
```

Here is an example implementation of IsUISupported:

```
STDMETHODIMP EnginePropertiesUI::IsUISupported(  
    const WCHAR* pszTypeOfUI,  
    void * /*pvExtraData*/,  
    ULONG /*cbExtraData*/,  
    IUnknown * /*punkObject*/,  
    BOOL *pfSupported )  
{  
    *pfSupported = false;  
  
    if ( wcscmp( pszTypeOfUI, SPDUI_EngineProperties ) ==  
0 )  
    {  
        *pfSupported = true;  
    }  
  
    return S_OK;  
}
```

SPDUI_EngineProperties is just the string, "EngineProperties" - this is the string which the SAPI 5.0 control panel uses to query

engines for UI components to be displayed when the user clicks the "Settings" button. If this function call returns true (using pfSupported), the application can then call ISpTokenUI::DisplayUI to display the UI component.

```
[local] HRESULT DisplayUI(  
    [in] HWND hwndParent,  
    [in] const WCHAR * pszTitle,  
    [in] const WCHAR * pszTypeOfUI,  
    [in] void * pvExtraData,  
    [in] ULONG cbExtraData,  
    [in] ISpObjectToken * pToken,  
    [in] IUnknown * punkObject  
);
```

Here is an example implementation of DisplayUI:

```
STDMETHODIMP SpTtsEngUI::DisplayUI(  
    HWND hwndParent,  
    const WCHAR * pszTitle,  
    const WCHAR * pszTypeOfUI,  
    void * /* pvExtraData */,  
    ULONG /* cbExtraData */,  
    ISpObjectToken * pToken,  
    IUnknown * /* punkObject */)   
{  
    HRESULT hr = S_OK;  
  
    if ( SUCCEEDED( hr ) )  
    {  
        if ( wcsncmp( pszTypeOfUI, SPDUI_EngineProperties )  
            == 0 )  
        {
```

```
    EnginePropertiesDialog dlg;
    dlg.hInstance = g_hInstance;
    dlg.hwndParent = hwndParent;
    hr = dlg.Run();
  }
}
return hr;
}
```

Using SAPI Lexicons

SAPI provides lexicons so that users and applications may specify pronunciation and part of speech information for words important to them. As such, all SAPI compliant TTS engines should use these lexicons to guarantee uniformity of pronunciation and part of speech information.

There are two types of lexicons in SAPI:

§ User Lexicons: Each user who logs onto a computer will have a User Lexicon. These are initially empty, but can have words added to them either programmatically, or using an engine's add/remove words UI component (for example, the sample application Dictation Pad provides an Add/Remove Words dialog).

§ Application Lexicons: Applications can create and ship their own lexicons of specialized words - these are read only.

Each of these lexicon types implements the ISpLexicon interface and can be created directly, but SAPI provides a Container Lexicon class which combines the user lexicon and all application lexicons into a single entity, making manipulating the lexicon information much simpler. Here is an example of how to create a Container Lexicon (which will contain the user lexicon and all the application lexicons):

```
CComPtr<ISpContainerLexicon> cpContainerLexicon;
```

```
cpContainerLexicon.CoCreateInstance( CLSID_SpLexicon );
```

The main lexicon function engines will want to use is
ISpLexicon::GetPronunciations:

```
HRESULT GetPronunciations(  
    [in] const WCHAR *pszWord,  
    [in] LANGID LangId,  
    [in] DWORD dwFlags,  
    [out][in] SPWORDPRONUNCIATIONLIST  
    *pWordPronunciationList  
);
```

Here is an example of how to get pronunciations out of a
Container Lexicon:

```
HRESULT hr = S_OK;  
DWORD dwLexFlags = eLEXTYPE_USER | eLEXTYPE_APP;  
SPWORDPRONUNCIATIONLIST SPList;  
  
ZeroMemory( &SPList, sizeof( SPWORDPRONUNCIATIONLIST  
    ) );  
  
hr = cpContainerLexicon->GetPronunciations( pszWord,  
    1033, dwLexFlags, &SPList );  
  
if ( SUCCEEDED( hr ) )  
{
```

```

    for ( SPWORDPRONUNCIATION *pWordPron =
SPList.pFirstWordPronunciation; pWordPron;
        pWordPron = pWordPron->pNextWordPronunciation )
    {
        //--- Do something with each pronunciation
    }
}

if ( SPList.pvBuffer )
{
    ::CoTaskMemFree( SPList.pvBuffer );
}

```

SPWORDPRONUNCIATIONLIST is the structure SAPI uses to return a list of pronunciations for a word:

```

typedef struct SPWORDPRONUNCIATIONLIST
{
    ULONG ulSize;
    BYTE *pvBuffer;
    SPWORDPRONUNCIATION *pFirstWordPronunciation;
} SPWORDPRONUNCIATIONLIST;

```

This structure should be initialized to zeroes before GetPronunciations is called (see the ZeroMemory call in the sample code, above). Furthermore, the memory allocated for

the pronunciations which are returned in this structure must be freed by the engine after GetPronunciations is called - this memory is all pointed to by pvBuffer, hence a single ::CoTaskMemFree call will free all of the allocated memory (see the sample code, above). SPWORDPRONUNCIATIONLIST is just a linked list of SPWORDPRONUNCIATIONS:

```
typedef [restricted] struct SPWORDPRONUNCIATION
{
    struct SPWORDPRONUNCIATION
    *pNextWordPronunciation;
    SPLEXICONTYPE eLexiconType;
    LANGID LangID;
    WORD wReserved;
    SPPARTOFSPEECH ePartOfSpeech;
    SPPHONEID szPronunciation[1];
} SPWORDPRONUNCIATION;
```

eLexiconType indicates which type of lexicon this pronunciation came from - in the above sample code, eLexiconType will be either eLEXTYPE_USER or eLEXTYPE_APP for each returned SPWORDPRONUNCIATION. szPronunciation is a NULL-terminated array of SPPHONEIDs which runs of the end of the SPWORDPRONUNCIATION structure into the pvBuffer member of SPWORDPRONUNCIATIONLIST;

If a word has a pronunciation in the User Lexicon, that pronunciation should take precedence over pronunciations in engine internal lexicons and pronunciations in Application Lexicons. Application Lexicon pronunciations should similarly

take precedence over pronunciations in engine internal lexicons.

For more information on SAPI Lexicons, including adding and removing words from the User Lexicon, or using the basic SAPI Lexicon classes (SpCompressedLexicon, SpUncompressedLexicon) for an engines internal lexicons, see the [Lexicon Manager](#) section).

Appendix A - SAPI 5 Phonemes

SYM	Example	PhoneID
-	syllable boundary (hyphen)	1
!	Sentence terminator (exclamation mark)	2
&	word boundary	3
,	Sentence terminator (comma)	4
.	Sentence terminator (period)	5
?	Sentence terminator (question mark)	6
_	Silence (underscore)	7
1	primary stress	8
2	secondary stress	9
aa	f <u>a</u> ther	10
ae	ca <u>t</u>	11
ah	cu <u>t</u>	12

<i>ao</i>	<u>d</u> og	13
<i>aw</i>	fo <u>u</u> l	14
<i>ax</i>	<u>a</u> go	15
<i>ay</i>	bi <u>t</u> e	16
<i>b</i>	<u>b</u> ig	17
<i>ch</i>	<u>ch</u> in	18
<i>d</i>	<u>d</u> ig	19
<i>dh</i>	<u>th</u> en	20
<i>eh</i>	pe <u>t</u>	21
<i>er</i>	fu <u>r</u>	22
<i>ey</i>	<u>a</u> te	23
<i>f</i>	<u>f</u> ork	24
<i>g</i>	<u>g</u> ut	25
<i>h</i>	<u>h</u> elp	26
<i>ih</i>	fi <u>l</u> l	27
<i>iy</i>	fee <u>l</u>	28
<i>jh</i>	jo <u>y</u>	29

<i>k</i>	<u>c</u> ut	30
<i>l</i>	l <u>l</u> id	31
<i>m</i>	<u>m</u> at	32
<i>n</i>	<u>n</u> o	33
<i>ng</i>	si <u>ng</u>	34
<i>ow</i>	g <u>o</u>	35
<i>oy</i>	to <u>y</u>	36
<i>p</i>	<u>p</u> ut	37
<i>r</i>	<u>r</u> ed	38
<i>s</i>	<u>s</u> it	39
<i>sh</i>	<u>sh</u> e	40
<i>t</i>	<u>t</u> alk	41
<i>th</i>	<u>th</u> in	42
<i>uh</i>	bo <u>o</u> k	43
<i>uw</i>	to <u>o</u>	44
<i>v</i>	<u>v</u> at	45
<i>w</i>	<u>w</u> ith	46

<i>y</i>	y <u>ar</u> d	47
<i>z</i>	<u>z</u> ap	48
<i>zh</i>	pleas <u>ur</u> e	49

Appendix B - SAPI 5 Visemes

VISEME	Described SAPI Phonemes
SP_VISEME_0	Silence
SP_VISEME_1	ae, ax, ah
SP_VISEME_2	aa
SP_VISEME_3	ao
SP_VISEME_4	ey, eh, uh
SP_VISEME_5	er
SP_VISEME_6	y, iy, ih, ix
SP_VISEME_7	w, uw
SP_VISEME_8	ow
SP_VISEME_9	aw
SP_VISEME_10	oy
SP_VISEME_11	ay
SP_VISEME_12	h
SP_VISEME_13	r

SP_VISEME_14	l
SP_VISEME_15	s, z
SP_VISEME_16	sh, ch, jh, zh
SP_VISEME_17	th, dh
SP_VISEME_18	f, v
SP_VISEME_19	d, t, n
SP_VISEME_20	k, g, ng
SP_VISEME_21	p, b, m



SAPI Speech Recognition Engine Guide

1 Contents

[SAPI Speech Recognition Engine Guide](#)

[1 Contents](#)

[2 Summary](#)

[3 Introduction](#)

[3.1 SAPI SR OBJECTS AND INTERFACES](#)

[3.2 SAMPLE SR ENGINE](#)

[4 Engine Initialization and Setup](#)

[4.1 ENGINE CREATION](#)

[4.2 OBJECT TOKEN LAYOUT](#)

[4.3 SETOBJECTTOKEN](#)

[4.4 RECOFILES](#)

[4.5 RECOCONTEXTS](#)

[4.6 RECOGNIZER PROPERTIES](#)

[5 Grammar handling](#)

[5.1 GRAMMAR CREATION AND DELETION](#)

[5.2 CFG GRAMMARS](#)

[5.2.1 Introduction and terminology](#)

[5.2.2 Grammar Notifications](#)

[5.2.3 Word Notifications](#)

[5.2.4 Rule Notifications](#)

[5.2.5 States](#)

[5.2.6 Transitions](#)

[5.2.7 Special Transitions](#)

- [5.2.8 Semantic Properties](#)
- [5.2.9 Additional topics](#)
- [5.3 DICTATION GRAMMARS](#)
- [5.3.1 Language model adaptation](#)
- [5.4 PROPRIETARY GRAMMARS](#)
- [5.4.1 Porting other grammar formats](#)
- [6 Lexicon handling](#)
- [6.1 USING LEXICONS](#)
- [6.2 PHONE CONVERTERS](#)
- [7 Recognition and audio](#)
- [7.1 RECOGNIZESTREAM](#)
- [7.1.1 Active always state](#)
- [7.2 READING AUDIO](#)
- [7.2.1 How audio formats are represented in SAPI](#)
- [7.2.2 Setting the audio format](#)
- [7.2.3 Reading data](#)
- [7.2.4 Information about the audio input](#)
- [7.2.5 Setting the input gain](#)
- [7.3 THREADING MODEL](#)
- [7.4 SYNCHRONIZATION](#)
- [7.4.1 Pause and auto-pause](#)
- [7.5 EVENTS AND RECOGNITIONS](#)
- [7.5.1 Standard events](#)
- [7.5.2 Event ordering](#)
- [7.5.3 Other events](#)

- [7.6 COMPLETION OF PROCESSING](#)
- [8 Recognition Results](#)
 - [8.1 RECOGNITION CALL](#)
 - [8.2 DICTATION PHRASES](#)
 - [8.3 CFG PHRASES](#)
 - [8.4 CONFIDENCE SCORING AND REJECTION](#)
 - [8.4.1 Word Confidence](#)
 - [8.4.2 Property and Rule Confidence](#)
 - [8.4.3 Required Confidence and Rejection](#)
 - [8.4.4 Ambiguous Results](#)
 - [8.5 INVERSE TEXT NORMALIZATION \(ITN\)](#)
 - [8.6 INTERPRETERS](#)
- [9 Alternates](#)
 - [9.1 RETURNING ALTERNATES IN A RECOGNITION](#)
 - [9.2 ALTERNATES ANALYZER](#)
- [10 User-Interface](#)
- [11 Engine extensions](#)
 - [11.1 IMPORTANT NOTES ABOUT COM INTERFACE POINTER HANDLING BY THE SR EXTENSION AGGREGATES](#)

2 Summary

This document describes fully the Speech Recognition engine interface in SAPI 5.0. Speech Recognition engines and applications use this interface to connect to SAPI. This document is aimed at engine vendors wishing to port their SR engine using SAPI 5.0, and at general developers who are interested in understanding more about SAPI. This document explains which interfaces and objects SAPI implements, and which interfaces an SR engine should implement. It describes how engines are registered and initialized; how grammar and lexicon information is communicated to the engine; how engines read data and perform recognition; and how engines return events and results back to the application.

3 Introduction

The Microsoft Speech API (SAPI) is a software layer used by speech-enabled applications to communicate with Speech Recognition (SR) engines and Text-to-Speech (TTS) engines. SAPI includes an Application Programming Interface (API) and a Device Driver Interface (DDI). Applications communicate with SAPI using the API layer and speech engines communicate with SAPI using the DDI layer.

A speech-enabled application and an SR engine do not directly communicate with each other – all communication is done using SAPI. SAPI controls a number of aspects of a speech system, such as:

- Controlling audio input, whether from a microphone, files, or a custom audio source; and converting audio data to a valid engine format.
- Loading grammar files, whether dynamically created or created from memory, URL or file; and resolving grammar imports and grammar editing.
- Compiling standard SAPI XML grammar format, and conversion of custom grammar formats, and parsing semantic tags in results.
- Sharing of recognition across multiple applications using the shared engine, as well as all marshaling between engine and applications.
- Returning results and other information back to the application and interacting with its message loop or other notification method. Using these methods, an engine can have a much simpler threading model than in SAPI 4, because SAPI 5 does much of the thread handling.

- Storing audio and serializing results for later analysis.
- Ensuring that applications do not cause errors - preventing applications from calling the engine with invalid parameters, and dealing with applications hanging or crashing.

The SR engine performs the following tasks:

- Uses SAPI grammar interfaces and loads dictation.
- Performs recognition.
- Polls SAPI for information about grammar and state changes.
- Generates recognitions and other events to provide information to the application.

3.1 SAPI SR Objects and Interfaces

In order for an SR engine to be a SAPI 5 engine, it must implement at least one COM object. Each instance of this object represents one SR engine instance. The main interface this object must implement is the ISpSREngine interface. SAPI calls the engine using the methods of this interface to pass details of recognition grammars. It also uses these methods to inform the engine when to start and stop recognition. SAPI itself implements the interface ISpSREngineSite. A pointer to this is passed to the engine and the engine calls SAPI using this interface to read audio, and return recognition results.

ISpSREngine is the main interface to be implemented, but there are other interfaces that an engine may implement. The SR engine can implement the ISpObjectWithToken interface. This provides a mechanism for the engine to query and edit information about the object token in the registry used to create the engine. Information about object tokens is provided in the [Object Tokens and Registry Settings White Paper](#) and in [SetObjectToken](#).

There are two other interfaces that the engine can also implement. Each needs to be implemented in a separate COM object, because SAPI needs to create and delete them independently of the main engine. These interfaces are:

- ISpSRAlternates, which can be used by the SR engine to generate alternates for dictation results. It is possible to generate alternates without this interface, but this interface generates alternates off-line, after the result has

been serialized. (See [Alternates](#)).

- ISpTokenUI implements UI components that can be initialized from an application. These can be used to perform user training, add and remove user words, and calibrate the microphone. (See [User-Interface](#))

The engine can also implement another COM object enabling engine-specific calls between the application and the engine. This object can implement any interface, which the application is able to use QueryInterface for. (See [Engine extensions](#))

3.2 Sample SR Engine

The SAPI 5.0 SDK contains working Microsoft speech recognition engines for US English, Japanese and Chinese. These engines are not shipped with source. The SDK also contains a Sample SR Engine, which is shipped with source (In directory **Microsoft Speech SDK5.0\Samples\CPP\Engines\SR**). This is a sample engine - it implements all the functionality of an SR engine and can be created and used in applications, but it does not actually perform any recognition - instead it generates valid, but random, results. This is very useful example code for understanding how a real SR engine might be implemented.

4 Engine Initialization and Setup

4.1 Engine Creation

When an application wants to perform recognition, it can create a recognizer in one of two ways. The application can create an in-process (InProc) ISpRecognizer object. In this case, SAPI creates the SR engine COM object from the object token representing an engine. Alternatively, an application can create the shared recognizer. In this case, SAPI will create the SR engine in a separate process (named sapiusr.exe) and all applications will share this recognizer. This process is completely invisible to the SR engine and all marshaling is handled by SAPI.

In order to create the SR engine, SAPI uses the SetRecognizer call to look at the object token of the default recognizer or the object token that the application has specified. The object token contains the class ID (CLSID) of the main SR engine and this class is created. The SR engine COM classes must register themselves with "ThreadingModel = Both" or they may not be successfully created.

SetSite on the ISpSREngine interface is then called to give the engine a reference to the ISpSREngineSite interface it will use to call back to SAPI. Like all COM interfaces, the engine should use AddRef to maintain the correct reference count.

4.2 Object token layout

As part of the SR engine installation process, an object token is added that represents the engine into the user's system. Otherwise, SAPI will have no information about the SR engine. To add an object token, add a key to this point in the registry:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Speech\Recognizers\T
```

On a computer with SAPI 5 installed, there are several registry keys here for the Microsoft English, Chinese, Japanese and Sample SR Engine.

Inside this key there must be the value CLSID containing the CLSID of the main SR engine class. Using this key SAPI determines how to create the engine. There are values for the other CLSIDs that the engine can implement - AlternatesCLSID for the class implementing the ISpSRAlternates alternates analyzer, and RecoExtension for the class implementing any engine-specific private call interfaces. The key should also contain a {Default} value set with the name of the engine so that the Speech properties in Control Panel can display it.

In this key, there can also be a subkey Attributes, used by applications to query for engines matching certain attributes. Typically, the engine sets a value Language to indicate which languages the engine supports; Dictation to indicate the engine supports dictation; CommandAndControl to indicate the engine supports command and control, and so on. (See the [Object Tokens and Registry Settings White Paper](#) for more information on object tokens).

There can also be a key UI indicating the types of user-interface components the engine supports (See [User-Interface](#)).

4.3 SetObjectToken

Once SAPI creates the SR engine COM object, it determines if the engine supports the `ISpObjectWithToken` interface. If it does, `SetObjectToken` is called. This passes a pointer to the object token this engine was created from. This is useful for two reasons:

- The engine can use the token to store information. It can store information directly in the object token using the `ISpDataKey` methods, or it can store file paths to other engine data. The `ISpObjectToken` interface method, `GetStorageFileName`, provides an easy way for an engine to find a file path to store data. This path is stored in the `Files` subkey of the engine object token.
- The engine can also read information from the token, such as file paths set during install, or user options set using an engine properties UI component. It is also possible for several object tokens to share the same engine CLSID. For example, they can share the CLSID for different language engines, telephony, or desktop variants of the engine. In this case, the engine needs to know from which object token it is being created.

4.4 RecoProfiles

SetRecoProfile is called next to give the engine an ISpObjectToken pointer referring to the current user profile. RecoProfiles are added or removed by the user inside Speech properties in Control Panel. The engine can create a subkey under the profile object token and use it to store any data. The engine must store the data in a subkey named after its CLSID. This prevents other engines on the system recognizing the same profile.

To provide user enrollment, the engine implements a UI component User Training (See [User-Interface](#)) (SPDUI_UserTraining is defined as this in sapi.idl). This is instantiated using Control Panel->Speech properties->SR tab->Train Profile. Engines can also request that an application display the UI using AddEvent (See [Events and Recognitions](#)) to request UI.

The user-training UI might produce some adapted model files. These can be saved and their location stored in the RecoProfile object token. The engine can read the location of these files from the object token later.

4.5 RecoContexts

Each application using speech has at least one RecoContext object implementing ISpRecoContext. It is from this interface that the application creates and loads grammars and activates recognition. SAPI informs the SR engine of each RecoContext associated with it using OnCreateRecoContext and OnDeleteRecoContext. The SR engine returns a pointer to SAPI from the OnCreateRecoContext function, which is then passed to the engine in any future calls that need to refer to the RecoContext. It is not essential for an engine to keep track of each RecoContext unless it is using private calls or proprietary grammars.

4.6 Recognizer Properties

SAPI provides a means for applications to set certain settings and configurations on the SR engine. This is done using the application calling methods on the ISpProperties interface, implemented on the RecoContext objects. There are four methods on this interface to get and set string and integer values. When these methods are called, SAPI calls equivalent methods on the SR engine: GetPropertyString, SetPropertyString, GetPropertyNum, SetPropertyNum.

In SAPI 5.0, each method has a SPPROPSRC parameter, which is always set to SPPROPSRC_RECO_INST, and a pointer *pvSrcObj*, which is always set to NULL.

A number of these properties are already defined by SAPI (See [SAPI 5.0 SR Properties White Paper](#)). Ideally, the engine should implement these if they have equivalent parameters that can be controlled.

When an application sets one of these values, and calls the SR engine, it returns S_OK if it supports this property and the value is updated, and S_FALSE if it doesn't.

Note that these properties exist to alter run-time settings for this instance of the engine, and are reset every time the engine is deleted. For permanent changes to engine results, use an engine properties UI component, or include additional values in its object token that applications can read and set.

5 Grammar handling

Each speech application can have one or more ISpRecoGrammar objects associated with it. Within each grammar object there are several types of grammar:

- Command and Control grammars. These are context-free grammars (CFG) created from either a SAPI XML grammar, or dynamically from the application, or from some other grammar format using the SpGramCompBackend object. In all cases, SAPI reports the contents of the grammars to the engine using the SpGramCompBackend object.
- Dictation grammars. Here the engine loads and unloads its own dictation language model.
- Proprietary grammars. There are various calls in SAPI to support engine-specific grammar formats.

Each grammar object can contain a dictation grammar and either a CFG or proprietary grammar.

Each application can have several grammars. In the shared recognizer case, multiple applications can be connected to one recognizer. Thus, grammars can be loaded, unloaded, modified, activated, and deactivated independently of each other. However, the SR engine controls when it is informed of these grammar state changes during recognition (See [Synchronization](#)).

5.1 Grammar Creation and Deletion

When an application creates a grammar object, this is reported to the engine using `OnCreateGrammar`. This passes the engine a grammar handle, as well as the pointer the SR engine returned from the call to `OnCreateRecoContext`. From this method the engine must also return a pointer, which is used to identify the grammar in later calls from SAPI. `OnDeleteGrammar` is called to delete grammars.

5.2 CFG Grammars

5.2.1 Introduction and terminology

Each CFG grammar contains one or more rules. Rules can be top-level, indicating that they can be activated for recognition. Each rule has an initial state and additional states, which are connected by transitions. Each transition can be one of several types:

- A word transition indicating a word to be recognized
- A rule transition indicating a reference to a sub-rule
- An epsilon (null) transition
- Some special transitions for such features as embedding dictation within a CFG.

References to sub-rules can be recursive, i.e., rules can reference themselves, either directly or indirectly. Left recursion is not supported and SAPI will reject these grammars upon loading. Inside a grammar, transitions can have semantic properties, although the engine does not normally need to recognize these.

SAPI takes full control of loading a grammar when an application requests it. SAPI can load from a file, URL, resource, or from memory, and can load either binary or XML forms of the grammar, and resolve imports. SAPI then notifies the SR engine about the contents of the grammar through various DDI methods.

5.2.2 Grammar Notifications

WordNotify and RuleNotify notify the engine about CFG grammar information. SAPI calls both methods before recognition begins, when a grammar is first loaded, and during recognition within a Synchronize call if grammars change (See [Synchronization](#)).

5.2.3 Word Notifications

The WordNotify call informs the engine about the words in the grammar. A single call is made to either add or remove words. SAPI keeps a reference count internally so that each word will be added only if it is not present in any existing grammar. Each word is represented by an SPWORDENTRY structure:

```
typedef struct SPWORDENTRY
{
    SPWORDHANDLE    hWord;
    LANGID          LangID;
    WCHAR           *pszDisplayText;
    WCHAR           *pszLexicalForm;
    SPPHONEID      *aPhoneld;
    void            *pvClientContext;
} SPWORDENTRY;
```

The *hWord* is a unique handle identifying the word. The *pvClientContext* is an arbitrary pointer that the SR engine sets with a call to SetWordClientContext. Subsequent calls to GetWordInfo will return the same structure with this field filled

in. The LangID field represents the language of the word. Currently this will be the same for all words in a grammar, but in the future SAPI may support multi-lingual grammars.

The *pszDisplayText* and *pszLexicalForm* fields give the text of the word. Words can be defined in a grammar to have a different textual display form to the actual spoken lexical form used to look up the words in a lexicon. The grammar can also specify the pronunciation of the word. This is given as an array of SPHONEIDs. See [Phone Converters](#) for more detail on phones and phone converters.

5.2.4 Rule Notifications

The RuleNotify call informs the engines when rules are added, changed or removed. There are five actions that are performed on rules:

- New rules can be added.
 - Existing rules can be removed.
- Rules can be activated.
 - Rules can be deactivated for recognition.
- Rules can be invalidated, which means the rule has been edited by the application and thus the engine needs to reread the contents of the rule.

Each rule is represented by an SPRULEENTRY structure:

```
typedef struct SPRULEENTRY
{
```

```

SPRULEHANDLE  hRule;
SPSTATEHANDLE hInitialState;
DWORD        Attributes;
void *       pvClientRuleContext;
void *       pvClientGrammarContext;
} SPRULEENTRY;

```

The *hRule* is a unique handle identifying the rule. The *pvClientRuleContext* is a pointer that the engine sets using `SetRuleClientContext`. Subsequent calls to `GetRuleInfo` return the same structure but with the *pvClientRuleContext* field filled in. The *pvClientGrammarContext* is the pointer that the engine set in `OnCreateGrammar`. This indicates which grammar the rule belongs to. The `Attributes` field, of type `SPCFGRULEATTRIBUTES`, contains flags with extra information about the rule:

- `SPRAF_TopLevel` if the rule is top-level and thus can be activated for recognition.
- `SPRAF_Active` if the rule is currently activated.
- `SPRAF_Interpreter` if the rule is associated with an Interpreter object for semantic processing (See [Interpreters](#)).
- `SPRAF_AutoPause` if the rule is auto-pause (See [Pause and auto-pause](#)).

The *hInitialState* gives the initial state of the rule.

5.2.5 States

The SR engine determines the full contents of the rule (either

immediately, or later during recognition), using `GetStateInfo`. This method passes information about all the subsequent states following from any given state. The engine passes a state handle into this method (starting with the *hInitialState* of the rule), and a pointer to an `SPSTATEINFO` structure (with all its fields initially set to zero). This structure is filled out with information on all of the transitions out of that state in the *pTransitions* array. SAPI uses `CoTaskMemAlloc` to create this array. The engine can call this method again on each of the states following the current state in order to get information about all of the states in the rule. Loop-back transitions are possible in a rule and the engine needs to check that it has not visited the current state before.

When the engine calls `GetStateInfo` subsequent times, it can call it with the *cAllocatedEntries* and *pTransitions* fields unchanged. SAPI re-uses the memory from the transition array, if possible, rather than re-allocating it. Alternatively, the engine can use `CoTaskMemFree` to free the *pTransitions* memory, and set these fields to `NULL`. SAPI will then re-allocate the memory every time.

5.2.6 Transitions

Each transition represents a link from one state to another state and is represented by an `SPTRANSITIONENTRY` structure. This structure contains an ID field that uniquely identifies the transition, an *hNextState* handle that indicates the state the transition is connected to, and a `Type` field that indicates what type of transition this is.

There are three common types of transition that all engines need to support:

- Word transitions (SPTRANSWORD). These represent single words that the recognizer recognizes before advancing to the next state. The handle to the word and the word pointer are supplied inside the SPTRANSITIONENTRY structure, which the engine uses to find the full text of the word with GetWordInfo. To produce recognition results, the engine needs to keep track of the transition IDs of word transitions as they are used in ParseFromTransitions.
- Rule transitions (SPTRANSRULE). These represent transitions into sub-rules. This transition is only passed when a path through the sub-rule has been recognized. The rule handle, engine's rule pointer, and initial state of the sub-rule are supplied. Rules can be recursive, but not left recursive.
- Epsilon transitions (SPTRANSEPSILON). These are null or transitions that can be traversed without recognizing anything.

A state with a transition to a null state handle indicates the end of a rule. There can also be void states, which are blocking and indicate that there is no recognition path from this state. These void paths are indicated by a state having zero transitions out of it.

5.2.7 Special Transitions

There are a number of special transitions that may not be supported by all engines. Attributes in the engine object token indicate whether these are supported:

- Wildcard transition (SPTRANSWILDCARD). This indicates a transition that matches any word or words (sometimes called a "garbage" model). The engine does not try and

recognize the spoken words. The engine includes the string value `WildcardInCFG` as an attribute in its object token to inform the application that it is capable of supporting this.

- Dictation transition (`SPTRANDICTATION`). This is used to embed dictation within a CFG. Each transition means one word should be recognized. The attribute `DictationInCFG` in the engine object token indicates support for this feature.
- Text buffer transition (`SPTRANSTEXTBUF`). This indicates that the engine is to recognize a sub-string of words from the text-buffer, if it has been set. (See [Text-buffers](#)).

5.2.8 Semantic Properties

Application developers are able to put properties (also known as semantic tags) within a grammar. This provides a powerful means for semantic information to be easily embedded inside a grammar.

By default, the engine does not recognize these properties. Typically, an engine simply recognizes the speech from the words in the grammar, and SAPI parses and adds the property information in the `ParseFromTransitions` call. However, it is possible for an engine to receive this information by calling `GetTransitionProperty` on any transition. If there is a property on this transition, the property name or ID, and value are returned in the `SPTRANSITIONPROPERTY` structure. When finished, `SPTRANSITIONPROPERTY` must be freed using `CoTaskMemFree`.

5.2.9 Additional topics

5.2.9.1 Ordering of actions

SAPI notifies engines about the contents of CFGs in a logical order. When a grammar is loaded, all new words are notified first with a WordNotify call, then all rules with a RuleNotify call. After that, rules are activated and deactivated. When a grammar is deleted, all rules are removed first and then all words.

All the rules and words for each grammar are added with single calls to RuleNotify and WordNotify. If a number of grammars are being loaded and rules are being activated, separate calls are made for each grammar. For engines that have a time-consuming internal grammar compilation to do before starting recognition, try to avoid unnecessary recompilations. Where possible, applications should try and combine separate CFG grammars into one to minimize compilations.

5.2.9.2 Grammar Weights

Grammar designers use a Weight field during each transition to change the likelihood of certain paths being taken. This Weight field is a probability – the range of values is 0.0 to 1.0, and the values of the transitions out of any state sum to 1.0. A value of 0.0 should always be interpreted as making this transition impossible to pass during recognition. Engines may or may not incorporate the other weight values into their recognition search. By default, grammars do not have weights set, so each transition weight will be 1.0 divided by the number of transitions out of the preceding state.

5.2.9.3 Required Confidence

Each transition also contains a RequiredConfidence field.

Grammar designers use this field to set how easily recognitions are to be accepted or rejected. For example, this field is used to avoid false positives on critical actions, such as delete file, while less critical actions, such as scroll down, remain unaffected. See [Required Confidence and Rejection](#) for more information on how this field can be used for rejection.

5.2.9.4 Text-buffers

Using this feature, an application can define a text buffer. When a text-buffer transition is reached in a CFG, the engine attempts to recognize a sub-string of words from the text buffer.

The text buffer is set by the application using `ISpRecoGrammar::SetWordSequenceData`, and reported to the engine by `ISpSREngine::SetWordSequenceData`. The format of the buffer is a sequence of one or more null-terminated strings, with a double null-termination at the end. The engine recognizes any sub-string of words from any of the strings in the buffer. This provides a very simple way for applications to select from a set of text.

It is also possible for the application to alter the areas of the buffer to be used for recognition. This is done using `SetTextSelection` with the structure `SPTXTSELECTIONINFO`. The *ulStartActiveOffset* and *cchActiveChars* indicate which area of the buffer should be active for recognition.

The other two fields of the `SPTXTSELECTIONINFO`, *ulStartSelection* and *cchSelection*, are used with dictation to indicate which area of the buffer is currently selected on screen. If *cchSelection* is zero, this indicates where the insertion point currently is. The engine could use this insertion point to get

extra language model context from the preceding words in the dictated text.

This text buffer feature is optional for engines, and support for it is defined in the WordSequences attribute. See [Object Tokens and Registry Settings](#) for more information.

5.2.9.5 Rule invalidations

When a rule is invalidated with a RuleNotify call with SPCFGN_INVALIDATE action, the engine needs to discard any cached information it has regarding this rule and parse the rule again from the new initial state. If a sub-rule is edited in a grammar, only that rule is invalidated, and not other rules referring to this rule.

5.2.9.6 Grammar Resources

It is possible for resource data to be included in a grammar. Each rule can contain one or more named strings containing arbitrary data. The engine recovers this data using GetResource, and passing in the rule handle and resource name.

5.3 Dictation Grammars

The mechanism used for handling dictation grammars is considerably simpler than for CFG grammars. SAPI instructs the engine to load a dictation grammar (or Statistical Language Model) with the LoadSLM method. LoadSLM supplies the engine's pointer to the grammar and a topic name. The topic name is by default NULL, and uses the standard dictation language model, but an application can request a specific topic. Currently, the only topic SAPI defines is Spelling for a spelling-mode grammar, although engines are free to define others. Dictation is unloaded with UnloadSLM and activated or deactivated for recognition with SetDictationState.

5.3.1 Language model adaptation

An application supplies text data to the engine for language model adaptation with SetAdaptationData. The engine is free to do nothing or anything with this data, and to either persist the adaptation in the current RecoProfile or reset every session. Because some engines take considerable processing to do adaptation, it is recommended that applications submit adaptation data in chunks. When the engine is ready to receive more data, it fires an event SPEI_ADAPTATION. If an engine does not have this performance issue, it can send the event immediately.

5.4 Proprietary grammars

If an engine and application wish to use a grammar format different from the standard SAPI binary or XML formats, this is possible. `LoadProprietaryGrammar` and `UnloadProprietaryGrammar` are used to load and unload such grammars. `LoadProprietaryGrammar` is called when the application calls `ISpRecoGrammar::LoadProprietaryGrammar`. The application can supply the engine with string data, binary data, or a GUID, or some combination of these. SAPI does not touch this data in any way apart from marshaling it between the application and shared engine.

To activate rules in a proprietary grammar, call `SetProprietaryRuleState` or `SetProprietaryRuleIdState`. The first of these methods takes an optional string pointer, where `NULL` is interpreted to mean activate all top-level rules. These methods are used so that `ISpRecoGrammar::SetRuleState` works consistently on SAPI CFGs and Proprietary grammars. The engine must set the *pcRulesChanged* value to inform SAPI how many rules have been activated or deactivated.

There are some extra methods that an SR engine needs to recognize if it is using proprietary grammars. When a grammar is activated or deactivated, it calls `SetGrammarState`. `SetContextState` activates or deactivates a context (with CFG grammars SAPI will automatically deactivate or activate all relevant rules so these methods are not needed).

5.4.1 Porting other grammar formats

A disadvantage of proprietary grammar formats is that they are not engine or application independent. It is possible to write code that will convert any finite-state or CFG-based grammar format into the compiled SAPI binary format. For example, SAPI XML grammars are compiled using the object CLSID_SpGrammarCompiler. This object uses the CLSID_SpGramCompBackend object to add the correct rules, states, and transitions into the grammar, and then saves the grammar to an IStream using SetSaveObjects and Commit.

Exactly the same approach can be used for other formats. A new COM object could be created using the compiler back-end methods to add the correct information and save the grammar (See the ISpGrammarBuilder interface documentation for more info). The grammar produced is a standard SAPI compiled grammar and can be used by any SAPI engine or application, without having to use the proprietary grammar support.

6 Lexicon handling

SAPI provides a means for users or applications to specify new words and their pronunciations in lexicons. An engine should look at these words and pronunciations and use them during recognition.

There are two types of lexicons in SAPI:

- User Lexicon. There is a User Lexicon for the current user logged onto the computer. It is initially empty but the user can add to it, either programmatically, or using an engine's add/remove words UI component. For example, running the Dictation Pad sample application and choosing Add/Remove Words will access the Microsoft engine UI where the user can add new words to the user lexicon.
- Application lexicons. Applications can create and ship their own lexicons of specialized words. These are read-only.

Both types of lexicon are represented by object tokens.

(Application Lexicons are stored in

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Appl

and the current user lexicon in

HKEY_CURRENT_USER\SOFTWARE\Microsoft\Speech\Curre

Each lexicon object implements the ISpLexicon interface.

Lexicons can be individually created from their object tokens.

However, SAPI provides a Container Lexicon that combines the user and application lexicons into one single entity, making manipulating the lexicon information simpler. This class is

SpLexicon (CLSID_SpLexicon).

6.1 Using lexicons

Typically, when an engine runs dictation, it will use CoCreate to create the SpLexicon object and determine which words and pronunciations are in it. The engine adds these to its dictation vocabulary and language model if it supports this feature. If the lexicon contains words with no pronunciation, the engine should try and generate pronunciations for them if it supports this feature. Similarly, if a CFG grammar is loaded containing words that are in the lexicon, the engine acknowledges these pronunciations.

Applications can call IsPronounceable on the RecoContext objects to determine if the engine can generate a pronunciation for a word. Engines set the BOOL pointer passed in to TRUE, if a pronunciation was found, otherwise, FALSE.

An engine can also add words into the user lexicon using AddPronunciations on the SpLexicon. ([Lexicon interfaces](#) for more information).

As an option, an engine also uses the basic SAPI lexicon classes (CLSID_SpCompressedLexicon for read-only lexicons and CLSID_SpUncompressedLexicon for read-write lexicons) for its own internal lexicons.

6.2 Phone Converters

For each language, SAPI defines a phone set describing which phones can be used in defining the pronunciation of a word. Currently, phone sets are defined for US English, Japanese and Chinese, and more will be added in later updates. Phone converters are represented as object tokens in the registry. Each phone converter object implements the interface `ISpPhoneConverter`. This has two methods, `PhoneToId` and `IdToPhone`, to convert between phone strings and phone IDs.

The pronunciations returned from a lexicon are returned as null-terminated arrays of `SPPHONEID`. All the methods the engine sees which use pronunciations also use these arrays of phone IDs, so the engine may never need to use a phone converter directly. However, to create a phone converter for a specific language, an engine uses `SpCreatePhoneConverter` in `sphelper.h`:

```
ISpPhoneConverter *pPhoneCon;
```

```
HRESULT hr = SpCreatePhoneConverter(409, NULL, NULL,  
&pPhoneCon);
```

If a vendor wishes to implement an engine in a language for which Microsoft does not currently define a phone converter, it will not be possible to use lexicons or grammars with pronunciations in them; otherwise, the engine vendor will need to define its own engine-specific phone converter.

7 Recognition and audio

SAPI indicates that the engine should start recognition by calling `RecognizeStream` on the SR engine. From that point on, the engine can read data, perform recognition, and send results and events back to SAPI. When all the data has been recognized or the application has deactivated recognition, the engine finishes processing and returns from the `RecognizeStream` call.

Thus the basic actions that take place are as follows:

- SAPI calls `RecognizeStream` on the engine.
- The engine starts reading data and doing recognition.
- The engine calls `Synchronize` and `UpdateRecoPos` to be informed of grammar changes.
- The engine returns events, hypotheses, and recognitions to SAPI.
- Recognition continues until the stream is terminated and the engine returns from `RecognizeStream`.

7.1 RecognizeStream

RecognizeStream is normally called after grammars have been loaded and activated. The engine recognizes from the rules that are active. If multiple rules or dictations are active, the engine recognizes from all things in parallel, i.e., the user is able to say a word from any available rule or dictation that is active.

7.1.1 Active always state

In only one case, RecognizeStream can be called when there are no active rules. That is, when the application has set the RecoState to SPRST_ACTIVE_ALWAYS (with ISpRecognizer::SetRecoState). This is done by an application when it wants audio running to display a VU-meter (by listening to SPEI_SR_AUDIO_LEVEL events). In this case, RecognizeStream is called regardless of whether there are any active rules. The engine is free to throw the data away; although, engines can use this data to perform environmental adaptation or noise level estimation.

7.2 Reading audio

7.2.1 How audio formats are represented in SAPI

Two fields are used to define audio formats in SAPI: A GUID defining the class of format; and for wav format types, a WAVEFORMATEX structure that contains type, sample rate, bits per sample etc. All wav format types have the GUID SPDFID_WaveFormatEx. Engines can use other GUIDs for any engine-specific formats they have. There is a helper class CSpStreamFormat in sphelper.h that converts to and from this format. Also, the SPSTREAMFORMAT enumeration lists commonly used formats.

The WAVEFORMATEX structure has the following definition:

```
typedef struct WAVEFORMATEX
{
    WORD    wFormatTag;
    WORD    nChannels;
    DWORD   nSamplesPerSec;
    DWORD   nAvgBytesPerSec;
    WORD    nBlockAlign;
    WORD    wBitsPerSample;
    WORD    cbSize;
} WAVEFORMATEX;
```

For example, the format for mono, PCM linear, 16-bit, 16kHz audio would be indicated by:

```
GUID guid = SPDFID_WaveFormatEx and  
WAVEFORMATEX wfx = { WAVE_FORMAT_PCM, 1, 16000,  
32000, 2, 16, 0 }
```

7.2.2 Setting the audio format

To determine what audio formats the engine supports, SAPI calls `GetInputAudioFormat` on the SR engine. The first pair of parameters of this method indicates the format SAPI is determining if the engine supports; the engine fills in the second pair of parameters to indicate which format it supports. Alternatively, SAPI calls this method with the first format set to `NULL`, in which case the engine sets the second pair of parameters to its preferred format.

When `GetInputAudioFormat` finishes, the `RecognizeStream` call specifies the current audio format in the *rguidFmtId* and *pWavFormatEx* parameters. This will not change during the `RecognizeStream`.

7.2.3 Reading data

`Read` is used to read data from the audio source. The SR engine requests the amount of data to be read. SAPI returns this data immediately if it is available, or will block until that amount of data is available. If the `Read` call returns a failure code or it reports that less data has been read than requested, the stream has ended and the engine can return from the `RecognizeStream` call once it has processed any data it has buffered.

It is possible to see how much data is available for reading immediately using the `DataAvailable` call. This call is used to read only the data that is already available without blocking. A Win32 event parameter, *hDataAvailable* on `RecognizeStream`, is set when a certain amount of data is available. `SetBufferNotifySize` sets the amount of data to trigger this parameter.

The engine tries to read data as close to real-time as possible. SAPI has only a finite buffer of audio data, and if the engine's reading lags by more than a certain amount of time (approx. 30 seconds currently), SAPI terminates the stream. The engine has the option of keeping its own buffer of data that has been read but has not yet processed.

The amount of data requested in the `Read` call is in bytes. All stream positions used in sending events, recognition information etc. back to SAPI are in bytes also.

7.2.4 Information about the audio input

The SR engine does not directly have access to the audio input device. SAPI handles this so that the SR engine is consistent regardless of the type of input. However, there is some information about the stream that the engine receives from parameters in the `RecognizeStream` call:

- The *rguidFmtId* and *pWavFormatEx* parameters indicate the format of the stream.
- *pAudioObjectToken* points to the object token that represents the audio input device.

- The *fRealTimeAudio* Boolean indicates whether the input is real-time. Real-time inputs in SAPI are those that implement the ISpAudio interface. An example of this is the standard multi-media microphone input. Non-real time streams are those that only implement ISpStreamFormat. An example of this is inputting wav files using the ISpStream object. With non real-time streams, all the data is available for reading immediately. The *hDataAvailable* event is always set and DataAvailable always returns INFINITE.
- *fNewAudioStream* specifies whether this call to RecognizeStream is a restarting of an existing stream or a new stream. For example, if an application deactivates the active rules and the RecognizeStream returns. If later the application activates some rules, the RecognizeStream call will have this parameter set as FALSE. Only if the application activates a new SetInput will this return TRUE. Some engines might find this useful because they could preserve environmental information between calls to RecognizeStream and only reset this when the input is really changed.

7.2.5 Setting the input gain

The engine can alter the gain of the input, if the input is from an audio device passing through the Windows Mixer. The engine can store a value in the RecoProfile for each device indicating what the gain should be, and SAPI sets the gain on this device in the mixer every time the audio is opened. The engine can set this value in the RecoProfile either when calibrating a microphone within its Microphone Training UI component, or at any other point.

The value stored in the profile must be in a subkey with the

same name as the CLSID of the main engine object. The value name should have the token ID of the audio input object currently being used (found by calling GetId on the audio input object token). The value should be a DWORD value between 0 and 10000 indicating the mixer level to set.

7.3 Threading model

With SAPI, engines use a very simple threading model. When recognition does not occur (i.e., when `RecognizeStream` is not being called on the engine), SAPI calls the engine on only one thread. During the `RecognizeStream`, SAPI only calls the engine when the engine itself calls `Synchronize`. Thus, the engine can control when it is called back. These call backs also occur only on one thread.

Because the engine does not return from `RecognizeStream` until all recognition is complete, SAPI has effectively given the engine one thread on which to operate. It is possible to write an engine to do all its work on this one thread and thus require no additional threads, critical sections or other thread-locking. This one thread system works best if the engine is not blocked unnecessarily during `Read` calls when it could be performing recognition. Use the `DataAvailable` to achieve this.

An alternative would be to have one additional thread. In this case, one thread could read the data, and possibly perform feature extraction, while another thread could do the actual recognition processing. Other threading arrangements are possible, and SAPI makes no restrictions about which threads call which methods or whether the methods are called simultaneously.

7.4 Synchronization

Synchronize informs SAPI that the engine is ready to receive any pending actions, such as grammar changes, rule activations/deactivations, or private calls. When the engine calls Synchronize, the engine is called from SAPI to perform these actions. For example, if the application has changed a grammar and called Commit, the next time the engine calls Synchronize, SAPI will call the engine back on RuleNotify and WordNotify with details of the modified grammar. When the engine returns from these methods, SAPI returns back from Synchronize as long as the engine is not in a paused state. When the engine sends a final recognition, (See [Events and Recognitions](#)) Synchronize is also called internally by SAPI.

The engine has complete control over when Synchronize is called. For example, engines may want to handle grammar changes only when the user is not speaking; and not handle changes when they are actually performing recognition. Also, a Win32 event *fRequestSync* is passed as a parameter in the RecognizeStream call. This event is set when an action is queued for the engine to respond to when it calls Synchronize. An engine can call Synchronize regularly, or only when this event is set.

The more quickly the engine responds to queued tasks, the more responsive it will seem to a user. An engine should not wait too long before calling Synchronize because in some cases an application will hang until the engine does so. For example, the sample SR Sample engine does not normally call Synchronize when it has detected speech. However, if speech has been detected for a long time and the *hRequestSync* event

is set, the engine will always call Synchronize to prevent hanging.

The stream position given as a parameter to Synchronize indicates the point before which the engine will fire any events. SAPI discards its stored audio before this point and thus the engine cannot fire any events or report recognitions before this point. This position does not need to be exactly where the engine is currently recognizing; it can be a point in the stream before the engine fires any events.

UpdateRecoPos is another method that an engine should call regularly during recognition. It informs SAPI of the engine's position in recognizing the stream. This is currently used to ensure that Bookmarks are fired correctly, and to keep the application informed of the recognizer position (using the *ullRecognitionStreamPos* field returned from *ISpRecognizer::GetStatus*).

7.4.1 Pause and auto-pause

It is possible to put an engine into a paused state. This happens for one of three reasons:

- The application called *ISpRecoContext::Pause*.
- A rule, which the application activated as *SPRS_ACTIVE_WITH_AUTO_PAUSE*, was recognized.
- A bookmark event of type *SPBO_PAUSE* has been reached.

When in the paused state, SAPI does not return to the engine from a call to Synchronize or a final recognition. Instead, control is kept by SAPI and it calls back into the engine to inform it of

any grammar changes. that may occur. In fact, these are the results of a normal call to Synchronize or final recognition, except that SAPI waits inside the call until the application resumes the engine. While in the paused state, the engine is still able to Read data if it has another thread running. If the engine also performs sound start/end detection in this thread, it could fire those events to the application.

In the paused state, an application can make grammar and state changes at specific points. Normally grammar changes only occur the next time an engine calls Synchronize. Thus, if an application wanted to make a number of changes, some might occur during one Synchronize call and some in another, which might not produce the best results. Using pause, the application makes changes: after each recognition using Auto-Pause; at a specific point in the stream using Bookmarks; or just as soon as possible using Pause. The application can make as many grammar changes as needed. When the application calls Resume, the engine continues recognizing, without having lost any data, but with all the grammar changes having been reported.

7.5 Events and Recognitions

7.5.1 Standard events

In order to report to the application information about what is being recognized, there are several events the engine can report. These indicate for example, that the engine has detected the start or end of speech, or that it has a hypothesis or a completed recognition result. The main events used to report the progress of recognition are as follows:

- Sound Start. Used to indicate that the start of some speech-like sound has been detected. Reported by calling `AddEvent` with event type `SPEI_SOUND_START`.
- Sound End. Used to indicate that the end of some speech-like sound has been detected. Reported by calling `AddEvent` with event type `SPEI_SOUND_END`.
- Phrase Start. Used to report the start of some speech that the engine recognizes as an utterance matching the currently active grammar. Reported by calling `AddEvent` with event type `SPEI_PHRASE_START`.
- Final recognition. Used to return results of the recognition of an utterance. Reported by calling [Recognition](#).
- False recognition. Used to indicate that the engine attempted recognition of the utterance but rejected it on the basis of low confidence scores, inability to find a valid path, etc. Indicated by calling `Recognition` with the `eResultType` having the `SPRT_FALSE_RECOGNITION` flag set.
- Hypothesis. Used to report a partial recognition of the utterance. Indicated by calling `Recognition` with the `fHypothesis` flag set.

AddEvent takes as parameters an SPEVENT structure, and an SPRECOCONTEXTHANDLE which should be set to NULL. The SPEVENT has the following fields:

```
SPEVENTENUM    eEventId;  
SPEVENTLPARAMTYPE elParamType;  
ULONG          ulStreamNum;  
ULONGLONG     ullAudioStreamOffset;  
WPARAM        wParam;  
LPARAM        lParam;
```

The *elParamType* should be set to SPET_LPARAM_IS_UNDEFINED and the *lParam* and *wParam* are set to NULL to display no extra information is returned with these events. The *ulStreamNumber* can also be set to 0 as SAPI fills this field in before returning the event to the application. The *eEventId* indicates the type of event (SPEI_SOUND_START, SPEI_SOUND_END, or SPEI_PHRASE_START). The stream position indicates the position in the audio stream where the engine decides this event has happened.

[Recognition](#) is used to send hypotheses and final or false recognitions.

7.5.2 Event ordering

There are various requirements for the chronological ordering and stream positions of how these events are reported:

- Sound start and sound end events form a pair. Every sound start call must later have a sound end call with a later stream position.
- Each phrase start event forms a pair with either a final or false recognition. The recognition must be fired later and have stream positions later than the phrase start event. Between the phrase start and recognition some hypotheses can optionally be located.
- Each phrase start/recognition pair must have stream positions inside a sound start/sound end pair, i.e., if part of the stream has been determined to be a phrase it must also be speech. Zero, one, or more than one phrase start/recognition pairs can sit between a sound start/end. Different phrase start/recognition pairs cannot overlap. If part of the stream is determined to be in one utterance, it cannot belong to other utterances.
- Although the phrase starts and recognitions must have stream positions within sound start/end events, the actual time sequence of the firing of these events can vary. For example, if an engine has an independent speech detector, which can determine the end of speech before the recognition has completed, it can fire the sound end event before a recognition event.

7.5.3 Other events

There are several other event types the engine can fire to indicate other information to SAPI and applications:

- SPEI_ADAPTATION is used to indicate that the engine is ready to receive more text adaptation data (See [Language model adaptation](#)).
- SPEI_REQUEST_UI is used to request a display of one of

the engine's UI components. Both these events are called with a stream position that is either valid or zero (See [User-Interface](#)).

- SPEI_INTERFERENCE is used to indicate to applications that there is a problem with the audio stream or user speech. This event is called with the LPARAM set to one of the SPINTERFERENCE values to indicate the nature of the problem (no signal, clipping, user speaking too fast etc). Applications can choose whether to respond to these events.
- SPEI_SR_PRIVATE is an event type that engines use for engine-specific communication with applications. Engines should include some unique identifier in the data sent with the event to distinguish the use of this event from another engine's. The SPRECOCONTEXTHANDLE parameter in the AddEvent call can also be used to send this event to a particular RecoContext, rather than to all contexts.

7.6 Completion of processing

An engine should continue recognizing as long as data is available and it is not signaled to stop. Typically, after an engine reports a recognition, it checks for grammar changes and continues reading data and recognizing.

The Read call indicates that the engine should finish recognition when there is no more data to read from the stream. The engine should finish processing the data it has, sending events and recognitions as necessary, and return from RecognizeStream. A Win32 event, passed as the *hExit* parameter to RecognizeStream, which is set to indicate that the recognizer should exit immediately, without necessarily reading all the data. This condition is also indicated by Recognition or Synchronize returning S_FALSE rather than S_OK.

Recognition is complete for one of several reasons:

- All active rules in the applications connected to the engine were deactivated.
- An application set the recognition state (with SetRecoState) to inactive or inactive with purge.
- No data left in the stream (e.g., for wav file streams).
- An error or buffer overflow in the stream reading.

Note that the default model in SAPI is for an engine to continue recognizing unless it is explicitly informed to stop. An example of this would be a desktop application where everything the user says is being listened to and acted upon. For some systems

an utterance-by-utterance method is required, where recognition stops after each thing the user says. This is best implemented by the application using the auto-pause feature when it activates rules. Then the engine pauses after each recognition and the application can process information and terminate recognition by deactivating all active rules.

8 Recognition Results

In this chapter, the process of returning results is explained in more detail.

8.1 Recognition Call

The Recognition call itself takes a pointer to the following structure as a parameter:

```
typedef struct SPRECORESULTINFO
{
    ULONG          cbSize;
    SPRESULTTYPE  eResultType;
    BOOL          fHypothesis;
    BOOL          fProprietaryAutoPause;
    ULONGLONG     ullStreamPosStart;
    ULONGLONG     ullStreamPosEnd;
    SPGRAMMARHANDLE hGrammar;
    ULONG         ulSizeEngineData;
    void          *pvEngineData;
    IspPhraseBuilder *pPhrase;
    SPPHRASEALT   *aPhraseAlts;
    ULONG         ulNumAlts;
} SPRECORESULTINFO;
```

These fields are set as follows:

- *cbSize* is simply size of (SPRECORESULTINFO).
- *eResultType* is either SPRT_CFG, SPRT_SLM, or SPRT_PROPRIETARY. The SPRT_FALSE_RECOGNITION flag is set to indicate a false recognition
- *fHypothesis* indicates whether the result is a hypothesis or final recognition.
- *fProprietaryAutoPause* is set to FALSE unless using auto-pause with proprietary grammars.
- *ullStreamPosStart/End* provides the start and end positions of the result. This informs the application of the position of the result, and controls what audio data is retained in the result. The start position must be equal or later than the stream position reported for the phrase start corresponding to this recognition, and later than the stream positions reported in previous Synchronize calls.
- *hGrammar* is set to the grammar handle for dictation or proprietary results, and NULL for CFG results.
- *pvEngineData* returns arbitrary data of size *ulSizeEngineData* with the result. This data is used for generating alternates if an [alternates analyzer](#) object is being used.
- *pPhrase* contains the main information about the result. This is created in differently for CFG, dictation, or proprietary results. This can also be NULL for false recognitions if the engine has no phrase to report. See below for details on how to create.
- *aPhraseAlts* is an array containing *ulNumAlts* alternates for this recognition. These are optional (See [Returning alternates in a Recognition](#)).

8.2 Dictation Phrases

The standard method used to construct a phrase builder object to hold a dictation result is as follows:

1. The engine creates an SPPHRASE structure using the needed information.
2. Then the engine uses CoCreate to create an SpPhraseBuilder object (CLSID_SpPhraseBuilder).
3. The SPHRASE information is added with IspPhraseBuilder::InitFromPhrase.

The SPPHRASE has the following fields:

- *cbSize*. This is the size of (SPPHRASE).
- *LangID*. The LANGID of the result. *ullGrammarID*, *wReserved*, *ftStartTime*, *ulRetainedSizeBytes*, *ullAudioSizeTime*. Set by SAPI and can be left as 0.
- *ullAudioStreamPosition* and *ullAudioSizeBytes*. Indicate the position of the result in the audio stream. The position is in bytes, relative to the start of the stream. The part of the stream spanned by this phrase must be the same as or less than the range in the *ullStreamPosStart/End* fields in the SPRECORESULTINFO.
- *Rule*. Set to zero for dictation results apart from the *ulCountOfElements*, which must be filled in with the number of words in the result.
- *pProperties*. The array of semantic properties, which will be NULL for a dictation result.

- *pElements*. The array of actual words in the result.
- *pReplacements*. This holds an array of size *cReplacements* of ITN text replacements the may fill in (See [Inverse Text Normalization \(ITN\)](#)).
- *SREngineID*. An arbitrary GUID that could be the CLSID of the engine, for example.
- *pSREnginePrivateData*. This enables arbitrary engine-specific data to be returned with the phrase object, of size *ulSREnginePrivateDataSize*.

The words of the result are represented by an array of SPPHRASEELEMENT structures in the *pElements* field. The rule *ulCountOfElements* field indicates the number of words. The elements are filled in either directly before the *InitFromPhrase* call, or afterward with an *AddElements* call.

Each SPPHRASEELEMENT contains the following information:

- Result times: *ulAudioTimeOffset* and *ulAudioSizeTime* and retained audio format details, *ulRetainedStreamOffset* and *ulRetainedSizeBytes*. These are all filled in by SAPI and are set to 0.
- Audio stream start position and size for this word: These are relative to the start stream position of the parent phrase. These can be left 0, although the application then cannot obtain word position information.
- Display text, lexical form, and pronunciation information for the word. The display text must be set so that the application displays the result. Optionally, the lexical form and pronunciation can also be set.
- Display attributes information. This is of type *SPDISPLAYATTRIBUTES* and provides information to SAPI

about how to format the result. For European languages, this would usually be set to `SPAF_ONE_TRAILING_SPACE` so that each word is printed with a space between it.

- RequiredConfidence. This is filled in by SAPI.
- Actual and SR confidence. (See [Confidence Scoring and Rejection](#))

Once the PhraseBuilder has been filled in, it passes to SAPI as the *pPhrase* field in the Recognition call.

Results for proprietary grammars are completed in the same way as dictation results.

8.3 CFG Phrases

The engine does not directly create a phrase object for a CFG result, but it calls `ParseFromTransitions`. The engine provides to this method information about the words in the result, and SAPI parses the active rules to fill in semantic property and other information correctly. The engine then passes the returned phrase builder object to SAPI as the *pPhrase* pointer in `Recognition`.

- `ParseFromTransitions` uses the `SPPARSEINFO` structure as a parameter containing the following information *cbSize*. Set to size of (`SPPARSEINFO`).
- *hRule*. The handle of the top-level rule this result refers to.
- *ullAudioStreamPosition* and *ulAudioSize* indicate the position of the result in the audio stream. The position is in bytes, relative to the start of the stream. The part of the stream spanned by this phrase must be the same as or less than the range given in the *ullStreamPosStart/End* fields in the `SPRECORESULTINFO`.
- *pPath*. An array, of size *cTransitions*, of `SPPATHENTRY` structures.
- An `SREngineID` GUID that can be used by the application to identify the engine.
- Optional private engine data *pSREnginePrivateData* of size *ulSREnginePrivateDataSize*.
- A flag *fHypothesis* indicating whether the result is to be used for a hypothesis or final recognition.

The SPPATHENTRY array contains information about the words in the result. Each word transition needs an entry in the result. The engine does not include rule or epsilon transitions; ParseFromTransitions is able to process the result without these. Each SPPATHENTRY contains a transition ID for the word transition, and an SPPHRASEELEMENT structure, which is filled in the same way as dictation results. Neither the display text, lexical form nor pronunciation needs to be filled in for word transitions. ParseFromTransitions will automatically do this.

There should also be entries for any special transitions:

- For a wildcard transition, the engine sets the transition ID to the value it received for the wildcard transition, and fills in the phrase element information as it would for a normal word.
- For a dictation or text-buffer transition the engine includes an SPPATHENTRY for each word recognized in the dictation or text-buffer. The display text needs to be set to the text for each word. Optionally, the lexical form and pronunciation can be set also.

8.4 Confidence Scoring and Rejection

8.4.1 Word Confidence

It is possible for confidence score information to be included in recognition results. On each phrase element there are two confidence fields that the engine can set. These have both a Confidence (three-level) field and an SREngineConfidence (floating-point) field. If the engine does not explicitly set any of these values, SAPI will try and produce reasonable default values for them. It will produce the Confidence values by averaging the levels for each of the words in the phrase or property, and it will set the SREngineConfidence values to -1.0.

The first, ActualConfidence, is a three level value to indicate low, medium or high confidence (SP_LOW_CONFIDENCE, SP_NORMAL_CONFIDENCE, SP_HIGH_CONFIDENCE for C/C++, or of type [SpeechEngineConfidence](#) for OLE automation). This is designed to give applications a simple, and engine-independent, confidence value.

The second value, SREngineConfidence is a positive floating-point value. This can be used by engines to give more detailed confidence information, but is not necessarily engine-independent. SAPI defines that this value should be positive, with zero indicating the lowest confidence. It can be used to optimize an application's performance with a specific engine. Using this value will improve the application with a particular speech engine but more than likely will make it worse with other engines and should be used with care. This value is more useful with speaker-independent engines because it allows a large corpus of recorded usage to correctly optimize the overall accuracy of the application. See Confidence Scoring and Rejection in SAPI Speech Recognition Engine Guide for additional details. If this field is not being used, the engine sets

this confidence to -1.0.

8.4.2 Property and Rule Confidence

It is also possible for confidences to be associated with rules and semantic properties in CFG results. This application looks at confidence on individual words rather than at confidence on groups of words. The confidence for a rule is the overall confidence for all the words in the phrase contained within that rule. Thus, for the top-level rule, this gives an overall confidence for the whole phrase. The confidence for a semantic property is the confidence for all the words within the rule, if the property is on a rule or rule reference; or the confidence for the word, if the property is on a word transition.

It is possible for the engine to override these settings if it has an alternative method of estimating phrase confidences. Since these fields cannot be directly manipulated on the ISpPhraseBuilder interface, it is necessary to convert them back to an SPPHRASE structure first. This is done using the following sequence of actions:

- Calling GetPhrase on the SpPhraseBuilder object to get an SPPHRASE.
- Modifying the chosen fields in the SPPHRASE (Note this may require casting the fields away from const to do this).
- Calling InitFromPhrase on the original SpPhraseBuilder to set the modified phrase information in the object.
- The engine can then use the SpPhraseBuilder in Recognition calls.

This method can also be used to override other information in a CFG result phrase after ParseFromTransitions has been called.

8.4.3 Required Confidence and Rejection

Each transition on a CFG contains a RequiredConfidence field. This is set to one of three values like the ActualConfidence field. This field is set to SP_NORMAL_CONFIDENCE by default, and is changed in the XML grammar by preceding words with “+” or “-”. The purpose of this field is to indicate how much confidence in the recognition an application requires for the result to be returned. In principle, if the engine has a result and the confidence of any of the words in the phrase is lower than the required confidence, the engine should reject the result. Note that the engine may have a different mechanism for rejecting phrases so SAPI does not enforce this particular confidence. The engine should, however, acknowledge the required confidence fields if possible.

Most applications will listen for and act only upon final recognitions (SPEI_RECOGNITION events), and probably do not analyze the confidence scores of these results. Engines send these events only after they have determined that the confidence is high enough to accept the result.

When a result is rejected, the engine sends a false recognition rather than a final recognition. A false recognition can include exactly the same phrase information as a final recognition, or it can have a NULL phrase. Returning a phrase with a false recognition could be useful to applications so that they can analyze why the phrase was rejected, and recover the audio from the phrase. Advanced applications can perform their own rejection, and thus could look at both final and false recognition events and analyze the confidence scores returned in each

result.

8.4.4 Ambiguous Results

Sometimes the words that have been recognized may match more than one possible path, either with dictation or CFG grammars. SAPI does not currently provide a means to resolve ambiguity or send the result to multiple contexts, so the engine must select one path to return the result as. The guidelines for this are as follows:

- If several CFG paths match the recognized words, set the result for the most recently activated rule. This should mean that applications with focus should receive the rule over those in the background.
- If the rule matches both a dictation and a CFG, pick the CFG (unless the path has a very low CFG weight score indicating that the dictation path should be chosen).
- If several dictations are active, pick the most recently activated dictation.

Neither engines nor SAPI can determine what the user meant to say. Applications are encouraged to try and avoid potentially ambiguous grammars.

8.5 Inverse Text Normalization (ITN)

For dictation results, it is possible for the SR engine to specify a normalized form as well as the raw text of the recognized words. To accomplish this, add one or more SPPHRASEREPLACEMENT structures into the result phrase and use either `AddReplacements` or directly set the *pReplacements* and *cReplacements* fields in the SPPHRASE. It is possible to have more than one replacement because each can refer to a sub-set of the full text.

SAPI does not provide an automatic inverse-normalization (ITN) facility. However, it does provide a mechanism for engine vendors to write a grammar describing their ITN rules which can be automatically parsed.

SAPI provides an object `SpITNProcessor` that implements `ISpITNProcessor`. The engine can use `CoCreate` to create this (`CLSID_SpITNProcessor`), and then call `LoadITNGrammar` on this object. The engine must pass in the `CLSID` of an object that implements the `ISpCFGInterpreter` interface. Engine vendors must implement this object, which has two methods. `InitGrammar` is called by SAPI when the `LoadITNGrammar` is called. This method should load a SAPI binary grammar containing the ITN information and return it as serialized data.

The ITN grammar that the engine implements should have rules for each of the phrase fragments that need to be normalized. These rules need to have the attributes “`TOPLEVEL=ACTIVE`” so that SAPI will activate them for parsing, and “`INTERPRETER=1`” so that SAPI calls the engine’s `ISpCFGInterpreter` object when

the rule is fired.

When the engine has a result phrase to normalize, ITNPhrase is called on the SpITNProcessor object. This will parse the grammar, and if a fragment of text matches any of the rules in the grammar, ISpCFGInterpreter::Interpret will be called. This method will be passed in a phrase containing the result text and matching rule information, and an ISpInterpreterSite pointer. The engine's implementation of this should look at the reported text and call ISpInterpreterSite::AddTextReplacement to add the normalized text.

Because the ITN grammar is a standard SAPI grammar, it is possible to use all the features of such a grammar. For example, it may be useful to include the normalized text as a semantic property. This will be included in the phrase passed to Interpret so that this method could just set the replacement text to be the property string. For example, the following is a simple grammar rule to convert dollar and pound signs to their respective symbols:

```
<RULE NAME="currency_type" INTERPRETER="1"
TOPLEVEL="ACTIVE">
    <L PROPNAME="CURRENCY_TYPE"
PROPID="CURRENCY_TYPE">
        <P VALSTR="$">dollar</P>
        <P VALSTR="$">dollars</P>
        <P VALSTR="£">pound</P>
        <P VALSTR="£">pounds</P>
    </L>
</RULE>
```

When trying to reorder symbols (e.g., so that the dollar sign comes before the currency amount), more sophisticated processing would have to be done in Interpret.

8.6 Interpreters

Sometimes an application will want extract semantic information from CFG results after processing. Using this process, it is possible for a grammar to be associated with an object. The object would implement `ISpCFGInterpreter`, and would be called each time a result matching a rule in the grammar was passed into `ParseFromTransitions`. Then, rather than filling in the property information statically from the grammar, this object would do so with `ISpInterpreterSite::AddProperty`.

This process is basically invisible to the SR engine. It recognizes the words in the grammar and calls `ParseFromTransitions` and then `Recognition`. The one difference is that `GetTransitionProperty` may report that there are no properties on a transition, which are later added into the result in the `ParseFromTransitions` calls. Rules with an associated interpreter have the `SPRAF_Interpreter` flag set in the `SPRULEENTRY Attributes` field.

9 Alternates

There are two ways the engine can supply alternates back to the application. It either supplies the alternates directly in the Recognition call, or produces alternates using an alternates analyzer object.

9.1 Returning alternates in a Recognition

The field *aPhraseAlts* in the SPRECORESULTINFO structure can contain an array of alternates. The size of the array is given by *ulNumAlts*. Each entry in the array is a SPPHRASEALT structure. This contains an ISpPhraseBuilder structure generated directly or from ParseFromTransitions, depending on whether the alternate is for a CFG or dictation result. This entry contains the full phrase of the alternate, not just the words that are different from the main result.

- *ulStartElementInParent* and *cElementsInParent* indicate which words in the main phrase are different in this alternate.
- *cElementsInAlternate* indicate which words these are being replaced within this alternate.
- *ulStartElementInAlternate* is not needed because it would be the same as *ulStartElementInParent*.
- *pvAltExtra* and *cbAltExtra* fields can be used to add engine-specific extra data to the alternate.

The number of alternates that an application requires is obtained with either `GetMaxAlternates` or `GetContextMaxAlternates`. The first gives the maximum number of alternates for any rule, and the latter, the maximum for any `RecoContext`.

Sometimes when multiple applications are using the shared recognizer and all have rules active, the engine may generate alternates from different applications. Currently, SAPI cannot process such alternate lists and it is not possible for the engine

to send alternates referring to different RecoContexts in the main result. To simplify detecting which context an alternate belongs to, the engine can use IsAlternate, using the handle of the main and alternate top-level rule. SAPI returns S_OK if the alternate is valid and S_FALSE otherwise.

9.2 Alternates Analyzer

The engine can also implement a separate alternates analyzer COM object. This must implement the interface `ISpSRAlternates`. The CLSID for this object is stored in the engine's object token in the `AlternatesCLSID` string value. This type of analyzer can only be used for dictation alternates, not for CFG alternates.

When an application asks for alternates with `ISpRecoResult::GetAlternates`, and if the engine has already supplied alternates within the Recognition call, SAPI supplies these to the application. Otherwise, if the engine has an alternates analyzer object, SAPI creates this and requests the alternates from it, using `GetAlternates`.

As well as passing in the main `ISpPhrase` result and the number of alternates requested, the alternates analyzer is also receives any extra data the engine supplied in the *pvEngineData* field of the `SPRECORESULTINFO` structure in the Recognition call. The engines stores serialized lattice information or similar information here, which the alternates analyzer uses to generate the alternates.

The analyzer is also passed a pointer to the `IspRecoContext`, requesting the alternates. The analyzer can use this to query for an engine-specific extension interface (See [Engine extensions](#)) to make a private call to the main engine object. These objects may be in different processes. Because the result object can be serialized and then re-created later, it may not be possible to provide a `RecoContext` referring to the SR engine. In this case `NULL` is passed in.

If a user makes a correction after looking at the alternates, the application can commit that alternate with `ISpRecoResult::Commit`. `Commit` is called on the analyzer, giving the engine the `SPPHRASEALT` that the application selected. The engine can use this to perform adaptation to improve subsequent recognition performance.

10 User-Interface

There are a variety of user-interface components an engine supplies along with the main engine object itself. Examples of these are user enrollment, microphone set-up, adding and removing user exception words and engine properties. Engines should not create UIs directly. Instead, SAPI provides mechanisms for engines to describe what UI components they have and to request the display of these components.

The engine's object token contains the details of the UI that an engine supports. The token can contain a UI subkey. Within this key can be subkeys for each component type the engine implements. For example the Sample SR engine (Token: **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Reco** has a key UI\AddRemoveWord. This contains a value CLSID, which holds the CLSID of the class to be created when this UI component is displayed. The engine setup installs and registers this class. This class must implement the interface ISpTokenUI.

An SR engine can display a certain UI by calling `AddEvent` with an *eEventId* of `SPEI_REQUEST_UI`. The name of the UI component requested is in the *IParam* field, with the *eIParamType* set to `SPET_LPARAM_IS_STRING`. To cancel the UI request, the engine calls this method again with a `NULL IParam`.

This event is sent to all applications connected to the engine that are waiting for the event. An application can either ignore the engine's request, or respond to it. An application can ask to create an engine UI at any time, not just after it has received a request.

An application determines if an engine object token supports a particular UI component by calling `IsUISupported`, either on the `ISpObjectToken` itself, or on the `ISpRecognizer`. If the object token has a CLSID for this type of UI, SAPI will create the object and call `ISpTokenUI::IsUISupported` on the engine's UI class. To actually display the UI, the application calls `DisplayUI`, which leads to a call to `ISpTokenUI::DisplayUI` on the engine's class.

Both `DisplayUI` and `IsUISupported` on `ISpTokenUI` take an `IUnknown` parameter *punkObject*. If an application calls these methods on the `ISpRecognizer`, this parameter will point to this `ISpRecognizer` object. The UI component finds the `RecoContext` and makes a private call to the main SR engine object. If the methods are called by the application directly from the object token, this parameter may be `NULL` or point to a different object. If the engine requires its parent SR engine to be created in order to run the UI, it may not be able to display the UI.

11 Engine extensions

SAPI 5 aims to provide interfaces for all the main objectives on the SR engine. However, there are often additional features that SR engines can implement when connected to certain applications. For example, there are a number of places where engine-specific data can be passed to an application. And, there are engine-specific data fields on results phrases, the SPEI_SR_PRIVATE event, methods to support proprietary grammars, new object token attributes and properties etc. There is also a mechanism for an engine to implement additional interfaces for an application to use if it wants to be specifically connected to a particular engine. To do this, the engine implements a new COM object. The CLSID for this object is stored in the engine object token, in the string value RecoExtension. This object can implement any interfaces that the engine vendor wants to implement. To use this object, an application must use QueryInterface on the RecoContext for an interface that it supports. SAPI then creates the engine extension object as an aggregate, and queries it for the interface. The application is then free to call any methods on this interface and the methods will be passed to the extension object. For example, the Sample SR Engine implements the interface ISampleSREngine, with which an application can use QueryInterface to call RecoContext.

The extension object can make calls to its main SR engine class. It does this by querying on the RecoContext for the _ISpPrivateEngineCall interface. The RecoContext is the outer unknown of the extension object. The sample engine does this (in srengext.h) by:

```
hr = OuterQueryInterface(IID__ISpPrivateEngineCall, (void  
**) &m_pEngineCall);
```

This query for `_ISpPrivateEngineCall` must be done in the constructor or ATL `FinalConstruct` of the engine extension object. It cannot be done later.

This gives a pointer to an `_ISpPrivateEngineCall` interface. See 11.1 Important notes regarding handling interface pointers in an SR engine extension. Both methods send a serialized chunk of data to the engine. The difference between these two methods is that `CallEngineEx` can return a chunk of data back from the engine that is larger than the data passed in.

It is possible for applications to know what engine they are connected to. If the engine is not one that supports the extension interface, the `QueryInterface` will fail. Alternatively, the application can look at the CLSID of the engine using `ISpRecognizer::GetStatus`. Thus, applications that plan to use special features of a particular engine will be able to detect if they do not have this interface and either perform with limited functionality or fail gracefully. This provides maximum interoperability between engines and still enables applications to take advantage of engine-specific information.

11.1 Important notes about COM interface pointer handling by the SR extension aggregates

An SR engine extension can only query for the IID_ISpPrivateEngineCall during creation of the object (for example, in the FinalConstruct call of an ATL object). RecoContext makes the interface visible only during this time. This interface does not use AddRef for the QueryInterface call, so the extension should never call Release on this interface.

Because the SR engine extension is created as a COM aggregate, if it were to hold a reference to its outer IUnknown interface (the IUnknown of the ISpRecoContext interface), it would prevent the context from ever being released. If the extension object calls QueryInterface on the ISpRecoContext interface for any interface except for _ISpPrivateEngineCall, Release() must be called immediately on the outer unknown object to prevent a self-reference. Continue to use this interface even though Release has already been called. This is because the lifetime of the ISpRecoContext interface and the extension object are guaranteed to be the same.

The sample SR engine extension shows an example of how to handle these cases in FinalConstruct.



Object Tokens and Registry Settings

1 Contents

[1 Contents](#)

[2 Summary](#)

[3 Overview: Tokens, Categories and the Registry](#)

[3.1 TOKENS](#)

[3.2 CATEGORIES](#)

[3.3 TOKENIDS AND CATEGORYIDS](#)

[3.4 USER DEFAULTS](#)

[3.5 TOKEN ENUMERATORS](#)

[4 Using Tokens and Categories](#)

[4.1 HELPER FUNCTION EXAMPLES](#)

[4.2 ENUMERATING TOKENS](#)

[4.3 INSTANTIATING AN OBJECT FROM A TOKEN](#)

[5 Tokens and Categories For Engine Developers](#)

[5.1 MAKING RESOURCES AVAILABLE THROUGH SAPI](#)

[5.2 ASSOCIATING FILES WITH TOKENS](#)

[5.3 INSPECTING UNDERLYING KEYS OF A TOKEN](#)

[5.4 CREATING NEW KEYS IN THE REGISTRY](#)

[6 Registry Settings](#)

[6.1 CATEGORY: VOICES](#)

[6.2 CATEGORY: RECOGNIZERS](#)

[6.3 CATEGORY: RECOPROFILES](#)

[6.4 CATEGORY: AUDIOINPUT](#)

[6.5 CATEGORY: AUDIOOUTPUT](#)

[6.6 CATEGORY: APPEXICONS](#)

[6.7 CATEGORY: PHONECONVERTERS](#)

[6.8 USERLEXICONS](#)

[7 Index of Tables](#)

2 Summary

This document is intended to help developers of speech-enabled applications discover and use resources (Voices/Recognizers) on a computer that has SAPI installed. A speech-enabled application is one that attempts to either recognize or synthesize speech. Developers of speech recognition (SR) and speech synthesis (Text to Speech or TTS) engines make their resources available to applications.

This spec answers the following questions:

- What are Tokens and Categories in SAPI?
- Where is information about tokens stored in the Registry?
- How does an application find tokens and initialize resources (i.e., Voices or Recognizers) from them?
- What are the SAPI-defined attributes that engines should document in the registry?
- How are files associated with tokens?

Note:

The Speech SDK documentation section on Object Tokens, which provides a complete description of the **ISpObjectToken** and **ISpObjectTokenCategory** and their methods, complements this document.

3 Overview: Tokens, Categories and the Registry

3.1 Tokens

A **token** is an object representing a resource that is available on a computer, such as a voice, recognizer, or an audio input device. A token provides an application an easy way to inspect the various attributes of a resource without having to instantiate it. The Vendor of a Recognizer, and Gender of a Voice are examples of attributes of resources. In many cases, applications should use SAPI-provided helper functions for common scenarios. For example, an application can use the **SpCreateBestObject** helper function to rapidly create the object, given a certain type of resource. The application can also query for tokens meeting certain criteria without using the helper function. To do this, the application calls the **EnumTokens** method on the **ISpObjectTokenCategory** interface to get an enumerator, and inspect the tokens in the enumerator further if it chooses to. Finally, the application selects one of the tokens in the enumerator to instantiate a resource. Once the resource (such as SR Engine) is instantiated, if it implements the **ISpObjectWithToken** interface, then it is handed a pointer to the token that was used to create it. This way, the resource contains a handle to more information about itself.

Conceptually, a token contains the following information:

- The language-independent name. This is the name that should be displayed wherever the name of the token is displayed. It is marked as (Default) in the registry. The implementer of the token may also choose to provide a set of language-dependent names in several languages.
- The CLSID used to instantiate the object from the token.
- A set of Attributes, which are the only set of queryable values in a token. This means SAPI provides a mechanism to query for tokens whose attributes match certain values. Details on how to query for tokens that match a set of attributes are in Sections 4.1 and 4.2.

A token may also contain the following:

- If a token has user interfaces (UIs), such as the properties of a Recognizer or a wizard to customize a Voice to display, then the token will also contain the CLSID for the COM object used to instantiate each type of UI.
- The set of Files from which SAPI returns the paths to all the associated files for the token.

SAPI stores information about tokens in the registry. A token is represented in the registry by a key and the key's underlying keys and values. When an application queries SAPI for tokens of all the female voices on the computer, SAPI will look at the

HKEY_LOCAL_MACHINE\Software\Microsoft\Speech\Voices area. This corresponds to a Category and categories are discussed in the Section 3.2. SAPI searches for tokens that match the criteria (in this case, a voice with the Gender attribute set to female) and uses one of these matching tokens to initialize the voice. The application may also specify a different fully qualified registry path to specify any non-standard (from a SAPI) location in the registry for SAPI to search for a token. In addition to the keys SAPI recommends, the entry for the token may contain any other bits of information that the implementer of the token can store here. In the registry, a token looks like this:

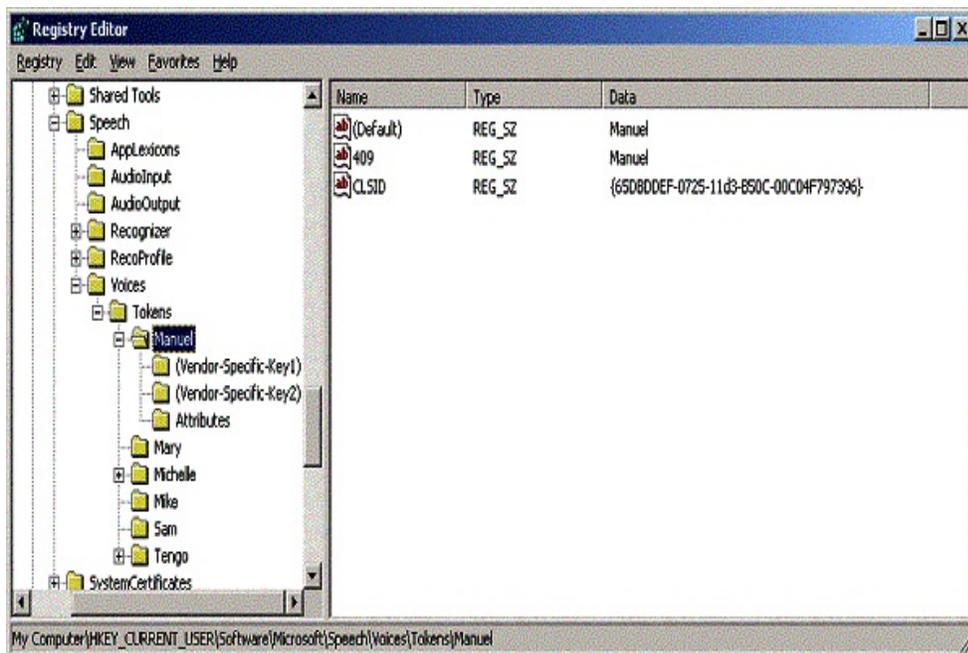
Table 1 Parts of a Token in the Registry

RegKey	ValueName	Sample Value	Comments
SampleTokenKey			Required. This RegKey is the 'SampleTokenKey'.
	(Default)	Joe	Required. Language name. Index of the voice. Name of the voice.
	409	Joe	Name of the voice. Hexadecimal.

	809	Joe	409 is E The be s thes one Lan whi Tok nar no l Ox t Lan
	CLSID	{8021D50E- D93C-4075- 8504- FC4E124D64E9}	Req Sam CLS obje inst the
SampleTokenKey/Attributes			Attr for t are this
	Language	409;809	The be s thes one attri is q See 4 fo expl of e the attri
	Vendor	VoiceVendor	

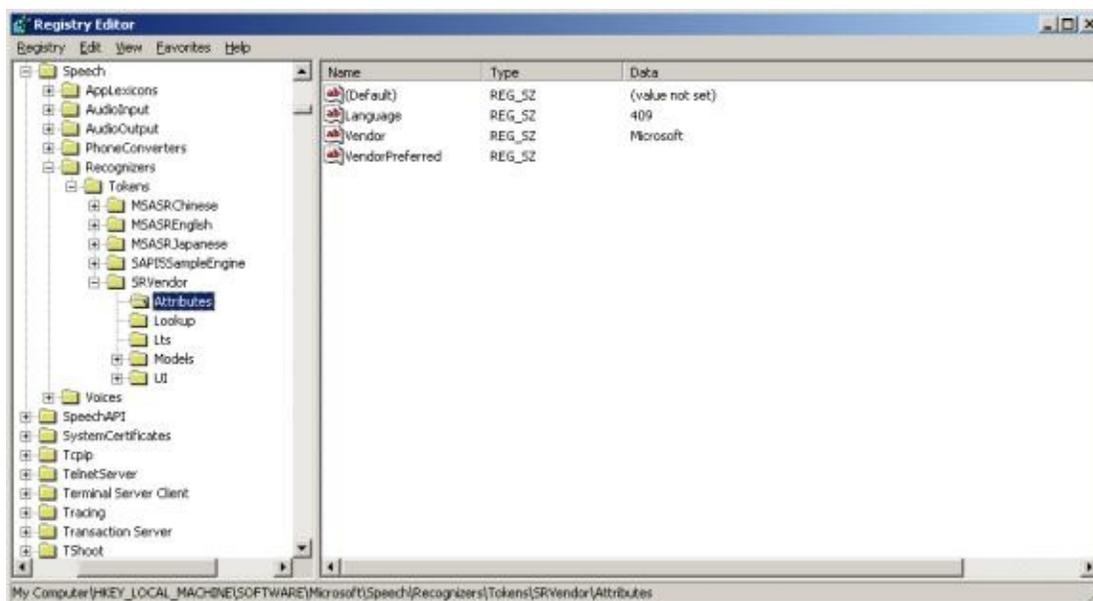
In the registry, this looks like:

Figure 1 A Token Key in the Registry



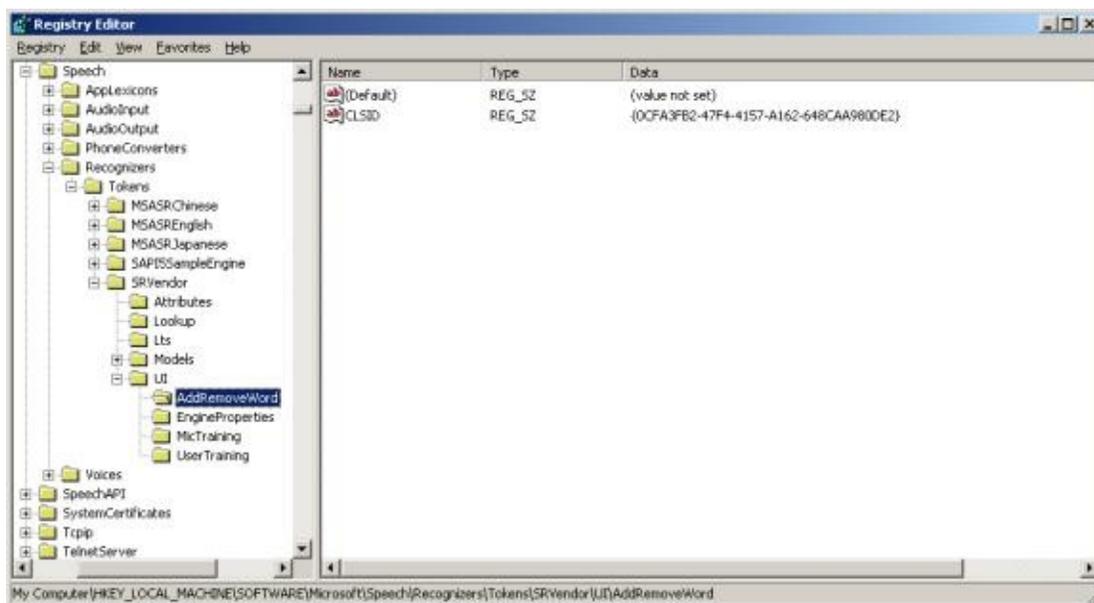
The Attributes key contains all the queryable values for the token. Section 4.2 discusses in detail how an application queries a token.

Figure 2 Attributes of a Token



If the token is capable of displaying UI, then each UI has its own key under the token. Fig 3 shows the token for a Recognizer that supports four types of UI: AddWord, EngineProperties, MicTraining and UserTraining, as well as the CLSID underlying each UI type.

Figure 3 A Token that supports UI has a token for each UI type

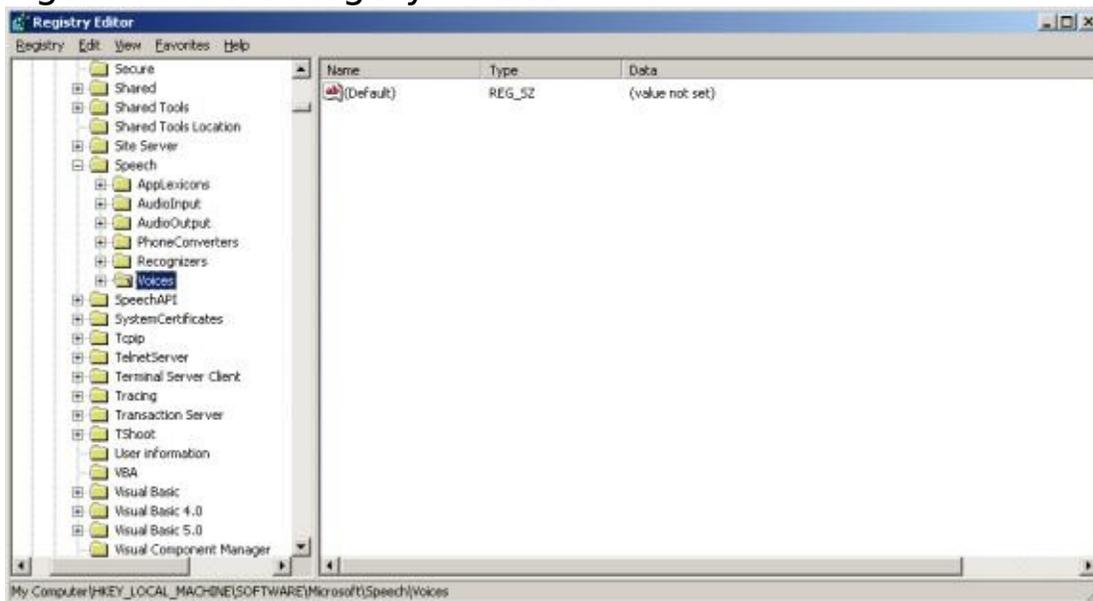


SAPI provides a comprehensive set of helper functions for the common scenarios using tokens. Section 4.1 provides a number of examples. SAPI also provides a way for engines and applications to implement tokens in their own proprietary manner. See Section 3.4 on token enumerators, for further discussion. Sections 4 and 5 explore common scenarios using these interfaces from application and engine coding perspectives.

3.2 Categories

A `ObjectTokenCategory` (hereafter referred to as category) is the highest level of grouping of registry entries in SAPI. A category is a class of tokens (or of resources, since each token represents an actual resource on the computer). Intuitively, a category is a type of SAPI resource. It is represented in the registry by a key containing one or more token keys under it. It is created and manipulated using helper functions such as `SpCreateDefaultObjectFromCategoryID` or methods on the `ISpObjectTokenCategory` interface. Please refer to the SAPI documentation for details on either of these. Examples of categories are Recognizers and Voices. Figure 4 shows the default SAPI categories, with the category Voices selected.

Figure 4 The Category Voices



SAPI organizes tokens in the Registry under seven categories.

By default, the following tokens for six of the SAPI categories are located under `HKEY_LOCAL_MACHINE\Software\Microsoft\Speech` (HKLMS). This is where all system-specific SAPI keys and values should be stored as recommended by Windows Application guidelines. Examples include settings

and files for Voices and the Recognizers (also known as Speech Recognition engines) installed on a computer, as shown in Figure 1.

1. Voices
2. Recognizers
3. AppLexicons
4. AudioInput
5. AudioOutput
6. PhoneConverter

The tokens for the other category, Recoprofiles, are located under **HKEY_CURRENT_USER\Software\Microsoft\Speech** (HKCUS).HKCUS also contains all other user-specific keys and values in the registry, such as user defaults for Voices, Recognizers, as well the location of the user lexicon file.

Categories contain the following items:

- A single key called Tokens, and the keys for the tokens that belong to that category under it. For example, the Voices category has a key for the voice called *Manuel*. All the keys and values for *Manuel* are located under HKLMS/Tokens/Manuel.
- Keys for token enumerators. A token enumerator is a special type of token that generates other tokens for the same category. This token provides a way for Vendors to generate tokens that are generated in non-standard way, such as, reading data from a stored file stored. Those engine vendors following SAPI guidelines for registering resources (Sections 4 and 5) can safely

ignore these and regard them as generators for another set of tokens. Section 3.4 explains token enumerators in more detail.

3.3 TokenIDs and CategoryIDs

A **CategoryID** uniquely identifies a category in the registry. For SAPI defined categories they take the form of

**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\
{CategoryName}**. For example,

**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Recognizers\
for the Recognizers category. All SAPI CategoryIDs should be referenced using
the constants defined in sapi.idl file:**

1. SPCAT_AUDIOOUT
2. SPCAT_AUDIOIN
3. SPCAT_VOICES
4. SPCAT_RECOGNIZERS
5. SPCAT_APPLEXICONS
6. SPCAT_PHONECONVERTERS
7. SPCAT_RECOPROFILES

Similarly, TokenIDs uniquely identify tokens in the registry. For tokens located in SAPI defined categories, they take the form of:

- **CATID\Tokens\TokenKeyName** - a static token from the registry. For example,
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech
- **CATID\TokenEnums\TokenEnumKeyName** - a static token from the registry that represents a token enumerator. This token instantiates a token enumerator used to enumerate dynamic tokens. SAPI uses this for its own implementation of audio input

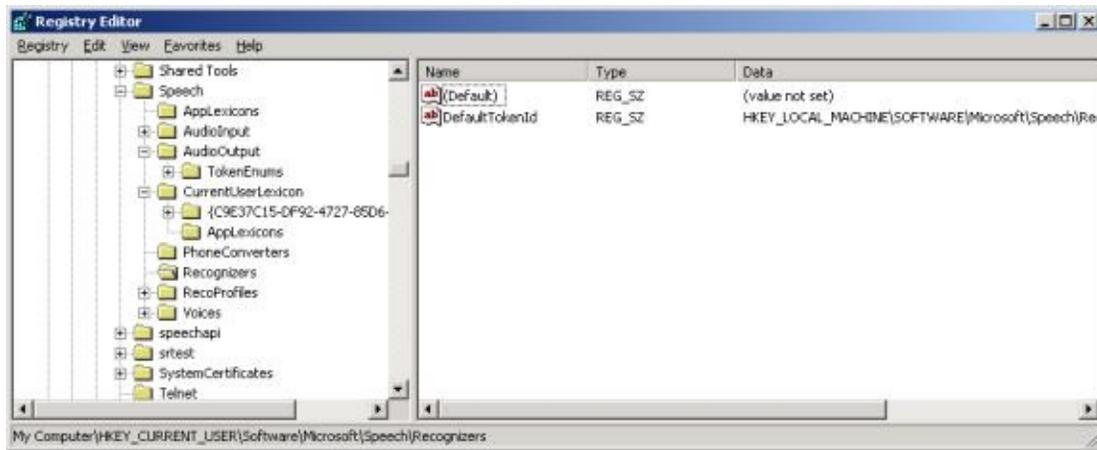
and output to list just the channels available on the computer at runtime. Token enumerators can also read tokens from other areas of the registry, or from remote computers. For example, **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech**

- **CATID\TokenEnums\TokenEnumKeyName** - a dynamic token representing the default token that the specified token enumerator generates. For example, SPDSOUND_AUDIO_IN_TOKEN_ID creates the default Dsound audio in an object. For example:
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech
- **CATID\TokenEnums\TokenEnumKeyNameEnumExtra...** - a specific dynamic token from the specified token enumerator. For example:
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\ Sound Crystal WDM Audio, which generates the Direct Sound Crystal WDM audio object.

3.4 User Defaults

In addition to the category defaults mentioned in Section 3.2, the categories Voices, Recognizers, AudioInput, AudioOutput and RecoProfile, also have user defaults and settings. As shown in Figure 5, these are located in the HKCUS area, under their respective category keys. Section 6 explains each category of tokens. This section also lists out the user-specific entries in the HKCUS and the system-wide entries in HKLMS.

Figure 5 The User category for Recognizers



3.5 Token Enumerators

Note: This section is relevant **only** for Engine or Application developers who need to store tokens in a separate part of the registry or even on the file system, and dynamically enumerate them.

SAPI provides a way for third parties to store their registry settings without following any of the SAPI-recommended guidelines. SAPI can find these tokens as long as the parties have implemented token enumerators. Token enumerators are COM objects that enumerate the necessary entries for the tokens under it. All token enumerators are stored under CategoryName/TokenEnums. Each token enumerator listed under a category needs to have the CLSID of the COM object that implements it under the token enumerator.

The token enumerator

- Must implement the methods Next, Skip, Reset, Clone, Item, GetCount on the IEnumSpObjectToken interface.
- May choose to implement methods SetObjectToken and GetObjectToken on ISpObjectWithToken interface. As mentioned in the end of Section 3.1, these give a resource a handle to the token that was used to instantiate it.

These tokens can be located in a separate part of the registry or somewhere else (possibly on the flusters). It is the responsibility of the token enumerator to return correctly on the above methods so an application does not know the difference between tokens coming from the token enumerator and tokens coming from the SAPI-specific part of the registry.

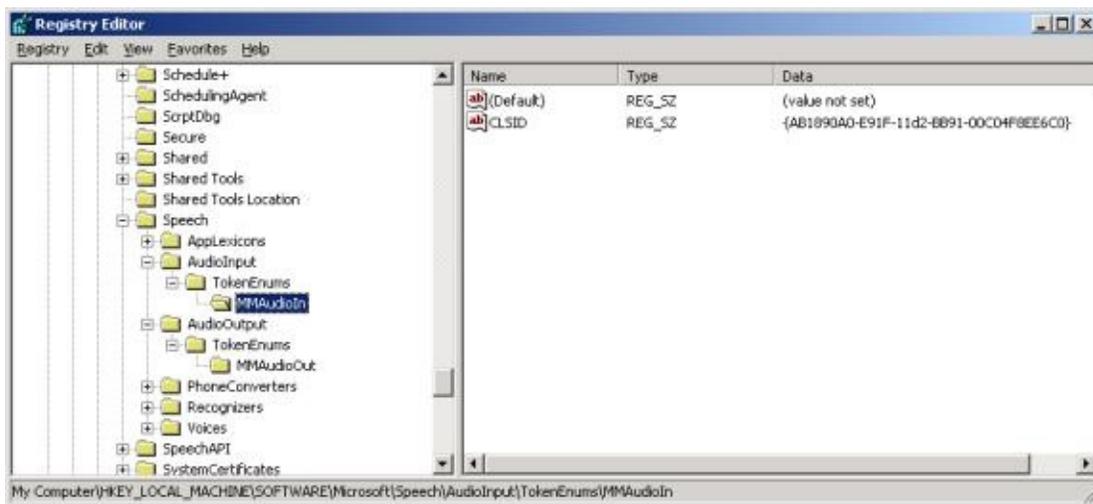
SAPI itself uses token enumerators only for the AudioInput and AudioOutput categories. Refer to Sections 6.4 and 6.5 for more

HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Speech

HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Speech

Figure 6 AudioInput token enumerator in the registry

Figure 6 illustrates how the AudioInput token enumerator looks in the registry.



4 Using Tokens and Categories

4.1 Helper Function Examples

A SAPI 5 application needs to find tokens and instantiate objects that meet certain criteria from the resources available on a computer. Helper functions distributed in the sphelper.h file are the recommended way for applications to interact with tokens and categories whenever possible. Table 3 provides a list of helper functions and the scenarios they address. The helper functions have been broken up into Common Helper Functions and Engine Developer Helper Functions based on likelihood of use. If the specific helper function is not found in either section, refer to the SAPI documentation for the comprehensive listing.

Table 3 Common Helper Functions

Helper Function	Action	Example
<code>SpGetDefaultTokenFromCategoryID</code>	Creates the default token from a CategoryID. The last argument tells SAPI to create the token if it does not currently exist.	<code>CcomPtr<...> hr = SpGetDe &m_cpE</code>
<code>SpFindBestToken</code>	Finds the	<code>CComPtr<</code>

	<p>most appropriate token given a set of required and optional criteria. For details on attribute matching see Section 4.2</p>	<pre>hr = SpFir L"Langua</pre>
<p>SpEnumTokens</p>	<p>Returns a token enumerator containing all tokens meeting a set of required and optional attributes. Tokens in the enumerator are sorted in the order specified in the Section 4.2.</p>	<pre>CcomPtr< hr = SpEnu L"Gender= L"Vendor= , &pEnur</pre>
<p>SpCreateDefaultObjectFromCategoryID</p>	<p>Creates the</p>	<pre>CComPtr<</pre>

	default object in a category, such as <code>AudioInput</code> or <code>Recognizer</code>	<code>SpCreateI &cpVoice</code>
SpCreateBestObject	Instantiates a resource that best matches a set of required and optional criteria. For details on attribute matching see Section 4.2	<code>CComPtr< SpCreateE L"Vendor: &cpVoice</code>
SpCreateObjectFromToken	Creates an object from a token.	<code>CComPtr< CComPtr< /--like las SpFindBe L"Vendor: /--now cre SpCreateC }</code>

Table 4 Engine Developer Helper Functions

Helper Function	Action	Example Helper
<p>SpCreateNewToken</p>	<p>Creates a new object token in the registry with CategoryID, but without specifying a keyname. This creates a token with a GUID as its registry key.</p>	<pre> CComPtr<ISpObj hr = SpCreateNe &cpUserToken); cpUserToken; </pre>
<p>SpGetTokenFromID</p>	<p>Creates a token from a TokenID of an enumerator or a new token if the token does not already exist. The last argument of FALSE tells SAPI not to create the token if it does not already exist.</p>	<pre> CComPtr<ISpObj hr = SpGetToken &cpAudioInTok, </pre>

<p>SpCreateObjectFromSubToken</p>	<p>Creates an object from a subtoken of a token. In this case, the engine token pEngineToken has the Lts key under it, which in turn has a CLSID value under it. This CLSID is used to instantiate the object.</p>	<pre> CComPtr<ISpObj hr = SpGetDefaultTok &m_cpEngineTok ISpLexicon * HRESULT hr = S L"Lts", &m_pLts </pre>
<p>SpGetSubTokenFromToken</p>	<p>Creates a subtoken under a token. This is useful, for example, when an Engine vendor would like to create a subtoken for custom data under its Recognizer token.</p>	<pre> CComPtr<ISpObjec hr = SpGetSubToker L"EngineProperties" </pre>

4.2 Enumerating tokens

The principal tasks related to tokens and categories that an application needs to accomplish are:

- Enumerating tokens
 - Inspecting and instantiating tokens

The two primary ways to enumerate tokens are by the helper function **SpEnumTokens**, or by the method **ISpObjectTokenCategory::EnumTokens**. Both methods allow the caller to specify a category and a set of required and optional attributes. The call then returns a token enumerator containing all the tokens matching those criteria. The method is defined as:

```
HRESULT EnumTokens(  
    [in] const WCHAR *pszCatName,  
    [in, string] const WCHAR *pReqAttrs,  
    [in, string] const WCHAR *pOptAttrs,  
    [out] IEnumSpObjectTokens **ppEnum);
```

When identifying matching tokens under in a category, an application needs to specify a fully qualified category identifier (FQCID). An FQCID is the full registry path to a category, such as

HKEY_CURRENT_USER\Software\Microsoft\Speech\Voices. It is recommended that these categories be referenced using the constants defined in the sapi.idl file below, and not using the full string to minimize typos in commonly used registry paths. SAPI maps the constant to the correct hive in the registry and returns matching tokens from the category. For instance, the SAPI defined **AudioInput** constant (from the sapi.idl file) is:

```
//--- Categories for speech resource management
const WCHAR SPCAT_AUDIOOUT[] =
L"HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Speech\\AudioOutput
```

Similarly, there are constants for the AudioInput, Voices, Recognizer, Applexicon, PhoneConverter, and RecoProfile categories.

An application may also specify a non-standard registry location by simply providing its FQCID, such as

HKEY_CURRENT_USER\\Software\\TTSVendor1\\Speech\\Voices

In both **SpEnumTokens** and **ISpObjectTokenCategory::EnumTokens** the following clauses are permitted in the ReqAttrs and OptAttrs strings, separated by semicolons.

Table 5 Query Operators

Condition	Example	Explanation
Exists	Telephony;Dictation	The valuenames Name and Dictation exist in the list of attributes for this token.
One of	Language=409	At least one of the values of the Valuename Language is 409. There may be other values, like 809, 512 as well.
		Values of Age that are

Not Equals	Age!=Child;Age!=Teen	neither “Child” nor “Teen”.
------------	----------------------	-----------------------------

The tokens are sorted “best matches” first using the following intuitive rules:

1. Only tokens matching the required attributes are returned.
2. Those tokens matching the optional attributes as well will be before those that just match the required attributes.
3. If there are no required or optional attributes (i.e., both are set to NULL), the first token is the default token for that category. If there is a valid DefaultTokenID in HKLMS/Category, that is returned as the default tokenID. If not, if there is a default tokenID in HKCUS/Category, that is returned. If none of these exist, SAPI searches for a DefaultdefaultTokenID in HKLMS/CategoryName, and that is returned.
4. Matching Rules: If a token matches an optional attribute, it gets a score of 1, otherwise, 0 for that attribute. The optional attributes mentioned earlier in the query string are more significant. These scores are concatenated as shown in Table 7. The tokens are then placed in descending order. This is illustrated in Tables 6 and 7.
5. Tokens having the same score are returned in random order in the enumerator.

A call to EnumTokens could look like:

```

CComPtr<IEnumSpObjectTokens> cpEnum;
CComPtr<ISpObjectTokenCategory> cpVoiceCat;

HRESULT hr =
cpTokenCategory.CoCreateInstance(CLSID_SpObjectTokenCategory
const WCHAR Req_Attrs[ ]=L"LanguagesSupported=409";
const WCHAR
Opt_Attrs[]=L"Vendor=VoiceVendor1;Age=Child;Gender=Female";

HRESULT hr = cpVoiceCat->EnumTokens(SPCAT_VOICES,
ReqAttrs , OptAttrs , &cpEnum);
// SPCAT_VOICES is defined in sapi.idl

```

If the following voices are installed on a computer as shown in Table 6:

Table 6 Voices installed on a computer

Voice	Vendor	Age	LanguagesSupported	Gender
Michelle	VoiceVendor1	Child	409; 411	Female
Mary	VoiceVendor1	Adult	409	Female
Jane	VoiceVendor2	Child	409	Female
Frank	VoiceVendor2	Adult	411	Male

Anna	VoiceVendor2	Adult	411	Female
------	--------------	-------	-----	--------

Then the order of the Voices returned in cpEnum will be as shown in Table 7:

Table 7 Scoring of tokens matching optional criteria

Optional Criteria ->	Vendor	Age	Gender	Net Score
Michelle	1	1	1	111
Mary	1	0	1	101
Jane	0	1	1	011

The final order is:

1. Michelle (meets all required criteria, scored 111 on optional criteria)
2. Mary (meets all required criteria, scores 101 on optional criteria)
3. Jane (meets only required criteria, score 11 optional criteria)

If the call to EnumTokens is changed to:

```
HRESULT hr = cpVoiceCat->EnumTokens(SPCAT_VOICES, NULL, NULL,
&cpEnum);
```

and the users default token in HKCUS\Voices\DefaultTokenID is set to:

HKEY_LOCAL_MACHINE\Software\Microsoft\Speech\Voices\T

then the enumerator `cpEnum` will contain all the tokens, with Jane being the first token.

What does SAPI do when `ISpObjectTokenCategory::EnumTokens` is called?

Consider a fictitious category that has both tokens and token enumerators under it. When an application calls the SAPI `ISpObjectTokenCategory::EnumTokens`, the following things happen:

1. SAPI creates an enumerator called `IEnumSpObjectTokens` that can enumerate all the matching tokens from these keys under `HKLMS/Voices/Tokens`.
2. Token enumerators Step (skip this step if not using token enumerators).
 - a. SAPI searches for a `CategoryName/TokenEnums` key. If found, it instantiates a token enumerator from each of the tokens under this key, one by one.
 - b. Each of the token enumerators return an `IEnumSpObjectToken` containing matching tokens under it that is merged with the `IEnumSpObjectToken` created in (i).
3. SAPI applies the required attributes so that the `IEnumSpObject` enumerator contains only those tokens that match these Attributes, then it sorts them according to how well they match the optional attributes (exact rules earlier in Section 4.2).
4. The application searches for an appropriate token and until one is found, it steps through each token, and further checks attributes and strings of each token with `ISpObjectToken` methods `GetData`, `GetStringValue`, and `GetDWORD` (inherited

from ISpDataKey).

5. The application identifies the token it is interested in and calls ISpObjectToken::CreateInstance and QIs the newly created object to see if the newly created object supports the ISpObjectWithToken interface. If it does, SAPI calls ISpObjectWithToken::SetDataKey to give the newly instantiated object a pointer to the token it was instantiated from.

4.3 Instantiating an Object from a Token

Continuing with this example, the application now has a pointer to the enumerator `IEnumSpObjectTokens`. An application may choose to step through the enumerator with the methods `Next`, `Skip` or `Reset` to find an `ISpObjectToken` that best meets its needs. Assume that the application is searching for a voice that sounds clear over a telephone. Also assume that such voices typically have a `ValueName` called `SupportsTelephony`, which is set to 1. There is no such protocol in SAPI; this is for illustration only. Because this is not a value under `Attributes`, it cannot be picked up by the standard query mechanism of required attributes. The variable `pCurVoiceToken` represents a token for that category. In the example below, the category is populated with tokens in `cpEnum` until a voice is found that also supports `Telephony`.

```
ISpObjectToken    *pCurVoiceToken;

bool              fFeature = false;

while (cpEnum->Next(1, &pToken, NULL) = S_OK)
{
    // At this point, all we know is that pToken is a pointer to a Voice
    token.

    hr = pToken->GetData(L"SupportsTelephony", fFeature);
    // Note, ISpObjectToken inherits from ISpDataKey

    if (( SUCCEEDED( hr ) ) && fFeature )
```

```
{  
  // this is the token for the Voice we want  
  pCurVoiceToken =pToken;  
  break;  
}  
}
```

At this point, store the selected Voice token in `pCurVoiceToken`. Now create the voice object from this token, so that `Speak` and other methods on it may be called. To create a voice object, `ISpVoice` must be created.

```
EXTERN_C const CLSID CLSID_SpVoice;

CComPtr<ISpVoice>    cpVoice;

// The Application may want to check to see if the token has any
// associated UI that it needs to display
hr = pCurVoiceToken-
>IsUISupported(SPDUI_EngineProperties, NULL, 0, NULL,
&fSupported);

// The Application calls the UI, or maybe enables a button in
// its own UI so the user can call the UI

// Next, CoCreate an instance of SpVoice called cpVoice
hr = cpVoice.CoCreateInstance(CLSID_SpVoice);

if( SUCCEEDED( hr ) )
{
```

```
// set cpVoice to our selected voice token  
hr = cpVoice->SetVoice(pCurVoiceToken);  
}
```

At this point the cpVoice object (of type ISpVoice) has been instantiated and is ready to speak, with a call such as:

```
hr = cpVoice->Speak( L"This audio file was created using SAPI five text  
to speech.", 0, NULL);
```

5 Tokens and Categories For Engine Developers

In addition to the enumerating and instantiating tokens, an engine vendor also needs to be able to:

- Create new tokens
- Associate files with tokens

5.1 Making Resources Available Through SAPI

There are several straightforward steps for an SR or TTS engine to be discoverable by SAPI:

1. Make an appropriate entry under the correct CategoryID/Tokens in the registry (details in Section 6).
2. Make an entry under CategoryID/TokenEnums if the vendor prefers dynamic tokens (i.e., the engine registry information is already stored in some other registry location or file). The enumerator should implement the interfaces outlined in Section 3.4.
3. Look at the standard attributes for a category in SAPI and identify the characteristics of the engines so that applications can query the engine for these properties.
4. Hand the SR engine a pointer to the recognition profile token once the RecoInstance has been created.

5.2 Associating Files with Tokens

One of the key issues for an engine vendor is to associate files with tokens in the registry, such as the language model files for a Recognizer or a RecoProfile token. A token can query for all the files under its Files key using the **ISpObjectToken::GetStorageFileName** method. SAPI searches for the file in a number of known locations. Because of the possibility of roaming, SAPI does not store fully qualified file paths in the registry (such as **C:/Documents and Settings/JoeUser/Local Settings/Application Data**), but stores paths such as **%1c%\Microsoft\Speech\Files\MSASR\SP_81738BE4B81F42F0BFC4BB98F** instead. SAPI queries the ShGetFolderPath .dll for the user's non-roaming directory on the individual computers. The calling application can specify (i) the specific name of the file if any, and (ii) the subdirectory to put the file in. Refer to the GetStorageFileName documentation for the exact interfaces. The engine may append any additional vendor-identifying directory names to indicate engine-specific data. Deleting the tokens with which the files are associated by calling **ISpObjectToken::RemoveStorageFileName**, will remove files from the file system as well.

Caveat: If roaming is enabled, the user's RecoProfiles in the HKCUS hive of the registry will roam (because the entire HKCUS hive roams); the associated files, situated in a non-roaming directory will not. This causes two unexpected effects:

1. When the Recognizer is initiated on the second computer, the Recoprofiles are likely to be missing. The Recognizer needs to be able to handle this and copy the necessary new-profile files. **Known issue:** Upon roaming the Microsoft SR Engine currently creates a new set of files, but these have entirely different names from the names on the original computer. As a result, when the registry is roamed back to the original computer, the original profile files become orphaned.
2. Subsequently, upon deleting the Recoprofiles from one computer, all the associated files and registry entries on that computer will be deleted. The rest will become orphans, that is, files without pointers to them.

5.3 Inspecting Underlying Keys of a Token

Besides helper functions, keys under a token can be inspected using a recognizer token, and opening the attributes key under it as a DataKey. Then all the ISpDataKey methods are available to inspect the values under the Attributes key. The sample below goes from the Recognizer token, to the attributes key under it, and finally to the “Desktop” and “Telephony” strings under that.

```
hr = SpGenericSetObjectToken(pToken,
m_cpEngineObjectToken);
if(FAILED(hr))
{
    return hr;
}

// Read attribute information
CComPtr<ISpDataKey> cpAttribKey;
hr = pToken->OpenKey(L"Attributes", &cpAttribKey);

if(SUCCEEDED(hr))
{
    WCHAR *psz = NULL;
    hr = cpAttribKey->GetStringValue(L"Desktop", &psz);
    ::CoTaskMemFree(psz);
}
```

```
if(SUCCEEDED(hr))
{
    // This instance of the engine is for doing desktop
recognition
}
else if(hr = SPERR_NOT_FOUND)
{
    hr = cpAttribKey->GetStringValue(L"Telephony", &psz);
    ::CoTaskMemFree(psz);
    if(SUCCEEDED(hr))
    {
        // This instance of the engine is for doing telephony
recognition
    }
}
}
```

5.4 Creating New Keys in the Registry

Below is another snippet of code where the Microsoft Sample Engine creates a new entry under a recognition profile. If the Recognition Profile does not exist for the engine (pszCLSID contains a pointer to the Engine GUID), it needs to be created it as well as the Gender and Age values under it.

```
// Read attribute information from Engine key;pProfile is the RecoProfile token
we obtain by calling GetRecoProfile on the Recognition Instance.
```

```
hr = pProfile->OpenKey(pszCLSID, &dataKey);
```

```
if(hr = SPERR_NOT_FOUND)
```

```
{
```

```
    // This user profile has not been seen before, so create a new registry key to
    hold info for it
```

```
    hr = pProfile->CreateKey(pszCLSID, &dataKey);
```

```
    // Now set some default values
```

```
    if(SUCCEEDED(hr))
```

```
    {
```

```
        hr = dataKey->SetStringValue(L"GENDER", L"UNKNOWN");
```

```
    }
```

```
    if(SUCCEEDED(hr))
```

```
    {
```

```
        hr = dataKey->SetStringValue(L"AGE", L"UNKNOWN");
```

```
}

// Now create some temporary file storage for trained models

// this will create a valuename called SampleEngTrainingFiles and
value C:\Documents and Settings\username\application data\microsof
speech\files\MSASR\LM7454901D23334AAF87707147726EC235.d

if(SUCCEEDED(hr))
{
    hr = pProfile->GetStorageFileName(CLSID_SampleSREngine,
L"SampleEngTrainingFile", "MSASR\LM%d.dat", CSIDL_FLAG_CREATE,
&pszPath);
}

// and request a UI for user training or properties -
SPDUI_RecoProfileProperties

hr = AddEventString(SPEI_REQUEST_UI, 0, SPDUI_UserTraining);
```

6 Registry Settings

This section documents in some detail, the registry settings of each category of tokens in both the HKCUS and the HKLMS hives. Each token entry needs to have the required keys and values for a token as outlined in Table 1. To find the most suitable token on the computer for the Recognizers, Voices, and Phone Converters categories of tokens, an application needs to define a standard set of attributes that applications can query for. It is important for engine vendors to implement these keys exactly as specified because the engines/voices must be discoverable through SAPI to applications.

It is important to note that in addition to the specified keys and values, a vendor may create any keys and values necessary to use as a resource in the registry. SAPI will ignore these values and not disturb them in any way, unless SAPI is uninstalled from the computer.

6.1 Category: Voices

The Voices category enumerates every voice installed on the computer by all TTS engines. The voice tokens should be located under the key:

HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Speech\Voices\Tok

The requirements for a voice token are listed below, with some sample values.

- Each voice token should meet the requirements for a standard token.
- Voices should document the SAPI-specific attributes that describe them so applications can search for them. Table 8 contains a full listing of Voices attributes and their locations. All Voice attributes are required. Section 3.1 and Section 4.2 have more information about attributes and querying them.
- Voices may have their own Vendor-specific UI implemented by the TTS Engine rendering the voice. If such UI is present, then the UI needs a separate token in the location described in Table 8. The minimum requirement is that the token contain the CLSID of the COM object implementing the UI. Click **Properties** on the Text-to-Speech tab of the Speech Control Panel to access the Vendor-specific UI. The Properties button will be unavailable if the EngineProperties token for the current default Voice is not supported

Table 8 provides a detailed listing of the registry entries that constitute a sample voice token called VoiceToken1 under

HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Speech\Voices

Table 8 Voice Registry and Attributes

RegKey	ValueName	Comments
VoiceToken1		Required - This is the RegKey for the Token.

	(Default)	Required - language independent name.
	409	Name in Hex_LangID 409, which is English. There may be several of these rows, one for each LangID in which the token has a name. Note, no leading 0x before the LangID
	809	
	CLSID	Required - Sample CLSID for object which instantiates the voice.
VoiceToken1/Attributes		Attributes for the Token are under this key.
	Age	Required - Value should be "Child," "Teen," "Adult," or "Senior" depending on Age of TTS Voice. Senior indicates an elderly voice.

		Vendors may choose to classify some voices as both “Senior” and “Adult”.
	Vendor	Required – TTS engine Vendor name.
	Language	Required - The LCID in hex of language this engine speaks.
	Gender	Required - Value should be “Male” if Male voice, “Female” if female.
	VendorPreferred	Required - If this is the Default voice for the vendor named in vendor.
	Name	Required - String representing language independent name
VoiceToken1\UI		Required, if the Voice has UI - U tokens for the voice token will be stored under

		this key.
VoiceToken1\UI\EngineProperties		The only SAPI-specific UI token is EngineProperties. Called when the user clicks Properties on the Text-to-speech tab.
	CLSID	Required - Sample CLSID for object which instantiates engine-specific UI from Speech properties in Control Panel.

Note: Please refer to the registry entries of the Microsoft recognizer and the Sample Engine, which ship in the SAPI 5 SDK, as an example of how the entries are created.

There is also a Voices category in the HKCUS hive that stores the following:

- The default TTS rate selected by the user using Speech properties in Control Panel.
- The default voice selected by the user.

Table 9 provides a listing of the user registry entries that constitute a voice token in

HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\Speech\Vo

Table 9 Voices - User Registry Settings

	ValueName	Value
VoiceToken1	DefaultTokenID	HKEY_LOCAL_MACHINE\SOFT
	TTSRate	5

Note: The TTS engine does not need to store any of these values, SAPI takes care of that.

Vendors may choose to store any additional keys and values in the same areas of the registry. Following is additional information relating to voice tokens:

- User specific entries for the voice (such as volume, pitch, rate, and any other information) should be stored in keys and values under **HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\Speech\Voices**. This creates a structure in the HKCUS hive parallel to the one in the HKLMS hive.
- Entries applying to all the voices using an engine should be stored under **HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\Speech**
- Non-user entries (pertaining to all users on the computer) for a voice should be stored in keys and values under the category **HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Spe**

6.2 Category: Recognizers

SAPI enumerates all the SR Engines installed on the computer from the tokens and token enumerators under

HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Speech

Below are the guidelines for registering recognizer tokens:

- Each recognizer token should meet the requirements for a standard token (Table 1).
- Each speech recognition engine installed on the computer should have a recognizer token. If vendors use a single recognizer for recognition in multiple languages (with different acoustic models), or a discrete and a continuous recognizer, they may choose to store the relevant data files and other initialization information under separate tokens, but use the same value for the CLSID. For example, a vendor may use the same recognition engine to recognize both Japanese and English. In this case, there are two tokens, both containing the CLSID of the same recognizer, but associated with different language and acoustic model files stored with the token.
- Recognizers should document the SAPI-specific attributes shown in Table 10 so that applications can search for them. Required attributes are also indicated Table 10.
- Most speech applications written with SAPI will be tested for a specific engine, or a few specific engines, if the application has a clear need for multiple engines. Typically applications will query for and use this engine by default. Use attributes when the application cannot find its preferred engine (or doesn't have one), and needs to locate the most suitable engine installed on the computer for its needs.
- Recognizer tokens may have an Alternate CLSID if they implement alternates.
- Recognizer tokens may have a RecoExtension CLSID for objects that extend SAPI's recognition context.

- The Recognizer may also have a number of engine-specific UIs that it exposes to SAPI. There should be a separate key under **{Recognizer TokenID}/UI/** for each such UI supported. The keys are listed and documented in Table 10 below.

Table 10 provides a detailed listing of the registry entries that constitute a sample voice token called VoiceToken1 under **HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Speech**

Table 10 Sample Entries of a Recognizer token

RegKey	Value
RecognizerToken1	
	(Default)
	409
	Alternate
	Recognition
RecognizerToken1\Attributes	
	Vendor

	Langu
	Speaki
	Dictati
	Comm Contro
	Deskte
	Teleph
	Vendo
	Altern

	Hypot
	WordS
	Dictati
	Wildc

RecognizerToken1\UI	
RecognizerToken1\UI\EngineProperties	
	CLSII
RecognizerToken1\UI\ AddRemoveWord	
	CLSII
RecognizerToken1\UI\MicTraining	

	CLSII
RecognizerToken1\UI\UserTraining	
	CLSII
RecognizerToken1\UI\RecoProfileProperties	

	CLSII

There is also a Recognizers category in the HKCUS hive that stores the selected default Recognizer. This is done exactly as for Voices, as shown in Table 8. The CategoryID is:

HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\Speech\Re

6.3 Category: RecoProfiles

RecoProfiles (RP) is a user-specific, engine-specific data file in which an SR Engine stores the user-specific acoustic and language data. The RP can be thought of as a bag of information that only the engine knows about. The RP also stores the attributes in a few keys under the RP's key in the registry (this is current Speech Recognition tab of Speech properties in Control Panel).

There are two key reasons for a user to have multiple acoustic profiles:

1. In a shared login case (for example, with a Win98 or Millennium home computer where the users typically press cancel to the login dialog box to enter the computer), multiple files allow two or more users to keep languages and acoustic data separate. In this case, the user will need to manually change the profile to the correct one in Speech properties in Control Panel before starting recognition (or an application may provide its own UI to do this).
2. On a laptop, to offer the user the choice of having different acoustic profiles for different acoustic settings, such as home and office.

A typical RP token is located in the user hive in the following location in the registry

HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\Speech\RecoProfiles\{ProfileGUID 1}

Initially, SAPI creates only one GUID, called Default User, for the RecoProfile. When the Recognizer is used for the first time, it should create a key under this GUID token of the Recognizer. For instance, if the default recognizer has the GUID XXX, the token

HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\Speech\RecoProfile\{ProfileGUID 1}\XXX is created. RecoProfile stores all the files and settings under this key. These settings may include paths to the acoustic and language model files for the profile that are modified during speaker enrollment and subsequently during recognition. It may also contain additional data about the profile that may improve the recognizer accuracy, such as age, gender, microphone gain setting and so on.

Under the Recoprofile token, there is a key for the GUID of each engine that has a profile. When keeping the profile the same, a user switches the default engine (say to YYY) in Speech properties in Control Panel. The new engine, on instantiation (or termination of the session) should create the key

HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\Speech\RecoProfile\{ProfileGUID 1}\YYY

FullPathToFile

All engine-specific (YYY-specific for instance) settings for its RecoProfile should be stored under this key.

6.4 Category: AudioInput

The HKEY_LOCAL_MACHINE/SOFTWARE/MICROSOFT/Speech/AudioInput category contains token enumerators that enumerate all the AudioInput devices present on the computer. There is a token enumerator for each class of AudioInput Device. By default, SAPI 5 will have only a single token enumerator for the MMSys technology. This token enumerator will create an audio token for each AudioInput device (microphone) on the computer and return it when an application or engine calls SpEnumTokens or IenumSpObjectTokens.

The AudioInput category does not have standard attributes, and if multiple technologies are installed, an application needs to inspect each token to find the most suitable one.

Any additional AudioIn token enumerators must meet the requirements for a token enumerator laid out in Table 2. Example of the AudioInput category at:

HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Speech\Recognize

Table 11 AudioInput Category

RegKey	ValueName	Comments
TokenEnums\MMSys		This is the category.
	DefaultTokenID	This Default can point to a token enumerator or token.

AudiInput1		This is the key for the audio input device.
AudiInput1\Attributes		Attributes for the Token are under this key.
	Technology	This is the technology, for example, "MMSys"
	Vendor	This is the vendor name.

6.5 Category: AudioOutput

The **HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Speech\AudioOut** category contains token enumerators that enumerate all the audio output devices present on the computer. As in the AudioInput category, there is a token enumerator for each technology of audio output Devices. By default, there will be a single token enumerator for MMSys. Under this, there will be entries for each audio output device installed on the computer.

RegKey	ValueName	Comments
TokenEnums\MMSys		This is the category.
	DefaultTokenID	This Default can point to a token enumerator or token.
AudioOutput1		This is the key for the audio output device.
AudioOutput1\Attributes		Attributes for the Token are under this key.
	NoSerializeAccess	Optional: Override serialization

		of multiple voices.
	Technology	This is the technology, for example, "MMSys"
	Vendor	This is the vendor name.

6.6 Category: AppLexicons

The AppLexicons category stores all the application lexicons SAPI knows about.

As in other categories, the lexicons are located under

HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Speech\Applexico

When called, the SpLexicon interface enumerates all the applexicons.

Applexicons have no attributes, and therefore, there is no way to load only specific Applexicons. These keys will be created by applications to make their own lexicons available through SAPI.

6.7 Category: PhoneConverters

The ISpPhoneConverter interface enables the application to convert from the SAPI character phoneset to the ID phoneset. Phone Converter keys should go under

HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Speech

SAPI has a single phoneconverter for each language. An engine can query for the phoneconverter whose language attribute matches the application's language of interest.

6.8 UserLexicons

SAPI stores the user lexicon keys under the **HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\Speech\Use** key. UserLexicon is not a category by itself. There is no interaction between an application or engine with the UserLexicon token. It is mentioned here only for the sake of completeness of the registry documentation.

7 Index of Tables

[Table 1: Parts of a Token in the Registry](#)

[Table 2 Parts of the AudioInput token enumerator](#)

[Table 3 Common Helper Functions](#)

[Table 4 Engine Developer Helper Functions](#)

[Table 5 Query Operators](#)

[Table 6 Voices installed on a computer](#)

[Table 7 Scoring of tokens matching optional criteria](#)

[Table 8 Voice Registry and Attributes](#)

[Table 9: Voices - User Registry Settings](#)

[Table 10 Sample Entries of a Recognizer token](#)

[Table 11: AudioInput Category](#)



Simple TTS Guide - Speak to a File and Speak a File

Overview

This document is intended to help developers of text-to-speech (TTS) applications use SAPI TTS functionality to speak text into a wav file and to speak a text file. The example illustrates how to use the Speak and SpeakStream methods, how to select a specific voice, and how to set the output audio stream to a wav file. The examples are written in both C++ and Visual Basic.

Speak to a wav file in C++

The following is an example that speaks a text string, "Hello World", to a wav file, "ttstemp.wav" in C++/ATL COM. The SAPI helper class, CSpStreamFormat, and helper method, SPBindToFile, which are defined in sphelper.h, are used in this example to set the audio wav format and bind the audio stream to the specific file. Since SPSF_22kHz16BitMono is the preferred wav format of the Microsoft English TTS engine, it is selected as the output audio format for the better audio effect. In the following example, the ISpVoice::SetOutput() method must be called to set the audio outputs to the right stream. This is because, by default, the output is set to the default audio device. For the simplification, ISpVoice::Speak() is called synchronously. If you want to speak asynchronously, change the speak flag to SPF_ASYNC and call ISpVoice::WaitUntilDone() after the ISpVoice::Speak() waiting for the completion of the speak process.

```
HRESULT                                     hr = S_OK;
CComPtr <ISpVoice>                          cpVoice;
CComPtr <ISpStream>                          cpStream;
CSpStreamFormat                              cAudioFmt;

//Create a SAPI Voice
hr = cpVoice.CoCreateInstance( CLSID_SpVoice );

//Set the audio format
if(SUCCEEDED(hr))
{
    hr = cAudioFmt.AssignFormat(SPSF_22kHz16BitMono);
}

//Call SPBindToFile, a SAPI helper method, to bind
if(SUCCEEDED(hr))
{
    hr = SPBindToFile( L"c:\\ttstemp.wav",  SPFM_
```

```

        &cpStream;, & cAudioFmt.FormatId(),c
    }

    //set the output to cpStream so that the output audi
    if(SUCCEEDED(hr))
    {
        hr = cpVoice->SetOutput( cpStream, TRUE );
    }

    //Speak the text "hello world" synchronously
    if(SUCCEEDED(hr))
    {
        hr = cpVoice->Speak( L"Hello World",  SPF_DEI
    }

    //close the stream
    if(SUCCEEDED(hr))
    {
        hr = cpStream->Close();
    }

    //Release the stream and voice object
    cpStream.Release ();
    cpVoice.Release();

```

Speak to a wav file in automation

The following example is written in Visual Basic. It has the same functionality as the above in C++. After the creation of an SpFileStream object, a default format, SAFT22kHz16BitMono, is assigned to the object so that user does not need to explicitly assign a wav format to it unless a specific wav format is needed. In this example, ISpeechFileStream.Open creates a wav file, ttstemp.wav, and binds the FileStream to the file. The third parameter of ISpeechFileStream.Open is the Boolean, DoEvents. The default of this parameter is set to False. However, the user should always set it to True to display SAPI events while playing back the wav file. If the parameter is set to False, no engine events will be stored in the file, resulting in that no engine events will be fired during the wav file play back.

```
Dim FileName As String
Dim FileStream As New SpFileStream
Dim Voice As SpVoice

'Create a SAPI voice
Set Voice = New SpVoice

'The output audio data will be saved to ttstemp.wav file
FileName = "c:\ttstemp.wav"

'Create a file; set DoEvents=True so TTS events will be saved
FileStream.Open FileName, SSFMCreatForWrite, True

'Set the output to the FileStream
Set Voice.AudioOutputStream = FileStream

'Speak the text
Voice.Speak "hello world"

'Close the Stream
FileStream.Close
```

```
'Release the objects  
Set FileStream = Nothing  
Set Voice = Nothing
```

Speak a Text File in C++

The following code snippet demonstrates how to speak a text file with a specific voice. In the example, `SpEnumTokens`, a SAPI helper method, is used to enumerate available voice tokens under the key:

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Voices\Tokens`
`SpEnumTokens` returns a token enumerator containing all tokens meeting a set of required and optional attributes. Tokens in the enumerator are sorted in the order of “best matches” rule. In the following example, the required voice attribute is “Name=Microsoft Sam” and there are no optional attributes. `SpEnumTokens` here will return all of the voice tokens with “Name=Microsoft Sam” voice attribute. Through `IEnumSpObjectTokens::Next()` method, you can find the best voice token and then set it as a current voice by calling `ISpVoice::SetVoice()` method. Now the “Microsoft Sam” voice has been chosen as a current voice.

Since the voice is speaking text from a file, the `ISpVoice::Speak` call, a speech flag, `SPF_IS_FILENAME`, must be set. Please note, you may choose to use `ISpVoice::SpeakStream` to speak a file. In that case, you need to call `SPBindToFile`, a helper function, to bind the text file to an `ISpStream` object, and then call `ISpVoice::SpeakStream`.

```
HRESULT          hr = S_OK;
CComPtr <ISpVoice>    cpVoice;
CComPtr <ISpObjectToken> cpToken;
CComPtr <IEnumSpObjectTokens> cpEnum;

//Create a SAPI voice
hr = cpVoice.CoCreateInstance( CLSID_SpVoice );

//Enumerate voice tokens with attribute "Name=Micros
if(SUCCEEDED(hr))
{
```

```
        hr = SpEnumTokens(SPCAT_VOICES, L"Name=Micro:
    }

    //Get the closest token
    if(SUCCEEDED(hr))
    {
        hr = cpEnum ->Next(1, &cpToken;, NULL);
    }

    //set the voice
    if(SUCCEEDED(hr))
    {
        hr = cpVoice->SetVoice( cpToken);
    }

    //set the output to the default audio device
    if(SUCCEEDED(hr))
    {
        hr = cpVoice->SetOutput( NULL, TRUE );
    }

    //Speak the text file (assumed to exist)
    if(SUCCEEDED(hr))
    {
        hr = cpVoice->Speak( L"c:\\ttstemp.txt", SPI
    }

    //Release objects
    cpVoice.Release ();
    cpEnum.Release();
    cpToken.Release();
```

Speak a Text File in Automation

The following code illustrates how to speak a text file in a specific voice in Visual Basic. This example assumes a text file (ttstemp.txt) containing the text to be spoken already exists. ISpeechVoice.SpeakStream is used here to speak an SpFileStream that has been bound to the file.

```
Dim FileName As String
Dim FileStream As New SpFileStream
Dim Voice As SpVoice

'Create SAPI voice
Set Voice = New SpVoice

'Assume that ttstemp.txt exists
FileName = "c:\ttstemp.txt"

'Open the text file
FileStream.Open FileName, SSFMOpenForRead, True

'Select Microsoft Sam voice
Set Voice.voice = voice.GetVoices("Name=Microsoft Sam", "Lan

'Speak the file stream
Voice.SpeakStream FileStream

'Close the Stream
FileStream.Close

'Release the objects
Set FileStream = Nothing
Set Voice = Nothing
```



SAPI 5.1 64-bit Issues

Overview

This document is intended to help application developers understand and use SAPI functionality on a 64-bit platform. All the discussions below are based on native Win64 programming.

64-bit Programming

Data types:

Win64 supports large memory requirements with 64-bit addressing. All pointers (including handles) are 64 bits long. LONG, INT, BOOL, etc. are still 32 bits long. WPARAM, LPARAM, and LRESULT are pointer based, thus are 64 bits long. Some new data types are defined. For example, fixed-precision data types such as INT32 and INT64; polymorphic data types such as LONG_PTR; specific pointer-precision data types like POINTER_32.

API changes:

There are some API changes in Win64. The constants GWL_XXX, GCL_XXX and DWL_XXX have been undefined. This allows the Win64 compiler to catch errors using the old Get/SetWindowLong APIs. Change your API calls to the new Get/SetWindowLongPtr, and use the newly defined GWLP_XXX, GCLP_XXX, and DWLP_XXX constants. For example, use SetWindowLong(hWnd, GWL_WNDPROC, (LONG)MyWndProc); in Win64 you will receive an error that GWL_WNDPROC is undefined. It should be changed to: SetWindowLongPtr(hWnd, GWLP_WNDPROC, (LONG_PTR)MyWndProc); To write code that is compatible with both 32-bit and 64-bit versions of Windows, use SetWindowLongPtr.

The following functions have been added to basetsd.h: PtrToLong() and PtrToUlong(), IntToPtr() and UIntToPtr(), HandleToLong() and LongToHandle(), etc. These can help convert values of one type to another. However, IntToPtr sign-extends the INT value, UIntToPtr zero-extends the unsigned int value, LongToPtr sign-extends the long value, and ULongToPtr zero-extends the unsigned long value. Also note that PtrToLong and HandleToLong will truncate the pointer to a 32bit value.

These values should not be used as pointers again.

Compiling the code:

Set /W3 compiler option, and clean up all Win64 related compiler warnings, particularly the following codes:

- C4305: Truncation warning. For example, "return": truncation from "unsigned int64" to "long."
- C4311: Truncation warning. For example, "type cast": pointer truncation from "int*_ptr64" to "int."
- C4312: Conversion to bigger-size warning. For example, "type cast": conversion from "int" to "int*_ptr64" of greater size.
- C4318: Passing zero length. For example, passing constant zero as the length to memset.
- C4319: Not operator. For example, "~": zero extending "unsigned long" to "unsigned _int64" of greater size.
- C4313: Calling the printf() family of routines with conflicting conversion type specifiers and arguments. For example, "printf": "%p" in format string conflicts with argument 2 of type "_int64." Another example is calling printf("%x", pointer_value); This causes a truncation of the upper 32 bits. The correct method is to call printf("%p", pointer_value).
- C4242 and C4244: return conversion. For example, "return": conversion from "_int64" to "unsigned int," possible loss of data.

Fix all Win64 related Compiler Errors, particularly --GWL_xxx and GCL_xxx not defined as discussed above.

For additional information on 64-bit Windows programming issues, check the [Platform SDK documentation](#). Select "Windows Development," then choose "Whistler 64-bit Edition."

SAPI issues

SAPI speech recognition (SR) interfaces are disabled in a 64-bit SAPI 5.1. SR related objects like Recognizer (both Inproc and Shared) cannot use CoCreateInstance on a 64-bit platform. Following is a list of the disabled interfaces: IspRecognizer, IspRecoContext, IspPhraseBuilder, ISpCFGEngine, IspGrammarCompiler, IspGramCompBackend, and ISpITNProcessor.

However, SAPI still supports TTS functionalities on a 64-bit platform. Because there is not a 64-bit version of sapi.lib, you cannot get the CLSIDs directly. But you can use CLSIDFromProgID() to get them. Here are the available SAPI ProgIDs on a 64-bit Windows:

CLSIDs	ProgIDs
CLSID_SpVoice	SAPI.SpVoice
CLSID_SpLexicon	SAPI.SpLexicon
CLSID_SpUnCompressedLexicon	SAPI.SpUnCompressedLexicon
CLSID_SpCompressedLexicon	SAPI.SpCompressedLexicon
CLSID_SpPhoneConverter	SAPI.SpPhoneConverter
CLSID_SpNullPhoneConverter	SAPI.SpNullPhoneConverter
CLSID_SpObjectTokenCategory	SAPI.SpObjectTokenCategory
CLSID_SpObjectTokenEnum	SAPI.SpObjectTokenEnum
CLSID_SpObjectToken	SAPI.SpObjectToken<
CLSID_SpDataKey	SAPI.SpDataKey
CLSID_SpMMAudioEnum	SAPI.SpMMAudioEnum
CLSID_SpMMAudioIn	SAPI.SpMMAudioIn
CLSID_SpMMAudioOut	SAPI.SpMMAudioOut
CLSID_SpStreamFormatConverter	SAPI.SpStreamFormatConverter
CLSID_SpRecPlayAudio	SAPI.SpRecPlayAudio
CLSID_SpStream	SAPI.SpStream
CLSID_SpResourceManager	SAPI.SpResourceManager

CLSID_SpNotifyTranslator SAPI.SpNotifyTranslator<

The following are used in Automation:

CLSIDs	ProgIDs
CLSID_SpTextSelectionInformation	SAPI.SpTextSelectionInformati
CLSID_SpAudioFormat	SAPI.SpAudioFormat
CLSID_SpWaveFormatEx	SAPI.SpWaveFormatEx
CLSID_SpCustomStream	SAPI.SpCustomStream
CLSID_SpFileStream	SAPI.SpFileStream
CLSID_SpMemoryStream	SAPI.SpMemoryStream

Here is a simple example of how to use the ProgIDs on 64-bit applications:

```
LPCOLESTR pProgID;
CLSID clsid = GUID_NULL;
CComPtr<ISpVoice> cpVoice;
pProgID = L"SAPI.SpVoice";
hr = CLSIDFromProgID(pProgID, &clsid);
if(SUCCEEDED(hr))
{
    hr = cpVoice.CoCreateInstance(clsid);
    if(SUCCEEDED(hr))
        hr = cpVoice->Speak(L"This is a 64 bit test.
//do some more stuff here.
}
```



Compliance Tests

Contents

- [1 Contents](#)
- [2 Table of Tables](#)
- [3 Compliance Testing Overview](#)
 - [3.1 SAPI Compliance Required Tests](#)
 - [3.2 SAPI Compliance Feature List Tests](#)
 - [3.3 Minimum Requirements](#)
- [4 Using the compliance testing tool](#)
- [5 Compliance Tests](#)
 - [5.1 Test Result Log:](#)
- [6 Compliance Testing Configuration Options](#)
 - [6.1 SAPI 5.0 Compliance Testing Application Toolbar](#)
 - [6.2 SAPI 5.0 Compliance Testing Application Menu Choices](#)
 - [6.3 SAPI 5.0 Compliance Testing Logging Options](#)
 - [6.4 SAPI 5.0 Compliance Testing Run Options](#)
 - [6.5 SAPI 5.0 Compliance Test Selection Options](#)
- [7 SAPI Compliance: SR](#)
 - [7.1 Required Tests](#)
 - [7.2 Feature Tests](#)
 - [7.3 SR Sample Engine](#)
 - [7.4 Compliance Test Customization](#)
 - [7.5 Multilingual Support](#)
 - [7.6 OS Language Incompatibility](#)
- [8 SAPI Compliance: TTS](#)

[8.1 Required Tests](#)

[8.2 Feature Tests](#)

[8.3 TTS Sample Engine](#)

[8.4 Multilingual Support](#)

[8.5 OS Language Incompatibility](#)

Table of Tables

[Table 1: Events Compliance Test](#)

[Table 2: Lexicon Compliance Test](#)

[Table 3: Command and Control Compliance Test](#)

[Table 4: Required Compliance Tests](#)

[Table 5: Events Feature Compliance Test](#)

[Table 6: Grammar Feature Compliance Test](#)

[Table 7: Feature Compliance Tests](#)

[Table 8: Sample Engine Required Compliance Test results](#)

[Table 9: Sample Engine Feature Compliance Test results](#)

[Table 10: Strings to be localized](#)

[Table 12: Speak Flag Tests](#)

[Table 13: Speak Tests](#)

[Table 14: Lexicon Tests](#)

[Table 15: SAPI XML tests](#)

[Table 16: Events Tests](#)

[Table 17: Sample Engine Required Test Results](#)

[Table 18: Sample Engine Feature List Test Results](#)

[Table 19: Strings to be localized for compliance tests](#)

[Table 21: Required Compliance Tests Failed](#)

[Table 22: Feature Compliance Tests Not Supported](#)

Compliance Testing Overview

This paper, directed toward engine vendors, describes the SAPI 5.0 compliance testing tool by answering the following questions:

- What does SAPI compliance for SAPI 5.0 imply?
- What are the SAPI compliance tests?
- What does each test look for?

The goals of the compliance tool are to help engine vendors test their speech engines for SAPI compliance and port these speech engines to SAPI 5.0. The tests also help vendors to support various SAPI features that are not required for compliance. These tests do not test the speech or performance quality of the engines. All compliance tests assume that SAPI will do parameter validation, and as such, they do not check the engine's ability to handle invalid parameters such as null, bad pointers, or values out of range.

To run the compliance tests, the SPcomp.exe tool is used and either the Text-to-Speech (TTS) or the Speech Recognition (SR) test suite is selected. This tool generates a log report indicating the results of the compliance tests.

There are two types of SAPI 5 compliance tests:

- 1) required tests
- 2) feature list tests

The compliance tests do not necessarily test the DDI directly, instead, they use the SAPI API function calls to test the engine's response to the DDI. The default engine is always used as the engine in the compliance test. Currently, the supported languages for the compliance tests are English, Japanese and Simplified Chinese^[1]. Please check [Microsoft® Speech.NET Technologies](#) for language pack updates and information.

SAPI Compliance Required Tests

The results of the required tests are of a pass/fail nature. These tests were designed to help an engine reach a minimal amount of functionality with the SAPI DDI layer. In order to be SAPI compliant, the engine must pass all required SAPI tests.

SAPI Compliance Feature List Tests

The results of the feature list tests are either “Supported” or “Unsupported”. Feature list tests were designed to help engine vendors port advanced features to SAPI 5.0. To be SAPI compliant, the engine does not need to pass any feature list test, although it is recommended that all features be implemented if possible.

Minimum Requirements

The minimum requirements for speech recognition for dictation are a 200 mhz Pentium with 64 MB for win 95/98 or 96 MB for NT. The recommended computer is a 300 Mhz Pentium II with 128 megs of RAM or better.

Using the compliance testing tool

The SAPI 5.0 compliance test tool, SPcomp.exe, enables you to load compliance test suites and determine the test result logging options. Please see Compliance Tests for more options.

The SPcomp.exe test tool creates the .pro file for a given test suite. Perform compliance tests by starting the SPcomp.exe application and loading a test suite from a .pro file. The .pro file loads the associated dynamic link library (.dll), which contains the SR or TTS compliance tests.

To start the SAPI 5.0 compliance test tool SPcomp.exe from Windows Explorer, double-click the compliance tool icon. Alternatively, you can perform each compliance test from the command line by running the compliance test tool and command line syntax.

For example, the command line syntax for running the SRcomp compliance test in the srcompreq.pro test suite is as follows:

```
C:> SPcomp.exe srcompreq.pro
```

SPcomp.exe srcompreq.pro

Starts the compliance test tool and loads the speech recognition (SR) required tests from the SRcomp.dll.

SPcomp.exe srcompopt.pro

Starts the compliance test tool and loads the speech recognition (SR) feature list tests from the SRcomp.dll.

SPcomp.exe ttscompreq.pro

Starts the compliance test tool and loads the text-to-speech (TTS) required tests from the TTScmp.dll.

SPcomp.exe ttscompopt.pro

Starts the compliance test tool and loads the text-to-

speech (TTS) feature list tests from the TTScomp.dll.

Compliance Tests

The SAPI 5.0 compliance tests verify that you have successfully implemented the required features to be considered compatible with SAPI 5.0. Your engine must successfully complete each of the following four compliance tests with 100 percent pass rate to be compliant with SAPI 5.0.

1. **srcompreq.bat**
—Speech Recognition (SR) required test batch file.
2. **srcompopt.bat**
—Speech Recognition (SR) feature list test batch file.
2. **ttscmpreq.bat**
—Text-to-speech (TTS) required test batch file.
4. **ttscmpopt.bat**
—Text-to-speech (TTS) feature list test batch file.

Test Result Log:

The SAPI 5.0 compliance tool generates a result log and you can configure it to display the result log information or you can save it to a file. The test result log contains pass or fail state information for each segment of the test suite. If a compliance test fails, you can review the result log to determine the origin of the failure.

Example result log with a 100% pass rate for all tests.

Total:

```
=====
PASS      FAIL      SUPPORTED  UNSUPPORTED  ABORTED
-----
1         0         0         0         0
=====
```

Status: PASS

Compliance Testing Configuration Options

The SAPI 5.0 compliance test application user interface (UI) enables you to configure the testing options. The following section provides additional compliance test configuration information.

- SAPI 5.0 Compliance Testing Application Toolbar
- SAPI 5.0 Compliance Testing Application Menu Choices
- SAPI 5.0 Compliance Testing Logging Options
- SAPI 5.0 Compliance Testing Run Options
- SAPI 5.0 Compliance Test Selection Options

SAPI 5.0 Compliance Testing Application Toolbar

The main window of the SAPI 5.0 compliance testing application contains a toolbar from which you can access the configuration options. Additionally, the configuration options are also available from the menu bar located at the top of the compliance testing application window.



Pause on an icon to display tooltip text. Click an icon to view the information associated with the feature.

Load the test DLL

Loads a test dynamic-link library (DLL).

You can run compliance tests using one of the following methods:

1. From the SAPI Engine Compliance Tool, click **File**, and then click **Load Test DLL**.
2. Load the test DLL into SPcomp.exe from the command line. For more information, see Using the compliance testing tool.

Note: loading a compliance test with either method results in automatically unloading any previously loaded compliance tests.

Load the test settings

Loads one of the pre-configured test suites.

You can run compliance tests using one of the following methods:

1. From the SAPI Engine Compliance Tool, click **File**, and then click **Load Settings**.
2. Load the test DLL into SPcomp.exe from the command line. For more information, see Using the compliance testing tool.

Note: loading a compliance test with either method results in automatically unloading any previously loaded compliance tests.

Save settings

Saves the configuration settings for the compliance test application.

Copy

Selects and copies content from the display log.

Clear Window

Clears the display contents of the result log.

Find

Searches for a specific word or phrase within the result log.

Find Next

Searches for the next occurrence of a specific word or phrase within the result log.

Run Test

Begins the compliance test.

Stop Test

Stops the compliance test.

Set Run Options

Configures the compliance test options.

Select Tests

Chooses which compliance test contained in the test suite to.

Set Logging

Determines location of the compliance test log information.

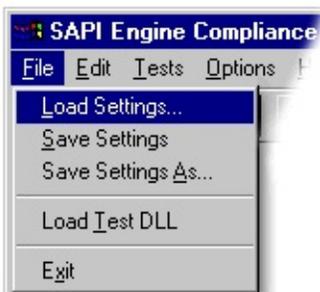
SAPI 5.0 Compliance Testing Application Menu Choices

The SAPI 5.0 compliance testing application configuration choices are accessible through the menu system. The following items are covered in this section:

- File menu
- Edit menu
- Test menu
- Options menu
- Help menu

1 File menu

Click **F**ile to set configuration options to load settings, save settings, or load the appropriate test DLL. Use the arrow keys to view various menu choices. Press ENTER to select a menu choice.



2 Edit menu

Click **E**dit to copy text from the result log and search for text within the result log. Use the arrow keys to view various menu

choices. Press ENTER to select a menu choice.



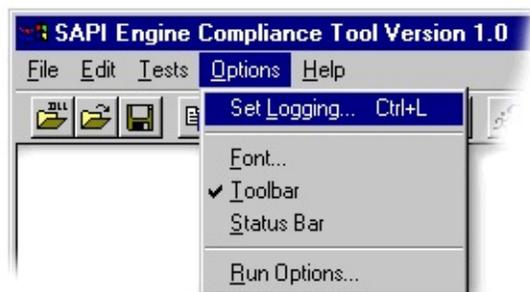
3 Test menu

Click **T**est to run the test or select a test. Use the arrow keys to view various menu choices. Press ENTER to select a menu choice.



4 Options menu

Click **O**ptions to view the various configuration settings. Use the arrow keys to view various menu choices. Press ENTER to select a menu choice.



5 Help menu

Click **H**elp and then click **A**bout to display the SAPI 5.0 Engine

Compliance Tool Version dialog box. Use the arrow keys to view the various menu choices. Press ENTER to select a menu choice.



SAPI 5.0 Compliance Testing Logging Options

From the **Options** menu, choose **Logging Settings** to set SAPI 5.0 compliance test result log configuration options.



Window

Displays the test result information in the main window of the compliance testing application.

Log File

Saves the test result information as text in a log file.

The log file is located at the same directory as *SPcomp.exe* tool and the file name will be the following style:

```
spcomp@442.log
```

The numbers "442" in the file name are generated by the *SPcomp.exe* tool and will be incremented by one each time you restart *SPcomp.exe* tool and run the test. A new log file is generated each time you start *SPcomp.exe* tool and run a compliance test.

Detailed

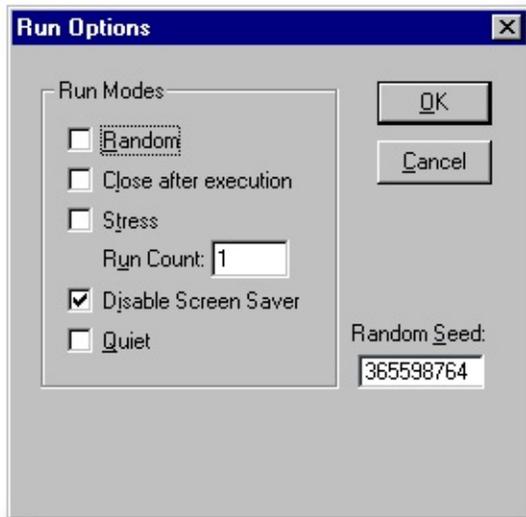
Specifies detailed result log information.

Summary

Specifies summary result log information.

SAPI 5.0 Compliance Testing Run Options

From the **Options** menu, click **Run Options** to configure SAPI 5.0 compliance testing run options.



Random

Randomizes the test order.

Close after execution

Closes the compliance testing application after the test sequence.

Stress

This option should not be selected for compliance tests.

Run count

Specifies the number of interactions the selected test should run.

Disable screen saver

Disable the screen saver.

Quiet

Runs the selected test in quiet mode.

Random Seed

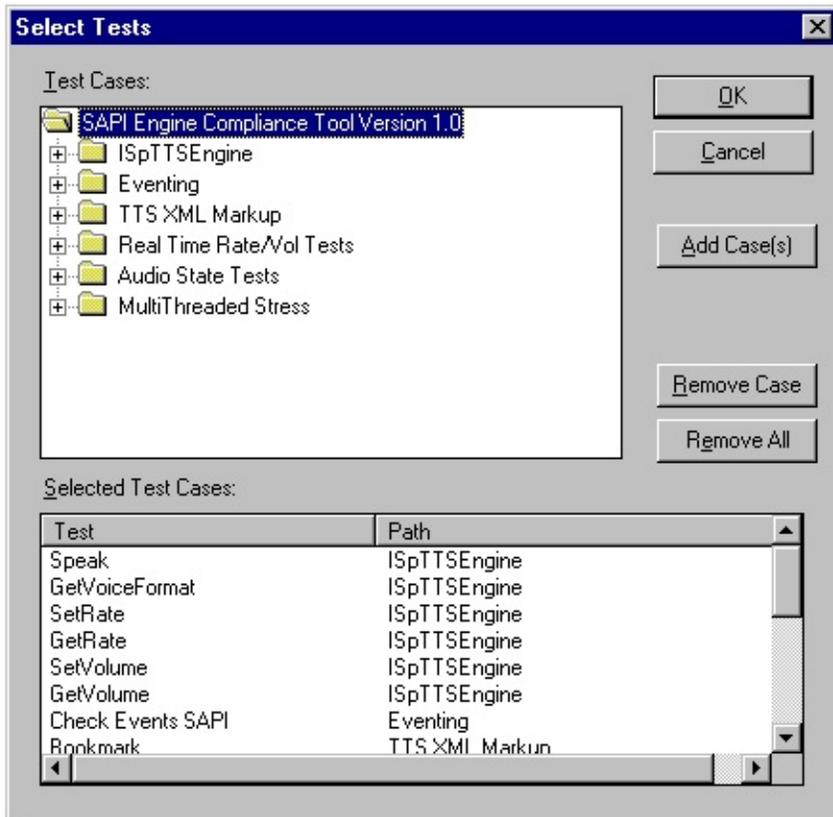
The random seed value set here is used for the next time you run the compliance test.

Note: When troubleshooting a failed compliance test, you need to enter the same seed value information that was used for the failed compliance test before you repeat the compliance test procedure.

You can obtain the compliance test seed value from the "Random Seed" field information in the SPcomp@xxx.log file that was generated during the unsuccessful compliance test.

SAPI 5.0 Compliance Test Selection Options

From the **Test** menu, click **Select Test** to configure SAPI 5.0 compliance test choices.



Test Cases

Displays the current test suite.

Selected Test Cases

Displays the current selected tests.

Add Case(s)

Adds test items to the list of selected test cases.
Alternatively, to add test cases, right-click the test case in the test case display window and click Add Item.



Remove Case

Removes the selected test case from the current test. However, removing the selected test does not affect the need to successfully pass this test case to satisfy SAPI compliancy.

Alternatively, to remove test cases, right-click the test cases in the selected test case display window and click Remove Case.



Remove All

Removes all test cases.

SAPI Compliance: SR

SAPI compliant SR engines must be able to perform the following^[2]:

- § Generate certain SR events
- § Interact with the SAPI lexicon
- § Handle Command and Control (C&C) grammars
- § Generate Phrase Elements
 - § Support auto pause on recognition
- § Support rule synchronization
- § Support multiple instances of the engine
- § Support multiple application contexts

Required Tests

1 Events

Events will be checked for with .wav files. The test will feed the wav file to the engine and expect a specific event notification to occur. Please note that whether or not the engine can fire a specific event depends on the confidence threshold of the engine. Engine vendors could change the .wav quality to meet their requirement.

For English:

Test	Description	Resource IDs
SoundStart	Test will check if a sound start event occurs.	IDS_WAV_SOUNDSTART IDR_L_GRAMMAR
SoundEnd	Test will check if a sound end event occurs.	IDS_WAV_SOUNDEND IDR_L_GRAMMAR
PhraseStart	A .wav file with audio the engine can do recognition on. Test insures that a phrase start	IDS_WAV_PHRASESTART IDR_L_GRAMMAR

event occurs

Recognition	A .wav with audio that the engine can do recognition on. Test insures that a recognition event occurs.	IDS_WAV_RECOGNITION_1 IDR_L_GRAMMAR
False Recognition	A wav file and a mismatching C&C grammar are loaded. Test insures that false recognition event occurs.	IDS_WAV_RECOGNITION_1 IDR_RULE_GRAMMAR
SoundStart/ SoundEnd	Test will check that the sound start event occurs before the sound end event.	IDS_WAV_SOUNDSTARTEND IDR_L_GRAMMAR
PhraseStart/ Recognition	Test will check that the phrasestart event occurs before the recognition event.	IDS_WAV_RECOGNITION_1 IDR_L_GRAMMAR
SoundStart/ PhraseStart/ Recognition/ SoundEnd/	A wav file with audio that the engine can do recognition on. Test insures that the audiooffsets of these events are	IDS_WAV_RECOGNITION_1 IDR_L_GRAMMAR

correct in terms of
value comparison.

Table 1: Events Compliance Test

2 Lexicon

It is expected that changes in the user and application lexicon will be synchronized with the engine both when the engine starts up and after it has loaded a command and control grammar.

Test	Description	Resource IDs
User Lexicon Before C&C Grammar Loaded	A made-up word with its customized pronunciation is added to the user lexicon. After command and control grammar is loaded, audio will be sent with the word added and the expected result is checked for.	IDS_WAV_SYNCH_BEFORE_LOAD IDR_SNORK_GRAMMAR IDS_RECO_SYNCH_BEFORE_LOAD IDS_RECO_NEWWORD_PRON
User Lexicon After C&C Grammar Loaded	After command and control grammar is loaded, a made-up word with its customized pronunciation is added to the user lexicon. Audio will be sent with the word added	IDS_WAV_SYNCH_AFTER_GRAM IDR_SNORK_GRAMMAR IDS_RECO_SYNCH_AFTER_GRAM IDS_RECO_NEWWORD_PRON

and the expected result is checked for.

Application
Lexicon
and C&C
Grammar

A made-up word with its customized pronunciation is added to the application lexicon. After command and control grammar is loaded, audio will be sent with the word added and the expected result is checked for.

IDS_WAV_APPLEX
IDR_SNORK_GRAMMAR
IDS_APPLEX_WORD
IDS_APPLEX_PROP

User
lexicon
before
application
lexicon

A made-up word is added to both user lexicon and application lexicon using the different customized pronunciations. After command and control grammar is loaded, audio will be sent

IDS_WAV_USERLEXBEFOREAPPLEX
IDR_SNORK_GRAMMAR
IDS_USERLEXBEFOREAPPLEX_WOR
IDS_USERLEXBEFOREAPPLEX_USEF
IDS_USERLEXBEFOREAPPLEX_APPP

with the word's
pronunciation
in user lexicon
and the
expected
result is
checked for.

Table 2: Lexicon Compliance Test

3 Command and Control Grammar

Testing the engine for grammar compliance is perhaps the most complex set of tests. The engine must process a grammar correctly. Each test will use a grammar specifically tailored for the particular feature.

Test	Description	Resource IDs
L Tag	A three-element list grammar is loaded. Audio with the middle item to be recognized with the sent to the engine and the result checked for this item.	IDS_RECO_L_TAG IDS_WAV_L_TAG IDR_L_GRAMMAR
Expected Rule	A grammar with two identical rules is loaded. The first rule will be activated. Audio that triggers this rule is sent and test verifies that the engine uses the first rule. The first	IDS_RECO_EXPRULE_FIRSTRULE IDS_RECO_EXPRULE_SECONDRULE IDS_WAV_EXPRULE_TAG IDR_EXPRULE_GRAMMAR

rule is then de-activated and the second rule is activated. The same audio is sent and the test verifies that the engine uses the second rule.

P Tag

A simple grammar with a single phrase. Audio is sent and recognition is expected. Audio that does not contain the phrase is sent and no recognition is expected.

IDS_RECO_P_TAG
IDS_WAV_P_TAG
IDR_P1_GRAMMAR

O Tag

A grammar will be defined with a phrase and an optional phrase preceding and following

IDS_RECO_O_TAG_1
IDS_RECO_O_TAG_2
IDS_RECO_O_TAG_3
IDS_WAV_O_TAG_1
IDS_WAV_O_TAG_2
IDS_WAV_O_TAG_3

it. Three audio streams will be sent. One with the first optional phrase, one for the second, and the third that does not contain any optional phrases. The appropriate recognition result is checked for in each case.

RULEREF Tag

A grammar with a phrase with a rule reference and a rule defined will be loaded. Audio that triggers the rule will be sent and the result checked.

IDS_RECO_RULE_TAG
IDS_WAV_RULE_TAG
IDR_RULE_GRAMMAR

/Disp/lex/pron format

Test ensures engine can support

IDS_CUSTOMPROP_NEWORD_PR
IDS_CUSTOMPROP_NEWORD_DI

customized pronunciation provided in the command and control grammar file.	IDS_CUSTOMPROP_NEWORD_LE IDS_CUSTOMPROP_RULE IDS_WAV_CUSTOMPROP
--	---

Table 3: Command and Control Compliance Test

4 Phrase Elements, Auto Pause, Rule invalidation, multiple instances and contexts.

Test	Description	Resource IDs
Phrase Elements	The audio offsets of SPPHRASEELEMENTs in one SPPHRASE are correctly filled in, which means that the audio offset of the first SPPHRASEELEMENT is less than the audio offset of the second SPPHRASEELEMENT, the audio offset of the second SPPHRASEELEMENT is less than the third one, etc.	IDS_WAV_RULE_TAG IDR_RULE_GRAMMAR
Auto Pause	The test makes sure engine can support auto pause feature provided by SAPI.	IDS_AUTOPAUSE_DYNAMICWC IDS_AUTOPAUSE_DYNAMICWC IDS_AUTOPAUSE_DYNAMICCRU IDS_AUTOPAUSE_DYNAMICCRU IDS_WAV_AUTOPAUSE

Top-level rule invalidation	Test verifies that engine can synchronize the rule information after SAPI notifies engine of top-level rule invalidation.	IDS_INVALIDATETOPLEVEL_DY IDS_INVALIDATETOPLEVEL_DY IDS_WAV_INVALIDATETOPLEV IDS_INVALIDATETOPLEVEL_DY IDS_WAV_INVALIDATETOPLEV
-----------------------------	---	---

None-top-level rule invalidation	Test verifies that engine can synchronize the rule information after SAPI notifies engine of non-top-level rule invalidation.	IDS_INVALIDATENONTOPLEVE IDS_INVALIDATENONTOPLEVE IDS_INVALIDATENONTOPLEVE IDS_INVALIDATENONTOPLEVE IDS_INVALIDATENONTOPLEVE IDS_INVALIDATENONTOPLEVE IDS_WAV_INVALIDATENONTOF IDS_INVALIDATENONTOPLEVE IDS_INVALIDATENONTOPLEVE IDS_WAV_INVALIDATENONTOF
----------------------------------	---	--

Multiple recognition contexts

Multiple recognition contexts will be created with different grammars. The test will verify that the recognition event is generated by the correct recognition contexts.

IDS_RECO_P_TAG
IDS_WAV_MULT_RECO
IDR_P1_GRAMMAR
IDR_P2_GRAMMAR

Multiple recognition engine

Basic tests are run separately on different threads to

NA

instances	see if engine can support multi instances.
-----------	--

Table 4: Required Compliance Tests

Feature Tests

Some of the features exposed through SAPI are useful from a competitive advantage point of view. Features are not required by SAPI compliance, but may be an attractive function for engine vendors to implement. SAPI features are:

- § Interference and hypothesis events
- § Dictation functionalities
- § Advanced command and control features
- § Command and control alternate
- § Engine properties
- § Inversed text normalization

1 Events

Events will be checked for with .wav. The test will feed the .wav to the engine and expect a specific event notification to occur. Please note that whether or not the engine can fire a specific event depends on the confidence threshold of the engine. Engine vendors may change the .wav files if it is felt that the .wav quality does not meet their requirements (Refer to Section 7.4).

Test	Description	Resource IDs	Description
Interference	A wav file with noises. Test will check that an interference event occurs.	IDS_WAV_INTERFERENCE IDR_L_GRAMMAR	Input . tag_l.w Input C gramm

Hypothesis	A .wav file with audio that engine can do recognition on. Test insures a hypothesis event occurs.	IDS_WAV_HYPOTHESIS IDR_EXPRULE_GRAMMAR	Input .tag_ex Input (gramm
------------	---	---	-------------------------------

Table 5: Events Feature Compliance Test

2 Dictation functionalities

This the required features if Engine wants to support dictation grammar. This include some basic functionalities for dictation grammar. This includes lexicon, dictation tag, dictation alternates.

Test	Description	Resource IDs	De
User Lexicon Before dictation Grammar Loaded	A made-up word with its customized pronunciation is added to the user lexicon. After dictation grammar is loaded, audio will be sent with the word added and the expected result is	IDS_WAV_SYNCH_BEFORE_LOAD IDS_RECO_SYNCH_BEFORE_LOAD IDS_RECO_NEWORD_PRON	Imp lex The for wo The pro the use

	checked for.		
User Lexicon After dictation Grammar Loaded	After dictation grammar is loaded, a made-up word with its customized pronunciation is added to the user lexicon. Audio will be sent with the word added and the expected result is checked for.	IDS_WAV_SYNCH_AFTER_DICT IDS_RECO_SYNCH_AFTER_DICT IDS_RECO_NEWWORD_PRON	Inp lex The for wo The pro the use
Dictation Tag	A rule with dictation tag is loaded. Audio is feed and the test verifies the recognition event is generated.	IDS_DICTATIONTAG_WORDS IDS_DICTATIONTAG_RULE IDS_WAV_DICTATIONTAG	The the tag The gra na Inp tag
Dictation alternates	Test ensures that engine can generate alternate results for dictation	IDS_WAV_EXPRULE_TAG	Inp tag

grammar.
The test
makes sure
that engine
has its own
alternate
object and
the object
can generate
some
alternate
results.

Table 5: Dictation Compliance Test

3 Grammar

Each test will use a grammar specifically tailored for the particular feature. Some tests would use dynamic grammar instead of the static grammar.

Test	Description	Resource IDs
WildCard Tag	A rule with wildcard tag is loaded. Audio is feed and the test verifies the recognition event is generated.	IDS_WILDCARD_WORDS IDS_WILDCARD_RULE IDS_WAV_WILDCARD
TextBuffer Tag	A grammar with <TextBuffer> tag will be loaded. Test fills in the content of TextBuffer on the fly. Audio with both static part and dynamic part of the grammar would be feed and the result would be checked.	IDS_CFGTEXTBUFFER_WORDS IDS_CFGTEXTBUFFER_BUFFERWOI IDS_CFGTEXTBUFFER_RULE IDS_WAV_CFGTEXTBUFFER
Use the correct grammar	Two unambiguous grammars are loaded to test if engine can use	IDS_RECO_RULE_TAG IDS_WAV_RULE_TAG IDR_L_GRAMMAR

	the correct grammar to do recognition.	IDR_RULE_GRAMMAR
Use the most recently activated grammar	Two ambiguous grammars are loaded to test if engine can use the most recently activated grammar to do the recognition.	IDS_WAV_RULE_TAG IDR_RULE_GRAMMAR

Table 6: Grammar Feature Compliance Test

4 Alternates, engine properties, inversed text normalization

Test	Description	Resource IDs
Command and Control alternates	Test ensures that the engine can generate alternate results for command and control grammar	IDS_ALTERNATESCFG_BESTWOR IDS_ALTERNATESCFG_ALTERNATI IDS_ALTERNATESCFG_ALTERNATI IDS_ALTERNATESCFG_WORDS IDS_WAV_ALTERMATESCFG
Engine numeric properties	If engine supports the numeric properties specified by SAPI	NA
Engine text properties	If engine can return S_FALSE on the text properties that are not supported.	NA
Inversed Text Normalization	The test uses a wav file and expects engine to pass back a result	IDS_RECO_GETITNRESULT IDS_WAV_GETITNRESULT IDR_RULE_GRAMMAR

containing
digits together
with the
normal result.
Please note
that this is a
very specific
ITN test and is
not coverage
of ITN related
issues.

Table 7: Feature Compliance Tests

7.3 SR Sample Engine

The sample engine is not fully SAPI compliant due to the fact that it does not have the full range of functionality that a true SR engine would have. Table 8 indicates which compliance tests will pass. Table 9 indicates which features are supported.

Test	Result	Description
<i>Events</i>		
SoundStart	Pass	
SoundEnd	Pass	
PhraseStart	Pass	
FalseRecognition	Fail	The sample engine doesn't generate this event based on the real SR job.
Recognition	Pass	
SoundStart/SoundEnd order	Pass	
PhraseStart/Recognition order	Pass	
Event offset	Pass	
<i>Lexicon</i>		

User Lexicon Before C&C Grammar Loaded	Fail	The sample engine doesn't use user lexicon.
User Lexicon After C&C Grammar Loaded	Fail	The sample engine doesn't use user lexicon.
App Lexicon	Fail	The sample engine doesn't use application lexicon.
Use user lexicon before application lexicon	Fail	The sample engine does not use either a user lexicon or an application lexicon.
<i>Grammar</i>		
L Tag	Fail	The result might be sometimes fail and sometimes pass. The sample engine randomly generates results based on the given grammar. It doesn't do any real recognition.
Expected Rule	Pass	
P Tag	Fail	The result might be sometimes fail and sometimes pass. The sample engine randomly generates results based on the given grammar. It doesn't do any real recognition.
O Tag	Fail	The result might be sometimes fail and sometimes pass. The sample engine randomly generates results based on the given grammar. It doesn't do any real recognition.

		grammar. It doesn't do any real recognition.
Ruleref Tag	Fail	The result might be sometimes fail and sometimes pass. The same engine randomly generates results based on the given grammar. It doesn't do any real recognition.
/Disp/lex/pron format	Fail	The result might be sometimes fail and sometimes pass. The same engine randomly generates results based on the given grammar. It doesn't do any real recognition.
<i>Other</i>		
Phrase Elements	Pass	
Auto Pause	Pass	
Top-level rule invalidation	Fail	The sample engine randomly generates results based on the given grammar. It doesn't do any real recognition.
Non-top-level rule invalidation	Fail	The sample engine randomly generates results based on the given grammar. It doesn't do any real recognition.
Multiple recognition	Pass	

contexts	
Multiple recognition engine instances	The sample engine randomly generates a cfg result based on the given grammar. It doesn't do any real recognition.

Table 8: Sample Engine Required Compliance Test results

Test	Result	Description
<i>Events</i>		
Hypothesis	SUPPORTED	
Interference	UNSUPPORTED	The sample engine doesn't generate the event correctly.
<i>Dictation</i>		
User lexicon synchronize before dictation grammar loaded	UNSUPPORTED	The sample engine doesn't use user lexicon.
User lexicon synchronize after dictation grammar loaded	UNSUPPORTED	The sample engine doesn't use user lexicon.
Dictation Tag	SUPPORTED	

Dictation alternates	SUPPORTED	
<i>Grammar</i>		
Wildcard Tag	SUPPORTED	
TextBuffer Tag	SUPPORTED	
Use the correct grammar	UNSUPPORTED	The sample engine randomly generates results based on the gi grammar. It doesn't do any real recognition.
Use the most recently activated grammar	UNSUPPORTED	The sample engine randomly generates results based on the gi grammar. It doesn't do any real recognition.
<i>Other</i>		
Command and Control alternates	UNSUPPORTED	The compliance test only uses one rule while the sample engine needs at least two rules.
Engine numeric properties	SUPPORTED	
Engine text properties	SUPPORTED	
Inversed Text Normalization	UNSUPPORTED	The sample engine doesn't have functionality.

Table 9: Sample Engine Feature Compliance Test results

Compliance Test Customization

Many of the tests do require that a specific recognition result be returned to verify proper handling of such things as the grammar format. To accommodate different engines variability with recognition of different voices and to support non-English engines, these tests will enable the engine vendor to supply a sound file that passes the test (Refer to Section 7.5). Since some tests might share the same .wav file, it is recommended to supply a .wav file with different name. Additionally the grammars can be changed to accommodate words that the engine is able to recognize better (Refer to Section 7.6).

Multilingual Support

The compliance tests will test engines for the supported languages[3]. To test an SR engine that uses another language, one must:

- § Ensure that the correct language pack is installed. For Windows 2000 and Millennium Edition, this may be done by installing the language pack from the Windows 2000 or Windows Millennium CD. For Windows 98 and Windows NT 4.0, install the language pack from the [Windows Update](#) web site.
- § Select the engine as the default engine using Speech Recognition tab in Speech properties.
- § Create and insert a string table in the sapi5sdk\tools\comp\sr\srcomp.rc that is localized for the language. (Refer to Table 10)
- § Create the .wav files[4] according to the new string table and place this under the specified directory (according to the search path precedence (Refer to Section 7.5.2)).
- § Create and compile the appropriate XML files using a grammar editor and compiler.
- § Include the CFG binaries into the .dll by importing the CFG file names into srcomp.rc[5].
- § Recompile the sr.dsp.
- § Run the compliance tests.

1 Example:

If you want to add resource for test SoundStart for language 888:

1. create and insert copy a string table in srcomp.rc for language 888.
2. Change the string “IDS_WAV_SOUNDSTART” to the new .wav file you want to use.
3. Insert the xml grammar file you want to use into the project. Modify the IDR_L_GRAMMAR reference to your cfg binary.

NOTE: If the default engine supports multiple languages, then the compliance test will only run on the first language specified in string “Language” under key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Recognizers. In other words you need to change the order of the languages in the attributes key under your speech recognizer token for each language you wish to test. (Refer to Section 7.6)

The strings that need to be localized are shown in Table 10.

String Number	English Text
IDS_RECO_L_TAG	put Trans string
IDS_RECO_EXPRULE_TAG_1	play Trans string
IDS_RECO_EXPRULE_TAG_2	the Trans

		string
IDS_RECO_P_TAG	white	Trans string
IDS_RECO_O_TAG_1	please	Trans string
IDS_RECO_O_TAG_2	walk	Trans string
IDS_RECO_O_TAG_3	slowly	Trans string
IDS_RECO_RULE_TAG	seven	Trans string
IDS_RECO_LN_TAG	red	Trans string
IDS_RECO_NEWWORD_PRON	s n ao l r k	Trans phone
IDS_AUTOPAUSE_DYNAMICWORD1	put	Trans string
IDS_AUTOPAUSE_DYNAMICWORD2	red	Trans string
IDS_AUTOPAUSE_DYNAMICRULE1	Action	Trans string
IDS_AUTOPAUSE_DYNAMICRULE2	color	Trans string
IDS_INVALIDATETOPLEVEL_DYNAMICWORDS	play the	Trans string

	oboe	
IDS_INVALIDATETOPLEVEL_DYNAMICRULE	Play	Trans string
IDS_INVALIDATETOPLEVEL_DYNAMICNEWWORDS	please play the seven	Trans string
IDS_INVALIDATENONTOPLEVEL_RULE1	option	Trans string
IDS_INVALIDATENONTOPLEVEL_RULE2	Thing	Trans string
IDS_INVALIDATENONTOPLEVEL_TOPLEVELRULE	play	Trans string
IDS_INVALIDATENONTOPLEVEL_OLDWORD1	empty	Trans string
IDS_INVALIDATENONTOPLEVEL_OLDWORD2	Oboe	Trans string
IDS_INVALIDATENONTOPLEVEL_NEWWORD2	Seven	Trans string
IDS_INVALIDATENONTOPLEVEL_TOPLEVELWORDS	Play the	Trans string
IDS_CFGTEXTBUFFER_WORDS	Play the	Trans string
IDS_CFGTEXTBUFFER_BUFFERWORD	oboe	Trans string

IDS_CFGTEXTBUFFER_RULE	play	Trans string
IDS_ALTERNATESCFG_BESTWORD	play	Trans string
IDS_ALTERNATESCFG_ALTERNATE1	played	Trans string
IDS_ALTERNATESCFG_ALTERNATE2	pay	Trans string
IDS_ALTERNATESCFG_WORDS	The oboe	Trans string
IDS_ALTERNATESCFG_RULE	play	Trans string
IDS_RECO_GETITNRESULT	please play the 7	Trans string
IDS_CUSTOMPROP_NEWWORD_PRON	s n ao l r k	Trans phone
IDS_CUSTOMPROP_RULE	play	Trans string
IDS_CUSTOMPROP_NEWWORD_DISP	abc	Trans string
IDS_CUSTOMPROP_NEWWORD_LEX	play	Trans string
IDS_DICTATIONTAG_WORDS	Play the	Trans string

IDS_DICTATIONTAG_RULE	play	Trans string
IDS_WILDCARD_WORDS	Please play	Trans string
IDS_WILDCARD_RULE	play	Trans string
IDS_APPLEX_PROP	s n ao l r k	Trans phone
IDS_USERLEXBEFOREAPPLEX_USERPROP	s n ao l r k	Trans phone
IDS_USERLEXBEFOREAPPLEX_APPPROP	P l ey	Trans phone
IDS_INVALIDATENONTOPLEVEL_NEWWORD1	Please	Trans string

Table 10: Strings to be localized

2 Search Path Precedence

The compliance tests use a search path precedence to find the various .wav files needed for the compliance tests. The order of search is:

- § current directory
- § '..\resources',
- § '..\..\resources'
- § '..\..\resources'

If the compliance tests cannot find the .wav files in these directories, the test will pop up a dialog asking the user to enter the customized dir to open the file. The new directory will be added to the search order and the new search order will persist for the life of SpComp.

It is important to note that for languages that do not have a SAPI standard phoneme set (i.e. languages which are not supported in this version of SAPI), the engine will fail the following required compliance tests:

Test	Result
<i>Lexicon</i>	
User Lexicon Before C&C Grammar Loaded	Fail
User Lexicon After C&C Grammar Loaded	Fail
App Lexicon	Fail
Use user lexicon before application lexicon	Fail
User lexicon synchronize before dictation grammar loaded	Fail

User lexicon
synchronize
after dictation
grammar
loaded

Fail

/Disp/lex/pron
format

Fail

OS Language Incompatibility

The compliance tests are not based on the language of the OS. They are based on the first language in the token of the default engine. In other words, a Japanese engine on an English OS will cause the compliance tests to load the Japanese resources and run the compliance tests expecting a Japanese engine. In order to run the compliance test for an engine that has a different language than the OS, you will need to set the default engine on the Speech Recognition tab in Speech properties.

NOTE: If the default engine supports multiple languages, then the compliance test will only run the first language. In other words you need to change the order of the languages in the attributes key under your speech recognizer token for each language you wish to test. For example, if your engine token supports both Japanese and English, to test English, the string "Language" must be like "409; 411" under HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Recognizers. To test Japanese, the string "Language" must be like "411; 409" under the same key.

SAPI Compliance: TTS

SAPI compliant TTS engines must be able to perform the following:

- § Speak with SAPI defined speak flags
- § Return the supported audio output formats
- § Interact with the SAPI lexicon
- § Interpret SAPI XML tags
- § React to programmatic volume changes
- § React to programmatic rate changes
- § Synthesize certain SAPI events
- § Skip forward and backward through a segment of text
- § Support multiple instances

Required Tests

8.1.1 Speak

The speak tests test the engine ability to interact with `ISpTTSEngine::Speak` as well as process and react to actions. Before SAPI passes a speak call to the engine, it Query Interfaces (QIs) the engine object for `ISpTTSEngine` and `ISpObjectWithToken` interfaces. If either of these are not implemented correctly, the speak call will fail. If both of the QIs pass, then SAPI will pass the speak call to the engine. The speak call contains a number of speak flags. Table 1 describes the individual tests.

Test	Description
SPF_DEFAULT	This is the normal, default speak flag. The engine should be able to render text to the output site under normal conditions. The engine should also be able to continue rendering when passed the continue action from SAPI
SPF_PURGEBEFORESPEAK	The engine should be able to stop rendering when passed to abort action
SPF_IS_XML	The engine should be able to interpret the SAPI XML when passed to it by SAPI.
SPF_ASYNC	The engine should be able to render text to the output site. The asynchronous environment should not affect the engine since SAPI handles the

	multithreading issues.
SPF_SPEAK_NLP_PUNC	the engine should be able to speak punctuation.

Table 12: Speak Flag Tests

The compliance tests also test a number of combination scenarios to help ensure that the engine is able to stop speaking and start speaking in various combinations that are shown in table 2.

Test	Description
Speak Destroy	SAPI sends the engine a speak call followed by an abort action.
Speak Stop	SAPI sends the engine a speak call followed by a purge call. This is to ensure that the engine is able to stop speaking and clear memory.

Table 13: Speak Tests

2 Output Format

The output format test checks to ensure that the engine is capable of passing its supported output format to SAPI. This is testing the `ISpTTSEngine::GetOutputFormat` function.

3 Lexicon

The lexicon tests check the interaction of the engine and SAPI. These tests add a word to the user and application lexicons and check to ensure that the engine is able to detect the words in the lexicon and is using the word pronunciation from

the user lexicon first, and then the application lexicon second if the word is present in both lexicons. There are two separate tests as shown:

Test	Description
User lexicon	A word is added to the user lexicon and the engine is requested to pronounce this word. The engine should be aware of the case sensitivity. The engine must support the SAPI phoneme, SAPI part of speech, as well the lexicon APIs.
Application lexicon	A word is added to the application lexicon and the engine is requested to pronounce this word. The engine must support the SAPI phoneme, SAPI part of speech, as well the lexicon APIs.

Table 14: Lexicon Tests

8.1.4 XML Tags

The SAPI XML tags are required for compliance. The tags tests shown are required:

Test	Description
Bookmark	Tests if the engine is able to process the bookmark tag and write the appropriate bookmark event.
Silence	Tests if the engine outputs the correct amount of silence as indicated by the silence tag.
Volume	Tests if the engine can change the volume by the correct amount.
Spell	Tests if the engine can handle the tag and spell out

	the text.
Pron	Tests if the engine can handle the SAPI defined phoneme set.
Rate	Tests if the engine can change the rate by the correct amount.
Pitch	Tests if the engine can change the pitch by the correct amount.
Context	Tests if the engine can handle the SAPI defined context tag.
Engine proprietary SAPI tags	Test if the engine can handle non-SAPI tags.

Table 15: SAPI XML tests

8.1.5 SetVolume

This test checks to see if the engine is capable of processing the volume change action. When the engine received this action, it should call the GetVolume function from SAPI to get the new volume, and reflect the change in the audio output.

6 SetRate

This test checks to see if the engine is capable of processing the rate change action. When the engine received this action, it should call the GetRate function from SAPI to get the new volume, and reflect the change in the audio output.

Events

The events test checks to ensure that the engine is writing the correct data, especially wParam and lParam, to the event structure. For the sentence boundary event, wParam is the character length of the sentence including punctuation in the current input stream being synthesized. lParam is the character position within the current text input stream of the sentence being synthesized. For the word boundary event, wParam is the character length of the word in the current input stream being synthesized. lParam is the character position within the current text input stream of the word being synthesized. Any leading and ending spaces will not be included in the length of the word or the sentence. There are three events which are required for SAPI compliance:

Test	Description
SPEI_TTS_BOOKMARK	Checks the engine's ability to properly fire bookmarks embedded in the text.
SPEI_WORD_BOUNDARY	Checks the engine's ability to generate word boundaries given a segment of text.
SPEI_SENTENCE_BOUNDARY	Checks the engine's ability to detect and generate sentence boundaries.

Table 16: Events Tests

8 Skip

This test examines the engine's ability to interact with the skip action. Once the engine receives this action, it should call the `ISpTTSEngineSite::GetSkipInfo` function. After it has completed the skip, it should call the `ISpTTSEngineSite::CompleteSkip` function.

Multi-Instance

The multiple instances test checks to ensure that the engine can handle multiple calls at the same time from SAPI. The tests contains a total of 4 threads and each thread has its own ISpVoice object and the test runs a random combination of the following tests 20 times consecutively:

- § Speak
- § Skip
 - § GetOutPutFormat
- § SetRate
- § SetVolume
- § Check SAPI required Event
- § XML Bookmark
 - § XML Silence
- § XML Spell
- § XML Pron
- § XML Rate
- § XML Volume
- § XML Pitch
- § Real time Rate changes
- § Real time Volume changes
- § Speak Stop
- § Lexicon
- § XML context
 - § Engine proprietary SAPI tags and other combination of XML tags

Feature Tests

Some of the features exposed through SAPI are useful from a competitive advantage point of view. Features are not required by SAPI compliance, but may be an attractive function for engine vendors to implement. SAPI features are:

- § Generation of phoneme events (determines if the engine can generate a phoneme event for a given string of text. The phonemes must correspond to the SAPI defined phonemes).
- § Generation of viseme events (determines if the engine can generate a viseme for a given string of text. The viseme must correspond with the SAPI defined visemes).
- § XML emphasis tag (the engine should change the volume, rate, or pitch of the audio rendered).
- § XML PartOfSp tag (the engine should handle the SAPI defined part of speech – the engine will need to implement this for the lexicon compatibility tests)

TTS Sample Engine

The sample engine is not fully SAPI compliant due to the fact that it does not have the full range of functionality that a true TTS engine would have. Table 6 indicates which compliance tests will pass. Table 7 indicates which features are supported.

Test	Result	Description
<i>Speak</i>		
SPF_DEFAULT	Pass	
SPF_PURGEBEFORESPEAK	Pass	
SPF_IS_XML	Pass	
SPF_ASYNC	Pass	
SPF_SPEAK_NLP_PUNC	Pass	
Speak Destroy	Pass	
Speak Stop	Pass	
GetOutput Format	Pass	
<i>Lexicon</i>		
User Lexicon	Fail	The sample engine does not use a lexicon.
App Lexicon	Fail	The sample engine does not use an

application lexic

XML Tags

Bookmark

Pass

Silence

Pass

Volume

Fail

The sample engine uses pre-recorded .wav files and cannot adjust the volume of the .wav files.

Spell

Fail

The sample engine uses pre-recorded .wav files and cannot spell each word.

Pron

Fail

The sample engine uses pre-recorded .wav files and cannot interpret SAPI phonemes.

Rate

Fail

The sample engine uses pre-recorded .wav files and cannot adjust the rate of the .wav files.

Pitch

Fail

The sample engine uses pre-recorded .wav files and cannot adjust the

			pitch of the .wav files.
	Context	Pass	
	Engine proprietary SAPI tags	Pass	
	<i>SetVolume</i>	Fail	The sample engine uses pre-recorded .wav files and cannot adjust the rate of the .wav files.
	<i>SetRate</i>	Fail	The sample engine uses pre-recorded .wav files and cannot adjust the volume of the .wav files.
	<i>Events</i>		
	SPEI_TTS_BOOKMARK	Pass	
	SPEI_WORD_BOUNDARY	Pass	
	SPEI_SENTENCE_BOUNDARY	Pass	
	<i>Skip</i>	Fail	The sample engine uses pre-recorded .wav files and cannot skip sentences since it does not have a sentence breaker
	Multiple Instances Test	Fail	The sample engine

uses pre-recorded .wav files and cannot skip sentences, change rate, pitch, and volume, or use lexicons.

Table 17: Sample Engine Required Test Results

Test	Result	Description
Phoneme events	UNSUPPORTED	The sample engine uses pre-recorded .wav files and cannot synthesize the phoneme events.
Viseme events	UNSUPPORTED	The sample engine uses pre-recorded .wav files and cannot synthesize the viseme events.
XML Emph Tag	UNSUPPORTED	The sample engine uses pre-recorded .wav files and cannot adjust the emphasis of the .wav files.
XML PartOfSp	UNSUPPORTED	The sample engine uses pre-recorded .wav files and cannot adjust the part of speech of the .wav files.

Table 18: Sample Engine Feature List Test Results

Multilingual Support

To test an engine that uses language aside for the supported languages, one must:

- § Ensure that the correct language pack is installed. For Windows 2000 and Millennium Edition, this may be done by installing the language pack from the Windows 2000 or Windows Millennium CD. For Windows 98 and Windows NT 4.0, install the language pack from the [Windows Update](#) web site.
- § Select the engine as the default engine using the TTS tab in Speech properties.
- § Create a string table in the `\sapi5sdk\tools\comp\tts\ttscomp.rc` which is localized for the language. (Refer to Table 19)
- o Go to ResourceView in the ttscomp workspace, right click the mouse on “String Table”, and select “Insert Copy”. The following window will appear. From the window, select the language that the engine supports, and then click OK.



- o Open `\sapi5sdk\tools\comp\tts\ttscomp.rc` to your editor and edit your language resources (Refer to Table 19), and then save `ttscomp.rc`
- o The following is an example how to support `GetOutputFormat` test in Korean using Microsoft FrontPage editor:
 - § Open `ttscomp.dsw` and create a string table in Korean, save `ttscomp.rc`
 - § Go to below table 19 and find strings used in `GetOutputFormat` test. Only one string, `IDS_STRING65`, is found corresponding to the test.
 - § Open `ttscomp.rc` in Notepad and find `IDS_STRING65` under Korean resources
 - § Launch Microsoft FrontPage and select File | New and Normal tab
 - § Translate “This is the TTS Compliance Test” to Korean.
 - § Select Preview tab, right click your mouse, select Encoding | Western European (Windows), and then cut/paste the string from MS FrontPage to `IDS_STRING65` under Korean resources in your notepad
 - § Save `ttscomp.rc`
 - § Recompile the `tts.dsp`.
 - § Run the compliance tests.

The strings that need to be localized are:

Test Name	String Number	English Text
Speak Destroy	IDS_STRING6	This is a long string of text t complete because it will be next line of code. The engine clean-up correctly and not fa
Speak	IDS_STRING8	Hello <BOOKMARK MARK= '
	IDS_STRING10	This is a test.
	IDS_STRING11	Blah blah ...
Phoneme & Viseme Events	IDS_STRING10	
SetVolume	IDS_STRING10	
SetRate	IDS_STRING10	
Check SAPI required Events	IDS_STRING20	Bookmark <BOOKMARK MAI

XML Bookmark	IDS_STRING20	
XML Silence	IDS_STRING23	Hello World
	IDS_STRING24	Hello <SILENCE MSEC = "80
XML Spell	IDS_STRING26	<SAPI> ENGLISH LANGUAGE
	IDS_STRING27	<SPELL> ENGLISH LANGUAGE
XML Rate	IDS_STRING30	<RATE SPEED= '-5'> hello w
	IDS_STRING31	<RATE SPEED= '5'> hello w
XML Volume	IDS_STRING33	<VOLUME LEVEL = '100'> h
	IDS_STRING34	<VOLUME LEVEL = '1'> hell
XML Pitch	IDS_STRING37	<PITCH MIDDLE = '-10'> a <

	IDS_STRING38	<PITCH MIDDLE ='+10'> a
XML PartOfSp	IDS_STRING48	H l ow
	IDS_STRING52	N ow n p r ow n ow aa aa ah er
	IDS_STRING76	test
Real time volume change	IDS_STRING53	This <BOOKMARK MARK='1 used in the real time rate ar It's rate and volume are adj stream. Engines should pick up.
Real time rate change	IDS_STRING53	
XML Pronounce	IDS_STRING56	A
	IDS_STRING57	<PRON SYM="aa n th ow p th ow p ow l ow jh iy aa n th iy">a</PRON>

Skip	IDS_STRING63	<SAPI>one. Two. Three. Four. Five. Six. Seven. Eight. Nine. Ten. <BOOKMARK="123"/>bookmark eleven. Twelve. Thirteen. Fourteen. Fifteen. Sixteen. Seventeen. Eighteen. Nineteen. Twenty. </SAPI>
User Lexicon Test	IDS_STRING64	Computer
	IDS_STRING71	dh aa n th ow p ow l ow jh iy l ow jh iy aa n th ow p ow l c ow ow p ow l ow jh ch ow ac
	IDS_STRING76	
	IDS_STRING94	h eh l ow w er l d h eh l ow v w er l d h eh l ow w er l d
GetOutputFormat	IDS_STRING65	This is the TTS Compliance Test
XML Non-SAPI tags	IDS_STRING67	<SOMEBOGUSTAGS> Non-SAPI </SOMEBOGUSTAGS>
XML Emph	IDS_STRING68	<SAPI>Do you hear me?</SAPI>

	IDS_STRING69	<SAPI><EMPH>Do you hea </SAPI>
App Lexicon Test	IDS_STRING76	
	IDS_STRING94	
XML Context	IDS_STRING99	<context id='date_mdy'>12/21/99</c <context id='date_mdy'>12.21.00</c <context id='date_mdy'>12 9999</context>
	IDS_STRING100	<context id='date_dmy'>21/12/00</c <context id='date_dmy'>21.12.33</c <context id='date_dmy'>21 1999</context>
	IDS_STRING101	<context id='date_ymd'>99/12/21</c <context id='date_ymd'>99.12.21</c <context id='date_ymd'>19 21</context>
	IDS_STRING102	<context id='date_ym'>99- <context id='date_ym'>199 <context id='date_ym'>99/
	IDS_STRING103	<context id='date_my'>12- <context id='date_my'>12. <context id='date_my'>12/
	IDS_STRING104	<context id='date_dm'>21.

<context id='date_dm'>21-
</context id='date_dm'>21/

IDS_STRING105 <context id='date_md'>12-
</context id='date_md'>12.
</context id='date_md'>12/

IDS_STRING106 <context ID = 'date_year'>
</context ID = 'date_year'>

IDS_STRING107 <context id='time'>12:30:1
</context id='time'>12:30<
</context id='time'>1'21" <

IDS_STRING108 <context
id='number_cardinal'>3432

IDS_STRING109 <context
id='number_digit'>3432</c

IDS_STRING110 <context
id='number_fraction'>3/15<

IDS_STRING111 <context
id='number_decimal'>423.1

IDS_STRING112 <context id='phone_numbe
2693</context>

IDS_STRING113 <context
id='currency'>\$12312.90</

IDS_STRING116 <context ID = 'address'>Or
Redmond, WA, 98052</cont

IDS_STRING117 <context ID = 'address_pos'
4X5</context>

IDS_STRING118 <CONTEXT ID = 'MS_My_Co
</CONTEXT>

Multiple-Instance
Test

IDS_STRING41 Hello <SILENCE MSEC = "%d"

IDS_STRING43 <PRON SYM = "aa n th ow p
hello </PRON>

IDS_STRING44 <RATE SPEED = "%d"> hello

IDS_STRING45 <VOLUME LEVEL = '%d'> he

IDS_STRING46 <PITCH MIDDLE = '%d'> he

IDS_STRING8 Hello <BOOKMARK MARK= '

IDS_STRING27

IDS_STRING67

IDS_STRING76

IDS_STRING94

IDS_STRING119

Table 19: Strings to be localized for compliance tests

It is important to note that for languages which do not have a SAPI standard phoneme set (i.e. languages which are not supported in this SAPI release), the engine will fail the following required compliance tests:

Test	Result
<i>Lexicon</i>	
User Lexicon	Fail
App Lexicon	Fail
<i>XML Tags</i>	
Pron	Fail

Table 21: Required Compliance Tests Failed

The engine will also fail the following feature tests:

Test	Result
Phoneme events	UNSUPPORTED
Viseme events	UNSUPPORTED
XML	UNSUPPORTED

PartOfSp

Table 22: Feature Compliance Tests Not Supported

OS Language Incompatibility

The compliance tests are not based on the language of the OS. They are based on the first language in the token of the default engine. In other words, a Japanese engine on an English OS will cause the compliance tests to load the Japanese resources and run the compliance tests expecting a Japanese engine. In order to run the compliance test for an engine that has a different language than the OS, you will need to set the default engine on the Speech Recognition tab of Speech properties.

[1] For SR Simplified Chinese, the .wav files, cfg files, and resources do not ship with the SAPI 5.0 SDK. To request these localized files, please send mail to sapi5@microsoft.com.

[2] All SR tests use the default recognition profile. To increase the accuracy of the tests, you may wish to change the .wav files used (Refer to Section 7.4) or train the recognition profile.

[3] The Simplified Chinese SR resource files, cfg files and .wav files are not included in the SAPI 5 SDK. Please e-mail sapi5@microsoft.com to obtain these files.

[4] These wav files MUST have different names than the original wav files. The new wav file names should be reflected in the string table.

[5] A basic assumption of the compliance tests is that the CFG files are included in the dll whereas the wav files are external.



XML TTS Tutorial

SAPI XML TTS for Application Developers

SAPI text-to-speech (TTS) extensible markup language (XML) tags fall into several categories.

- [Voice state control](#)
- [Direct item insertion](#)
- [Voice context control](#)
- [Voice selection](#)
- [Custom Pronunciation](#)

Voice state control tags

SAPI TTS XML supports five tags that control the state of the current voice: Volume, Rate, Pitch, Emph, and Spell.

Volume

The Volume tag controls the volume of a voice. The tag can be empty, in which case it applies to all subsequent text, or it can have content, in which case it only applies to that content.

The Volume tag has one required attribute: Level. The value of this attribute should be an integer between zero and one hundred. Values outside of this range will be truncated.

```
<volume level="50">
```

This text should be spoken at volume level fifty.

```
    <volume level="100">
```

This text should be spoken at volume level one hundred

```
    </volume>
```

```
</volume>
```

```
<volume level="80"/>
```

All text which follows should be spoken at volume level eight

One hundred represents the default volume of a voice. Lower values represent percentages of this default. That is, 50 corresponds to 50% of full volume.

Values specified using the Volume tag will be combined with values specified programmatically (using ISpVoice::SetVolume). For example, if you combine a SetVolume(50) call with a `<volume level="50">` tag, the volume of the voice should be 25% of its full volume.

Rate

The Rate tag controls the rate of a voice. The tag can be empty, in which case it applies to all subsequent text, or it can have content, in which case it only applies to that content.

The Rate tag has two attributes, Speed and AbsSpeed, one of which must be present. The value of both of these attributes should be an integer between negative ten and ten. Values outside of this range may be truncated by the engine (but are not truncated by SAPI). The AbsSpeed attribute controls the absolute rate of the voice, so a value of ten always corresponds to a value of ten, a value of five always corresponds to a value of five.

```
<rate absspeed="5">
```

```
    This text should be spoken at rate five.
```

```
    <rate absspeed="-5">
```

```
        This text should be spoken at rate negative five.
```

```
    </rate>
```

```
</rate>
```

```
<rate absspeed="10"/>
```

All text which follows should be spoken at rate ten.

Speed

The Speed attribute controls the relative rate of the voice. The absolute value is found by adding each Speed to the current absolute value.

```
<rate speed="5">  
  This text should be spoken at rate five.  
  <rate speed="-5">  
    This text should be spoken at rate zero.  
  </rate>  
</rate>
```

Zero represents the default rate of a voice, with positive values being faster and negative values being slower. Values specified using the Rate tag will be combined with values specified programmatically (using ISpVoice::SetRate).

Pitch

The Pitch tag controls the pitch of a voice. The tag can be empty, in which case it applies to all subsequent text, or it can have content, in which case it only applies to that content.

The Pitch tag has two attributes, Middle and AbsMiddle, one of which must be present. The value of both of these attributes should be an integer between negative ten and ten. Values outside of this range may be truncated by the engine (but are not truncated by SAPI).

The AbsMiddle attribute controls the absolute pitch of the voice, so a value of ten always corresponds to a value of ten, a value of five always corresponds to a value of five.

```
<pitch absmiddle="5">  
This text should be spoken at pitch five.  
  <pitch absmiddle="-5">  
    This text should be spoken at pitch negative five.  
  </pitch>  
</pitch>  
<pitch absmiddle="10"/>
```

All text which follows should be spoken at pitch ten.

The Middle attribute controls the relative pitch of the voice. The absolute value is found by adding each Middle to the current absolute value.

```
<pitch middle="5">  
This text should be spoken at pitch five.  
  <pitch middle="-5">  
    This text should be spoken at pitch zero.  
  </pitch>  
</pitch>
```

Zero represents the default middle pitch for a voice, with positive values being higher and negative values being lower.

Emph

The Emph tag instructs the voice to emphasize a word or section of text. The Emph tag cannot be empty. The following word should be emphasized.

```
<emph> boo </emph>!
```

The method of emphasis may vary from voice to voice.

Spell

The Spell tag forces the voice to spell out all text, rather than using its default word and sentence breaking rules, normalization rules, and so forth. All characters should be expanded to corresponding words (including punctuation, numbers, and so forth). The Spell tag cannot be empty.

```
<spell>  
These words should be spelled out.  
</spell>  
These words should not be spelled out.
```

Direct item insertion tags

Three tags are supported that applications the ability to insert items directly at some level: Silence, Pron, and Bookmark.

Silence

The Silence tag inserts a specified number of milliseconds of silence into the output audio stream. This tag must be empty, and must have one attribute, Msec.

Five hundred milliseconds of silence `<silence msec="500"/>` j

Pron

The Pron tag inserts a specified pronunciation. The voice will process the sequence of phonemes exactly as they are specified. This tag can be empty, or it can have content. If it does have content, it will be interpreted as providing the pronunciation for the enclosed text. That is, the enclosed text will not be processed as it normally would be.

The Pron tag has one attribute, Sym, whose value is a string of white space separated phonemes.

```
<pron sym="h eh 1 l ow & w er 1 l d " />  
<pron sym="h eh 1 l ow & w er 1 l d"> hello world </pron>
```

Bookmark

The Bookmark tag inserts a bookmark event into the output audio stream. Use this event to signal the application when the audio corresponding to the text at the Bookmark tag has been reached. The Bookmark tag must be empty.

The Bookmark tag has one attribute, Mark, whose value is a string. This value can then be used to differentiate between bookmark events (each of which will contain the string value from their corresponding tag).

The application will receive an event here,

```
<bookmark mark="bookmark_one"/>
```

and another one here

```
<bookmark mark="bookmark_two"/>
```

Voice context control tags

Two tags provide context to the current voice: PartOfSp and Context. Those tags enable the voice to determine how to deal with the text it is processing. With both of these tags, the extent to which voices use the context may vary.

PartOfSp

The PartOfSp tag provides the voice with the part of speech of the enclosed word(s). Use this tag to enable the voice to pronounce a word with multiple pronunciations correctly depending on its part of speech. The PartOfSp tag cannot be empty.

The PartOfSp tag has one attribute, Part, which takes a string corresponding to a SAPI part of speech as its attribute. Only SAPI defined parts of speech are supported - "Unknown", "Noun", "Verb", "Modifier", "Function", "Interjection".

```
<partofsp part="noun"> A </partofsp> is the first letter of
```

```
Did you <partofsp part="verb"> record </partofsp> that <partofsp
```

Context

The Context tag provides the voice with information which the voice may then use to determine how to normalize special items, like dates, numbers, and currency. Use this tag to enable the voice to distinguish between confusable date formats (see the example, below). The Context tag cannot be empty.

The Context tag has one attribute, Id, which takes a string corresponding to the context of the enclosed text. Several contexts are defined by SAPI and are more likely to be

recognized by SAPI compliant voices, but any string may be used. See documentation for a particular voice for more details.

```
<context id="date_mdy"> 03/04/01 </context> should be March  
<context id="date_dmy"> 03/04/01 </context> should be April  
<context id="date_ymd"> 03/04/01 </context> should be April
```

Voice Selection Tags

There are two tags which can be used (potentially) to change the current voice: Voice and Lang.

Voice

The Voice tag selects a voice based on its attributes, Age, Gender, Language, Name, Vendor, and VendorPreferred. The tag can be empty, in which case it changes the voice for all subsequent text, or it can have content, in which case it only changes the voice for that content.

The Voice tag has two attributes: Required and Optional. These correspond exactly to the required and optional attributes parameters to [ISpObjectTokenCategory_EnumerateTokens](#) and [SpFindBestToken](#) functions. The selected voice follows exactly the same rules as the latter of these two functions. That is, all the required attributes are present, and more optional attributes are present than with the other installed voices (if several voices have equal numbers of optional attributes one is selected at random). See [Object Tokens and Registry Settings](#) for more details.

In addition, the attributes of the current voice are always added as optional attributes when the Voice tag is used. This means that, a voice which is more similar to the current voice will be selected over one which is less similar.

If no voice is found that matches all of the required attributes, no voice change will occur.

The default voice should speak this sentence.

```
<voice required="Gender=Female;Age!=Child">
```

A female non-child should speak this sentence, if one exists.

```
<voice required="Age=Teen">
```

A teen should speak this sentence - if a female, non-child

```
</voice>
```

```
</voice>
```

Lang

The Lang tag selects a voice based solely on its Language attribute. The tag can be empty, in which case it changes the voice for all subsequent text; or it can have content, in which case it only changes the voice for that content.

The Lang tag has one attribute, LangId. This attribute should be a LANGID, such as 409 (U.S. English) or 411 (Japanese). Note that these numbers are hexadecimal, but without the typical "0x".

The Lang tag is a shortened version of the Voice tag with the Required attribute containing "Language=xxx". So the following examples should produce exactly the same results:

```
<voice required="Language=409">
```

A U.S. English voice should speak this.

```
</voice>
```

```
<lang langid="409">
```

A U.S. English voice should speak this.

```
</lang>
```

Custom Pronunciation

An alternative to using the <P> tag with the DISP and PRON attributes is to use custom pronunciation. Using custom pronunciation, tags in the form of the following.

```
<P DISP="disp" PRON="pron">word</P>
```

can be written as

```
<P>/disp/word/pron;</P>
```

More specifically, if you want to recognize the word hello only when it is pronounced as ah and display greeting when recognized, you would normally use something like the following.

```
<P DISP="greeting" PRON="ah">hello</P>
```

Using custom pronunciation, the above would translate to the following.

```
<P>/greeting/hello/ah;</P>
```



Text Normalization

You can perform simple text normalization for voice training using the text buffer provided to the engine. Text normalization is the process of changing the input buffer that allows the engine to use preferred word units. The engine word units affect how words are expected to be pronounced as well as how they appear in the voice training wizard.

The text provided to the engine is called an article. An article is composed of multiple phrases, each separated by a new-line character. The voice training wizard displays one phrase at a time.

```
<article> ::= { <phrase> "\n" }
```

A phrase is a sequence of word units, separated by white space characters. In this context, white space characters are all characters for which the C run time function `isspace()` returns TRUE.

```
<phrase> ::= { <word> | <literal_symbols> |  
<numeric_expression> }
```

Word units

Literals

The following symbols are recognized as units. They should be separated from adjacent text with white space; they will "snuggle" to the words appropriately when presented to the user.

```
<literal_symbols> ::=  
"!\\exclamation-point" | "\\end-quote" | "\\quote" | "#\\pound-  
sign" | "$\\dollar" | "%\\percent" | "&\\ampersand" | "\\end-quote" |  
"\\quote" | "(\\paren" | ")\\close-paren" | "*\\asterisk" | "+\\plus" |  
","\\comma" | "--\\double-dash" | "-\\hyphen" | "...\\ellipsis" | ".\\dot" |  
".\\period" | "\\slash" | ":\\colon" | ";\\semicolon" | "<\\less-than" |  
"=\\equals" | ">\\greater-than" | "?\\question-mark" | "@\\at-sign" |  
"[\\bracket" | "\\back-slash" | "]\\close-bracket" | "^\\circumflex" |  
"_\\underscore" | "`\\back-quote" | "{\\left-brace" | "|\\vertical-bar"  
| "}\\right-brace" | "~\\tilde"
```

Numerics

Numbers can be the following form:

```
<digit> ::= "0"-"9"
```

```
<non_zero_digit> ::= "1"-"9"
```

```
<numeric_expression> ::= <integer_expression> |  
<integer_expression> <cardinal_suffix> |  
<floating_expression>
```

```
<integer_expression> ::= ["-"] <non_zero_digit> [<digit>  
 [<digit>]] { ["," <digit><digit><digit> ] }
```

```
<floating_expression> ::= <integer_expression> "." <digit> [{  
<digit> }]
```

```
<cardinal_suffix> ::= "st" | "nd" | "rd" | "th"
```

Collections

The remainder of the buffer will be treated as a collection of words:

<alpha_char> ::= "a"-"z" | "A"-"Z"

<word_char> ::= <alpha_char> | "-" | "_" | "0"-"9"

<word> ::= <word_0> | <word_1> | <word_2> | <word_3>

<word0> ::= <alpha_char> [{<word_char>}]

<word1> ::= <alpha_char> [{<word_char>}] "s"|"in"

<word2> ::= <alpha_char> [{<word_char>}] "." <word2>

<word3> ::= <abbreviation_string> "."

<abbreviation_string> ::=

"al" | "apr" | "assn" | "assoc" | "atty" | "aug" | "bef" | "bldg" |
"ch" | "chg" | "co" | "com" | "cont" | "corp" | "dec" | "def" | "det" |
"dev" | "div" | "doc" | "etc" | "ext" | "feb" | "gov" | "in" | "ins" |
"int" | "intl" | "jan" | "jr" | "jul" | "jun" | "mar" | "messrs" | "mos" |
"mph" | "mr" | "mrs" | "ms" | "mt" | "no" | "nov" | "oct" | "oz" |
"par" | "pct" | "pfc" | "pp" | "pres" | "prov" | "pt" | "qtr" | "ref" |
"reg" | "rep" | "rev" | "sdn" | "sec" | "sep" | "sq" | "sr" | "tech" |
"vol" | "wm"

Microsoft Speech SDK

Speech Automation 5.1



Persisting Recognized Wav Audio from the Speech Recognition Engine

Overview

This document is intended to help developers of speech recognition (SR) applications use the Microsoft speech recognition and audio APIs to persist or store the wav audio recognized by a SR engine. The topics covered include:

- Typical file input scenario
- Typical audio storage scenario
- Relevant APIs for C++ and Visual Basic/Scripting developers
- Sample recognized audio storage source code for developers (written in both C++ and Visual Basic 6.0)

Typical file input scenario

The following are typical scenarios that would need to store the wav audio recognized by the SR engine:

- Transcription applications (e.g., convert voice mail to email)
- Audio correction user interface (e.g., replay and/or re-recognize audio snippets)
- SR engine testing (e.g., measure and improve engine accuracy with reproducible audio input data)

Typical audio storage scenario

Follow these basic steps to retrieve and store recognized wav audio:

1. Create an SR engine (InProc or shared).
2. Enable retained audio on the relevant recognition

context.

3. Set the retained audio format (specify lower quality for smaller storage size, higher quality for clearer audio). Default is the SR engine's audio format.
4. Set up and receive recognition events for relevant recognition context.
5. Retrieve audio stream from recognition result.
6. Copy result's audio stream to file-bound stream.

Relevant wav audio file input APIs for COM/C/C++ Developers:

- SpStream object, ISpStream interface: Basic SAPI audio stream
- ISpStream::BindToFile: Setup audio stream for wav file input
- SpBindToFile: Helper function to setup stream with a wav file
- ISpRecoContext::SetAudioOptions: To enable/disable retained audio
- ISpRecoResult::GetAudio: To retrieve recognized audio
- ISpStreamFormat::GetFormat: To retrieve audio format
- CSpStreamFormat helper object: Helper for handling audio formats
- ISpStream::Read/Write: Methods for reading and writing stream data
- SPEI_RECOGNITION/SPEI_FALSE_RECOGNITION: Events sent by SAPI when a recognition or false recognition has occurred

Relevant wav audio file input APIs for Automation/Visual Basic/Scripting Developers:

- SpFileStream object: Basic file-based SAPI audio stream
- SpMemoryStream object: Basic memory-based SAPI audio stream
- ISpeechRecoContext::RetainedAudio property: To enable/disable retained audio
- ISpeechBaseStream::Read/Write: Methods for reading and writing stream data
- ISpeechBaseStream::Format property: To retrieve audio format
- SpFileStream::Open/Close: Methods for opening and closing a file-based stream
- ISpeechRecoContext::Recognition/FalseRecognition: Events sent by SAPI when a recognition or false recognition has occurred

Sample recognized audio storage source code

Note: Error handling is omitted for brevity

COM/C++ Developers (C-style is very similar)

```
{
    HRESULT hr = S_OK;
    CComPtr<ISpRecoContext> cpRecoContext;
    CComPtr<ISpRecoGrammar> cpRecoGrammar;
    CComPtr<ISpRecoResult> cpRecoResult;
    CComPtr<ISpStreamFormat> cpStreamFormat;
    CSpEvent spEvent;
    WAVEFORMATEX* pexFormat = NULL;
    SPAUDIOOPTIONS eAudioOptions = SPAO_NONE;

    ' format for storing the audio
    const SPSTREAMFORMAT spFormat = SPSF_22kHz8BitMono;
    CSpStreamFormat Fmt( spFormat, &hr;);
    // Check hr

    // Create shared recognition context for receiving events
```

```

hr = cpRecoContext.CoCreateInstance(CLSID_SpSharedRecoCon
// Check hr

// Create a grammar
hr = cpRecoContext->CreateGrammar(NULL, &cpRecoGrammar);
// Check hr

// Load dictation
hr = cpRecoGrammar->LoadDictation(NULL, SPLO_STATIC);
// Check hr

// Enabled audio retention in the SAPI runtime, and set tl
hr = cpRecoContext->SetAudioOptions( SPAO_RETAIN_AUDIO, &
// Check hr

// activate dictation
hr = cpRecoGrammar->SetDictationState(SPRS_ACTIVE);
// Check hr

// wait 15 seconds for an event to occur (specifically, tl
hr = cpRecoContext->WaitForNotifyEvent(15000);
if (S_OK == hr)
{
    // retrieve the event from the recognition context
    hr = spEvent.GetFrom(cpRecoContext);
    if (S_OK == hr)
    {
        // verify that the event is a recognition event
        if (SPEI_RECOGNITION == spEvent.eEventId)
        {
            // store the recognition result pointer
            cpRecoResult = spEvent.RecoResult();
            // release recognition result pointer in event of
            spEvent.Clear();
        }
    }
}

// deactivate dictation (only processing one recognition :
hr = cpRecoGrammar->SetDictationState(SPRS_INACTIVE);
// Check hr
// unload dictation

```

```

hr = cpRecoGrammar->UnloadDictation();
// Check hr

// if recognition received, and result stored then store
if (cpRecoResult)
{
    // get stream pointer to recognized audio
    // Note: specifying NULL for the start element and element
    hr = cpRecoResult->GetAudio( 0, 0, &cpStreamFormat; );
    // Check hr

    // basic SAPI-stream for file-based storage
    CComPtr<ISpStream> cpStream;
    ULONG cbWritten = 0;

    // create file on hard-disk for storing recognized audio
    hr = SPBindToFile(L"c:\\recognized_audio.wav", SPFM_CREATE);
    // Check hr

    // Continuously transfer data between the two streams until done
    // Note only transfer 1000 bytes at a time to create a small file
    while (TRUE)
    {
        // for logging purposes, the app can retrieve the recognition
        STATSTG stats;
        hr = cpStreamFormat->Stat(&stats;, NULL);
        // Check hr

        // create a 1000-byte buffer for transferring
        BYTE bBuffer[1000];
        ULONG cbRead;

        // request 1000 bytes of data from the input stream
        hr = cpStreamFormat->Read(bBuffer, 1000, &cbRead);
        // if data was returned...
        if (SUCCEEDED(hr) && cbRead > 0)
        {
            // then transfer/write the audio to the file-based stream
            hr = cpStream->Write(bBuffer, cbRead, &cbWritten);
            // Check hr
        }
    }
}

```

```

        // since there is no more data being added to the i
        if (cbRead < 1000)
        {
            break;
        }
    }
}

' explicitly close the file-based stream to flush file da
hr = cpStream->Close();
}
}

```

Automation/Visual Basic 6.0 Developers

Scripting code is similar to Visual Basic.

Option Explicit

```

Dim WithEvents RecoContext As SpSharedRecoContext ' context
Dim Grammar As ISpeechRecoGrammar ' grammar

' Setup/Initialization code for application startup
Private Sub Form_Load()
    ' Create new shared recognition context (inproc works si
    Set RecoContext = New SpSharedRecoContext
    ' Create grammar
    Set Grammar = RecoContext.CreateGrammar
    ' Activate retained audio
    RecoContext.RetainedAudio = SRAORetainAudio
    ' Optionally, set the retained audio format to lower qua.
    ' RecoContext.RetainedAudioFormat = ???

    ' load and activate dictation
    Grammar.DictationLoad
    Grammar.DictationSetState SGDSActive
End Sub

' Recognition event was received
Private Sub RecoContext_Recognition(ByVal StreamNumber As Long)
    ' Create new file-based stream for audio storage
    Dim FileStream As New SpFileStream

```

```

' Variable for accessing the recognized audio stream
Dim AudioStream As SpMemoryStream

' Retrieve recognized audio from result object
' Note: application can also retrieve smaller portions of
Set AudioStream = Result.Audio

' Setup the file-based stream format with the same format
Set FileStream.Format = AudioStream.Format
' Create a file on the hard-disk for storing the recognized audio
FileStream.Open "c:\recognized_audio.wav", SSFMCreatFor

Dim Buffer As Variant ' Buffer for storing stream data
Dim lRead As Long ' Amount of data read from the stream
Dim lWritten As Long ' Amount of data written to the stream

' Continuously transfer data between the two streams until
' Note only transfer 1000 bytes at a time to creating large
Do While True
' read 1000 bytes of stream data
    lRead = AudioStream.Read(Buffer, 1000)
    ' if data was retrieved, then transfer/write it to the file
    If (lRead > 0) Then
        lWritten = FileStream.Write(Buffer)
    End If

    ' Since the input stream will not increase in size,
    If lRead < 1000
        Exit Do ' exit if no more data
    End If
Loop
' close the file-based stream
' Note: The stream will be closed automatically when the
FileStream.Close

' Sample code will deactivate and unload dictation, then
Grammar.DictationSetState SGDSInactive
Grammar.DictationUnload
Unload Me ' shutdown app
End Sub

```

Microsoft Speech SDK

Speech Automation 5.1



Using Wav File Input with Speech Recognition Engines

Overview

This document is intended to help developers of speech recognition (SR) applications use the Microsoft Speech API's speech recognition and audio APIs to connect a wav file with an SR engine. The topics covered include:

- Typical file input scenario
- In-process (InProc) versus shared engines
- Relevant to setting up and using wav files as input to the speech recognizer
- Sample source code (written in both C++ and Visual Basic 6.0) to help guide developers.

Typical file input scenario

There are many different types of audio input configurations used by SR applications, which include:

- A microphone shared by all desktop applications
- A telephony card communicating with one or more SR engines
- Sending audio from a persisted wav file to an SR engine

The shared desktop microphone scenario uses the default SR engine and the default audio input. The user selects each in Speech properties in Control Panel, and each is hosted in the shared speech server.

The telephony scenario can use either the SAPI 5 standard multimedia audio input object or a custom audio object combined with an InProc SR engine.

The wav file input scenario is special because it uses controlled, reproducible audio input and requires a dedicated SR engine, without interference from other applications (e.g., a shared desktop microphone). The file input scenario should use a generic SAPI audio stream connected to the input wav file and an InProc SR engine.

Typical scenarios that would use the wav file audio input configuration include:

- Offline transcription applications (e.g., convert voice mail to email)
- SR engine testing (e.g., measure and improve engine accuracy with reproducible audio input data)
- SR application testing (e.g., verify and improve application behavior when responding to reproducible voice commands)

Follow these basic steps to perform SR on a wav file:

1. Create and configure basic SAPI audio stream object for wav file input
2. Create an InProc SR engine using the code samples in this document
3. Set the audio stream object from step 1 as the SR engine's input
4. Activate grammars and begin SR
5. Respond to recognition events until end of audio stream is reached

Relevant wav audio file input APIs for COM/C/C++ Developers:

- SpStream object, ISpStream interface: Basic SAPI audio stream
- ISpStream::BindToFile: Set up audio stream for wav file input
- SpBindToFile: Helper function to setup stream with a wav file
- SpInprocRecognizer, ISpRecognizer: InProc SR engine
- ISpRecognizer::SetInput: Set stream object as engine's input
- SPEI_START_SR_STREAM, SPEI_END_SR_STREAM: Event signaling engine has reached the beginning or the end of the wav file, respectively

Relevant wav audio file input APIs for Automation/Visual Basic/Scripting Developers:

- SpFileStream object: Basic file-based SAPI audio stream
- SpInprocRecognizer, ISpeechRecognizer: InProc SR engine
- SpInprocRecoContext, ISpeechRecoContext: InProc SR context
- ISpeechRecognizer::AudioInputStream property: Set file stream object as engine's input
- ISpeechRecoContext::EndStream/StartStream events

Wav audio file input outcome specific to SAPI

Finite-length audio input stream

Unlike microphone input which has no predetermined stream length, a finite-length audio input stream is a file which has a specific length that is known before recognition begins. Similarly, applications that use microphone input will toggle between actively listening and not listening states until the

speech application is closed. However, transcription applications are typically designed to listen to one continuous audio stream, and then close when the stream ends. Consequently, the application must specifically acknowledge the end audio stream event (SPEI_SR_END_STREAM for C/C++, ISpeechRecoContext::EndStream event for Automation). Transcription applications can potentially record multiple recognitions on a single audio stream, if the speaker pauses or breaks between sections of audio. If the transcription application exits after the first recognition event is received, it will miss any further recognizable audio that remains.

Non-real-time audio input

Microphone input and networked audio streams are typically real-time audio objects. This means that the audio object is designed to support audio buffering and dynamic state manipulation (e.g. stop->play->pause->play->stop) to handle delays and latency in the audio source and/or the SR engine's processing.

Sample wav audio file input source code

COM/C++ Developers

C-style is very similar to C++ and COM

```
{
    CComPtr<ISpStream> cpInputStream;
    CComPtr<ISpRecognizer> cpRecognizer;
    CComPtr<ISpRecoContext> cpRecoContext;
    CComPtr<ISpRecoGrammar> cpRecoGrammar;

    // Create basic SAPI stream object
    // NOTE: The helper SpBindToFile can be used to perform this
    hr = cpInputStream.CoCreateInstance(CLSID_SpStream);
    // Check hr
    CSpStreamFormat sInputFormat;
    // generate WaveFormatEx structure, assuming the wav format
    hr = sInputFormat.AssignFormat(SPSF_22kHz16BitStereo);
    // Check hr

    // setup stream object with wav file MY_WAVE_AUDIO_FILENAME
    // for read-only access, since it will only be accessed by
    hr = cpInputStream->BindToFile(MY_WAVE_AUDIO_FILENAME,
        SPFM_OPEN_READONLY,
        sInputFormat.FormatId(),
        sInputFormat.WaveFormatExPtr(),
        SPFEI_ALL_EVENTS);

    // Check hr

    // Create in-process speech recognition engine
    hr = cpRecognizer.CoCreateInstance(CLSID_SpInprocRecognizer);
    // Check hr

    // connect wav input to recognizer
    // SAPI will negotiate mismatched engine/input audio format
    hr = cpRecognizer->SetInput(cpInputStream, TRUE);
    // Check hr

    // Create recognition context to receive events
```

```

hr = cpRecognizer->CreateRecoContext(&cpRecoContext);
// Check hr

// Create grammar, and load dictation
// ignore grammar ID for simplicity's sake
// NOTE: Voice command apps would load CFG here
hr = cpRecognizer->CreateGrammar(NULL, &cpRecoGrammar);
// Check hr
hr = cpRecoGrammar->LoadDictation(NULL, SPLO_STATIC);
// Check hr

// check for recognitions and end of stream event
hr = cpRecoContext->SetInterest(SPFEI(SPEI_RECOGNITION) |

// use Win32 events for command-line style application
hr = cpRecoContext->SetNotifyWin32Event();
// Check hr

// activate dictation, and begin recognition
hr = cpRecoGrammar->SetDictationState(SPRS_ACTIVE);
// Check hr

// while events occur, continue processing
// timeout should be greater than the audio stream length
BOOL fEndStreamReached = FALSE;
while (!fEndStreamReached && S_OK == cpRecoContext->WaitFor
{
    CSpEvent spEvent;
    // pull all queued events from the reco context's event

while (!fEndStreamReached && S_OK == spEvent.GetFrom(c
{
    // Check event type
    switch (spEvent.eEventId)
    {
        // speech recognition engine recognized some aud.
        case SPEI_RECOGNITION:
            // TODO: log/report recognized text
            break;

        // end of the wav file was reached by the speech
        case SPEI_SR_END_STREAM:

```

```

        fEndStreamReached = TRUE;
        break;
    }

    // clear any event data/object references
    spEvent.Clear();
} // END event pulling loop - break on empty event q
} // END event polling loop - break on event timeout OR

// deactivate dictation
hr = cpRecoGrammar->SetDictationState(SPRS_INACTIVE);
// Check hr

// unload dictation topic
hr = cpRecoGrammar->UnloadDictation();
// Check hr

// close the input stream, since we're done with it
// NOTE: smart pointer will call SpStream's destructor, a
hr = cpInputStream->Close();
// Check hr
}

```

Automation/Visual Basic 6.0 Developers

Scripting code is similar to Visual Basic.

Option Explicit

```

Dim WithEvents RecoContext as ISpeechRecoContext ' context for
Dim Grammar as ISpeechRecoGrammar ' grammar
Dim InputFile as SpeechLib.SpFileStream ' wav audio input file

```

' Setup InProc reco context and wav audio input file

```
Private Sub MyForm_Load()
```

```
    ' Create new recognizer
```

```
    Dim Recognizer as New SpInprocRecognizer
```

```
    ' create input file stream
```

```
    Set InputFile as New SpFileStream
```

```
    ' Defaults to open for read-only, and DoEvents false
```

```
    InputFile.Open MY_WAVE_AUDIO_FILENAME
```

```

' connect wav audio input to speech recognition engine
Set Recognizer.AudioInputStream = InputFile

' create recognition context
Set RecoContext = Recognizer.CreateRecoContext

' create grammar
Set Grammar = RecoContext.CreateGrammar
' ... and load dictation
Grammar.DictationLoad

' start dictating
Grammar.DictationSetState SGDSActive
End Sub

' Event fired on app shutdown
Private Sub MyForm_Unload(Cancel as Boolean)
    InputFile.Close ' close audio input file
End Sub

' Event fired when speech recognition engine recognizes audio
Private Sub RecoContext_Recognition(StreamNumber as Long, StreamNumber as Long)
    ' Log/Report recognized phrase/information
End Sub

' End of wav Input Stream reached by speech recognition engine
Private Sub RecoContext_EndStream(StreamNumber as Long, StreamNumber as Long)
    ' Disable dictation and unload grammars on app close
    Grammar.DictationSetState SGDSInactive
    Grammar.DictationUnload

    Unload Me ' shutdown app on end of stream
End Sub

```



Automation Interfaces and Objects

The **Automation Interfaces** present provide object-oriented access to the speech recognition and text-to-speech capabilities of SAPI.

Please note that all automation interface names begin with "ISpeech" and that all automation object names begin with "Sp." Applications can explicitly create object variables which instantiate automation objects, using the "CreateObject" statement or the "New" keyword in a "Dim" or "Set" statement. Object variables which instantiate automation interfaces, on the other hand, are only created by the methods, properties and events of automation objects.

Additionally, some automation interfaces are implemented by automation objects, and the properties and methods of those interfaces are inherited by the objects. For example, the ISpeechBaseStream interface defines a set of properties and methods for storing and manipulating audio data in memory. The SpFileStream, SpMemoryStream and SpCustomStream objects implement the ISpeechBaseStream interface; as a result, the methods and properties of the ISpeechBaseStream interface are available in all three objects.

Automation Interface and Objects

SAPI 5.1 Automation consists of the following interfaces and objects:

Interfaces	Description
ISpeechAudio	Supports the control of real-time audio streams, such as those connected to a live microphone or telephone line.
ISpeechAudioBufferInfo	Defines the audio stream l

	information.
ISpeechAudioStatus	Provides control over the control of real-time audio streams
ISpeechBaseStream	Defines properties and methods common to all audio streams
ISpeechDataKey	Provides access to the speech configuration database.
ISpeechGrammarRule	Defines the properties and methods of a speech grammar rule.
ISpeechGrammarRules	Represents a collection of ISpeechGrammarRule objects
ISpeechGrammarRuleState	Presents the properties and methods of a speech grammar state.
ISpeechGrammarRuleStateTransition	Returns data about a transition from one rule state to another, or from a rule state to the end of a rule.
ISpeechGrammarRuleStateTransitions	Represents a collection of ISpeechGrammarRuleStateTransition objects.
ISpeechLexiconPronunciation	Provides access to the pronunciation of a speech word.
ISpeechLexiconPronunciations	Represents a collection of ISpeechLexiconPronunciation objects.
ISpeechLexiconWord	Provides access to a speech word.
ISpeechLexiconWords	Represents a collection of ISpeechLexiconWord objects
ISpeechObjectTokens	Represents a collection of SpObjectToken objects.
ISpeechPhraseAlternate	Enables applications to retrieve alternate phrase information from an SR engine, and to update

	engine's language model t committed alternate chan
ISpeechPhraseAlternates	Represents a collection of ISpeechPhraseAlternate ob
ISpeechPhraseElement	Provides access to informa about a word or phrase.
ISpeechPhraseElements	Represents a collection of ISpeechPhraseElement ob
ISpeechPhraseInfo	Contains properties detaili elements.
ISpeechPhraseProperties	Represents a collection of ISpeechPhraseProperty ob
ISpeechPhraseProperty	Stores the information for semantic property.
ISpeechPhraseReplacement	Specifies a replacement, c normalization, of one or m spoken words.
ISpeechPhraseReplacements	Represents a collection of ISpeechPhraseElement ob
ISpeechPhraseRule	Contains information abou speech phrase rule.
ISpeechPhraseRules	Represents a collection of ISpeechPhraseRule objects
ISpeechRecognizerStatus	Returns the status of the s recognition engine represe the recognizer object.
ISpeechRecoGrammar	Enables applications to ma words and phrases for the engine.
ISpeechRecoResult	Returns information about recognition engine's hypot recognitions, and false rec
ISpeechRecoResultTimes	Contains the time informa speech recognition results
ISpeechVoiceStatus	Contains status informatio

an SpVoice object.

Objects	Description
SpAudioFormat	Defines an audio format.
SpCustomStream	Supports supports the use of existing IStream objects in SAPI.
SpFileStream	Provides the ability to open files as audio streams and save audio streams as files.
SpInProcRecoContext	Defines a recognition context, or a collection of settings, that requests a specific type of recognition as determined by the needs of an application.
SpInProcRecoContext (Events)	Defines the types of events that a recognition context can receive.
SpInProcRecognizer	Represents a speech recognition engine.
SpLexicon	Provides access to lexicons, which contain information about words that can be recognized or spoken.
SpMemoryStream	Supports audio stream operations in memory.
SpMMAudioIn	Represents the audio implementation for the standard Windows wave-in multimedia layer.
SpMMAudioOut	Represents the audio implementation for the standard Windows wave-out multimedia layer.
SpObjectToken	Supports object token entries.
SpObjectTokenCategory	Represents a class of object tokens.
SpPhoneConverter	Supports conversion from the SAPI character phoneset to the Id phoneset.

<u>SpPhraseInfoBuilder</u>	Provides the ability to rebuild phrase information from audio data saved to memory.
<u>SpSharedRecoContext</u>	Defines a recognition context, or a collection of settings, that requests a specific type of recognition as determined by the needs of an application.
<u>SpSharedRecoContext (Events)</u>	Defines the types of events that a recognition context can receive.
<u>SpSharedRecognizer</u>	Represents a speech recognition engine.
<u>SpTextSelectionInformation</u>	Provides access to the text selection information pertaining to a word sequence buffer.
<u>SpUnCompressedLexicon</u>	Provides access to lexicons, which contain information about words that can be recognized or spoken.
<u>SpVoice</u>	Enables an application to perform text synthesis operations.
<u>SpVoice (Events)</u>	defines the types of events that can be received by an SpVoice object.
<u>SpWaveFormatEx</u>	Defines the format of waveform-audio data.



Automation Overview

Automation is the ability of one entity (or object) to communicate with another object. These entities may be applications or ActiveX® controls, although ultimately they all use COM (component object model) as the common denominator. The underlying complexity is removed for the Visual Basic programmer. The result is that your application can access features and information from other sources. This interaction can be as involved as sharing information between the two applications, such as between your Visual Basic application exporting or importing data from a database or spreadsheet. Automation also allows you to issue basic commands such as Open and Print. This way, developers can integrate their favorite spreadsheet or word processor capabilities with methods unique to a particular corporate environment. For example, while you could write your own spell checker for your Visual Basic application, your application could access the Microsoft Word spell checker, thus saving time and effort. In Visual Basic, this scenario using standard applications is commonly referred to as Visual Basic for Applications (VBA). VBA is a specialized subset of Visual Basic and is designed to manage applications for custom solutions.

A second use of automation includes components. A component can be considered an additional functionality provided outside of the capabilities of any programming language itself. For example, Visual Basic provides a basic text box for displaying text and although that text may be italic or bold, all the text must share the same characteristics. For example, it must all be bold, or all the same font size. On the other hand, a rich text edit box may be added to the Visual Basic Toolbox. Using this rich text edit box, you can change the font, size and appearance of the text. However, this capability must be explicitly added using the appropriate component on the Components menu. In reality, that rich text edit box is a COM object. Because COM is intended to offer programmers flexibility, there is a robust set of

components available. Many come with the Visual Basic package, others are available through third party sources, and some are designed for specific purposes such as in-house corporate environments. This ability to compartmentalize features and functionality gives components their popularity.

A third use of automation is its ability to access dynamic link libraries (or DLLs). This aspect of automation involves applications without a user interface. SAPI, for instance, has no interface per se. While users can incorporate SAPI functions into their own applications, they cannot run SAPI in a conventional sense and have a user interface, as well as menus and documents. Developers however, have access to all of SAPI's functions in a programmatic way using dynamic link libraries (DLLs). If DLLs exist on the computer, a developer can access them by linking their development environment to them. In the case of Visual Basic, SAPI may be added using the *Microsoft Speech Object Library* on the References menu. If a particular DLL is not present, it can be installed using the manufacturer's installer. For example, sapi.dll is loaded when developers install SAPI.

DLLs are used frequently. A search for *.dll reveals the true extent of their use. Using them, developers can introduce new components or capabilities to an operating system without compiling them into the system. DLLs may be, and usually are, loaded after the operating system. However, their mere presence does not affect any application. As mentioned, developers need to make calls to DLLs before a DLL can have an effect on an operating system. Since SAPI has no interface, this document is the only guide to accessing its features. The application programming interface (API) acts as a guide and provides a list of all the possible calls, explanations of them, and guidelines for successfully using them. Certain calls must be made in a specific order to keep SAPI automation simple. This API explains these issues and eases development concerns. Also available with this documentation is the software development kit (SDK), which provides supplementary information such as

code examples and additional tools. Developers are encouraged to review the sample code and possibly base their applications on it.

Even though three different uses of automation are described above, ultimately all automation uses an identical foundation. Automation itself is a complex technology. It has evolved over the years and, no doubt, will continue to evolve to accommodate changing technologies and the increasing sophistication demanded by the marketplace. Older programmers will remember OLE (object linking and embedding), which evolved into ActiveX, COM, and COM+. All of these programs are progressions of what is commonly called automation. Fortunately automation's complexity is hidden. Support is usually built into the operating system now and accessing these technologies, or individual aspects of them, is nothing more than making a selection from a list within the development environment.

The concept of automation using COM is that of programming language independence. As long as the object complies with COM standards, any programming language may be used. This documentation suite addresses the SAPI automation programming using Visual Basic; however, Visual Basic is not actually required. In fact, many of the same calls and calling syntax may be used in other computer languages, provided they support automation. This includes the ability to use JScript®, Visual C#®, or possibly other solutions that support automation. Visual Basic is used here because of its popular and widespread support in marketplace. Also, it has supported automation virtually since its inception and the Visual Basic tool suite offers a variety of options.



Objects and Classes Overview

Classes

For automation to work, there must be technology common to all applications. Though still evolving, the current incarnation of that technology is the component object model (COM). Many applications are COM-compliant and the usage is widespread. Using this technology, COM-compliant applications can issue and receive commands from other applications. The details of COM and automation are complex, but almost all of the issues are hidden from the user or application developers.

The functionality of these components is encapsulated in distinct files called type libraries. Type libraries contain all the methods, properties, and supporting data for the applications to use; the application just needs to load the appropriate type library to gain access to the functionality. In Visual Basic, type libraries are also called references. Some type libraries are loaded automatically and others must be loaded explicitly. This will be covered later and in more detail. Once loaded, Visual Basic has complete access to the contents of the type library.

Libraries themselves contain many separate items. For example, The Microsoft Speech Object Library (once loaded, it is called SpeechLib in the references) contains over 400 individual methods and properties, not including supporting data. Therefore, to better organize functionality, related calls and data are grouped together in classes. For example, SAPI has an [ISpeechRecoResult](#) class. This class contains functionality needed to assess the results of successful speech recognition. To retrieve the text of the spoken recognition, use the GetText method from ISpeechRecoResult.

However, classes themselves are just a blueprint for the functionality. That is, classes describe the functions, but they cannot execute the function. To execute the function, an object

must be created (also called instantiated). In this way, one class may be instantiated any number of times.

Once an object is created, Visual Basic uses a standard notation for making these calls. This is an *object.method* notation (pronounced "object dot method"). A typical call will look like this: `ISpeechRecoResult.PhraseInfo.GetText()`. Complete ownership could also be used by including the library name: `SpeechLib.ISpeechRecoResult.PhraseInfo.GetText()`.

To determine what is available from a particular type library after loading it, Visual Basic offers an object browser. The browser is available by either pressing F2 or selecting View->Object Browser from the menu. The object browser displays the information in a graphical format. Clicking or double-clicking an item reveals additional information about it including properties, methods, events, or classes that it contains. The bottom pane displays a quick reference guide for the selection and includes the owning object memberships up through the library. This method of browsing available objects can be useful when learning a new library. In addition, two additional modules `SpeechConstants` and `SpeechStringConstants`, are available for viewing. These two modules contain names for numeric and string constants used by SAPI.

Objects

As mentioned, classes are just blueprints for a certain functionality. To use the calls of a class, an object must be created. An object can allocate memory and assign values to various elements contained by the object. Making an instance of the object is similar to declaring variables. For example, the variable type `String` is a common occurrence in Visual Basic programs. The type allows a character string to be used, saving information in it, and later retrieving information from it. However, before using the variable an instance must be created. In the following example, one `String` instance is created:

```
Dim myString As String
```

After this appears in the code, the variable *myString* is valid and may be referenced and changed as needed. In this sense, *String* can be considered the class and *myString* is an instance of the class. Likewise, it is possible to declare several instances of `String`.

```
Dim myString, userName, fileName As String
```

Now three strings are available for use, presumably to record information for vastly different purposes. Perhaps *userName* may be used to track the user's identity, and *fileName* for recording the location and name of a file. This is an example of one data type having several instances. The same will hold true for classes and interfaces.

Instantiating an object is similar, but requires slight modifications. The additional modifications are in the second line:

```
Dim myVoice As SpVoice  
Set myVoice = New SpVoice
```

The additional keywords are **Set** and **New**. Classes require these two keywords since classes are more complex than variables. Once declared and allocated, the object may be used and referenced. However, simply creating an object may not be enough and additional initialization may be required. Just as creating *myString* allocates the variable for use, the programmer must assign a value to it. In fact, it is possible for an object to have no valid reference. In this case, the value will be Nothing. The following code tests for an object which is Nothing.

If myVoice Is Nothing Then

 'Handle the situation here. You might want to allocate it or

 'warn the user an error could have occurred.

End If



Events Overview

Taken in the most general sense, the concept of automation is that of two objects communicating with each other. These objects may be two applications, or an application and a COM component. Ultimately, however, it comes down to two COM components, although that is not really the point. The point is two items are exchanging information between them.

In the case of SAPI, it is easy to get the impression that the Visual Basic application makes all the calls to SAPI and receives nothing in return. You can get that impression because most of the emphasis of this documentation suite features the several hundred available calls to SAPI. But in fact, there is a two-way communication going on. The feedback from SAPI and the speech engines plays a critical role in a speech application.

SAPI interacts with the host application using events. An event is a signal sent to the application by an outside source. The event indicates that some condition exists outside of the application that may be of interest to the user. The outside source in this case is SAPI and examples of events may indicate that a recognition is available or that the SAPI engine is ready to accept input.

Events are not limited or unique to SAPI. They are the standard method to get information back to the host application. It is common for other applications to send events. An example of an event sent back to an application is that of an e-mail application running in the background notifying users whenever a new e-mail arrives. Unlike a conventional method or property, which must be explicitly coded by the programmer and called at a specific time, events may be sent or received at any time. It is even possible for events to be sent faster than the host application can process. Yet this rarely is disruptive to the user or the current application.

Far from interfering with the application, events intend to

enhance the user experience. For example, moving the pointer over menu items alters the appearance of individual buttons and indicates that the menu item is available. This type of event is handled transparently to users, and they do not need to respond in any way. It is even transparent to programmers because the change in appearance is automatic and requires no code implementation. Nevertheless, the application is responding to an event.

Events can be considered to be of one of two types: user-initiated or background generated. As the name implies, a user-initiated event is one initiated by the user. A common instance is that of the user clicking a command button. This actually generates several events including a MouseDown event and Click event. A graphical user interface (GUI, pronounced gooey) is an example of many user-initiated events; that is, the user starts various activities such as clicking a menu, opening an application, or entering text. Conventional GUI programming relies extensively on an event loop. This endless loop waits for events. After receiving one, the event is dispatched and whatever action it is intended to do begins. Visual Basic applications have an event loop, although it is hidden from the programmer.

Background generated events, on the other hand, are events initiated not by the user but by some other source such as another application (such as SAPI) or the operating system. This is how applications inform one another about something of interest. For instance, if an application were using automation with Microsoft Word, it is possible for Word to send an event whenever a file is deleted, printed, or opened. The application could respond or even ignore the event if it were unimportant. SAPI sends back events in the same way. For example, when SAPI finishes processing a phrase and has a recognition ready, it sends a recognition event to the host application. Handling an event is similar to a function or subroutine except an event is invoked automatically.



Types of Automation Events

Introduction

Chances are you have already have used events and may not have known it. If your application has a command button, (a common interface tool) events generate the action for it. For example, clicking the command button when the application is running usually initiates a certain action. Clicking the button sends out a button Click event. However the programmer had to write the code for the action and chances are that code is contained in a procedure perhaps called *Command1_click*. The word *click* indicates that it is an action. This is an example of a user-initiated event; that is, the user must click the button to perform its action.

There are other examples of user initiated events. For example, the first time a form is displayed, a *Form_Load* event is generated. Perhaps not as common as a *Command_Click* event, the *Form_Load* event initializes information on a particular form. Though still considered to be a user-initiated event, the *Form_Load* event may or may not be a direct result of user interaction. You could have chosen a menu item that displays a new dialog box (and hence a form). Or perhaps the form could also be the result of an error handling routine initiated by an activity you caused indirectly. Regardless of what caused the event, the application can have code to handle the situation. The particular event could be ignored completely, in which case you do not need code for the event.

Background-generated events work in a similar fashion as user-initiated events. The difference is the background-generated event may be fired at anytime and is not the direct result of user interaction, at least not in the same way as clicking a button. SAPI events fall into this category. For either speech recognition or text-to-speech, SAPI returns events to the application. The application then addresses or ignores the

event. Furthermore, the application can selectively reject certain types of events if there is no need to work with them. Speech recognition returns a spectrum of events from noise interference prohibiting recognition to notifications (successful or otherwise). SAPI returns additional events for activities such as the start and end of media streams, or the start or end of sound input, which are required for potential recognition attempts. Text-to-speech has events related to it as well.

Speech Recognition Events

The enumeration [SpeechRecoEvents](#) contains the speech recognition events. Upon review of these events, many appear logical and obvious. The speech recognition event `SRERecognition` is returned upon a successful recognition; `SREPhraseStart` handles the start of a new phrase. Other events are less obvious and have specialized functions. For example, `SRERecoOtherContext` indicates that a successful recognition occurred but cannot be associated with the current recognizer context. For a detailed explanation of events and some situations where they might be used, please see the appropriate SAPI documentation.

For a complete list of speech recognition events see [ISpeechRecoContext \(Events\)](#)

Text-To-Speech Events

The enumeration [SpeechVoiceEvents](#) contains all the text-to-speech events. Similar to recognition events, some voice events may seem obvious, and others less so. See the appropriate SAPI documentation.

For a complete list of text-to-speech events see [SpVoice \(Events\)](#)

Exploring Types of Events

In general, there are two ways to discover what events are available. First, the application programming interface (API) reference lists them and describes their use in detail. In order to gain a better understanding of any event or method, it is important to review those event topics first.

A second method is to view the Object browser from within Visual Basic. Click F2 to display the browser or select View->Object Browser from the menu. Events, like other items, have an associated icon, in this case a lightning bolt.

In any case, SAPI has only two interfaces with events associated with them:

- [SpVoice](#)
- [ISpeechRecoContext](#)



Automation Event Handling

Event Handlers

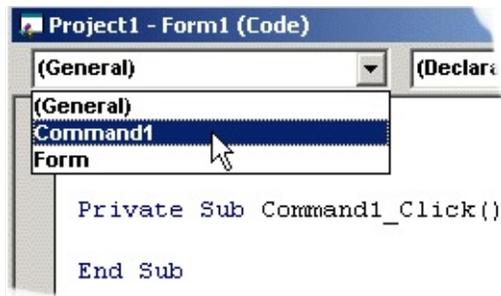
In order to respond to an event, the application needs a corresponding event procedure (also called an event handler). If an event does not have an associated procedure, then that event is not considered handled and the application does not act upon it. By default, no event procedures are included in any Visual Basic applications. For example, a programmer may include a command button on a form. But the mere presence of the button does not mean that there is an associated activity if the user clicks the button. Instead, the programmer must explicitly define the activity. In the same way SAPI returns several types of events by default. However, unless the host application has the code to process the event, it will appear that the event is not handled.

In Visual Basic, defining the procedure is simple and can be done in one of three ways.

1. Double-click the control while in design mode. Visual Basic programmer are probably most familiar with this method. For example, double-click the command button while in design mode. Visual Basic automatically includes the code necessary to support the function. This code includes the event type (again, for this example, the event type is button Click) and any necessary parameters. In fact, programmers do not need to know the number or kinds of parameters. Visual Basic includes them automatically.

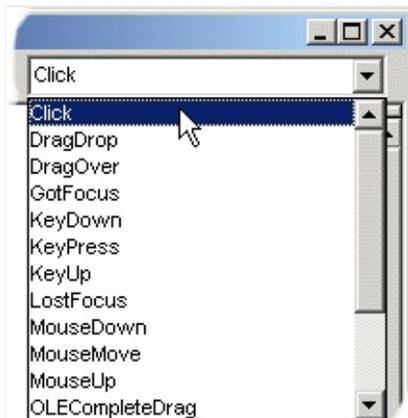
2. Use the Object or Procedures drop-down list boxes in the Visual Basic editor. The Object list presents all the available objects on the form. From this list, the programmers can directly choose the procedure they want rather than scrolling through the code or searching for the item name.





The Object Menu in Visual Basic. This lists all the objects on the form. In this example, the form only contains a single item: a button named Command1. However, the form itself is an object and can have events associated with it. By default, there is also a General item. This includes functions not associated directly with any control.

The Procedures list presents all the events available for the selected object. Keep in mind that clicking a specific control in the forms window only brings up one kind of event, usually the click event. Objects often have many other events associated with it. New events may be added for the item selected in the Object menu. Select the intended event from the Procedures menu. Visual Basic automatically opens the code window, and includes the supporting code necessary to the event. Again, programmers do not need to know parameter information.



The Procedures menu in Visual Basic. With an object selected from the Object menu, the available events are listed. Using the previous example, the events listed are for the Command1 button. Choosing an event (the click event is highlighted) will automatically insert the subroutine or function along with the required parameters.

3. Consult the API documentation for the event. This presents the information and parameter requirements for the event. The function can then be written or pasted in manually as code. This is the least automated method of the three. However, not all development environments support automatic generation of

code and this may be the only option in some cases.

In the simple case of a command button Click, any of the three methods above results in the following code:

```
Private Sub Command1_Click()
```

```
End Sub
```

Whenever Command1 is clicked, it uses the code in this routine. Naturally, it is up to programmers to write the procedure to handle the event as they see fit.

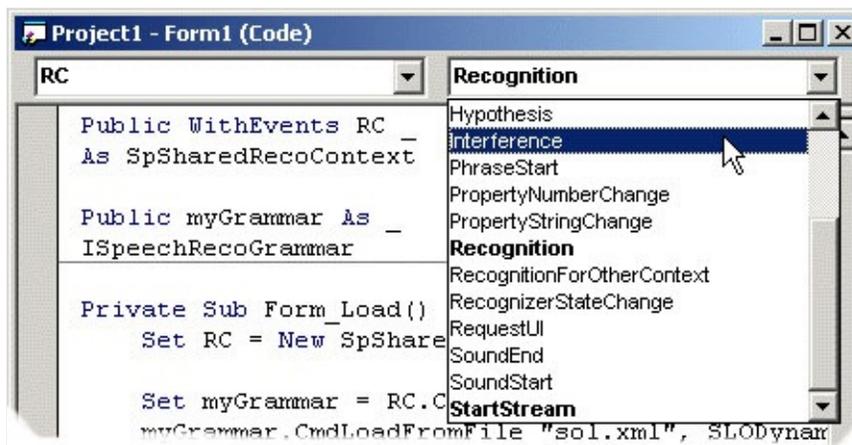
SAPI Event Handlers

Adding SAPI event handlers is similar to other controls. First, the object associated with the events must be declared. For example, declare a new recognizer object in the project

```
Public WithEvents RC As SpSharedRecoContext
```

Notice also the declaration has an additional keyword in it: `WithEvents`. This keyword is required to identify the object as an event source and thus allowing events to be associated with the recognition object. It may also be misleading since the application will still compile and run if it is omitted. If the keyword is missing, no events will be returned. Once declared, the Procedures menu will then list the events.

With the same ease as selecting an event for a conventional control, you can select an event for the SAPI object. For instance, selecting Recognition from the Procedures menu inserts the appropriate code in the open project file.



An example of the Procedures menu with SAPI. The recognition context is declared as the `RC` object. The Procedures menu lists all the available events for the recognition context. The two in bold indicate those events are already defined. The other events listed will have the code generated for them if event is selected.

A [Recognition](#) event is defined as

```
Private Sub RecoContext_Recognition(ByVal StreamNumber As Long
```

```
End Sub
```

Events in Scripting Languages

In scripting languages, use the HTML object tag to instantiate a SAPI object. Event procedures for such an object are identified by the object-ID and the event name. For example, the following JavaScript code snippet demonstrates the definition and creation of an SpVoice object and the definition of a typical Word event procedure.

```
...
<OBJECT ID="VoiceObj" CLASSID="clsid:96749377-3391-11D2-9EE3
<SCRIPT LANGUAGE="JScript">

function VoiceObj::Word(Number, Position, CharacterPosition,
{
    idTextBox.select();
    var CurrentRange = document.selection.createRange();
    var MyLen = CurrentRange.text.length;
    CurrentRange.moveStart("character", CharacterPosition);
    CurrentRange.moveEnd("character", CharacterPosition + MyLen);
    CurrentRange.select();
    idTextBox.focus();
}
...
```

There are two ways for applications developers to create a recognizer and a recognition context. The first is to create a recognizer object and then to derive a recognition context from the recognizer. The other is to create a recognition context and then to derive the recognizer from the context.

But in a scripting environment, events can only be received by objects created by the scripting host. And since recognition results are received exclusively through recognition context events, consequently, in a scripting environment, the

recognition context must be created first. As a further consequence, the recognizer created from the recognition context will be associated with the default SR engine.

Event Parameters

The SAPI engines return information back to the application through the event handler's parameters. SAPI events can come from different sources and there can even be multiple instances of recognition contexts or voices. However each event is self-contained and has enough information to relate the event to a specific and unique source. If the recognition context or even the recognizer itself is important, you can derive the source by tracing back information through the parameters.

In the case of a Recognition event, a common situation is that the application is interested only in the last parameter, the recognition result. This contains information about the recognized phrase including the associated text. Parameters are not required to be used. In some instances, it may be important to know only that a recognition occurred; the application may not process any information any further.

Note also the event's parameters represent one-way communications, that of SAPI to the application. It is not possible to change the parameter information and send the changed parameters back to the engine.



Using Events in Code

Associated With Recognition Contexts

Once the event handler is created, using that particular event is simple. No additional code is needed to initiate the event. That is, if the application receives a Recognition event from SAPI, the Recognition code is invoked automatically and immediately. For example, if the recognition event code were as follows for, notice two issues.

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal S  
    Form1.Label1.Caption = Result.PhraseInfo.GetText  
End Sub
```

First, a hypothetical Label1 text box on Form1 of the application would be updated with the text from the successful recognition. Second, notice each event is associated with a particular recognition context. In this case recognition context is presumed to be named RC. A possible declaration could be

```
Public WithEvents RC As SpSharedRecoContext
```

Associating each event with a recognition context simplifies handling the event. Since each application can have more than one recognition context, a particular recognition context can be active or inactive at any moment, and recognition contexts are generally used to isolate specific application areas (such as one recognition context for the menus, another for general dictation, and possible others for dialog boxes), when an event is received, the scope of the event is automatically defined. For example, in the RC_Recognition listed above, the event simply displays the results to a text box. The application does not have to test if the event related to menus, which would probably require a different set of actions.

Applications, however, can have additional processing within the event. For example, assuming there is one recognition context for all the menus, the following recognition event does have to test for which menu is intended. It uses a context free (CFG) grammar and all the rule names are defined inside that file.

```
Private Sub MenuBar_Recognition(ByVal StreamNumber As Long, B)
```

```
    Set RecoResult = Result
```

```
    Dim rp As ISpeechPhraseInfo
```

```
    Set rp = RecoResult.PhraseInfo
```

```
    If rp.Rule.Name = "filemenu" Then Form2.Command1_Click
```

```
    If rp.Rule.Name = "editmenu" Then Form2.EditMenu_Click
```

```
    If rp.Rule.Name = "aboutapplication" Then frmAbout.Show vbM
```

```
End Sub
```

Event Interests

Both speech recognition and text-to-speech have a wide variety of events. By default, SAPI using automation has all events active except for `SREAudioLevel` for speech recognition and `SVEAudioLevel` for text-to-speech. That means all the other kinds of events will be returned by SAPI to the application. Remember, the application does not have to support any particular events; it could simply omit an event handler for it. Nevertheless, the application still receives the event and it takes a minimal amount of time to process it through the queue. For simple applications, this processing time is negligible and has little or no effect. For other applications this additional time could represent an undesirable lag. For instance, if the application were animating a face by using visemes to change the mouth position of the on screen character, speed and timing is important. For speech recognition, a game application could rely on speed as well to process a voice command. While some amount of lag time is inherent to all speech systems (speech processing is not instantaneous), removing unwanted events is one way to minimize the demands on the application.

Controlling the flow of events is done by setting event interests. Event interests is a filtering mechanism by which specific events may be sent back to the application or repressed by the engine to begin with. All, none, or selective events may be chosen to be received. Both groups of events have a similar call for setting interests: speech recognition has [*`ISpeechRecoContext.EventInterests`*](#) and text-to-speech uses [*`SpVoice.EventInterests`*](#). Both methods sets event interest but for the appropriate set of interests.

Event interests work the same way for either technology. The `SetInterests` specifies only the events to be sent by the engines. Therefore the following call allows a single event to be sent, that of the successful recognition. The sample also assumes a valid *RecoContext*.

RecoContext.EventInterests = SRERecognition

Additional interests may be set by adding the values together. So that the next sample now allows two events be sent, a recognition and a sound start.

RecoContext.EventInterests = SRERecognition + SRESoundStart

If addition is used, the new value completely replaces the previous set of interests. For example, if the last two samples were executed in the reverse order than presented, the recognition context would only use one event afterward: the recognition. It is not recommended to use subtraction to modify interests. Instead, use logical operators.

Logical operators may also be applied to event interests. For example, an application may need to add one specific event for the moment. Instead of having to refine the entire set of interests, a potentially laborious task if many interests are currently set, use the logical And operator. For example, the following code tests if the current set of interests excludes SRERecognition (by using the logical And) and if not, adds it (by using the logical Or).

```
If (RC.EventInterests And SREAudioLevel) <> SREAudioLevel Then  
    RC.EventInterests = RC.EventInterests Or SREAudioLevel  
End If
```

Setting interests is more simple than retrieving them. Symbolic equivalent may be to set interests. That means to add recognition events the Symbolic equivalent *SRERecognition*. However, in retrieving the current set of interests, a numeric value is returned instead. For example, by default voice event interests are set to 33,287. The value is the sum of all the individual events. An event interest of SVEWordBoundary, SVEVoiceChange, and SVEViseme would have a value of 296.

Microsoft Speech SDK

Speech Automation 5.1



Using the Visual Basic Code Examples

Prerequisites

To run the code examples contained in this documentation, your computer must have the following installed:

- SAPI 5.1
- Visual Basic 5.0, Visual Basic 6.0, or Visual Basic.Net
- Speakers

A microphone is helpful, but not necessary, for demonstrating speech recognition.

Getting Started

The code examples in this document are designed for use with Microsoft Visual Basic. Each example is identified either as a "code snippet" or as "form code." Code snippets are small sections of code intended to be placed within a single Visual Basic procedure; form code examples are designed to be placed within a Visual Basic form. All forms, modules, controls and resources use Visual Basic's default names, such as form "Form1," command button "Command1," and resource "101."

Following are the normal steps for running a code example:

Opening a project in Visual Basic

1. Open Visual Basic.
2. In the New Project dialog box, double-click Standard exe.
3. Visual Basic will display a new form called Form1.

Adding a "code snippet" example to the project

1. Double-click on Form1 to display the code.

2. Select the example code from the documentation and copy (Ctl-C).
3. Click inside the Form_Load procedure and paste (Ctl-V).

Adding a form code example to the project

If the code example uses controls, use the Toolbox to add them to Form1

1. Double-click on Form1 to display the code.
2. Select the example code from the documentation and copy (Ctl-C).
3. Select all code in Form1 and paste (Ctl-V).

Adding a reference to SAPI

1. From the Project menu, click References.
2. In the References list box, select Microsoft Speech Object Library.
3. Click OK.

About the Examples

Recognition of text-to-speech voices

Several of the speech recognition code examples perform recognition of audio files created by text-to-speech (TTS) voices. Speech recognition is designed to recognize human voices rather than TTS voices, so the quality of recognition from TTS voices is not as high as that from human voices; however, TTS voices are also much more consistent than human voices. Some code examples need to demonstrate types of recognition result data that are dependent on subtle factors in the speaking voice. In the instances where TTS recognition takes place, users can enter their own text. In some cases, it may be difficult to find a

phrase that will produce a particular recognition result from a TTS voice. Once such a phrase is known, a TTS voice speaking that phrase will reproduce those results very consistently. Most of these types of results would be very difficult for a user to reproduce by speaking into a microphone.



Programming Notes for Visual Basic

Hidden members

Certain methods or properties (collectively called members) are designated as hidden. This indicates that member does not display in a browser such as Visual Basic's IntelliSense (or automatic statement completion) menu or in the Object Browser list. Members may be designated as hidden for several reasons. Most commonly, it represents an advanced feature or capability that is not needed in normal situations. Hidden members should be considered carefully before using them.

To view hidden members

1. On the **View** menu, click **Object Browser**.
2. Right-click the **Object Browser** window, and then click **Show Hidden Members**. Drag the scroll box to view hidden members that appear in gray text. With **Show Hidden Members** selected, these will appear in the IntelliSense menu as well.
3. Right-click the **Object Browser** window, and then select **Show Hidden Members** again to hide the members.

Determining SAPI's presence

SAPI requires no special error handling outside of Visual Basic's standard procedures. That is, SAPI errors may be trapped and handled using the same techniques presented by Visual Basic.

However, there are two special considerations when working with SAPI. First, the computer on which the application is being developed must have the SAPI library loaded. See [Using the Code Examples](#) for details on setting up a Visual Basic environment for SAPI. Second, the computer running the application with SAPI automation must have SAPI 5.1 or later installed.

It is possible to run applications that include SAPI automation without SAPI 5.1. If this is the case, surround the SAPI code with conditional statements to ensure that no SAPI commands are actually executed.

The easiest way to determine if an appropriate version of SAPI is present on a computer is to simply make calls to SAPI and see if they fail. Use error trapping to catch and work around failed SAPI calls. If all the SAPI calls are caught, it is possible to run SAPI-enabled applications on computers without a compatible version of SAPI. Although, those applications cannot use SAPI functions or functionality, but a separate version of the application does not have to be created or maintained.

For instance, if the following code snippet is executed and SAPI 5.1 is not installed, a run-time error results.

```
Private Sub Form_Load()  
    Set RC = New SpSharedRecoContext  
  
    Set myGrammar = RC.CreateGrammar  
    myGrammar.DictationSetState SGDSActive  
End Sub
```

The error would be related to creating an object from nonexistent source. On a computer with no version of SAPI loaded, this would commonly display

Run-time error '459':

Object or class does not support the set of events

Therefore, the SAPI calls need an error handler for them. For example, the same code above could be trapped by the following version. The code intends to use *gSAPIPresent* as a flag marking SAPI's presence. If all the SAPI-related calls are conditional based on *gSAPIPresent*, the application could run on computers lacking SAPI support. Although those applications could run, voice features could not be used.

```
Private Sub Form_Load()
```

```
    On Error GoTo SAPINotFound
```

```
    Set RC = New SpSharedRecoContext
```

```
    Set myGrammar = RC.CreateGrammar
```

```
    myGrammar.DictationSetState SGDSActive
```

```
    gSAPIPresent = True
```

```
    Exit Sub
```

```
SAPINotFound:
```

```
    If Err.Number = 459 Then
```

```
        MsgBox "SAPI not found"
```

```
    Else
```

```
        MsgBox "Error encountered : " & Err.Number
```

```
    End If
```

```
    gSAPIPresent = False
```

```
End Sub
```

If an application needs to explicitly test for the presence of a compatible version of SAPI, use the following code to load a small SAPI object. If the call fails, SAPI is not present.

```
Private Sub Form_Load()  
{  
    'Use Visual Basic's built in error checking  
    On Error GoTo Err_SAPILoad  
  
    Dim PIB As ISpeechPhraseInfoBuilder  
    'Now if this call fails Visual Basic's Error handling will kick in and s  
    Set PIB = New SpPhraseInfoBuilder  
  
Err_SAPILoad:  
    MsgBox "Error loading SAPI objects! Please make sure SAPI5.1 is c  
}
```

Error Codes

SAPI error codes are listed in [Error Codes](#).

COM Reliance

SAPI automation is built on a COM foundation. As a result many of the SAPI automation calls are virtually identical to correspondingly named calls the C/C++ section of the SAPI documentation. It may be helpful read the related sections for additional insight and suggestions for automation calls. While the C/C++ references will specify C-style programming terms and code samples, in many cases the principles will be the same.



VB Application Sample: Dictation Recognition (Shared)

The following code sample represents a simple, but functional, recognition application. It uses a dictation grammar and allows free dictation. The commented lines refer to hypothetical text boxes in a form to possibly display information. Of course you may modify this application as needed to fit your own requirements.

Before running the application, a speech reference must be included. Using the Project->References menu, find and select the Microsoft Speech Object Library.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal S
    'Label1.Caption = Result.PhraseInfo.GetText
End Sub
```

```
Private Sub RC_StartStream(ByVal StreamNumber As Long, ByVal S
    'Label2.Caption = Val(StreamNumber)
End Sub
```



VB Application Sample: Command and Control Recognition

The following code sample represents a simple, but functional, recognition application. It uses a command and control grammar which limits the recognizable text to those listed in the configuration file. In this case, the file is named sol.xml and the file contents are listed in the second code box below. The text is restricted to the single command "new game."

The commented lines in the Visual Basic code refer to hypothetical labels in a form to possibly display information. If a speech attempt does not match the new game rule pattern, it could also display "(no recognition)". Of course you may modify this application as needed to fit your own requirements.

Before running the application, a speech reference must be included. Using the Project->References menu, find and select the Microsoft Speech Object Library.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar, b As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.CmdLoadFromFile "sol.xml", SLODynamic
    myGrammar.CmdSetRuleIdState 0, SGDSActive
End Sub
```

```
Private Sub RC_FalseRecognition(ByVal StreamNumber As Long, By
    Label1.Caption = "(no recognition)"
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal S
    'Label1.Caption = Result.PhraseInfo.GetText
End Sub
```

```
Private Sub RC_StartStream(ByVal StreamNumber As Long, ByVal S
    'Label2.Caption = Val(StreamNumber)
End Sub
```

In addition to copying the code to the Visual Basic project, copy the following XML code into a new file named *sol.xml*.

```
<GRAMMAR LANGID="409">
  <DEFINE>
    <ID NAME="RID_NewGame" VAL="101"/>
  </DEFINE>

  <RULE NAME="newgame" ID="RID_NewGame" TOPLEVEL="A
    <P>new +game</P>
  </RULE>
</GRAMMAR>
```

In cases when a second grammar file is needed, copy the following XML code into a new file named *sol2.xml*.

```
<GRAMMAR LANGID="409">
  <DEFINE>
    <ID NAME="RID_FileGame" VAL="200"/>
  </DEFINE>

  <RULE NAME="filegame" ID="RID_FileGame" TOPLEVEL="AC
    <P>file +game</P>
  </RULE>
</GRAMMAR>
```



VB Application Sample: Dictation Recognition (Inproc)

The following code sample represents a simple, but functional, recognition application, using the in process (or InProc) recognizer. It uses a dictation grammar and allows free dictation. The commented lines refer to hypothetical labels in a form to possibly display information. To see the recognized phrase, add one label, named Label1. Of course you may modify this application as needed to fit your own requirements.

Before running the application, a speech reference must be included. Using the Project->References menu, find and select the Microsoft Speech Object Library.

An InProc recognizer requires additional lines that shared recognizers do not. For InProc recognizers, the audio object for either input or output must be explicitly assigned.

```
Dim WithEvents RC As SpInProcRecoContext
Dim Recognizer As SpInprocRecognizer
Dim myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpInProcRecoContext
    Set Recognizer = RC.Recognizer

    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive

    Dim Category As SpObjectTokenCategory
    Set Category = New SpObjectTokenCategory
    Category.SetId SpeechCategoryAudioIn

    Dim Token As SpObjectToken
```

```
Set Token = New SpObjectToken
Token.SetId Category.Default()
Set Recognizer.AudioInput = Token
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal S
Label1.Caption = Result.PhraseInfo.GetText
End Sub
```



Sample DLL Code Example

This example builds a Visual Basic ActiveX DLL in order to demonstrate general object-oriented programming techniques, and more specifically, to demonstrate the use of the `ISpeechRecoGrammar_CmdLoadFromResource` method.

The DLL contains one resource, called "101," which is a compiled version of the Solitaire recognition grammar. This resource is used in the `ISpeechRecoGrammar` [CmdLoadFromResource](#) sample.

The DLL contains one method, called "SpeakToFile," which is used in the `ISpeechPhraseAlternate` [code example](#).

To create the Solitaire grammar

- Open Notepad
- Copy the Solitaire Grammar text below, and paste it into Notepad
- Save it as "C:\sol.xml"

To compile the Solitaire Grammar

Click Start, and then click Run. Paste this text into the text box:

```
"C:\Program Files\Microsoft Speech SDK 5.1\Bin\GC" C:\SOL.XML
```

If your Speech SDK is installed at a different location, adjust the path accordingly. Click OK, and the grammar compiler will compile SOL.XML into SOL.CFG

Create the DLL project

In Visual Basic, create an ActiveX DLL project.

1. On the Project menu, select **Project1 Properties**.
2. In the Project1 Properties dialog box, click the **General** tab, and replace the Project Name (Project1) with SpeechDocs, click **OK**.
3. On the Project menu, click **References**.
4. Scroll down the References list, select **Microsoft Speech Object Library**, click **OK**.
5. Paste the DLL source code into the Declarations section of the module called Class1.

Load the Visual Basic Resource Editor

1. On the **Add-Ins** menu, click **Add-In Manager**.
2. From the **Available Add-Ins** list box, select the **VB 6 Resource Editor** .
3. From the **Load Behavior** box, select **Loaded/Unloaded**, click **OK**.

An icon for the Resource Editor will appear on Visual Basic's standard toolbar.

Add the Grammar to the DLL as a Resource

1. Click the Resource Editor toolbar icon.
2. In the Resource Editor, click the **Add Custom Resource** toolbar button.
3. In the **Open Custom Resource** dialog box, select the grammar file SOL.CFG, click **OK**. The Resource Editor

will now display an icon captioned 101.

4. Right-click the icon and select **Properties**.
5. In the Type text box, replace the word CUSTOM with CFGGRAMMAR, click **OK**.
6. Click the **Save** toolbar button and save file SpeechDocs.RES.

Compile the DLL

On the File menu, click **Make SpeechDocs.dll**.

Text of Solitaire Grammar

This is the text of the Solitaire grammar SOL.XML.

```
<GRAMMAR LANGID="409">
  <DEFINE>
    <ID NAME="FROM" VAL="1"/>
    <ID NAME="TO" VAL="2"/>
    <ID NAME="SUIT" VAL="3"/>
    <ID NAME="COLOR" VAL="4"/>
    <ID NAME="RANK" VAL="5"/>
    <ID NAME="ColorRed" VAL="11101"/>
    <ID NAME="ColorBlack" VAL="10011"/>
    <ID NAME="RID_NewGame" VAL="101"/>
    <ID NAME="RID_MoveCard" VAL="102"/>
    <ID NAME="RID_Rank" VAL="103"/>
  </DEFINE>
  <RULE NAME="newgame" ID="RID_NewGame" TOPLEVEL="ACTIVE">
    <P>new +game</P><0>-please</0>
  </RULE>
  <RULE NAME="playcard" TOPLEVEL="ACTIVE" EXPORT="1">
    <0>please</0>
    <P>play the</P>
    <RULEREF NAME="card"/>
    <0>please</0>
  </RULE>
  <RULE NAME="movecard" ID="RID_MoveCard" TOPLEVEL="ACTIVE":
    <0>please</0>
```

```

<P>
  <L>
    <P>move</P>
    <P>put</P>
  </L>
  <P>the</P>
</P>
<RULEREF PROPNAME="from" PROPID="FROM" NAME="card"/>
<0>
  <L>
    <P>on</P>
    <P>to</P>
  </L>
  <P>the</P>
  <RULEREF PROPNAME="to" PROPID="TO" NAME="card"/>
</0>
<0>please</0>
</RULE>
<RULE NAME="card">
  <L>
    <P>
      <L PROPNAME="color" PROPID="COLOR">
        <P VAL="ColorRed">red</P>
        <P VAL="ColorBlack">black</P>
      </L>
      <RULEREF NAME="rank"/>
    </P>
    <P>
      <RULEREF NAME="rank"/>
      <0>
        <P>of</P>
        <L PROPNAME="suit" PROPID="SUIT">
          <P VAL="0">clubs</P>
          <P VAL="1">hearts</P>
          <P VAL="2">diamonds</P>
          <P VAL="3">spades</P>
        </L>
      </0>
    </P>
    <L PROPNAME="suit" PROPID="SUIT">
      <P VAL="0">club</P>
      <P VAL="1">heart</P>
    </L>
  </L>

```

```

        <P VAL="2">diamond</P>
        <P VAL="3">spade</P>
    </L>
</L>
</RULE>
<RULE NAME="rank" ID="RID_Rank">
    <L PROPNAME="rank" PROPID="RANK">
        <P VAL="1">ace</P>
        <P VAL="2">two</P>
        <P VAL="3">three</P>
        <P VAL="4">four</P>
        <P VAL="5">five</P>
        <P VAL="6">six</P>
        <P VAL="7">seven</P>
        <P VAL="8">eight</P>
        <P VAL="9">nine</P>
        <P VAL="10">ten</P>
        <P VAL="11">jack</P>
        <P VAL="12">queen</P>
        <P VAL="13">king</P>
        <P VAL="12">lady</P>
        <P VAL="13">emperor</P>
    </L>
</RULE>
</GRAMMAR>

```

The DLL Source Code

This is the source code for SpeechDocs DLL.

Option Explicit

```

Private mV As SpeechLib.SpVoice
Private mF As SpeechLib.SpFileStream

```

```

Public Function SpeakToFile( _
    ByVal strText, _
    ByVal strFName, _
    Optional NameOrToken _
) As SpFileStream

```

```

Set mV = New SpVoice           'Create voice object with
If (Not IsMissing(NameOrToken)) Then
    On Error GoTo BadNameOrToken
    Select Case TypeName(NameOrToken) 'VBA.Information
        Case "String"           'Use string parameter as
            Set mV.Voice = mV.GetVoices("name=" & NameOrToken)
        Case "ISpeechObjectToken" 'Use object token parameter
            Set mV.Voice = NameOrToken
    End Select
End If

```

```

BadNameOrToken:           'Use default voice if Set from NameOrToken

```

```

On Error GoTo OtherErrors
Set mF = New SpFileStream           'Create stream object
mF.Open strFName, SSFMCreatForWrite, True 'Open as the
Set mV.AudioOutputStream = mF      'Set voice object
mV.Speak strText, SVSFIsXML        'Speak synchronously
mF.Close                            'Close file

```

```

OtherErrors:           'Exit on illegal file path or other error
Set mV = Nothing
Set SpeakToFile = mF      'Return FileStream object
End Function

```



Automation Enumerations

The **Automation Enumerations** specify the possible values for many of the types of data used in speech automation.

Automation Enumerations

The automation enumerations include the following:

Enumerations	Description
SpeechAudioFormatType	Lists the supported stream formats.
SpeechAudioState	Sets the audio input or output state to one of four possible states.
SpeechBookmarkOptions	Specifies whether or not the bookmark will pause a speech recognition(SR) engine.
SpeechDataKeyLocation	Specifies the top-level speech configuration database keys.
SpeechDiscardType	Indicates portions of a recognition result to be removed or eliminated once they are no longer needed.
SpeechDisplayAttributes	Specifies information about the display of a word.
SpeechEngineConfidence	Specifies levels of confidence.

SpeechFormatType	Used to request either the input format for the original audio source, or the format actually arriving at the speech engine.
SpeechGrammarRuleStateTransitionType	Lists the types of transitions for the speech recognition engine.
SpeechGrammarState	Defines the possible states of a speech grammar.
SpeechGrammarWordType	Specifies the type of the word(s) to be added to a grammar.
SpeechInterference	Lists possible causes of interference or poor recognition with the input stream.
SpeechLexiconType	Specifies the allowed lexicon types.
SpeechLoadOption	indicates how a speech grammar is loaded.
SpeechPartOfSpeech	Defines the parts-of-speech categories used in SAPI.
SpeechRecoContextState	Provides lists levels of control for setting and restoring recognition states for each recognition context.
SpeechRecoEvents	Lists the event interests for the recognition context.

SpeechRecognitionType	Specifies the state of the SR engine.
SpeechRecognizerState	Enumerates the states of a Recognizer object.
SpeechRetainedAudioOptions	Indicates the options for an audio stream.
SpeechRuleAttributes	Lists attribute's information about grammar rules.
SpeechRuleState	Defines the states of a speech grammar rule.
SpeechRunState	Lists the voice running state.
SpeechSpecialTransitionType	Lists special transitions for the speech recognition engine.
SpeechStreamFileMode	Specifies the file opening states.
SpeechStreamSeekPositionType	Lists the methods to seek a location within a stream.
SpeechTokenContext	Lists the context in which the code managing the newly created object runs.
SpeechTokenShellFolder	Lists possible locations storing token information.
SpeechVisemeFeature	Contains constants specifying features of phonemes and visemes.

SpeechVisemeType	Contains constants for each of the visemes supported by the SpVoice object.
SpeechVoiceEvents	Enumerates the types of events which the SpVoice object can raise.
SpeechVoicePriority	Enumerates the possible Priority settings of an SpVoice object.
SpeechVoiceSpeakFlags	Contains flags that control the SpVoice.Speak method.
SpeechWordPronounceable	Defines the set of return values from the IsPronounceable method of the SpRecoGrammar object.
SpeechWordType	Specifies the change state of a word/pronunciation combination in a lexicon.

Microsoft Speech SDK

Speech Automation 5.1



SpeechAudioFormatType Enum

The **SpeechAudioFormatType** enumeration lists the supported stream formats.

These enumeration elements are all common audio formats ranging from the uncompressed PCM formats to highly compressed formats. They are available as standard formats on the Windows operating systems and are supported by SAPI 5.

Definition

```
Enum SpeechAudioFormatType
    SAFTDefault = -1
    SAFTNoAssignedFormat = 0
    SAFTText = 1
    SAFTNonStandardFormat = 2
    SAFTExtendedAudioFormat = 3

    // Standard PCM wave formats
    SAFT8kHz8BitMono = 4
    SAFT8kHz8BitStereo = 5
    SAFT8kHz16BitMono = 6
    SAFT8kHz16BitStereo = 7
    SAFT11kHz8BitMono = 8
    SAFT11kHz8BitStereo = 9
    SAFT11kHz16BitMono = 10
    SAFT11kHz16BitStereo = 11
    SAFT12kHz8BitMono = 12
    SAFT12kHz8BitStereo = 13
    SAFT12kHz16BitMono = 14
    SAFT12kHz16BitStereo = 15
    SAFT16kHz8BitMono = 16
    SAFT16kHz8BitStereo = 17
    SAFT16kHz16BitMono = 18
    SAFT16kHz16BitStereo = 19
    SAFT22kHz8BitMono = 20
    SAFT22kHz8BitStereo = 21
    SAFT22kHz16BitMono = 22
```

```
SAFT22kHz16BitStereo = 23
SAFT24kHz8BitMono = 24
SAFT24kHz8BitStereo = 25
SAFT24kHz16BitMono = 26
SAFT24kHz16BitStereo = 27
SAFT32kHz8BitMono = 28
SAFT32kHz8BitStereo = 29
SAFT32kHz16BitMono = 30
SAFT32kHz16BitStereo = 31
SAFT44kHz8BitMono = 32
SAFT44kHz8BitStereo = 33
SAFT44kHz16BitMono = 34
SAFT44kHz16BitStereo = 35
SAFT48kHz8BitMono = 36
SAFT48kHz8BitStereo = 37
SAFT48kHz16BitMono = 38
SAFT48kHz16BitStereo = 39

// TrueSpeech format
SAFTTrueSpeech_8kHz1BitMono = 40

// A-Law formats
SAFTCCITT_ALaw_8kHzMono = 41
SAFTCCITT_ALaw_8kHzStereo = 42
SAFTCCITT_ALaw_11kHzMono = 43
SAFTCCITT_ALaw_11kHzStereo = 44
SAFTCCITT_ALaw_22kHzMono = 44
SAFTCCITT_ALaw_22kHzStereo = 45
SAFTCCITT_ALaw_44kHzMono = 46
SAFTCCITT_ALaw_44kHzStereo = 47

// u-Law formats
SAFTCCITT_uLaw_8kHzMono = 48
SAFTCCITT_uLaw_8kHzStereo = 49
SAFTCCITT_uLaw_11kHzMono = 50
SAFTCCITT_uLaw_11kHzStereo = 51
SAFTCCITT_uLaw_22kHzMono = 52
SAFTCCITT_uLaw_22kHzStereo = 53
SAFTCCITT_uLaw_44kHzMono = 54
SAFTCCITT_uLaw_44kHzStereo = 55
SAFTADPCM_8kHzMono = 56
SAFTADPCM_8kHzStereo = 57
```

```
SAFTADPCM_11kHzMono = 58
SAFTADPCM_11kHzStereo = 59
SAFTADPCM_22kHzMono = 60
SAFTADPCM_22kHzStereo = 61
SAFTADPCM_44kHzMono = 62
SAFTADPCM_44kHzStereo = 63

// GSM 6.10 formats
SAFTGSM610_8kHzMono = 64
SAFTGSM610_11kHzMono = 65
SAFTGSM610_22kHzMono = 66
SAFTGSM610_44kHzMono = 67

// Other formats
SAFTNUM_FORMATS = 68
End Enum
```

Remarks

SAFTNonStandardFormat

SAFTNonStandardFormat is a non-SAPI 5 standard format without a WAVEFORMATEX description.

SAFTExtendedAudioFormat

SAFTExtendedAudioFormat is a non-SAPI 5 standard format but has WAVEFORMATEX description.

Microsoft Speech SDK

Speech Automation 5.1



SpeechAudioState Enum

The **SpeechAudioState** enumeration lists the four possible audio input and output states.

This is used by [ISpeechAudioStatus.State](#) property and [ISpeechAudio.SetState](#) method.

Definition

```
Enum SpeechAudioState
    SASClosed = 0
    SASStop = 1
    SASPause = 2
    SASRun = 3
End Enum
```

Elements

SASClosed

Audio is stopped and closed. For multimedia audio input devices (sound cards etc.), the device will be released. It can be opened by other processes and potentially made unavailable to SAPI.

SASStop

Audio is stopped. For multimedia audio input devices (sound cards etc.), the audio device will not be closed. This guarantees that SAPI can restart it without an intervening process opening it.

SASPause

Audio is paused. Staying in this state for too long a period will cause audio loss.

SASRun

Audio is enabled.

Microsoft Speech SDK

Speech Automation 5.1



SpeechBookmarkOptions Enum

The **SpeechBookmarkOptions** enumeration lists bookmark options.

Definition

```
Enum SpeechBookmarkOptions
    SBONone = 0
    SBOPause = 1
End Enum
```

Elements

SBONone

The recognition context will not pause when it encounters the bookmark.

SBOPause

The recognition context will pause when it encounters the bookmark. This is the same as calling [*ISpeechRecoContext.Pause*](#). The pause stops the speech recognition engine from processing any more data until *Resume* is called. In a paused state, the application can perform tasks such as changing grammars while the engine is at a known position in the stream. The application must explicitly begin processing afterward with [*ISpeechRecoContext.Resume*](#).

Microsoft Speech SDK

Speech Automation 5.1



SpeechDataKeyLocation Enum

The **SpeechDataKeyLocation** enumeration lists the top-level speech configuration database keys.

Used with [ISpeechObjectTokenCategory.GetDataKey](#) to read and write token categories.

Definition

```
Enum SpeechDataKeyLocation
    SDKLDefaultLocation = 0
    SDKLCurrentUser = 1
    SDKLLocalMachine = 2
    SDKLCurrentConfig = 5
End Enum
```

Elements

SDKLDefaultLocation

The default location is set by ISpObjectTokenCategory.

SDKLCurrentUser

The speech configuration database key
HKEY_CURRENT_USER.

SDKLLocalMachine

The speech configuration database key
HKEY_LOCAL_MACHINE.

SDKLCurrentConfig

The speech configuration database key
HKEY_CURRENT_CONFIG.

Microsoft Speech SDK

Speech Automation 5.1



SpeechDiscardType Enum

The **SpeechDiscardType** enumeration lists flags indicating portions of a recognition result to be removed or eliminated once they are no longer needed.

Definition

```
Enum SpeechDiscardType
    SDTProperty = 1
    SDTReplacement = 2
    SDTRule = 4
    SDTDisplayText = 8
    SDTLexicalForm = 16
    SDTPronunciation = 32
    SDTAudio = 64
    SDTAlternates = 128
    SDTAll = 255
End Enum
```

Elements

SDTProperty

Removes the property tree.

SDTReplacement

Removes the phrase replacement text for inverse text normalization.

SDTRule

Removes the non-top level rule tree information for a phrase.

SDTDisplayText

Removes the display text.

SDTLexicalForm

Removes the lexicon from text.

SDTPronunciation

Removes the pronunciation text.

SDTAudio

Removes the audio data that is attached to a phrase.
However, the audio has to have been both set and retained.

SDTAlternates

Removes the alternate data that is attached to a phrase.
Discarding alternates loses the words permanently and they cannot be retrieved.

SDTAll

Remove all the features above.

Microsoft Speech SDK

Speech Automation 5.1



SpeechDisplayAttributes Enum

The **SpeechDisplayAttributes** enumeration lists the possible ways of displaying a word.

Used by [ISpeechPhraseReplacement.DisplayAttributes](#) and [ISpeechPhraseElement.DisplayAttributes](#) to retrieve SpeechDisplayAttributes for the recognition. This property cannot be set and it is determined by the particular engine and is specific for the language.

Definition

```
Enum SpeechDisplayAttributes
    SDA_No_Trailing_Space = 0
    SDA_One_Trailing_Space = 2
    SDA_Two_Trailing_Spaces = 4
    SDA_Consume_Leading_Spaces = 8
End Enum
```

Elements

SDA_No_Trailing_Space

Does not insert any trailing spaces after words.

SDA_One_Trailing_Space

Inserts one trailing space, used for most words.

SDA_Two_Trailing_Spaces

Inserts two trailing spaces, often used after a sentence's final period.

SDA_Consume_Leading_Spaces

Consume leading space, often used for periods. If this is absent, the word should have a leading space by default.

Microsoft Speech SDK

Speech Automation 5.1



SpeechEngineConfidence Enum

The **SpeechEngineConfidence** enumeration specifies levels of confidence.

Results returned by the speech recognition (SR) engine may be assigned one of these values to indicate the degree of confidence of the recognition. Alternatively, the engine may be assigned one of these values as a minimum requirement before passing back a recognition.

See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details.

Definition

```
Enum SpeechEngineConfidence
    SECLowConfidence = -1
    SECNormalConfidence = 0
    SECHighConfidence = 1
End Enum
```

Elements

SECLowConfidence

Indication of low confidence.

SECNormalConfidence

Indication of normal confidence.

SECHighConfidence

Indication of high confidence.

Microsoft Speech SDK

Speech Automation 5.1



SpeechFormatType Enum

The **SpeechFormatType** enumeration requests either the input format for the original audio source, or the format that actually arrives at the speech engine.

SAPI may be dynamically converting the stream to a different format and as a result, the original stream format may be different than the format received by the speech recognition (SR) engine.

Definition

```
Enum SpeechFormatType
    SFTInput = 0
    SFTSREngine = 1
End Enum
```

Elements

SFTInput

Request the format of the original audio input.

SFTSREngine

Request the format received by the SR engine.

Microsoft Speech SDK

Speech Automation 5.1



SpeechGrammarRuleStateTransitionType Enum

The **SpeechGrammarRuleStateTransitionType** enumeration lists the types of transitions for the speech recognition engine.

Definition

```
Enum SpeechGrammarRuleStateTransitionType
    SGRSTTEpsilon = 0
    SGRSTTWord = 1
    SGRSTTRule = 2
    SGRSTTDictation = 3
    SGRSTTWildcard = 4
    SGRSTTTextBuffer = 5
End Enum
```

Elements

SGRSTTEpsilon

Indicates there should be an epsilon transition. These are NULL transitions that can be traversed without recognizing anything.

SGRSTTWord

Indicates there should be a word transition. These represent single words that the recognizer will recognize before advancing to the next state.

SGRSTTRule

Indicates there should be a rule transition. These represent transitions into sub-rules. This transition is only passed when a path through the sub-rule has been recognized.

SGRSTTDictation

Indicates there should be a dictation transition.

SGRSTTDictation is a special transition and may not be supported by all engines. This is used to embed dictation within a context-free grammar (CFG). Each transition means one word should be recognized.

SGRSTTWildcard

Indicates there should be a wildcard transition.

SGRSTTWildcard is a special transition and may not be supported by all engines. This indicates a transition that matches any word or words. The engine does not try and recognize the spoken words. The engine includes the string value `WildcardInCFG` as an attribute in its object token to inform the application that it is capable of supporting this.

SGRSTTTextBuffer

Indicates there should be a text buffer transition.

SGRSTTTextBuffer is a special transition and may not be supported by all engines. This indicates that the engine is to recognize a sub-string of words from the text buffer, if it has been set.

Microsoft Speech SDK

Speech Automation 5.1



SpeechGrammarState Enum

The **SpeechGrammarState** enumeration lists the possible states of a speech grammar.

Definition

```
Enum SpeechGrammarState
    SGSDisabled = 0
    SGSEnabled = 1
    SGSExclusive = 3
End Enum
```

Elements

SGSEnabled

SGSEnabled indicates that the grammar can receive recognitions and that its rules are active. This is the default speech grammar state.

SGSDisabled

SGSDisabled indicates that the grammar cannot receive recognitions and that its rules are inactive. Rules can still be added to a grammar in this state.

SGSExclusive

SGSExclusive indicates that this grammar is the only active grammar and disables all rules that are not part of this grammar. Currently not implemented.

Microsoft Speech SDK

Speech Automation 5.1



SpeechGrammarWordType Enum

The **SpeechGrammarWordType** enumeration lists the types of words in a grammar.

Definition

```
Enum SpeechGrammarWordType
    SGDisplay = 0
    SGLexical = 1
    SGPronunciation = 2
End Enum
```

Elements

SGDisplay

Each word to be added is in display form. That is, it possibly will have to be converted into lexical form(s). For example, the word "23" (display form) would have to be converted into "twenty three" (lexical form). This is currently not implemented.

SGLexical

Each word to be added is in lexical form and can be used to access the lexicon. This type is specified in the speech text grammar format as <GRAMMAR WORDTYPE="LEXICAL">.

SGPronunciation

Each word is specified solely by its pronunciation. This is currently not implemented.

Microsoft Speech SDK

Speech Automation 5.1



SpeechInterference Enum

The **SpeechInterference** enumeration lists possible causes of interference or poor recognition with the input audio stream.

Definition

```
Enum SpeechInterference
    SINone = 0
    SInoise = 1
    SInoSignal = 2
    SITooLoud = 3
    SITooQuiet = 4
    SITooFast = 5
    SITooSlow = 6
End Enum
```

Elements

SINone

Private. Do not use.

SInoise

The sound received is interpreted by the speech recognition engine as noise. This event is generated when there is a SOUND_START followed by a SOUND_END without an intervening PHRASE_START. The event will be also generated during dictation if, after a series of hypotheses, it is determined that the signal is noise.

SInoSignal

A sound is received but it is of a constant intensity. This also includes the microphone being unplugged or muted.

SITooLoud

A sound is received but the stream intensity is too high for discrete recognition.

SITooQuiet

A sound is received but the stream intensity is too low for discrete recognition.

SITooFast

The words are spoken too quickly for discrete recognition.

SITooSlow

The words are spoken too slowly and indicates excessive time between words.

Microsoft Speech SDK

Speech Automation 5.1



SpeechLexiconType Enum

The **SpeechLexiconType** enumeration lists the allowed lexicon types.

Currently there are only two types in use: user and application lexicons. Additional types may be added in the future or created uniquely by the application.

Definition

```
Enum SpeechLexiconType {  
    SLTUser    = 1  
    SLTApp     = 2  
} End Enum
```

Elements

SLTUser

Indicates the user lexicon. Each Windows user has a unique user lexicon.

SLTApp

Indicates the application lexicon. An application lexicon is shared by all users.

Microsoft Speech SDK

Speech Automation 5.1



SpeechLoadOption Enum

The **SpeechLoadOption** enumeration lists the options available when loading a speech grammar.

Definition

```
Enum SpeechLoadOption
    SLOStatic = 0
    SLODynamic = 1
End Enum
```

Elements

SLOStatic

Specifies that the grammar is loaded statically.

SLODynamic

Specifies that the grammar is loaded dynamically, meaning that rules can be modified and committed at run time.

Microsoft Speech SDK

Speech Automation 5.1



SpeechPartOfSpeech Enum

The **SpeechPartOfSpeech** enumeration lists the parts-of-speech categories used in SAPI.

This list of known parts-of-speech types is intentionally small and broad and will be expanded and refined in future releases. **SpeechPartOfSpeech** in its minimal form is required to support look ups from the standard SAPI lexicon. This information is useful to TTS engines in order to determine the correct pronunciation of ambiguous words based on their context.

Definition

```
Enum SpeechPartOfSpeech
    SPSNotOverriden = -1
    SPSUnknown = 0
    SPSNoun = 4096
    SPSVerb = 8192
    SPSModifier = 12288
    SPSFunction = 16384
    SPSInterjection = 20480
End Enum
```

Elements

SPSNotOverriden

Indicates that the part of speech already present in the lexicon should not be overridden.

SPSUnknown

Indicates that the part of speech is unknown and is probably from the user lexicon.

SPSNoun

Indicates that the part of speech is a noun.

SPSVerb

Indicates that the part of speech is a verb.

SPSModifier

Indicates that the part of speech is a modifier.

SPSFunction

Indicates that the part of speech is a function.

SPSInterjection

Indicates that the part of speech is an interjection.

Microsoft Speech SDK

Speech Automation 5.1



SpeechRecoContextState Enum

The **SpeechRecoContextState** enumeration lists the states of a recognition context.

Used by [ISpeechRecoContext.State](#)

Definition

```
Enum SpeechRecoContextState
    SRCS_Disabled = 0
    SRCS_Enabled = 1
End Enum
```

Elements

SRCS_Disabled

Specifies that grammars associated with this recognition context are disabled.

SRCS_Enabled

Specifies that grammars associated with this recognition context are enabled.

Microsoft Speech SDK

Speech Automation 5.1



SpeechRecoEvents Enum

The **SpeechRecoEvents** enumeration lists speech recognition (SR) events.

Used in [ISpeechRecoContext.VoicePurgeEvent](#) and [ISpeechRecoContext.EventInterests](#).

Definition

```
Enum SpeechRecoEvents
    SREStreamEnd = 1
    SRESoundStart = 2
    SRESoundEnd = 4
    SREPhraseStart = 8
    SRERecognition = 16
    SREHypothesis = 32
    SREBookmark = 64
    SREPropertyNumChange = 128
    SREPropertyStringChange = 256
    SREFalseRecognition = 512
    SREInterference = 1024
    SRERequestUI = 2048
    SREStateChange = 4096
    SREAdaptation = 8192
    SREStreamStart = 16384
    SRERecoOtherContext = 32768
    SREAudioLevel = 65536
    SREPrivate = 262144
    SREAllEvents = 393215
End Enum
```

Elements

SREStreamEnd

SR engine has reached the end of an input stream.

SRESoundStart

SR engine has detected the start of non-trivial audio data.

SRESoundEnd

SR engine has detected the end of non-trivial audio data.

SREPhraseStart

SR engine has detected the start of a recognizable phrase.

SRERecognition

SR engine's best hypothesis for the audio data.

SREHypothesis

SR engine's interim hypothesis for the result of the audio data.

SREBookmark

SR engine has reached the specified point in the audio stream.

SREPropertyNumChange

LPARAM points to a string WPARAM that is the attribute value.

SREPropertyStringChange

LPARAM pointer to a buffer. Two concatenated null-terminated strings.

SREFalseRecognition

Apparent speech with no valid recognition.

SREInterference

LPARAM is any combination of [SpeechInterference](#) flags.

SRERequestUI

LPARAM is string.

SREStateChange

WPARAM contains new recognition state.

SREAdaptation

The adaptation buffer is now ready to be accepted.

SREStreamStart

SR engine has reached the start of an input stream.

SRERecoOtherContext

Phrase finished and recognized but for other context.

SREAudioLevel

Input audio volume level

SREPrivate

Private engine-specific event.

SREAllEvents

All events listed above.

Microsoft Speech SDK

Speech Automation 5.1



SpeechRecognitionType Enum

The **SpeechRecognitionType** enumeration lists the types of speech recognition.

The [Recognition](#) method of the `SpInProcRecognizer` and `SpSharedRecognizer` objects returns a `SpeechRecognitionType` member indicating the type of recognition which produced the recognition result.

Definition

```
Enum SpeechRecognitionType
    SRTStandard = 0
    SRTAutopause = 1
    SRTEmulated = 2
End Enum
```

Elements

SRTStandard

Indicates that the recognition result was produced by standard recognition.

SRTAutopause

Indicates that the recognition result was produced by standard recognition and that the engine is paused.

SRTEmulated

Indicates that the recognition result was produced by recognition emulation.

Microsoft Speech SDK

Speech Automation 5.1



SpeechRecognizerState Enum

The **SpeechRecognizerState** enumeration lists the states of a Recognizer object.

Definition

```
Enum SpeechRecognizerState
    SRInactive = 0
    SRActive = 1
    SRActiveAlways = 2
    SRInactiveWithPurge = 3
End Enum
```

Elements

SRInactive

The engine and audio input are inactive and no audio is being read, even if there are rules active. The audio device will be closed in this state.

SRActive

Recognition takes place if there are any active rules. If a rule is active, audio will be read and passed to the SR engine and recognition will take place.

SRActiveAlways

Indicates the audio is running regardless of the rule state. Even if there are no active rules, audio will still be read and passed to the engine.

SRInactiveWithPurge

Indicates the engine state is set to inactive and all active

audio data is purged. This state is used when an application wishes to shut an engine down as quickly as possible, without waiting for it to finish processing any audio data that is currently buffered.

Microsoft Speech SDK

Speech Automation 5.1



SpeechRetainedAudioOptions Enum

The **SpeechRetainedAudioOptions** enumeration lists the options for retaining data from an audio stream.

Used in [ISpeechRecoContext.RetainedAudio](#).

Definition

```
Enum SpeechRetainedAudioOptions
    SRAONone = 0
    SRAORetainAudio = 1
End Enum
```

Elements

SRAONone

Indicates that the audio should not be retained, but discarded after use.

SRAORetainAudio

Flag indicates that the audio stream should be retained (e.g., serialization of recognition object, playback of recognized audio, etc.).

Microsoft Speech SDK

Speech Automation 5.1



SpeechRuleAttributes Enum

The **SpeechRuleAttributes** enumeration lists the possible attributes of a grammar rule.

Used in [ISpeechGrammarRules.Add](#) and [ISpeechGrammarRule.Attributes](#).

Definition

```
Enum SpeechRuleAttributes
    SRATopLevel = 1
    SRADefaultToActive = 2
    SRAExport = 4
    SRAImport = 8
    SRAInterpreter = 16
    SRADynamic = 32
End Enum
```

Elements

SRATopLevel

Specifies that the rule is defined as a top-level rule. Top-level rules are the entry points into the grammar and can be activated or deactivated programmatically. Set a rule as top-level by using the TOPLEVEL attribute in the Speech Text Grammar Format.

SRADefaultToActive

Specifies that the rule is defined as a top-level rule that is activated by default. This can be set using the TOPLEVEL="ACTIVE" attribute-value pair in the Speech Text Grammar Format.

SRAExport

Specifies the rule is exported and hence can be referred to by a rule in another grammar. This can be set using the EXPORT="YES" attribute-value pair in the Speech Text Grammar Format.

SRAImport

Specifies the rule is imported from another grammar and is therefore not defined in this grammar.

SRAInterpreter

Specifies the rule has an interpreter (custom C/C++ code implementing the ISpCFGInterpreter interface) associated with it.

SRADynamic

Specifies the rule is dynamic (can be changed programmatically through the ISpGrammarBuilder interface). Note that the CFG must be loaded with the SPLO_DYNAMIC flag to enable changes at run time.

Microsoft Speech SDK

Speech Automation 5.1



SpeechRuleState Enum

The **SpeechRuleState** enumeration lists the states of a speech grammar rule.

Definition

```
Enum SpeechRuleState
    SGDSInactive = 0
    SGDSActive = 1
    SGDSActiveWithAutoPause = 3
End Enum
```

Elements

SGDSInactive

Grammar rule is inactive.

SGDSActive

Grammar rule is active.

SGDSActiveWithAutoPause

SR engine will be placed in a paused state when the grammar rule is recognized.

Microsoft Speech SDK

Speech Automation 5.1



SpeechRunState Enum

The **SpeechRunState** enumeration lists the running states of a TTS voice.

Used with [ISpeechVoiceStatus.RunningState](#).

Definition

```
Enum SpeechRunState
    SRSEDone = 1
    SRSEIsSpeaking = 2
End Enum
```

Elements

SRSEDone

The voice has finished rendering all queued phrases.

SRSEIsSpeaking

The SpVoice currently claims the audio queue.

Remarks

A SpeechRunState value of zero represents a state in which the voice is waiting to speak. This condition is returned by *ISpeechVoiceStatus.RunningState* before the voice has begun speaking, and when the voice is interrupted by an alert voice.

Microsoft Speech SDK

Speech Automation 5.1



SpeechSpecialTransitionType Enum

The **SpeechSpecialTransitionType** enumeration lists special transitions for the speech recognition engine.

Special transitions may not be supported by all speech engines. Used in [ISpeechGrammarRuleState.AddSpecialTransition](#).

Definition

```
Enum SpeechSpecialTransitionType
    SSTTWildcard = 1
    SSTTDictation = 2
    SSTTTextBuffer = 3
End Enum
```

Elements

SSTTWildcard

Indicates there should be a wildcard transition. SGRSTTWildcard is a special transition and may not be supported by all engines. This indicates a transition that matches any word or words. The engine does not try to recognize the spoken words. The engine includes the string value WildcardInCFG as an attribute in its object token to inform the application that it is capable of supporting this.

SSTTDictation

Indicates there should be a dictation transition. SGRSTTDictation is a special transition and may not be supported by all engines. This is used to embed dictation within a context-free grammar (CFG). Each transition means one word should be recognized.

SSTTTextBuffer

Indicates there should be a text buffer transition.

SGRSTTTextBuffer is a special transition and may not be supported by all engines. This indicates that the engine is to recognize a sub-string of words from the text-buffer, if it has been set.

Microsoft Speech SDK

Speech Automation 5.1



SpeechStreamFileMode Enum

The **SpeechStreamFileMode** enumeration lists the access modes of a file stream.

Used by [SpFileStream.Open](#).

Definition

```
Enum SpeechStreamFileMode
    SSFMOpenForRead = 0
    [hidden] SSFMOpenReadWrite = 1
    [hidden] SSFMCreate = 2
    SSFMCreateForWrite = 3
End Enum
```

Elements

SSFMOpenForRead

Opens an existing file as read-only.

SSFMOpenReadWrite

[hidden] Opens an existing file as read-write. Not supported for wav files.

SSFMCreate

[hidden] Opens an existing file as read-write. Else, it creates the file then opens it as read-write. Not supported for wav files.

SSFMCreateForWrite

Creates file even if file exists and so destroys or overwrites the existing file.



SpeechStreamSeekPositionType Enum

The **SpeechStreamSeekPositionType** enumeration lists the types of positioning from which a Seek method can be performed.

Used by [ISpeechBaseStream.Seek](#).

Definition

```
Enum SpeechStreamSeekPositionType
    SSSPTRelativeToStart = 0
    SSSPTRelativeToCurrentPosition = 1
    SSSPTRelativeToEnd = 2
End Enum
```

Elements

SSSPTRelativeToStart

Calculates the stream offset relative from the start of the stream.

SSSPTRelativeToCurrentPosition

Calculates the stream offset relative from the current position.

SSSPTRelativeToEnd

Calculates the stream offset relative from the end of the stream.

Microsoft Speech SDK

Speech Automation 5.1



SpeechTokenContext Enum

The **SpeechTokenContext** enumeration lists the context in which the code managing the newly created object runs.

This is associated with COM and the use of `CoRegisterClassObject` to make new objects. Used in [SpObjectToken.CreateInstance](#)

Definition

```
Enum SpeechTokenContext
    STCInprocServer = 1
    STCInprocHandler = 2
    STCLocalServer = 4
    STCRemoteServer = 16
    STCALL = 23
End Enum
```

Elements

STCInprocServer

Creates and manages objects in the same process as the caller of the function.

STCInprocHandler

Creates and manages objects as an in process (InProc) handler. This is a DLL running in the client process and implements client-side structures of this class when instances are accessed remotely.

STCLocalServer

Creates and manages objects that are loaded in a separate process space; that is, it runs on same computer but in a

different process.

STCRemoteServer

Creates and manages objects on a remote machine context.

STCAII

Creates and manages objects for all class contexts.

Microsoft Speech SDK

Speech Automation 5.1



SpeechTokenShellFolder Enum

The **SpeechTokenShellFolder** enumeration lists possible locations storing token information.

This is a standard Win32 CSIDL value identifying the folder whose path is returned as a String. Used in [SpObjectToken.GetStorageFileName](#).

Definition

```
Enum SpeechTokenShellFolder
    STSF_AppData = 26
    STSF_LocalAppData = 28
    STSF_CommonAppData = 35
    STSF_FlagCreate = 32768
End Enum
```

Elements

STSF_AppData

Stores information in the application data of the user's profile.

STSF_LocalAppData

Stores information in the My Computer folder. This is a virtual folder containing everything on the local computer

STSF_CommonAppData

Stores information in the file system directory that serves as a common repository for application-specific data.

STSF_FlagCreate

Forces the creation of a folder. This flag is used in combination with another CSIDL value to create the item.

Microsoft Speech SDK

Speech Automation 5.1



SpeechVisemeFeature Enum

The **SpeechVisemeFeature** enumeration lists the features of phonemes and visemes.

Definition

```
Enum SpeechVisemeFeature
    SVF_None = 0
    SVF_Stressed = 1
    SVF_Emphasis = 2
End Enum
```

Elements

SVF_None

Indicates that a viseme or phoneme has no stress or emphasis.

SVF_Stressed

Indicates that a viseme or phoneme is stressed relative to the other phonemes within a word.

SVF_Emphasis

Indicates that the word containing the viseme or phoneme is emphasized relative to the other words within a sentence.

Microsoft Speech SDK

Speech Automation 5.1



SpeechVisemeType Enum

The **SpeechVisemeType** enumeration lists the visemes supported by the SpVoice object. This list is based on the original Disney visemes.

Definition

```
Enum SpeechVisemeType
    SVP_0 = 0          'silence
    SVP_1 = 1          'ae ax ah
    SVP_2 = 2          'aa
    SVP_3 = 3          'ao
    SVP_4 = 4          'ey eh uh
    SVP_5 = 5          'er
    SVP_6 = 6          'y iy ih ix
    SVP_7 = 7          'w uw
    SVP_8 = 8          'ow
    SVP_9 = 9          'aw
    SVP_10 = 10        'oy
    SVP_11 = 11        'ay
    SVP_12 = 12        'h
    SVP_13 = 13        'r
    SVP_14 = 14        'l
    SVP_15 = 15        's z
    SVP_16 = 16        'sh ch jh zh
    SVP_17 = 17        'th dh
    SVP_18 = 18        'f v
    SVP_19 = 19        'd t n
    SVP_20 = 20        'k g ng
    SVP_21 = 21        'p b m
End Enum
```

Elements

SVP_0

The viseme representing silence.

SVP_1

The viseme representing ae, ax, and ah.

SVP_2

The viseme representing aa.

SVP_3

The viseme representing ao.

SVP_4

The viseme representing ey, eh, and uh.

SVP_5

The viseme representing er.

SVP_6

The viseme representing y, iy, ih, and ix.

SVP_7

The viseme representing w and uw.

SVP_8

The viseme representing ow.

SVP_9

The viseme representing aw.

SVP_10

The viseme representing oy.

SVP_11

The viseme representing ay.

SVP_12

The viseme representing h.

SVP_13

The viseme representing r.

SVP_14

The viseme representing l.

SVP_15

The viseme representing s and z.

SVP_16

The viseme representing sh, ch, jh, and zh.

SVP_17

The viseme representing th and dh.

SVP_18

The viseme representing f and v.

SVP_19

The viseme representing d, t and n.

SVP_20

The viseme representing k, g and ng.

SVP_21

The viseme representing p, b and m.

Microsoft Speech SDK

Speech Automation 5.1



SpeechVoiceEvents Enum

The **SpeechVoiceEvents** enumeration lists the types of events which a text-to-speech (TTS) engine can send to an SpVoice object.

Definition

```
Enum SpeechVoiceEvents
    SVEStartInputStream = 2
    SVEEndInputStream = 4
    SVEVoiceChange = 8
    SVEBookmark = 16
    SVEWordBoundary = 32
    SVEPhoneme = 64
    SVESentenceBoundary = 128
    SVEViseme = 256
    SVEAudioLevel = 512
    SVEPrivate = 32768
    SVEAllEvents = 33790
End Enum
```

Elements

SVEStartInputStream

Represents the StartStream event, which occurs when the engine begins speaking a stream.

SVEEndInputStream

Represents the EndStream event, which occurs when the engine encounters the end of a stream while speaking.

SVEVoiceChange

Represents the VoiceChange event, which occurs when the

engine encounters a change of Voice while speaking.

SVEBookmark

Represents the Bookmark event, which occurs when the engine encounters a bookmark while speaking.

SVEWordBoundary

Represents the WordBoundary event, which occurs when the engine completes a word while speaking.

SVEPhoneme

Represents the Phoneme event, which occurs when the engine completes a phoneme while speaking.

SVESentenceBoundary

Represents the SentenceBoundary event, which occurs when the engine completes a sentence while speaking.

SVEViseme

Represents the Viseme event, which occurs when the engine completes a viseme while speaking.

SVEAudioLevel

Represents the AudioLevel event, which occurs when the engine has completed an audio level change while speaking.

SVEPrivate

Represents a private engine event.

SVEAllEvents

Represents all speech voice events.

Microsoft Speech SDK

Speech Automation 5.1



SpeechVoicePriority Enum

The **SpeechVoicePriority** enumeration lists the possible Priority settings of an SpVoice object.

The priority level defines the order in which the text-to-speech (TTS) engine processes a voice object's speech requests relative to requests from other voice objects. Higher priority levels are assigned to error-handling voices and the lowest priority level is assigned to normal voices.

The default Priority setting of a voice is SVPNormal.

Definition

```
Enum SpeechVoicePriority
    SVPNormal = 0
    SVPAAlert = 1
    SVPOver = 2
End Enum
```

Elements

SVPNormal

The priority of a normal voice. Text streams spoken by a normal voice are added to the end of the voice queue. A voice with SVPNormal priority cannot interrupt another voice.

SVPAAlert

The priority of an alert voice. Text streams spoken by an alert voice are inserted into the voice queue ahead of normal voice streams. An alert voice will interrupt a normal voice, which will resume speaking when the alert voice has finished speaking.

SVPOver

The priority of an over voice. Text streams spoken by an over voice go into the voice queue ahead of normal and alert streams. An over voice will not interrupt, but speaks over (mixes with) the voices of lower priorities.

Microsoft Speech SDK

Speech Automation 5.1



SpeechVoiceSpeakFlags Enum

The **SpeechVoiceSpeakFlags** enumeration lists flags that control the [SpVoice.Speak](#) method.

Definition

```
Enum SpeechVoiceSpeakFlags
    'SpVoice flags
    SVSFDefault = 0
    SVSFlagsAsync = 1
    SVSFPurgeBeforeSpeak = 2
    SVSFIsFilename = 4
    SVSFIsXML = 8
    SVSFIsNotXML = 16
    SVSFPersistXML = 32

    'Normalizer flags
    SVSFNLPSpeakPunc = 64

    'Masks
    SVSFNLPMask = 64
    SVSFVoiceMask = 127
    SVSFUnusedFlags = -128
End Enum
```

Elements

SVSFDefault

Specifies that the default settings should be used. The defaults are:

- To speak the given text string synchronously (override with SVSFlagsAsync),
- Not to purge pending speak requests (override with SVSFPurgeBeforeSpeak),

- To parse the text as XML only if the first character is a left-angle-bracket (override with *SVSFIsXML* or *SVSFIsNotXML*),
- Not to persist global XML state changes across speak calls (override with *SVSFPersistXML*), and
- Not to expand punctuation characters into words (override with *SVSFNLPSpeakPunc*).

SVSFlagsAsync

Specifies that the Speak call should be asynchronous. That is, it will return immediately after the speak request is queued.

SVSFPurgeBeforeSpeak

Purges all pending speak requests prior to this speak call.

SVSFIsFilename

The string passed to the Speak method is a file name rather than text. As a result, the string itself is not spoken but rather the file the path that points to is spoken.

SVSFIsXML

The input text will be parsed for XML markup.

SVSFIsNotXML

The input text will not be parsed for XML markup.

SVSFPersistXML

Global state changes in the XML markup will persist across speak calls.

SVSFNLPSpeakPunc

Punctuation characters should be expanded into words (e.g.

"This is it." would become "This is it period").

SVSFNLPMask

Flags handled by SAPI (as opposed to the text-to-speech engine) are set in this mask.

SVSFVoiceMask

This mask has every flag bit set.

SVSFUnusedFlags

This mask has every unused bit set.

Microsoft Speech SDK

Speech Automation 5.1



SpeechWordPronounceable Enum

The **SpeechWordPronounceable** enumeration lists the possible return values from the `IsPronounceable` method of the [ISpeechRecoGrammar](#) interface.

Definition

```
Enum SpeechWordPronounceable
    SWPUnknownWordUnpronounceable = 0
    SWPUnknownWordPronounceable = 1
    SWPKnownWordPronounceable = 2
End Enum
```

Elements

SWPUnknownWordUnpronounceable

The word is not pronounceable by the SR engine, and is not located in the lexicon and/or the engine's dictionary.

SWPUnknownWordPronounceable

The word is pronounceable by the SR engine, but is not located in the lexicon and/or the engine's dictionary.

SWPKnownWordPronounceable

The word is pronounceable by the SR engine, and is located in the lexicon and/or the engine's dictionary.

Microsoft Speech SDK

Speech Automation 5.1



SpeechWordType Enum

The **SpeechWordType** enumeration lists the change state of a word/pronunciation combination in a lexicon.

The [ISpeechLexiconWord.Type](#) property returns a SpeechWordType value.

Definition

```
Enum SpeechWordType  
    SWTAdded = 1  
    SWTDeleted = 2  
End Enum
```

Elements

SWTAdded

Indicates that the word has been added to the lexicon.

SWTDeleted

Indicates that the word has been deleted from the lexicon.



ISpeechAudio

The **ISpeechAudio** automation interface supports the control of real-time audio streams, such as those connected to a live microphone or telephone line.

The Format property and the Read, Write and Seek methods are inherited from the [ISpeechBaseStream](#) interface.

Automation Interface Elements

The ISpeechAudio automation interface contains the following elements:

Properties	Description
BufferInfo Property	Returns the audio buffer information as an ISpeechAudioBufferInfo object.
BufferNotifySize Property	Returns the audio stream buffer size information.
DefaultFormat Property	Returns the default audio format as an SpAudioFormat object.
EventHandle Property	Returns a Win32 event handle that applications can use to wait for status changes in the I/O stream.
Format Property	Gets and sets the cached wave format of the audio stream or device.
Status Property	Returns the audio status as an ISpeechAudioStatus object.
Volume Property	Gets and sets the volume level.
Methods	Description
Read Method	Reads data from an audio stream.
Seek Method	Returns the current read position of

	the audio stream in bytes.
<u>SetState</u> Method	Sets the audio state with a SpeechAudioState constant.
<u>Write</u> Method	Writes data to the audio stream.



Interface: [ISpeechAudio](#)

BufferInfo Property

The **BufferInfo** property returns the audio buffer information as an `ISpeechAudioBufferInfo` object.

Syntax

Set: (This property is read-only)

Get: [ISpeechAudioBufferInfo](#) = `ISpeechAudio`.**BufferInfo**

Parts

ISpeechAudio

The owning object.

ISpeechAudioBufferInfo

Set: (This property is read-only)

Get: An `ISpeechAudioBufferInfo` object which gets the buffer data.



Interface: [ISpeechAudio](#)

BufferNotifySize Property

The **BufferNotifySize** property gets and sets the audio stream buffer size information.

This information is used to determine when the event returned by the `EventHandle` method is set or reset.

Syntax

```
Set: ISpeechAudio.BufferNotifySize = Long
```

```
Get: Long = ISpeechAudio.BufferNotifySize
```

Parts

ISpeechAudio

The owning object.

Long

Set: A Long variable that sets the property.

Get: A Long variable that gets the property.

Remarks

For read streams, the event is set if the audio buffered is greater than or equal to the value set in *pcbSize*, otherwise the event information is reset.

For write streams, the event is set if the audio buffered is less than the value set in *pcbSize*, otherwise the event information is reset.



Interface: [ISpeechAudio](#)

DefaultFormat Property

The **DefaultFormat** property returns the default audio format as an SpAudioFormat object.

Syntax

Set: (This property is read-only)

Get: [SpAudioFormat](#) = *ISpeechAudio*.**DefaultFormat**

Parts

ISpeechAudio

The owning object.

SpAudioFormat

Set: (This property is read-only)

Get: A SpAudioFormat that gets the audio format.



Interface: [ISpeechAudio](#)

EventHandle Property

The **EventHandle** property returns a Win32 event handle that applications can use to wait for status changes in the I/O stream.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechAudio*.**EventHandle**

Parts

ISpeechAudio

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the handle.

Remarks

The handle may use one of the various Win32 wait functions, such as `WaitForSingleObject` or `WaitForMultipleObjects`.

For read streams, set the event when there is data available to read and reset it whenever there is no available data. For write streams, set the event when all of the data has been written to the device, and reset it at any time when there is still data available to be played.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechBaseStream](#)

Format Property

The **Format** property gets and sets the cached wave format of the stream as an SpAudioFormat object.

Syntax

Set: *ISpeechBaseStream*.**Format** = [SpAudioFormat](#)

Get: [SpAudioFormat](#) = *ISpeechBaseStream*.**Format**

Parts

ISpeechBaseStream

The owning object.

SpAudioFormat

Set: An SpAudioFormat object that sets the wave format.

Get: An SpAudioFormat object that gets the wave format.

Example

For an example of the use of the Format property, see the code example in the SpAudioFormat [GetWaveFormatEx](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechBaseStream](#)

Read Method

The **Read** method reads data from a stream object.

The Read method reads text or audio data from the stream into a Variant variable. Data is read from the Seek pointer of the stream until the specified number of bytes has been copied, or the end of the stream has been reached. When the method has completed, it returns the actual number of bytes read and resets the Seek pointer one byte past the last byte read.

```
ISpeechBaseStream.Read(  
    Buffer As Variant,  
    NumberOfBytes As Long  
) As Long
```

Parameters

Buffer

Specifies a Variant variable to receive the data.

NumberOfBytes

Specifies the number of bytes of data to attempt to read from the audio stream.

Return Value

A Long variable containing the actual number of bytes read from the stream object.

Example

Use of the Read method is demonstrated in a [code example](#) at the end of the ISpeechBaseStream section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechBaseStream](#)

Seek Method

The **Seek** method returns the current read position of the stream in bytes.

The Seek method may also move the Seek pointer forward or backward in the stream. The parameter Position, specifies a number of bytes to move the Seek pointer forward in the stream; negative values specify moving the Seek pointer backward. The parameter Origin, specifies the point from which the forward or backward movement will begin. When the method has completed, it returns a Variant variable containing the new Seek pointer.

```
ISpeechBaseStream.Seek(  
    Position As Variant,  
    [Origin As SpeechStreamSeekPositionType = SSSPTRelative  
) As Variant
```

Parameters

Position

Specifies the number of bytes to move the Seek pointer forward in the stream. Negative values move the pointer backward.

Origin

[Optional] Specifies the Origin. Default value is SSSPTRelativeToStart.

Return Value

A Variant variable containing the new Seek pointer.

Remarks

The following are examples. This statement sets the Seek pointer 23,456 bytes past the start of the stream and returns a Variant containing the new Seek pointer:

```
varCurPos = S.Seek(23456, SSSPTRelativeToStart)
```

This statement moves the Seek pointer forward 23,456 bytes and returns a Variant containing the new Seek pointer:

```
varCurPos = S.Seek(23456, SSSPTRelativeToCurrentPosi
```

This statement sets the Seek pointer to the end of the stream and returns a Variant containing the new Seek pointer, which in this case is equal to the size of the stream:

```
varCurPos = S.Seek(0, SSSPTRelativeToEnd)
```

Example

Use of the Seek method is demonstrated in a [code example](#) at the end of the ISpeechBaseStream section.



Interface: [ISpeechAudio](#)

Type: Hidden

SetState Method

The **SetState** method sets the audio state with a `SpeechAudioState` constant.

```
ISpeechAudio.SetState(  
    State As SpeechAudioState  
)
```

Parameters

State

Specifies a member of the `SpeechAudioState` enumeration.

Return Value

None.



Interface: [ISpeechAudio](#)

Status Property

The **Status** property returns the audio status as an ISpeechAudioStatus object.

Syntax

Set: (This property is read-only)

Get: [ISpeechAudioStatus](#) = *ISpeechAudio*.**Status**

Parts

ISpeechAudio

The owning object.

ISpeechAudioStatus

Set: (This property is read-only)

Get: An ISpeechAudioStatus variable that gets the audio status.



Interface: [ISpeechAudio](#)

Volume Property

The **Volume** property gets and sets the volume (loudness) level.

The volume level is on a linear scale from zero to 10,000.

Syntax

```
Set: ISpeechAudio.Volume = Long
```

```
Get: Long = ISpeechAudio.Volume
```

Parts

ISpeechAudio

The owning object.

Long

Set: A Long variable that sets the property.

Get: A Long variable that gets the property.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechBaseStream](#)

Write Method

The **Write** method writes data to a stream object.

The Write method writes text or audio data from a Variant variable into the stream, starting at the Seek pointer and writing until all data has been copied. When the method has completed, it returns the number of bytes written, and resets the Seek pointer one byte past the last byte written.

```
ISpeechBaseStream.Write(  
    Buffer As Variant  
) As Long
```

Parameters

Buffer

A Variant variable containing the data to be written.

Return Value

A Long variable indicating the number of bytes written.

Example

Use of the Write method is demonstrated in a [code example](#) at the end of the *ISpeechBaseStream* section.



ISpeechAudioBufferInfo

The **ISpeechAudioBufferInfo** automation interface defines the audio stream buffer information.

Automation Interface Elements

The ISpeechAudioBufferInfo automation interface contains the following elements:

Properties	Description
BufferSize Property	Gets and sets the size of the audio object's buffer, in milliseconds.
EventBias Property	Gets and sets the amount of time, in milliseconds, by which event notifications precede the actual occurrence of the events.
MinNotification Property	Gets and sets the minimum preferred time, in milliseconds, between the actual time an event notification occurs and the ideal time.



Object: [ISpeechAudioBufferInfo](#)

BufferSize Property

The **BufferSize** property gets and sets the size of the audio object's buffer, in milliseconds.

For readable audio objects, this is simply a preferred size because readable objects will automatically expand their buffers to accommodate data. For writable audio objects, this is the amount of audio data that will be buffered before a call to Write will block.

This value must be greater than or equal to 200 milliseconds. A reasonable default is 500ms.

Syntax

```
Set: ISpeechAudioBufferInfo.BufferSize = Long
```

```
Get: Long = ISpeechAudioBufferInfo.BufferSize
```

Parts

ISpeechAudioBufferInfo

The owning object.

Long

Set: A Long variable that sets the buffer size.

Get: A Long variable that gets the buffer size.



Object: [ISpeechAudioBufferInfo](#)

EventBias Property

The **EventBias** property gets and sets the amount of time, in milliseconds, by which event notifications precede the actual occurrence of the events.

For example, setting a value of 100 for the event bias would cause all events to be notified 100 milliseconds prior to the audio data being played. This can be useful for applications needing time to animate mouths for TTS voices.

Syntax

```
Set: ISpeechAudioBufferInfo.EventBias = Long
```

```
Get: Long = ISpeechAudioBufferInfo.EventBias
```

Parts

ISpeechAudioBufferInfo

The owning object.

Long

Set: A Long variable that sets the property.

Get: A Long variable that gets the property.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechAudioBufferInfo](#)

MinNotification Property

The **MinNotification** property gets and sets the minimum preferred time, in milliseconds, between the actual time an event notification occurs and the ideal time.

More CPU resources are needed when the amount of time is shorter; however, the event notifications are more timely. This value must be greater than zero and no more than one quarter the size of the BufferSize property.

Syntax

Set: *ISpeechAudioBufferInfo*.**MinNotification** = *Long*

Get: *Long* = *ISpeechAudioBufferInfo*.**MinNotification**

Parts

ISpeechAudioBufferInfo

The owning object.

Long

Set: A Long variable that sets the property.

Get: A Long variable that gets the property.



ISpeechAudioStatus

The **ISpeechAudioStatus** automation interface provides control over the operation of real-time audio streams.

It is intended for use when the audio input or output source is not a standard Windows multimedia device. An audio stream connected to a microphone or a telephone line would be a typical use.

Automation Interface Elements

The ISpeechAudioStatus automation interface contains the following elements:

Properties	Description
CurrentDevicePosition Property	Returns the current read or write position of the stream or device in bytes.
CurrentSeekPosition Property	Returns the current seek position in the stream or device in bytes.
FreeBufferSpace Property	Returns the size of the free space in the stream or device in bytes.
NonBlockingIO Property	Returns the amount of data which can be read from or written to the stream or device without blocking.
State Property	Returns the state of the audio stream or device.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechAudioStatus](#)

CurrentDevicePosition Property

The **CurrentDevicePosition** property returns the current read or write position of the stream or device in bytes.

This is the position in the stream where the device is currently reading or writing. For readable streams, this value will always be greater than or equal to the CurrentSeekPosition property. For writable streams, this value will always be less than or equal to the CurrentSeekPosition property.

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechAudioStatus*.**CurrentDevicePosition**

Parts

ISpeechAudioStatus

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant variable returning the position.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechAudioStatus](#)

CurrentSeekPosition Property

The **CurrentSeekPosition** property returns the current seek position in the audio stream or device in bytes.

This is the position in the stream where the next read or write will be performed.

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechAudioStatus*.**CurrentSeekPosition**

Parts

ISpeechAudioStatus

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant variable returning the position.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechAudioStatus](#)

FreeBufferSpace Property

The **FreeBufferSpace** property returns the size of the free space in the stream or device in bytes.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechAudioStatus*.**FreeBufferSpace**

Parts

ISpeechAudioStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the size.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechAudioStatus](#)

NonBlockingIO Property

The **NonBlockingIO** property returns the amount of data which can be read from or written to the stream or device without blocking.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechAudioStatus*.**NonBlockingIO**

Parts

ISpeechAudioStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the count in bytes.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechAudioStatus](#)

State Property

The **State** property returns the state of the audio stream or device.

Syntax

Set: (This property is read-only)

Get: [SpeechAudioState](#) = ISpeechAudioStatus.**State**

Parts

ISpeechAudioStatus

The owning object.

SpeechAudioState

Set: (This property is read-only)

Get: A SpeechAudioState object returning the state of the stream or device.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechBaseStream

The **ISpeechBaseStream** automation interface defines properties and methods for manipulating data streams.

ISpeechBaseStream objects normally contain audio data, but may also be used for text data.

The Read, Write and Seek methods maintain a pointer referred to as the Seek pointer. The [Read](#) and [Write](#) methods begin reading or writing at the Seek pointer, and reset the Seek pointer one byte past the last byte read or written. The [Seek](#) method returns the current pointer, and can also move the Seek pointer forward or backward in the stream, starting from the Seek pointer, or relative to the beginning or the end of the stream.

Use of the Read, Write and Seek methods are demonstrated in a [code example](#) at the end of the ISpeechBaseStream section.

The ISpeechBaseStream is not an object in its own right, but is implemented by other objects, such as [SpFileStream](#) and [SpMemoryStream](#). SAPI does not call ISpeechBaseStream methods, but uses the underlying COM interfaces. For this reason, a custom object cannot be created using the ISpeechBaseStream interface.

Automation Interface Elements

The ISpeechBaseStream automation interface contains the following elements:

Properties	Description
Format Property	Gets and sets the cached wave format of the stream as an

SpAudioFormat object.

Methods	Description
<u>Read</u> Method	Reads data from the audio stream.
<u>Seek</u> Method	Returns the current read position of the audio stream in bytes.
<u>Write</u> Method	Writes data to the audio stream.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechBaseStream](#)

Example

The following Visual Basic form code demonstrates the use of the ISpeechBaseStream methods Read, Write and Seek. These ISpeechBaseStream methods are inherited by SpCustomStream, SpFileStream and SpMemoryStream. This example uses SpFileStream, but the methods work the same in all three objects.

To run this code, create a form with the following controls:

- Two command buttons called Command1 and Command2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object and two SpFileStream objects. The Command1_Click procedure speaks the words "one," "point," "five," and "SAPI" into the first file stream, using the Seek method to find the ending byte position of each word. It then reads the audio data for each word separately, and writes a new stream with the order of the words reversed. The Command2_Click procedure plays back the two files created by Command1.

```
Option Explicit
```

```
Const FILENAME1 = "c:\First.wav"  
Const FILENAME2 = "c:\Second.wav"
```

```
Dim V As SpeechLib.SpVoice  
Dim S1 As SpeechLib.SpFileStream  
Dim S2 As SpeechLib.SpFileStream
```

```
Private Sub Command1_Click()
```

```

Dim varT(3) As Variant      'text to be spoken
Dim varP(3) As Variant      'positions in output stream
Dim varD(3) As Variant      'audio data chunks
Dim varStart As Variant
Dim ii As Integer

varT(0) = "one": varT(1) = "point": varT(2) = "five": va

'Create WAV file of "one point five SAPI"
'Speak the words into a single filestream object,
'and remember the end-of-stream position of each word.

S1.Open FILENAME1, SSFMCreateForWrite
Set V.AudioOutputStream = S1
For ii = 0 To UBound(varT)
    V.Speak varT(ii)
    varP(ii) = S1.Seek(0, SSSPTRelativeToCurrentPosition)
Next ii
S1.Close

'Read the words from the first file into the variant array
'Write them back into the second file in reverse order.

S1.Open FILENAME1, SSFMOpenForRead
S2.Open FILENAME2, SSFMCreateForWrite

varStart = 0
For ii = 0 To UBound(varT)
    S1.Read varD(ii), varP(ii) - varStart
    varStart = varP(ii)
Next ii

For ii = UBound(varT) To 0 Step -1
    S2.Write varD(ii)
Next ii

S2.Close
S1.Close

'After using AudioOutputStream, reset voice AudioOutput
Set V.AudioOutput = V.GetAudioOutputs("").Item(0)

```

```
Command1.Enabled = False
Command2.Enabled = True
```

```
End Sub
```

```
Private Sub Command2_Click()
```

```
    S1.Open FILENAME1, SSFMOpenForRead
    S2.Open FILENAME2, SSFMOpenForRead
```

```
    'Use first male voice to announce the results
    Set V.Voice = V.GetVoices("gender=male").Item(0)
```

```
    V.Speak "This is the first sound file", SVSFlagsAsync
    V.SpeakStream S1, SVSFlagsAsync
```

```
    V.Speak "This is the second sound file", SVSFlagsAsync
    V.SpeakStream S2, SVSFlagsAsync
```

```
    Do
        DoEvents
    Loop Until V.WaitUntilDone(1)
```

```
    S1.Close
    S2.Close
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpeechLib.SpVoice
    Set S1 = New SpFileStream           'Create stream1
    Set S2 = New SpFileStream           'Create stream2
    Command2.Enabled = False
```

```
End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechDataKey

The **ISpeechDataKey** automation interface provides read and write access to the speech configuration database.

The Speech configuration database contains folders which represent the resources on a computer which are used by SAPI 5.1 SR and TTS. These folders are organized into resource categories, such as voices, lexicons, and audio input devices. The [SpObjectTokenCategory](#) object provides access to a category of resources, and the [SpObjectToken](#) object provides access to a single resource.

An ISpeechDataKey object is typically created by the [DataKey](#) property of an SpObjectToken or the [GetDataKey](#) method of an SpObjectTokenCategory object. Such an ISpeechDataKey object provides read and write access to the database folder represented by its parent token or token category object. Further ISpeechDataKey objects can be created by [CreateKey](#) and [OpenKey](#) calls on existing ISpeechDataKey objects. ISpeechDataKey methods can create, delete and enumerate subfolders and values in the database folder represented by an ISpeechDataKey object.

Automation Interface Elements

The ISpeechDataKey automation interface contains the following elements:

Methods	Description
CreateKey Method	Creates the specified subkey within the data key.
DeleteKey Method	Deletes the specified subkey from the data key.

<u>DeleteValue</u> Method	Deletes the specified value from the data key.
<u>EnumKeys</u> Method	Returns the name of one subkey of the data key, specified by its index.
<u>EnumValues</u> Method	Returns the name of one value of the data key, specified by its index.
<u>GetBinaryValue</u> Method	Gets the specified binary value from the data key.
<u>GetLongValue</u> Method	Gets the specified Long value from the data key.
<u>GetStringValue</u> Method	Gets the specified String value from the data key.
<u>OpenKey</u> Method	Opens the specified subkey of the data key as another data key object.
<u>SetBinaryValue</u> Method	Sets the specified binary value in the data key.
<u>SetLongValue</u> Method	Sets the specified Long value in the data key.
<u>SetStringValue</u> Method	Sets the specified String value in the data key.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

CreateKey Method

The **CreateKey** method creates the specified subkey within the data key.

```
ISpeechDataKey.CreateKey(  
    SubKeyName As String  
) As ISpeechDataKey
```

Parameters

SubKeyName

The name of the subkey.

Return Value

An *ISpeechDataKey* object representing the new subkey.

For an example of the use of the `CreateKey` method, see the code example in the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

DeleteKey Method

The **DeleteKey** method deletes the specified subkey from the data key.

```
ISpeechDataKey.DeleteKey(  
    SubKeyName As String  
)
```

Parameters

SubKeyName

The name of the subkey.

Return Value

None.

For an example of the use of the DeleteKey method, see the code example in the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

DeleteValue Method

The **DeleteValue** method deletes the specified value from the data key.

```
ISpeechDataKey.DeleteValue(  
    ValueName As String  
)
```

Parameters

ValueName

The name of the value.

Return Value

None.

Example

For an example of the use of the DeleteValue method, see the code example in the [EnumKeys](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

EnumKeys Method

The **EnumKeys** method returns the name of one subkey of the data key, specified by its index.

The starting index is zero. A count of subkeys or an enumeration of subkey names can be performed by calling this method repetitively, starting with an index of zero, and increasing the index until all items are enumerated. This is indicated by an SPERR_NO_MORE_ITEMS error.

```
ISpeechDataKey.EnumKeys(  
    Index As Long  
) As String
```

Parameters

Index

The index of the subkey to be returned.

Return Value

A String variable containing the name of the subkey.

Example

The following Visual Basic form code demonstrates several *ISpeechDataKey* methods, including *OpenKey*, *EnumKeys*, *SetStringValue*, and *DeleteValue*. It also demonstrates the creation and deletion of custom object token attributes, and the selection of tokens by these custom attributes. A custom

attribute called "SenseOfHumor" is created for each voice token. This attribute name is intentionally unrelated to any actual characteristics of a voice or a voice token; it is intended to be obviously meaningless and easily distinguishable from installed voice attributes. To run this code, create a form with the following controls:

- A list box called List1
- Two command buttons called Command1 and Command2

Copy the code below and paste it into the Declarations section of the form.

The Form_Load procedure creates an SpObjectTokenCategory object and sets it to the category of voices in the speech configuration database.

The Command1 procedure sets one data key object to the category of voices, and another data key object to the Tokens subfolder. It then enumerates the voice tokens contained in Tokens and writes a value called SenseOfHumor in the Attributes subfolder of each voice token. The values written are "0" and "1" alternately, so that every second voice has a non-zero SenseOfHumor attribute value. Finally, the procedure calls a subroutine which uses the SenseOfHumor attribute in a GetVoices call.

The Command2 procedure enumerates the voice token "Attributes" subfolders and removes all "SenseOfHumor" attributes. It then calls the subroutine which uses "SenseOfHumor" attributes in a GetVoices call after these attributes have been removed.

```
Option Explicit
```

```
Dim C As SpeechLib.SpObjectTokenCategory      'one object token
Dim T As SpeechLib.SpObjectToken             'one object token
Dim V As SpeechLib.SpVoice                   'Voice used for

Dim Kc As SpeechLib.ISpeechDataKey          'DataKey of the
Dim Kf As SpeechLib.ISpeechDataKey          'DataKey of its
```

```
Dim Kt As SpeechLib.ISpeechDataKey           'DataKey of one voice
```

```
Const SPERR_NO_MORE_ITEMS = &H80045039;
```

```
Private Sub Command1_Click()
```

```
    Dim nn As Long
```

```
    Dim strKey As String, strValue As String
```

```
    Set Kc = C.GetDataKey                     'Get DataKey of voice
```

```
    Set Kf = Kc.OpenKey("Tokens")           'Get DataKey of voice
```

```
    nn = 0
```

```
    On Error Resume Next
```

```
    Do
```

```
        strKey = Kf.EnumKeys(nn)
```

```
        If Err.Number = SPERR_NO_MORE_ITEMS Then Exit Do
```

```
        'Write an attribute for each voice:
```

```
        'Attribute "SenseOfHumor" will be true for alternative voices
```

```
        Set Kt = Kf.OpenKey(strKey)         'Kt = the DataKey of voice
```

```
        Set Kt = Kt.OpenKey("Attributes")   'Kt = the DataKey of voice
```

```
        If nn Mod 2 Then strValue = "1" Else strValue = "0"
```

```
        Call Kt.SetStringValue("SenseOfHumor", strValue)
```

```
        nn = nn + 1
```

```
    Loop
```

```
    Err.Clear
```

```
    On Error GoTo 0
```

```
    Call TestAttributes                       'Select voices using this attribute
```

```
End Sub
```

```
Private Sub Command2_Click()
```

```
    Dim nn As Long
```

```
    Dim strKey As String
```

```
    Set Kc = C.GetDataKey                     'Get DataKey of voice
```

```
    Set Kf = Kc.OpenKey("Tokens")           'Get DataKey of voice
```

```
    On Error Resume Next
```

```
    Do
```

```

    strKey = Kf.EnumKeys(nn)
    If Err.Number = SPERR_NO_MORE_ITEMS Then Exit Do
    Set Kt = Kf.OpenKey(strKey)           'Kt = the DataKey
    Set Kt = Kt.OpenKey("Attributes")    'Kt = the DataKey
    Call Kt.DeleteValue("SenseOfHumor")
    nn = nn + 1

Loop
Err.Clear
On Error GoTo 0

    Call TestAttributes      'Select voices using this attrib

End Sub

Private Sub Form_Load()

    'Create new token object, and set its ID to voice tokens

    Set C = New SpObjectTokenCategory
    C.SetId SpeechCategoryVoices

    Set V = New SpVoice

End Sub

Private Sub TestAttributes()

    List1.Clear
    List1.AddItem "Voices with a sense of humor"

    For Each T In V.GetVoices("senseofhumor=1")
        List1.AddItem "    " & T.GetDescription
    Next

    List1.AddItem ""
    List1.AddItem "Voices with no sense of humor"

    'When the "SenseOfHumor" attribute is not present,
    'the selection "SenseOfHumor!=1" shows all voices,
    'bau the selection "SenseOfHumor=0" would show no voices

```

```
For Each T In V.GetVoices("senseofhumor!=1")
    List1.AddItem " " & T.GetDescription
Next
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

EnumValues Method

The **EnumValues** method returns the name of one value of the data key, specified by its index.

The starting index is zero. A count of values or an enumeration of value names can be performed by calling this method repetitively, starting with an index of zero, and increasing the index until all items are enumerated.

```
ISpeechDataKey.EnumValues(  
    Index As Long  
) As String
```

Parameters

Index

The index of the value to be returned.

Return Value

A String variable containing the name of the value.

For an example of the use of the EnumValues method, see the code example in the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

GetBinaryValue Method

The **GetBinaryValue** method gets the specified binary value from the data key.

```
ISpeechDataKey.GetBinaryValue(  
    ValueName As String  
) As Variant
```

Parameters

ValueName

The name of the Value.

Return Value

A Variant variable.

For an example of the use of the GetBinaryValue method, see the code example in the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

GetLongValue Method

The **GetLongValue** method gets the specified Long value from the data key.

```
ISpeechDataKey.GetLongValue(  
    ValueName As String  
) As Long
```

Parameters

ValueName
The name of the Value.

Return Value

A Long variable.

For an example of the use of the GetLongValue method, see the code example in the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

GetStringValue Method

The **GetStringValue** method gets the specified String value from the data key.

```
ISpeechDataKey.GetStringValue(  
    ValueName As String  
) As String
```

Parameters

ValueName

The name of the Value.

Return Value

A String variable.

For an example of the use of the GetStringValue method, see the code example in the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

OpenKey Method

The **OpenKey** method opens the specified subkey of the data key as another data key object.

```
ISpeechDataKey.OpenKey(  
    SubKeyName As String  
) As ISpeechDataKey
```

Parameters

SubKeyName
Name of the subkey.

Return Value

An *ISpeechDataKey* variable representing the subkey.

Example

For an example of the use of the `OpenKey` method, see the code example in the [EnumKeys](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

SetBinaryValue Method

The **SetBinaryValue** method sets the specified binary value in the data key.

```
ISpeechDataKey.SetBinaryValue(  
    ValueName As String,  
    Value As Variant  
)
```

Parameters

ValueName

The name of the value.

Value

The binary data value.

Return Value

None.

For an example of the use of the SetBinaryValue method, see the code example in the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

SetLongValue Method

The **SetLongValue** method sets the specified Long value in the data key.

```
ISpeechDataKey.SetLongValue(  
    ValueName As String,  
    Value As Long  
)
```

Parameters

ValueName

The name of the value.

Value

The Long data value.

Return Value

None.

For an example of the use of the SetLongValue method, see the code example in the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

SetStringValue Method

The **SetStringValue** method sets the specified String value in the data key.

```
ISpeechDataKey.SetStringValue(  
    ValueName As String,  
    Value As String  
)
```

Parameters

ValueName

The name of the value.

Value

The String data value.

Return Value

None.

Example

For an example of the use of the SetStringValue method, see the code example in the [EnumKeys](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechDataKey](#)

Code Example

The following Visual Basic form code demonstrates the methods of the ISpeechDataKey interface. To run this code, create a form containing the following controls:

- Three command buttons, called Command1, Command2, and Command3

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object and gets its object token.

The Command1 procedure creates a data key, called K1, from the voice's token. This key provides data access to the token's folder in the Speech configuration database. The procedure then uses that datakey's CreateKey method to create a subfolder called "ISpeechDataKey" within its folder; the new datakey K2, which provides access to the new subfolder, is returned by the CreateKey method. Next, the procedure uses datakey K2 to create a subfolder called "Test1" within the "ISpeechDataKey" folder. The CreateKey method returns datakey K3, which accesses the folder. The procedure then creates a default value, a string value, a long value and a binary value within the "Test1" folder. Finally, the procedure creates a subfolder called "Test2" within the "ISpeechDataKey" folder, and writes the same four values into it.

The Command2 procedure opens the "ISpeechDataKey" subfolder, and enumerates the subfolders "Test1" and "Test2" and enumerates the values with each subfolder.

The Command3 procedure enumerates and deletes the values within the "Test1" and "Test2" subfolders, deletes these folders, and finally deletes the "ISpeechDataKey" folder.

Option Explicit

```
Dim V As SpeechLib.SpVoice  
Dim T As SpeechLib.SpObjectToken
```

```
Dim K1 As SpeechLib.ISpeechDataKey  
Dim K2 As SpeechLib.ISpeechDataKey  
Dim K3 As SpeechLib.ISpeechDataKey
```

```
Const SPERR_NO_MORE_ITEMS = &H80045039;
```

```
Private Sub Command1_Click()
```

```
    'Create subkey voice\ISpeechDataKey  
    Set K1 = T.DataKey  
    Set K2 = K1.CreateKey("ISpeechDataKey")
```

```
    'Create subkey voice\ISpeechDataKey\Test1  
    Set K3 = K2.CreateKey("Test1")
```

```
    Call K3.SetStringValue("", "The default value")  
    Call K3.SetStringValue("Description", "A test string value")  
    Call K3.SetLongValue("Long_10K", 10000)  
    Call K3.SetBinaryValue("Binary_100K", 100000)
```

```
    'Create subkey voice\ISpeechDataKey\Test2  
    Set K3 = K2.CreateKey("Test2")
```

```
    Call K3.SetStringValue("", "The default value")  
    Call K3.SetStringValue("Description", "A test string value")  
    Call K3.SetLongValue("Long_10K", 10000)  
    Call K3.SetBinaryValue("Binary_100K", 100000)
```

```
End Sub
```

```
Private Sub Command2_Click()
```

```
    Dim nn As Long, mm As Long  
    Dim strKey As String
```

```
    Set K1 = T.DataKey  
    Debug.Print "Key " & T.GetDescription
```

```

Set K2 = K1.OpenKey("ISpeechDataKey")
Debug.Print "    Key ISpeechDataKey"

'Enumerate keys within "ISpeechDataKey"
nn = 0
On Error Resume Next
Do
    strKey = K2.EnumKeys(nn)
    If Err.Number = SPERR_NO_MORE_ITEMS Then Exit Do
    Debug.Print "        Key "" & strKey & """"
    nn = nn + 1

    'Enumerate values within each subkey
    Set K3 = K2.OpenKey(strKey)
    Call EnumerateValues(K3)

Loop
Err.Clear
On Error GoTo 0

End Sub

Private Sub Command3_Click()
    Dim strKey As String, strValue As String

    Set K1 = T.DataKey
    Set K2 = K1.OpenKey("ISpeechDataKey")

    'Enumerate and delete keys loop
    On Error Resume Next
    Do
        strKey = K2.EnumKeys(0)
        If Err.Number = SPERR_NO_MORE_ITEMS Then Exit Do

        'Enumerate and delete values loop
        Set K3 = K2.OpenKey(strKey)
        Do
            strValue = K3.EnumValues(0)
            If Err.Number = 0 Then
                Call K3.DeleteValue(strValue)
            End If
        Loop Until Err.Number = SPERR_NO_MORE_ITEMS
    Loop

```

```

        Err.Clear

        Call K2.DeleteKey(strKey)
    Loop

    Call K1.DeleteKey("ISpeechDataKey")

End Sub

Private Sub Form_Load()

    'T is object token for first available voice
    Set V = New SpVoice
    Set T = V.GetVoices().Item(0)

End Sub

Private Sub EnumerateValues(DK As ISpeechDataKey)
    Dim nn As Long, strValue As String
    Dim varVariant As Variant
    Dim lngLong As Long
    Dim strString As String

    nn = 0
    On Error Resume Next
    Do
        strValue = DK.EnumValues(nn)
        If Err.Number = SPERR_NO_MORE_ITEMS Then Exit Do

        If Left(strValue, 6) = "Binary" Then
            'Binary
            varVariant = DK.GetBinaryValue(strValue)
            strString = Format(varVariant)
        ElseIf Left(strValue, 4) = "Long" Then
            'Long
            lngLong = DK.GetLongValue(strValue)
            strString = Format(lngLong)
        Else
            'String
            strString = DK.GetStringValue(strValue)
        End If
    End Sub

```

```
Debug.Print "  
nn = nn + 1
```

```
Val "" & strValue & "" = ""
```

```
Loop  
Err.Clear
```

```
End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechGrammarRule

The **ISpeechGrammarRule** automation interface defines the properties and methods of a speech grammar rule.

Automation Interface Elements

The ISpeechGrammarRule automation interface contains the following elements:

Properties	Description
Attributes Property	Returns information about the attributes of a speech grammar rule.
Id Property	Specifies the ID of the speech grammar rule.
InitialState Property	Specifies the initial state of the speech grammar rule.
Name Property	Specifies the name of the speech grammar rule.

Methods	Description
AddResource Method	Adds a string to a speech rule.
AddState Method	Adds a state to a speech rule.
Clear Method	Clears a rule, leaving only its initial state.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRule](#)

AddResource Method

The **AddResource** method adds one or more string name/value pair associated with a rule.

This method is used with an interpreter rule, in which case the interpreter can call the C/C++ function [ISpCFGInterpreterSite::GetResourceValue](#) to get the value of a specified resource name.

```
ISpeechGrammarRule.AddResource(  
    ResourceName As String,  
    ResourceValue As String  
)
```

Parameters

ResourceName
Specifies the ResourceName.

ResourceValue
Specifies the ResourceValue.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRule](#)

AddState Method

The **AddState** method adds a state to a speech rule.

This method can be used with the `ISpeechGrammarRuleState` methods `AddRuleTransition`, `AddSpecialTransition`, or `AddWordTransition` to modify speech rules programmatically.

ISpeechGrammarRule.**AddState()** As [ISpeechGrammarRuleState](#)

Parameters

None.

Return Value

An `ISpeechGrammarRuleState` object.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRule](#)

Attributes Property

The **Attributes** property returns information about the attributes of each grammar rule.

This property consists of one or more members of the `SpeechRuleAttributes` enumeration.

Syntax

Set: (This property is read-only)

Get: [SpeechRuleAttributes](#) = `ISpeechGrammarRule.Attributes`

Parts

ISpeechGrammarRule

The owning object.

SpeechRuleAttributes

Set: (This property is read-only)

Get: One or more `SpeechRuleAttributes` flags representing the attributes of the rule.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRule](#)

Clear Method

The **Clear** method clears a rule, leaving only its initial state.

ISpeechGrammarRule.**Clear()**

Parameters

None.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRule](#)

Id Property

The **Id** property specifies the ID of the speech grammar rule.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechGrammarRule.Id*

Parts

ISpeechGrammarRule

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the ID of the rule.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRule](#)

InitialState Property

The **InitialState** property specifies the initial state of the speech grammar rule.

Syntax

Set: (This property is read-only)

Get: [ISpeechGrammarRuleState](#) =
ISpeechGrammarRule.**InitialState**

Parts

ISpeechGrammarRule

The owning object.

ISpeechGrammarRuleState

Set: (This property is read-only)

Get: An *ISpeechGrammarRuleState* variable that gets the initial rule state.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRule](#)

Name Property

The **Name** property specifies the name of the speech grammar rule.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechGrammarRule*.**Name**

Parts

ISpeechGrammarRule

The owning object.

String

Set: (This property is read-only)

Get: A String variable that gets the name of the rule.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechGrammarRules

The **ISpeechGrammarRules** automation interface represents a collection of [ISpeechGrammarRule](#) objects.

Automation Interface Elements

The ISpeechGrammarRules automation interface contains the following elements:

Properties	Description
Count Property	Returns the count of objects in the collection.
Dynamic Property	Determines whether the grammar rules contained in the collection were created as dynamic.

Methods	Description
Add Method	Creates a new grammar rule in an ISpeechGrammarRules collection.
Commit Method	Compiles the rules in the rule collection.
CommitAndSave Method	Compiles the rules in the rule collection and saves the result.
FindRule Method	Returns a grammar rule, specified by Name or by ID.
Item Method	Returns a member of the collection specified by its index.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRules](#)

Add Method

The **Add** method creates a new `ISpeechGrammarRule` object in an `ISpeechGrammarRules` collection.

RuleName, or *RuleId*, or both must be specified to identify the rule. If *RuleName* is specified (not an empty string such as ""), it must be unique within the grammar. If *RuleId* is specified (a value other than zero), it must be unique within the grammar.

```
ISpeechGrammarRules.Add(  
    RuleName As String,  
    Attributes As SpeechRuleAttributes,  
    [RuleId As Long = 0]  
) As ISpeechGrammarRule
```

Parameters

RuleName

Specifies the `RuleName` of the new rule.

Attributes

Specifies the `Attributes` of the new rule.

RuleId

Specifies the `RuleId` of the new rule. Default value is zero.

Return Value

The `Add` method returns the newly-created `ISpeechGrammarRule` object.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRules](#)

Commit Method

The **Commit** method compiles the rules in the rule collection.

ISpeechGrammarRules.**Commit()**

Parameters

None.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRules](#)

CommitAndSave Method

The **CommitAndSave** method compiles the rules in the rule collection and saves the result.

```
ISpeechGrammarRules.CommitAndSave(  
    ErrorText As String  
) As Variant
```

Parameters

ErrorText

The text of any errors that may result from compiling and saving the grammar upon calling the method.

Return Value

The CommitAndSave method returns a Variant variable.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRules](#)

Count Property

The **Count** property returns the number of `ISpeechGrammarRule` objects in the `ISpeechGrammarRules` object.

Syntax

Set: (This property is read-only)

Get: `Long = ISpeechGrammarRules.Count`

Parts

ISpeechGrammarRules

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the count.

Remarks

The `ISpeechGrammarRules` object is a collection of `ISpeechGrammarRule` objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The `Count` property is one of these common properties.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRules](#)

Dynamic Property

The **Dynamic** property determines whether the grammar rules contained in the collection were created as dynamic.

Syntax

Set: (This property is read-only)

Get: *Boolean* = *ISpeechGrammarRules*.**Dynamic**

Parts

ISpeechGrammarRules

The owning object.

Boolean

Set: (This property is read-only)

Get: A Boolean variable returning the value of the property.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRules](#)

FindRule Method

The **FindRule** method returns a grammar rule, specified by Name or by ID.

```
ISpeechGrammarRules.FindRule(  
    RuleNameOrId As Variant  
) As ISpeechGrammarRule
```

Parameters

RuleNameOrId
Specifies the RuleNameOrId.

Return Value

The FindRule method returns an ISpeechGrammarRule variable.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRules](#)

Item Method

The **Item** method returns a member of the ISpeechGrammarRules collection by its index.

```
ISpeechGrammarRules.Item(  
    Index As Long  
) As ISpeechGrammarRule
```

Parameters

Index
Specifies the Index of the rule.

Return Value

The Item method returns an ISpeechGrammarRule object as specified by the *Index* parameter.

Remarks

The ISpeechGrammarRules object is a collection of ISpeechGrammarRule objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Item method is one of these common methods.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechGrammarRuleState

The **ISpeechGrammarRuleState** automation interface presents the properties and methods of a speech grammar rule state.

Automation Interface Elements

The ISpeechGrammarRuleState automation interface contains the following elements:

Properties	Description
Rule Property	Specifies the rule to which the rule state belongs.
Transitions Property	Specifies the set of transitions out of the rule state.

Methods	Description
AddRuleTransition Method	Adds a rule reference transition from the current rule state to another rule state in the same rule.
AddSpecialTransition Method	Adds a special transition from the current rule state to another rule state in the same rule
AddWordTransition Method	Adds a word transition from this rule state to another rule state in the same rule

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleState](#)

AddRuleTransition Method

The **AddRuleTransition** method adds a rule reference transition from the current rule state to another rule state in the same rule.

When the word sequence accepted by the referenced rule is spoken, the rule can go from the current state to the `DestinationState`. A rule accepts a word sequence when the rule can go from the initial state to the end state using the word sequence.

```
ISpeechGrammarRuleState.AddRuleTransition(  
    DestinationState As ISpeechGrammarRuleState,  
    Rule As ISpeechGrammarRule,  
    [PropertyName As String = ""],  
    [PropertyId As Long = 0],  
    [PropertyValue As Variant = 0],  
    [Weight As Single = 1.0]  
)
```

Parameters

DestinationState

Specifies the `DestinationState`, or the state where the transition ends. `DestinationState` of `Nothing` means the end state of the rule.

Rule

Specifies the Rule.

PropertyName

[Optional] Specifies the `PropertyName`.

PropertyId

[Optional] Specifies the PropertyId.

PropertyValue

[Optional] Specifies the PropertyValue.

Weight

[Optional] Specifies the Weight.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleState](#)

AddSpecialTransition Method

The **AddSpecialTransition** method adds a special transition from the current rule state to another rule state in the same rule.

When the word sequence accepted by the special transition is spoken, the rule can go from the current state to the DestinationState. A rule accepts a word sequence when the rule can go from the initial state to the end state using the word sequence.

Special transitions may not be supported by all speech engines. There are three types of special transitions:

- Wildcard: A Wildcard accepts any one word and ignores it.
- Dictation: A Dictation accept any one word and returns it in the result.
- Textbuffer: A Textbuffer accepts the word sequence and can be set later using [ISpeechRecoGrammar.SetWordSequenceData](#) and [ISpeechRecoGrammar.SetTextSelection](#).

```
ISpeechGrammarRuleState.AddSpecialTransition(  
    DestinationState As ISpeechGrammarRuleState,  
    Type As SpeechSpecialTransitionType,  
    [PropertyName As String = ""],  
    [PropertyId As Long = 0],  
    [PropertyValue As Variant = 0],  
    [Weight As Single = 1.0]  
)
```

Parameters

DestinationState

Specifies the DestinationState, or the state where the transition ends. DestinationState of Nothing means the end state of the rule.

Type

Specifies the Type.

PropertyName

[Optional] Specifies the PropertyName.

PropertyId

[Optional] Specifies the PropertyId.

PropertyValue

[Optional] Specifies the PropertyValue.

Weight

[Optional] Specifies the Weight.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleState](#)

AddWordTransition Method

The **AddWordTransition** method adds a word transition from this rule state to another rule state in the same rule.

When the word sequence specified by the word transition is spoken, the rule can go from this state to the DestinationState. A rule accepts a word sequence when the rule can go from the initial state to the end state using the word sequence.

```
ISpeechGrammarRuleState.AddWordTransition(  
    DestinationState As ISpeechGrammarRuleState,  
    Words As String,  
    [Separators As String = " "],  
    [Type As SpeechGrammarWordType = SGLexical],  
    [PropertyName As String = ""],  
    [PropertyId As Long = 0],  
    [PropertyValue As Variant = 0],  
    [Weight As Single = 1.0]  
)
```

Parameters

DestinationState

Specifies the DestinationState, or the state where the transition ends. DestinationState of Nothing means the end state of the rule.

Words

Specifies the Words.

Separators

[Optional] Specifies the Separators.

PropertyName

[Optional] Specifies the PropertyName.

PropertyId

[Optional] Specifies the PropertyId.

PropertyValue

[Optional] Specifies the PropertyValue.

Weight

[Optional] Specifies the Weight.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleState](#)

Rule Property

The **Rule** property specifies the rule to which the rule state belongs.

Syntax

Set: (This property is read-only)

Get: [ISpeechGrammarRule](#) = *ISpeechGrammarRuleState*.**Rule**

Parts

ISpeechGrammarRuleState

The owning object.

ISpeechGrammarRule

Set: (This property is read-only)

Get: An *ISpeechGrammarRule* variable that gets the property.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleState](#)

Transitions Property

The **Transitions** property specifies the set of transitions out of the rule state.

Syntax

```
Set: (This property is read-only)  
Get: ISpeechGrammarRuleStateTransitions =  
     ISpeechGrammarRuleState.Transitions
```

Parts

ISpeechGrammarRuleState

The owning object.

ISpeechGrammarRuleStateTransitions

Set: (This property is read-only)

Get: An *ISpeechGrammarRuleStateTransitions* object that gets the property.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechGrammarRuleStateTransition

The **ISpeechGrammarRuleStateTransition** automation interface returns data about a transition from one rule state to another, or from a rule state to the end of a rule.

Automation Interface Elements

The ISpeechGrammarRuleStateTransition automation interface contains the following elements:

Properties	Description
<u>NextState</u> Property	Specifies the rule state to which the transition leads.
<u>PropertyId</u> Property	Specifies the Id of a property contained in a semantic tag.
<u>PropertyName</u> Property	Specifies the name of a property contained in a semantic tag.
<u>PropertyValue</u> Property	Returns the value of a property contained in a semantic tag.
<u>Rule</u> Property	Specifies the speech rule to which the transition leads.
<u>Text</u> Property	Returns the recognition text associated with a transition.
<u>Type</u> Property	Specifies the type of the transition.
<u>Weight</u> Property	Assigns the transition a weight relative to its sibling transitions.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleStateTransition](#)

NextState Property

The **NextState** property specifies the rule state to which the transition leads.

Syntax

Set: (This property is read-only)

Get: [ISpeechGrammarRuleState](#) =
ISpeechGrammarRuleStateTransition.**NextState**

Parts

ISpeechGrammarRuleStateTransition

The owning object.

ISpeechGrammarRuleState

Set: (This property is read-only)

Get: An *ISpeechGrammarRuleState* variable that gets the property.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleStateTransition](#)

PropertyId Property

The **PropertyId** property specifies the ID of a property contained in a semantic tag.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechGrammarRuleStateTransition*.**PropertyId**

Parts

ISpeechGrammarRuleStateTransition

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the property.

Remarks

Semantic tags provide a way for grammar designers to put semantic information about a transition into the grammar. The PropertyId, PropertyName and PropertyValue properties provide access to that information at run time.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleStateTransition](#)

PropertyName Property

The **PropertyName** property specifies the name of a property contained in a semantic tag.

Syntax

Set: (This property is read-only)

Get: *String* =

ISpeechGrammarRuleStateTransition.**PropertyName**

Parts

ISpeechGrammarRuleStateTransition

The owning object.

String

Set: (This property is read-only)

Get: A String variable that gets the property.

Remarks

Semantic tags provide a way for grammar designers to put semantic information about a transition into the grammar. The PropertyId, PropertyName and PropertyValue properties provide access to that information at run time.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleStateTransition](#)

PropertyValue Property

The **PropertyValue** property returns the value of a property contained in a semantic tag.

Syntax

Set: (This property is read-only)

Get: *Variant* =

ISpeechGrammarRuleStateTransition.**PropertyValue**

Parts

ISpeechGrammarRuleStateTransition

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant variable returning the value of the semantic tag property.

Remarks

Semantic tags provide a way for grammar designers to put semantic information about a transition into the grammar. The PropertyId, PropertyName and PropertyValue properties provide access to that information at run time.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleStateTransition](#)

Rule Property

The **Rule** property specifies the speech rule to which the transition leads.

Syntax

Set: (This property is read-only)

Get: [ISpeechGrammarRule](#) =
ISpeechGrammarRuleStateTransition.Rule

Parts

ISpeechGrammarRuleStateTransition

The owning object.

ISpeechGrammarRule

Set: (This property is read-only)

Get: An *ISpeechGrammarRule* variable that gets the property.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleStateTransition](#)

Text Property

The **Text** property returns the recognition text associated with a transition. This is text which is added to the recognition results when the transition is processed.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechGrammarRuleStateTransition*.**Text**

Parts

ISpeechGrammarRuleStateTransition

The owning object.

String

Set: (This property is read-only)

Get: A String variable that gets the property.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleStateTransition](#)

Type Property

The **Type** property specifies the type of the transition. The `SpeechGrammarRuleStateTransitionType` enumeration contains the possible transition type values.

Syntax

```
Set: (This property is read-only)  
Get: SpeechGrammarRuleStateTransitionType =  
     ISpeechGrammarRuleStateTransition.Type
```

Parts

ISpeechGrammarRuleStateTransition
The owning object.

SpeechGrammarRuleStateTransitionType
Set: (This property is read-only)
Get: A `SpeechGrammarRuleStateTransitionType` constant returning the transition type.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleStateTransition](#)

Weight Property

The **Weight** property assigns the transition a weight relative to its sibling transitions.

Syntax

Set: (This property is read-only)

Get: *Variant = ISpeechGrammarRuleStateTransition*.**Weight**

Parts

ISpeechGrammarRuleStateTransition

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant variable that gets the property.

Remarks

Recognition proceeds by means of transitions from one rule state to another. Each rule state has a collection of transitions, which represent the possible recognition paths to subsequent rule states.

In the absence of weighting, each transition is considered equally probable. For example, in a rule state with five transitions, each has a 20 percent probability of being followed. The Weight property enables grammar designers to specify a transition as more probable, or less probable, than its sibling transitions.

The weight property for a transition is a fractional number with

a range of zero to one, and the sum of the weights of a rule state's transition should be one.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechGrammarRuleStateTransitions

The **ISpeechGrammarRuleStateTransitions** automation interface represents a collection of [ISpeechGrammarRuleStateTransition](#) objects.

Automation Interface Elements

The ISpeechGrammarRuleStateTransitions automation interface contains the following elements:

Properties	Description
Count Property	Returns the number of objects in the collection.

Methods	Description
Item Method	Returns a member of the collection specified by its index.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleStateTransitions](#)

Count Property

The **Count** property returns the number of `ISpeechGrammarRuleStateTransition` objects in the `ISpeechGrammarRuleStateTransitions` object.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechGrammarRuleStateTransitions*.**Count**

Parts

ISpeechGrammarRuleStateTransitions

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the Count property.

Remarks

The `ISpeechGrammarRuleStateTransitions` object is a collection of `ISpeechGrammarRuleStateTransition` objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all objects in the collection. The `Count` property is one of these common properties.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechGrammarRuleStateTransitions](#)

Item Method

The **Item** method returns a member of the `ISpeechGrammarRuleStateTransitions` collection by its index.

```
ISpeechGrammarRuleStateTransitions.Item(  
    Index As Long  
) As ISpeechGrammarRuleStateTransition
```

Parameters

Index

Specifies the Index of the member to be returned.

Return Value

The Item method returns an `ISpeechGrammarRuleStateTransition` variable.

Remarks

The `ISpeechGrammarRuleStateTransitions` object is a collection of `ISpeechGrammarRuleStateTransition` objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Item method is one of these common methods.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechLexiconPronunciation

The **ISpeechLexiconPronunciation** automation interface provides access to the pronunciations of a speech lexicon word.

Automation Interface Elements

The ISpeechLexiconPronunciation automation interface contains the following elements:

Properties	Description
LangId Property	Returns the language id of the pronunciation.
PartOfSpeech Property	Returns a word's part of speech.
PhonIds Property	Returns the pronunciation of a word as a Variant array of numeric phone ids.
Symbolic Property	Returns the pronunciation of a word as a string of phone symbols.
Type Property	Returns the type of the pronunciation.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechLexiconPronunciation](#)

LangId Property

The **LangId** property returns the language ID of the pronunciation.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechLexiconPronunciation.LangId*

Parts

ISpeechLexiconPronunciation

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the LangId.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechLexiconPronunciation](#)

PartOfSpeech Property

The **PartOfSpeech** property returns a word's part of speech.

Syntax

Set: (This property is read-only)

Get: [SpeechPartOfSpeech](#) =
ISpeechLexiconPronunciation.**PartOfSpeech**

Parts

ISpeechLexiconPronunciation

The owning object.

SpeechPartOfSpeech

Set: (This property is read-only)

Get: A *SpeechPartOfSpeech* object returning the value of the property.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechLexiconPronunciation](#)

Phonelds Property

The **Phonelds** property returns the pronunciation of a word as a Variant array of numeric phone ids.

Pronunciations represented in Phonelds can be converted to phones with the `IdToPhone` method of the [SpPhoneConverter](#) object.

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechLexiconPronunciation*.**Phonelds**

Parts

ISpeechLexiconPronunciation

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant variable returning the value of the property.

Example

The following code snippet creates a lexicon, adds a pronunciation to it, then gets the pronunciation object, and formats the values of the Phonelds as a string. The resulting string is " 46 12 33."

```
Dim objLEX As SpeechLib.SpLexicon
Dim objPRO As SpeechLib.ISpeechLexiconPronunciation
Dim colPRO As SpeechLib.ISpeechLexiconPronunciations
```

```
Set objLEX = New SpeechLib.SpLexicon
```

```
Call objLEX.AddPronunciation("one", 1033, SPSNoun, "w ah n")
```

```
'Get item(0) of the pronunciations collection  
Set colPRO = objLEX.GetPronunciations("one", 1033)  
Set objPRO = colPRO(0)
```

```
Dim varPhoneIds As Variant 'A Variant array of numeric values  
Dim strOut As String 'Display the PhoneId values
```

```
varPhoneIds = objPRO.PhoneIds  
strOut = Str(varPhoneIds(0)) & Str(varPhoneIds(1)) & Str(vari
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechLexiconPronunciation](#)

Symbolic Property

The **Symbolic** property returns the pronunciation of a word as a string of phone symbols.

Pronunciations represented in phones can be converted to Phonelds with the PhoneToId method of the [SpPhoneConverter](#) object.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechLexiconPronunciation*.**Symbolic**

Parts

ISpeechLexiconPronunciation
The owning object.

String

Set: (This property is read-only)

Get: A String variable returning the value of the property.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechLexiconPronunciation](#)

Type Property

The **Type** property returns the type of the pronunciation.

Syntax

Set: (This property is read-only)

Get: [SpeechLexiconType](#) = *ISpeechLexiconPronunciation*.**Type**

Parts

ISpeechLexiconPronunciation

The owning object.

SpeechLexiconType

Set: (This property is read-only)

Get: A *SpeechLexiconType* constant returning the type.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechLexiconPronunciations

The **ISpeechLexiconPronunciations** automation interface represents a collection of [ISpeechLexiconPronunciation](#) objects.

Automation Interface Elements

The ISpeechLexiconPronunciations automation interface contains the following elements:

Properties	Description
Count Property	Returns the number of objects in the collection.

Methods	Description
Item Method	Returns a member of the collection by its index.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechLexiconPronunciations](#)

Count Property

The **Count** property returns the number of objects in the collection.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechLexiconPronunciations*.**Count**

Parts

ISpeechLexiconPronunciations

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the Count property.

Remarks

The *ISpeechLexiconPronunciations* object is a collection of *ISpeechLexiconPronunciation* objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all objects in the collection. The Count property is one of these common properties.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechLexiconPronunciations](#)

Item Method

The **Item** method returns a member of the ISpeechLexiconPronunciations collection by its index.

```
ISpeechLexiconPronunciations.Item(  
    Index As Long  
) As ISpeechLexiconPronunciation
```

Parameters

Index
Specifies the Index.

Return Value

The Item method returns an ISpeechLexiconPronunciation object.

Remarks

The ISpeechLexiconPronunciations object is a collection of ISpeechLexiconPronunciation objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Item method is one of these common methods.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechLexiconWord

The **ISpeechLexiconWord** automation interface provides access to a lexicon word.

Automation Interface Elements

The ISpeechLexiconWord automation interface contains the following elements:

Properties	Description
<u>LangId</u> Property	Returns the language Id of a lexicon word.
<u>Pronunciations</u> Property	Returns the pronunciations of a lexicon word.
<u>Type</u> Property	Returns the type of a lexicon word.
<u>Word</u> Property	Returns the text of a lexicon word.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechLexiconWord](#)

LangId Property

The **LangId** property returns the language ID of a lexicon word.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechLexiconWord*.**LangId**

Parts

ISpeechLexiconWord
The owning object.

Long
Set: (This property is read-only)
Get: A Long variable returning the LangId.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechLexiconWord](#)

Pronunciations Property

The **Pronunciations** property returns the pronunciations of a lexicon word.

The pronunciations are returned as a collection of [ISpeechLexiconPronunciation](#) objects.

Syntax

Set:	(This property is read-only)
Get:	ISpeechLexiconPronunciations = <i>ISpeechLexiconWord</i> . Pronunciations

Parts

ISpeechLexiconWord
The owning object.

ISpeechLexiconPronunciations
Set: (This property is read-only)
Get: An *ISpeechLexiconPronunciations* object returning the pronunciations of the lexicon word.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechLexiconWord](#)

Type Property

The **Type** property returns the type of a lexicon word.

Syntax

Set: (This property is read-only)

Get: [SpeechWordType](#) = *ISpeechLexiconWord*.**Type**

Parts

ISpeechLexiconWord
The owning object.

SpeechWordType
Set: (This property is read-only)
Get: A *SpeechWordType* constant returning the word type.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechLexiconWord](#)

Word Property

The **Word** property returns the text of a lexicon word.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechLexiconWord*.**Word**

Parts

ISpeechLexiconWord
The owning object.

String
Set: (This property is read-only)
Get: A String variable returning the word.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechLexiconWords

The **ISpeechLexiconWords** automation interface represents a collection of [ISpeechLexiconWord](#) objects.

Automation Interface Elements

The ISpeechLexiconWords automation interface contains the following elements:

Properties	Description
Count Property	Returns the number of objects in the collection.

Methods	Description
Item Method	Returns a member of the collection by its index.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechLexiconWords](#)

Count Property

The **Count** property returns the number of `ISpeechLexiconWord` objects in the `ISpeechLexiconWords` collection.

Syntax

Set: (This property is read-only)

Get: `Long = ISpeechLexiconWords.Count`

Parts

ISpeechLexiconWords

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the Count property.

Remarks

The `ISpeechLexiconWords` object is a collection of `ISpeechLexiconWord` objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Count property is one of these common properties.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechLexiconWords](#)

Item Method

The **Item** method returns a member of the ISpeechLexiconWords collection by its index.

```
ISpeechLexiconWords.Item(  
    Index As Long  
) As ISpeechLexiconWord
```

Parameters

Index
Specifies the Index.

Return Value

The Item method returns an ISpeechLexiconWord object.

Remarks

The ISpeechLexiconWords object is a collection of ISpeechLexiconWord objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Item method is one of these common methods.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechObjectTokens

The **ISpeechObjectTokens** automation interface represents a collection of [SpObjectToken](#) objects.

Automation Interface Elements

The ISpeechObjectTokens automation interface contains the following elements:

Properties	Description
Count Property	Returns the number of objects in the collection.

Methods	Description
Item Method	Returns a member of the collection by its index.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechObjectTokens](#)

Count Property

The **Count** property returns the number of SpObjectToken objects in the ISpeechObjectTokens object.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechObjectTokens.Count*

Parts

ISpeechObjectTokens

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the Count property.

Remarks

The ISpeechObjectTokens object is a collection of SpObjectToken objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Count property is one of these common properties.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechObjectTokens](#)

Item Method

The **Item** method returns a member of the ISpeechObjectTokens collection by its index.

```
ISpeechObjectTokens.Item(  
    Index As Long  
) As SpObjectToken
```

Parameters

Index
Specifies the Index.

Return Value

The Item method returns an SpObjectToken variable.

Remarks

The ISpeechObjectTokens object is a collection of SpObjectToken objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Item method is one of these common methods.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechPhraseAlternate

The **ISpeechPhraseAlternate** automation interface enables applications to retrieve alternate phrase information from a speech recognition (SR) engine, and to update the SR engine's language model to reflect committed alternate changes.

There are two types of speech recognition in SAPI: Command and control (C and C) recognition, which is guided by one or more grammars; and dictation recognition, in which the SR engine uses statistical calculations to determine the most likely recognitions of a requested phrase. It should be noted that the two types are not mutually exclusive, and that applications do not request one type of recognition as opposed to the other. Applications simply request recognition; they can determine after the fact which type of recognition produced a particular recognition result.

Dictation recognition is inherently less accurate than C and C recognition, which is guided by grammars. Because of this, there is a practical need for returning alternate recognition results, and for the user to select an alternate as the correct recognition of the phrase.

By default, the [ISpeechRecoResult](#) object contains a single recognition result. If the result was obtained using dictation recognition, then alternate recognitions are available. These results can be accessed using the [ISpeechRecoResult.Alternates](#) method. This method returns a specific number of recognitions in an [ISpeechPhraseAlternates](#) object. It should be noted that the default recognition result is the first [ISpeechPhraseAlternate](#) object in the collection.

Use of the [ISpeechPhraseAlternate](#) object is demonstrated in a [code example](#) at the end of this section.

Automation Interface Elements

The ISpeechPhraseAlternate automation interface contains the following elements:

Properties	Description
NumberOfElementsInResult Property	Returns the count of phrase elements in the alternate's parent ISpeechRecoResult object.
PhraseInfo Property	Returns the ISpeechPhraseInfo object of the alternate's parent ISpeechRecoResult object.
RecoResult Property	Returns the alternate's parent ISpeechRecoResult object.
StartElementInResult Property	Specifies the starting phrase element of the alternate's parent ISpeechRecoResult object.

Methods	Description
Commit Method	Specifies that the alternate recognition should replace the recognition selected by the SR engine.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseAlternate](#)

Commit Method

The **Commit** method specifies that the alternate recognition should replace the recognition selected by the speech recognition (SR) engine.

ISpeechPhraseAlternate.Commit()

Parameters

None.

Return Value

None.

Example

Use of the Commit method is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseAlternate](#)

NumberOfElementsInResult Property

The **NumberOfElementsInResult** property returns the count of phrase elements in the alternate's parent ISpeechRecoResult object.

Syntax

Set: (This property is read-only)

Get: *Long* =
ISpeechPhraseAlternate.**NumberOfElementsInResult**

Parts

ISpeechPhraseAlternate
The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the value of the property.

Example

Use of the ISpeechPhraseAlternate object is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseAlternate](#)

PhraseInfo Property

The **PhraseInfo** property returns the ISpeechPhraseInfo object of the alternate's parent ISpeechRecoResult object.

Syntax

Set: (This property is read-only)

Get: [ISpeechPhraseInfo](#) = *ISpeechPhraseAlternate*.**PhraseInfo**

Parts

ISpeechPhraseAlternate

The owning object.

ISpeechPhraseInfo

Set: (This property is read-only)

Get: An ISpeechPhraseInfo that gets the value of the property.

Example

Use of the ISpeechPhraseAlternate object is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseAlternate](#)

RecoResult Property

The **RecoResult** property returns the alternate's parent ISpeechRecoResult object.

Syntax

```
Set: (This property is read-only)  
Get: ISpeechRecoResult =  
     ISpeechPhraseAlternate.RecoResult
```

Parts

ISpeechPhraseAlternate
The owning object.

ISpeechRecoResult
Set: (This property is read-only)
Get: The alternate's parent ISpeechRecoResult object.

Example

Use of the ISpeechPhraseAlternate object is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseAlternate](#)

StartElementInResult Property

The **StartElementInResult** property specifies the starting phrase element of the alternate's parent ISpeechRecoResult object.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseAlternate*.**StartElementInResult**

Parts

ISpeechPhraseAlternate

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the starting element.

Example

Use of the ISpeechPhraseAlternate object is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseAlternate](#)

ISpeechPhraseAlternate Code Example

The following Visual Basic form code displays use of the ISpeechPhraseAlternate object. It demonstrates recognition and the EmulateRecognition method. In addition, it demonstrates the use of the SpeechDocs DLL created in the [Sample DLL](#) section. This DLL must be created before this code can be run.

To run this code, create a form with the following controls:

- A text box called Text1
- Two list boxes, called List1 and List2
- Two command buttons, called Command1 and Command2

Paste this code into the Declarations section of the form.

Because this code uses the SpeechDocs DLL, it is necessary to add a reference to "SpeechDocs" in the Project, References dialog.

The solitaire grammar is located in the default directory for a standard SAPI SDK installation. If the file is not there, use the path to an existing version. If no file is available, use the example code from [VB Application Sample: Command and Control Recognition](#) if needed.

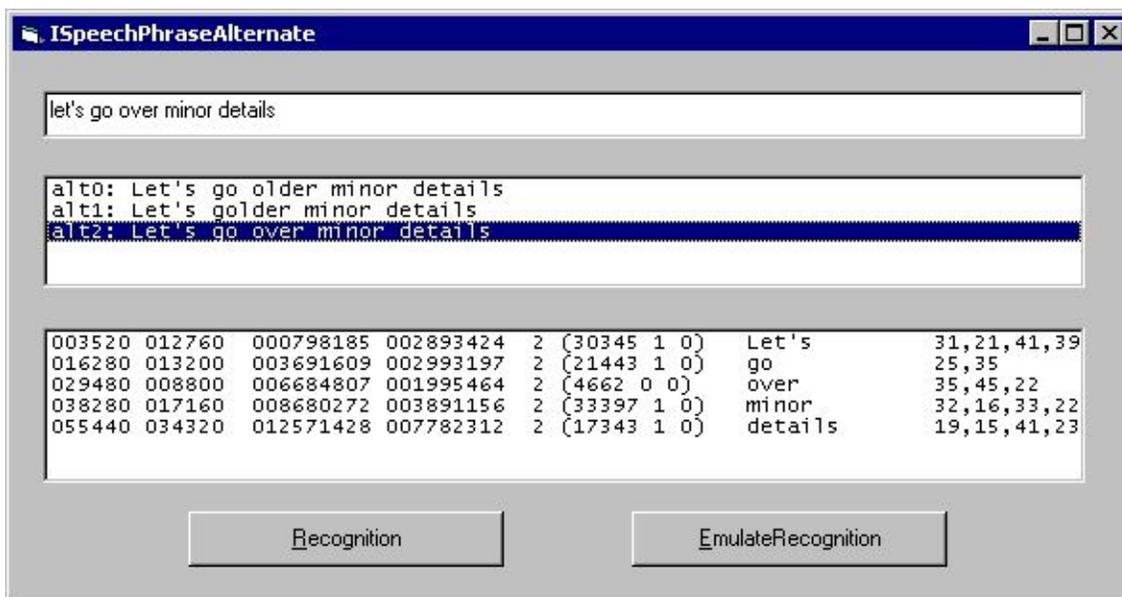
The Form_Load procedure creates a recognizer, a recognition context, and a grammar object. It loads the grammar object with sol.xml, the Solitaire grammar from the SAPI sample code, then activates the command and control (C and C) and dictation components of the grammar, and places a Solitaire command in the text box. Text matching the rules of the Solitaire grammar will produce better recognition results, but in order to demonstrate recognition alternates, it is necessary to enter phrases that do not satisfy grammar rules.

The command button, captioned Recognition, speaks text from

the text box into an audio file, and then performs speech recognition of that file. The command button captioned EmulateRecognition simply calls the EmulateRecognition method.

When alternates are available, the code displays three alternates in the top list box, and shows the phrase elements of one alternate in the lower list box. Click any alternate to display its phrase elements in the lower list box. Double click any alternate to perform the Commit method on that alternate. In the illustration below, the first alternate is incorrect; the second alternate is highlighted, and its phrase elements are displayed in the lower list box.

The data displayed in the lower list box is the same data displayed in the ISpeechPhraseElement [code example](#). Please see that page for full details.



Option Explicit

```
Dim D As SpeechDocs.Class1 'DLL created from Speech doc:
```

```
Const WAVEFILENAME = "C:\ISpeechPhraseElement.wav"
```

```
Private blnAlternatesExist As Boolean
```

```

Dim R As SpeechLib.SpInprocRecognizer
Dim G As SpeechLib.ISpeechRecoGrammar
Dim F As SpeechLib.SpFileStream
Dim V As SpeechLib.SpVoice

Dim PI As SpeechLib.ISpeechPhraseInfo
Dim A As SpeechLib.ISpeechPhraseAlternate
Dim AA As SpeechLib.ISpeechPhraseAlternates
Dim E As SpeechLib.ISpeechPhraseElement
Dim EE As SpeechLib.ISpeechPhraseElements

Dim WithEvents C As SpeechLib.SpInProcRecoContext

Private Sub Command1_Click()

    List1.Clear
    List2.Clear
    Screen.MousePointer = vbHourglass

    'Call this method in the DLL to speak into a file
    Set F = D.SpeakToFile(Text1.Text, WAVEFILENAME, "Microso

    'Set the file as recognizer's input stream
    F.Open WAVEFILENAME
    Set R.AudioInputStream = F

End Sub

Private Sub Command2_Click()

    List1.Clear
    List2.Clear
    Screen.MousePointer = vbHourglass
    C.Recognizer.EmulateRecognition Text1.Text

End Sub

Private Sub Form_Load()

    ' Create Recognizer, RecoContext, Grammar, and Voice
    Set R = New SpInprocRecognizer

```

```

Set C = R.CreateRecoContext
Set G = C.CreateGrammar(16)
Set V = New SpVoice

Set D = New SpeechDocs.Class1 'Create Class1 object from
                                ' Load Grammar with solitaire XML, set active
                                G.CmdLoadFromFile "C:\Program Files\Microsoft Speech SDK
                                G.CmdSetRuleIdState 0, SGDSActive 'Set C & C at
                                G.DictationSetState SGDSActive 'Set Dictati

Text1.Text = "let's go over minor details"
Command1.Caption = "&Recognition;"
Command2.Caption = "&EmulateRecognition;"

```

```
End Sub
```

```

Private Function PhonesToString(ByVal arrV As Variant) As String
    Dim ii As Integer, S As String
    If IsEmpty(arrV) Then
        PhonesToString = ""
    Else
        For ii = 0 To UBound(arrV)
            If Len(S) Then
                S = S & "," & arrV(ii)
            Else
                S = arrV(ii)
            End If
        Next ii
        PhonesToString = S
    End If
End Function

```

```

Private Sub C_Recognition(ByVal StreamNumber As Long, _
                        ByVal StreamPosition As Variant, _
                        ByVal RecognitionType As SpeechLib.Speech
                        ByVal Result As SpeechLib.ISpeechRecoRes

    Dim nn As Integer

    Set AA = Result.Alternates(3) 'May or may not be a
    If AA Is Nothing Then

```

```

'No alternates when speech matches a grammar rule
'No alternates when using EmulateRecognition
blnAlternatesExist = False
List1.AddItem Result.PhraseInfo.GetText
If Len(Result.PhraseInfo.Rule.Name) Then
    List1.AddItem " matches rule "" & Result.PhraseInfo.Rule.Name
End If
Set PI = Result.PhraseInfo
Else

    blnAlternatesExist = True
    nn = 0
    For Each A In AA
        List1.AddItem "alt" & Format(nn) & ": " & A.PhraseInfo.GetText
        nn = nn + 1
    Next
    Set PI = AA.Item(0).PhraseInfo
End If

List1.ListIndex = 0 'Highlight the selected alternate
Call DisplayPhraseElements(0)
Screen.MousePointer = vbDefault

End Sub

Private Sub C_EndStream(ByVal StreamNumber As Long, _
    ByVal StreamPosition As Variant, _
    ByVal StreamReleased As Boolean)

    'Recognition uses the Filestream, EmulateReco does not
    If ActiveControl.Caption = "&Recognition;" Then
        F.Close
        DoEvents
        F.Open WAVEFILENAME
        V.SpeakStream F
        F.Close
    End If
    Screen.MousePointer = vbDefault

End Sub

Private Sub DisplayPhraseElements(Index As Integer)

```

```

Dim X As String
Dim T As String
Dim A1 As Long, A2 As Long
Dim T1 As Long, T2 As Long
Dim C1 As Single, C2 As Integer, C3 As Integer

List2.Clear

'If no alternates, then there is only one PhraseInfo
If blnAlternatesExist = True Then
    Set PI = AA.Item(Index).PhraseInfo
End If

For Each E In PI.Elements

    'Audio data
    A1 = E.AudioStreamOffset
    A2 = E.AudioSizeBytes
    X = Format(A1, "000000") & " " & Format(A2, "000000")

    'Time data
    T1 = E.AudioTimeOffset
    T2 = E.AudioSizeTime
    X = X & Format(T1, "0000000000") & " " & Format(T2, "0000000000")

    'Display attributes
    X = X & Format(E.DisplayAttributes) & " "

    'Confidences
    C1 = E.EngineConfidence
    C2 = E.ActualConfidence
    C3 = E.RequiredConfidence
    T = "(" & Format(C1) & " " & Format(C2) & " " & Format(C3) & ")"
    X = X & Left(T & " ", 14)

    'Text and pronunciation
    X = X & Left(E.DisplayText & " ", 14)
    X = X & PhonesToString(E.Pronunciation)

    List2.AddItem X
Next

```

```
End Sub
```

```
Private Sub List1_Click()  
    Call DisplayPhraseElements(List1.ListIndex)  
End Sub
```

```
Private Sub List1_DblClick()  
    AA.Item(List1.ListIndex).Commit  
End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechPhraseAlternates

The **ISpeechPhraseAlternates** automation interface is a collection of [ISpeechPhraseAlternate](#) objects.

Automation Interface Elements

The ISpeechPhraseAlternates automation interface contains the following elements:

Properties	Description
Count Property	Returns the number of objects in the collection.
Methods	Description
Item Method	Returns a member of the collection by its index.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseAlternates](#)

Count Property

The **Count** property returns the number of `ISpeechPhraseAlternate` objects in the `ISpeechPhraseAlternates` object.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseAlternates*.**Count**

Parts

ISpeechPhraseAlternates

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the Count property.

Remarks

The `SpeechPhraseAlternates` object is a collection of `ISpeechPhraseAlternate` objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The `Count` property is one of these common properties.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechPhraseAlternates](#)

Item Method

The **Item** method returns a member of the ISpeechPhraseAlternates collection by its index.

```
ISpeechPhraseAlternates.Item(  
    Index As Long  
) As ISpeechPhraseAlternate
```

Parameters

Index
Specifies the Index.

Return Value

The Item method returns an ISpeechPhraseAlternate variable.

Remarks

The ISpeechPhraseAlternates object is a collection of ISpeechPhraseAlternate objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Item method is one of these common methods.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechPhraseElement

The **ISpeechPhraseElement** automation interface provides access to information about a word or phrase.

Use of the ISpeechPhraseElement is demonstrated in a [code example](#) at the end of this section.

Automation Interface Elements

The ISpeechPhraseElement automation interface contains the following elements:

Properties	Description
ActualConfidence Property	Returns the actual confidence for the phrase element.
AudioSizeBytes Property	Returns the size, in bytes, of the audio for this element.
AudioSizeTime Property	Returns the length of the element in 100-nanosecond units.
AudioStreamOffset Property	Returns the starting offset of the element in bytes relative to the start of the phrase in the original input stream.
AudioTimeOffset Property	Returns the starting offset of the element in 100-nanosecond units relative to the start of the phrase.
DisplayAttributes Property	Returns a set of SpeechDisplayAttributes constants defining information about the display of this word.
DisplayText Property	Returns the display text for the element.

<u>EngineConfidence</u> Property	Returns the confidence score computed by the SR engine.
<u>LexicalForm</u> Property	Returns the lexical form of the element.
<u>Pronunciation</u> Property	Returns the pronunciation of the element as phonemes.
<u>RequiredConfidence</u> Property	Returns the required confidence for this element.
<u>RetainedSizeBytes</u> Property	Returns the size, in bytes, of the element in the retained audio stream.
<u>RetainedStreamOffset</u> Property	Returns the starting offset of the element in bytes relative to the start of the phrase in the retained audio stream.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

ActualConfidence Property

The **ActualConfidence** property returns the actual confidence for the phrase element.

This value consists of a SpeechEngineConfidence constant, and it is always at least as high as the [ISpeechPhraseElement.RequiredConfidence](#) property. The confidence rating is an enumerated value of type [SpeechEngineConfidence](#). See [Confidence Scoring and Rejection in SAPI Speech Recognition Engine Guide](#) for additional details.

Syntax

```
Set: (This property is read-only)
Get: SpeechEngineConfidence =
     ISpeechPhraseElement.ActualConfidence
```

Parts

ISpeechPhraseElement
The owning object.

SpeechEngineConfidence
Set: (This property is read-only)
Get: A SpeechEngineConfidence constant that gets the value of the property.

Example

Use of the *ISpeechPhraseElement* is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

AudioSizeBytes Property

The **AudioSizeBytes** property returns the size, in bytes, of the audio for this element.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseElement*.**AudioSizeBytes**

Parts

ISpeechPhraseElement

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the value of the property.

Example

Use of the *ISpeechPhraseElement* is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

AudioSizeTime Property

The **AudioSizeTime** property returns the length of the element in 100-nanosecond units.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseElement*.**AudioSizeTime**

Parts

ISpeechPhraseElement

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the value of the property.

Example

Use of the *ISpeechPhraseElement* is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

AudioStreamOffset Property

The **AudioStreamOffset** property returns the starting offset of the element, in bytes, relative to the start of the phrase in the original input stream.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseElement*.**AudioStreamOffset**

Parts

ISpeechPhraseElement

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the value of the property.

Example

Use of the *ISpeechPhraseElement* is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

AudioTimeOffset Property

The **AudioTimeOffset** property returns the starting offset of the element in 100-nanosecond units relative to the start of the phrase.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseElement*.**AudioTimeOffset**

Parts

ISpeechPhraseElement

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the value of the property.

Example

Use of the *ISpeechPhraseElement* is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

DisplayAttributes Property

The **DisplayAttributes** property returns a set of SpeechDisplayAttributes constants defining information about the display of this word.

Syntax

```
Set: (This property is read-only)
Get: SpeechDisplayAttributes =
     ISpeechPhraseElement.DisplayAttributes
```

Parts

ISpeechPhraseElement
The owning object.

SpeechDisplayAttributes
Set: (This property is read-only)
Get: One or more SpeechDisplayAttributes constants that gets the value of the property.

Example

Use of the *ISpeechPhraseElement* is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

DisplayText Property

The **DisplayText** property returns the display text for the element with normalization of numbers, currency values, and ordinals (for example, displays "\$2" for the spoken words "two dollars").

The [LexicalForm](#) property displays text without normalization of numbers, currency values, and ordinals.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechPhraseElement*.**DisplayText**

Parts

ISpeechPhraseElement
The owning object.

String
Set: (This property is read-only)
Get: A String variable that gets the value of the property.

Example

Use of the *ISpeechPhraseElement* is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

EngineConfidence Property

The **EngineConfidence** property returns the confidence score computed by the speech recognition (SR) engine.

Syntax

Set: (This property is read-only)

Get: *Single* = *ISpeechPhraseElement*.**EngineConfidence**

Parts

ISpeechPhraseElement

The owning object.

Single

Set: (This property is read-only)

Get: A *Single* variable that gets the value of the property.

Remarks

The value range is engine dependent. It can be used to optimize an application's performance with a specific engine. Using this value will improve the application with a particular speech engine, but it is likely to make it worse with other engines and should be used with care. This value is more useful with speaker-independent engines because it allows a large corpus of recorded usage to correctly optimize the overall accuracy of the application.

Example

Use of the *ISpeechPhraseElement* is demonstrated in a [code](#)

[example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

LexicalForm Property

The **LexicalForm** property returns the lexical form of the element without normalization of numbers, currency values, and ordinals (for example, displays "two dollars" for the spoken words "two dollars").

The [DisplayText](#) property displays text with normalization of numbers, currency values, and ordinals.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechPhraseElement*.**LexicalForm**

Parts

ISpeechPhraseElement

The owning object.

String

Set: (This property is read-only)

Get: A String variable that gets the value of the property.

Example

Use of the *ISpeechPhraseElement* is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

Pronunciation Property

The **Pronunciation** property returns the pronunciation of the element as phonemes.

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechPhraseElement*.**Pronunciation**

Parts

ISpeechPhraseElement

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant variable that gets the value of the property.

Example

Use of the *ISpeechPhraseElement* is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

RequiredConfidence Property

The **RequiredConfidence** property returns the required confidence for this element.

This value is an enumerated value of type [SpeechEngineConfidence](#). See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details. Required confidence values are specified in grammars. Preceding a word with '-' sets its RequiredConfidence as low (SECLowConfidence); preceding a word with '+' sets its RequiredConfidence as high (SECHighConfidence)

Syntax

Set: (This property is read-only)

Get: [SpeechEngineConfidence](#) =
ISpeechPhraseElement.**RequiredConfidence**

Parts

ISpeechPhraseElement
The owning object.

SpeechEngineConfidence

Set: (This property is read-only)

Get: A SpeechEngineConfidence variable that gets the value of the property.

Example

Use of the ISpeechPhraseElement is demonstrated in a [code](#)

[example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

RetainedSizeBytes Property

The **RetainedSizeBytes** property returns the size, in bytes, of the element in the retained audio stream.

If the current RecoContext is retaining audio, the value of RetainedSizeBytes will be equal to the value of [AudioSizeBytes](#). If the current RecoContext is not retaining audio, RetainedSizeBytes will be zero.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseElement*.**RetainedSizeBytes**

Parts

ISpeechPhraseElement

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the value of the property.

Example

Use of the ISpeechPhraseElement is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

RetainedStreamOffset Property

The **RetainedStreamOffset** property returns the starting offset of the element, in bytes, relative to the start of the phrase in the retained audio stream.

If the current RecoContext is retaining audio, the value of RetainedStreamOffset will be equal to the value of [AudioStreamOffset](#). If the current RecoContext is not retaining audio, RetainedStreamOffset will be zero.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseElement*.**RetainedStreamOffset**

Parts

ISpeechPhraseElement

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the value of the property.

Example

Use of the *ISpeechPhraseElement* is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseElement](#)

ISpeechPhraseElement Code Example

The following Visual Basic form code displays the properties of the ISpeechPhraseElement object. The ISpeechPhraseElement object is contained in the [ISpeechRecoResult](#) object returned by a RecoContext's [Recognition](#) event. This code shows two ways to create a recognition result object:

- Perform recognition directly
- Emulate recognition using the [EmulateRecognition](#) method

The example also demonstrates the use of the [SpFileStream](#) object in conjunction with the [AudioOutputStream](#) of the voice and the [AudioInputStream](#) of the recognizer.

To run this code, create a form with the following controls:

- Two command buttons called Command1 and Command2
- A list box called List1
- A text box called Text1

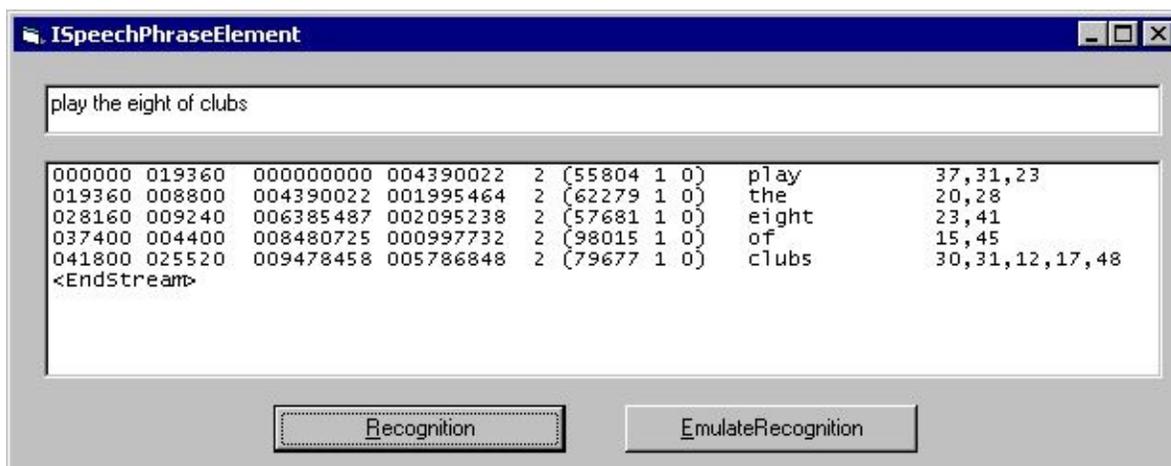
Paste this code into the Declarations section of the form.

The code will set the command button captions as shown in the illustration.

The Form_Load procedure creates a recognizer, a recognition context, and a grammar object. It loads the grammar object with sol.xml, the Solitaire grammar from the SAPI sample code. It then activates the command and control (C and C) and dictation components of the grammar, and places a Solitaire command in the text box. Users can enter whatever text they like in the text box, but best recognition results will be obtained from phrases matching the rules in the C and C grammar; for example, sentences such as "Move the black ten to the jack of diamonds," or "Play the red queen."

The command button captioned Recognition speaks text from the text box into an audio file, and then performs speech recognition of that file. The command button captioned EmulateRecognition simply calls the EmulateRecognition method.

When the speech recognition (SR) engine has completed recognition, it generates a Recognition Event that returns an ISpeechRecoResult object. The Recognition procedure instantiates each ISpeechPhraseElement in the result object's Elements property (a collection of ISpeechPhraseElement objects), and displays selected phrase element properties in columns in the list box.



AudioStreamOffset and AudioSizeBytes

The first two columns show the [AudioStreamOffset](#) and [AudioSizeBytes](#) properties. These two properties indicate the boundaries of an element in the input audio stream.

AudioStreamOffset points to the beginning of the element and AudioSizeBytes is the element's length. The sum of an element's AudioStreamOffset and AudioSizeBytes is the same as the AudioStreamOffset of the next element.

In an ISpeechPhraseElement object created by the EmulateRecognition method, the AudioStreamOffset and AudioSizeBytes properties are zero.

AudioTimeOffset and AudioSizeTime

The next two columns are the [AudioTimeOffset](#) and [AudioSizeTime](#) properties, which delimit the phrase elements in 100-nanosecond units of time. AudioTimeOffset indicates the beginning time of the element, and AudioSizeTime is its time length. The sum of an element's AudioTimeOffset and AudioSizeTime is approximately the same as the AudioTimeOffset of the next element.

In an ISpeechPhraseElement object created by the EmulateRecognition method, the AudioTimeOffset and AudioSizeTime properties are zero.

DisplayAttributes

The next column is the [DisplayAttributes](#) property, which defines how the text of the element is displayed relative to text from other phrase elements. The [SpeechDisplayAttributes](#) enumeration lists the possible values of this property. All elements in the example above have a DisplayAttributes property of two, which is the value of the SDA_One_Trailing_Space constant, indicating that the element should be displayed with a trailing space.

EngineConfidence, ActualConfidence, and RequiredConfidence

The three numbers in parentheses are the three property values involving confidence in the recognition of the phrase element. The first of these is the [EngineConfidence](#) property, which represents the SR engine's level of confidence in the recognition. The [ActualConfidence](#) property reduces the EngineConfidence to one of three confidence levels: low, normal or high. The [RequiredConfidence](#) property specifies the confidence level that the ActualConfidence property must equal or surpass.

In an ISpeechPhraseElement object created by the EmulateRecognition method, the EngineConfidence and

RequiredConfidence properties are zero. If the emulated phrase matches C and C rules, the ActualConfidence property is one; otherwise it is zero.

DisplayText and LexicalForm

The [DisplayText](#) property is the next column in the example. It consists of the recognized text of the phrase element, with normalization of numbers, ordinals, and currency values. The [LexicalForm](#) property, not shown in this example, returns the same text, but without normalization.

Pronunciation

To the right of the DisplayText is data from the [Pronunciation](#) property. Each number represents a phoneme, and the phonemes represent the pronunciation of the phrase element.

In an ISpeechPhraseElement object created by the EmulateRecognition method, the Pronunciation property is Empty.

RetainedStreamOffset and RetainedSizeBytes

The [RetainedStreamOffset](#) and [RetainedSizeBytes](#) properties are not shown in this example. If the current recognition context is retaining audio data, then RetainedStreamOffset and RetainedSizeBytes are the same as AudioStreamOffset and AudioSizeBytes, respectively; otherwise, both properties are zero.

In an ISpeechPhraseElement object created by the EmulateRecognition method, the RetainedStreamOffset and RetainedSizeBytes properties are zero.

```
Option Explicit
```

```
Const WAVEFILENAME = "C:\ISpeechPhraseElement.wav"
```

```
Dim R As SpeechLib.SpInprocRecognizer
```

```

Dim G As SpeechLib.ISpeechRecoGrammar
Dim F As SpeechLib.SpFileStream
Dim E As SpeechLib.ISpeechPhraseElement
Dim V As SpeechLib.SpVoice

Dim WithEvents C As SpeechLib.SpInProcRecoContext

Private Sub Command1_Click()

    List1.Clear
    Call SpeakToFile(Text1.Text, WAVEFILENAME)
    F.Open WAVEFILENAME
    Set R.AudioInputStream = F

End Sub

Private Sub Command2_Click()

    List1.Clear
    C.Recognizer.EmulateRecognition Text1.Text

End Sub

Private Sub Form_Load()

    ' Create Recognizer, RecoContext, Grammar, and Voice
    Set R = New SpInProcRecognizer
    Set C = R.CreateRecoContext
    Set G = C.CreateGrammar(16)
    Set V = New SpVoice
    Set V.Voice = V.GetVoices("gender=male").Item(0)

    ' Load Grammar with solitaire XML, set active
    G.CmdLoadFromFile "c:\sol.xml", SLOStatic
    G.CmdSetRuleIdState 0, SGDSActive           'Set C &
    G.DictationSetState SGDSActive             'Set Dic

    Text1.Text = "play the eight of clubs"
    Command1.Caption = "&Recognition;"
    Command2.Caption = "&EmulateRecognition;"

End Sub

```

```

Private Sub SpeakToFile(ByVal strText As String, ByVal strFN:
    Set F = New SpFileStream 'Create stre
    F.Open strFName, SSFMCreatForWrite, True 'Open as the
    Set V.AudioOutputStream = F 'Set voice o
    V.Speak strText, SVSFisXML 'Speak synch
    F.Close 'Close file
End Sub

```

```

Private Function PhonesToString(ByVal arrV As Variant) As St
    Dim ii As Integer, S As String
    If IsEmpty(arrV) Then
        PhonesToString = ""
    Else
        For ii = 0 To UBound(arrV)
            If Len(S) Then
                S = S & "," & arrV(ii)
            Else
                S = arrV(ii)
            End If
        Next ii
        PhonesToString = S
    End If
End Function

```

```

Private Sub C_Recognition(ByVal StreamNumber As Long, _
    ByVal StreamPosition As Variant, _
    ByVal RecognitionType As SpeechLib.Speech
    ByVal Result As SpeechLib.ISpeechRecoRes

    Dim X As String
    Dim T As String
    Dim A1 As Long, A2 As Long
    Dim T1 As Long, T2 As Long
    Dim C1 As Single, C2 As Integer, C3 As Integer

    For Each E In Result.PhraseInfo.Elements

        'Audio data
        A1 = E.AudioStreamOffset
        A2 = E.AudioSizeBytes
        X = Format(A1, "000000") & " " & Format(A2, "000000"

```


Microsoft Speech SDK

Speech Automation 5.1



ISpeechPhraseElements

The **ISpeechPhraseElements** automation interface represents a collection of [ISpeechPhraseElement](#) objects.

Automation Interface Elements

The ISpeechPhraseElements automation interface contains the following elements:

Properties	Description
Count Property	Returns the number of objects in the collection.
Methods	Description
Item Method	Returns a member of the collection by its index.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechPhraseElements](#)

Count Property

The **Count** property returns the number of ISpeechPhraseElement objects in the ISpeechPhraseElements object.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseElements*.**Count**

Parts

ISpeechPhraseElements

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the count.

Remarks

The ISpeechPhraseElements object is a collection of ISpeechPhraseElement objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Count property is one of these common properties.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK



Speech Automation 5.1

Object: **[ISpeechPhraseElements](#)**

Item Method

The **Item** method returns a member of the ISpeechPhraseElements collection by its index.

```
ISpeechPhraseElements.Item(  
    Index As Long  
) As ISpeechPhraseElement
```

Parameters

Index
Specifies the Index.

Return Value

The Item method returns an ISpeechPhraseElement variable.

Remarks

The ISpeechPhraseElements object is a collection of ISpeechPhraseElement objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Item method is one of these common methods.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechPhraseInfo

The **ISpeechPhraseInfo** automation interface contains properties detailing phrase elements. This includes information about the audio streams, start and stop times for the audio, and elements contained in the phrase.

Automation Interface Elements

The ISpeechPhraseInfo automation interface contains the following elements:

Properties	Description
<u>AudioSizeBytes</u> Property	Returns the size of audio data in bytes for this phrase.
<u>AudioSizeTime</u> Property	Returns the length of phrase's audio in 100-nanosecond units.
<u>AudioStreamPosition</u> Property	Returns the start time in the audio stream for the phrase.
<u>Elements</u> Property	Returns information about the elements of the phrase.
<u>EngineId</u> Property	Returns a string containing the GUID of the engine recognizing this phrase.
<u>EnginePrivateData</u> Property	Returns the private data of the engine.
<u>GrammarId</u> Property	Returns the ID of the grammar that contains the top-level rule used to recognize the phrase.
<u>LanguageId</u> Property	Returns the language ID for the phrase elements.
<u>Properties</u> Property	Returns the root property for the result.

Replacements Property	Returns possible text replacements.
RetainedSizeBytes Property	Returns the size in bytes of the retained audio data for the audio format specified.
Rule Property	Retrieves information about the top-level rule that was used to recognize the phrase.
StartTime Property	Returns the start time for start of phrase audio in absolute time.

Methods	Description
GetDisplayAttributes Method	Returns the display attribute for the text.
GetText Method	Returns the text from a recognition as a single string.
SaveToMemory Method	Saves the entire recognition result to memory.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

AudioSizeBytes Property

The **AudioSizeBytes** property returns the size of audio data in bytes for this phrase.

AudioSizeBytes is directly tied to the input audio format. AudioSizeBytes increases as the quality of the format increases (and in doing so, requires more bytes). However, it is possible that the application needs the retained audio in a format different from the input audio stream (perhaps for lower-quality persistence, for example). If an application scales the audio, RetainedAudioSizeBytes will change, but not AudioSizeBytes.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseInfo*.**AudioSizeBytes**

Parts

ISpeechPhraseInfo

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable containing the size of audio data.

Example

The following code snippet assumes a valid recognition, *RecoResult*, although it still checks for validity. AudioSizeBytes displays in a message box. A short phrase such as "Now is the

time" might be 56,000 bytes.

```
If Not RecoResult Is Nothing Then
    Dim rp As ISpeechPhraseInfo
    Set rp = RecoResult.PhraseInfo

    MsgBox "AudioTime: " & rp.AudioSizeBytes
End If
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

AudioSizeTime Property

The **AudioSizeTime** property returns the phrase audio length in 100-nanosecond units.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseInfo*.**AudioSizeTime**

Parts

ISpeechPhraseInfo

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable containing the phrase audio length.

Example

The following code snippet assumes a valid recognition, *RecoResult*, although it still checks for validity. `AudioSizeTime` displays in a message box. A short phrase such as "Now is the time" might be 26,000,000 100-nanosecond units, or about 2.6 seconds.

```
If Not RecoResult Is Nothing Then
    Dim rp As ISpeechPhraseInfo
    Set rp = RecoResult.PhraseInfo

    MsgBox "AudioTime: " & rp.AudioSizeTime
End If
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

AudioStreamPosition Property

The **AudioStreamPosition** property returns the start time in the audio stream for the phrase.

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechPhraseInfo*.**AudioStreamPosition**

Parts

ISpeechPhraseInfo

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant variable that gets the property.

Example

The following code snippet assumes a valid recognition, *RecoResult*, although it still checks for validity. *AudioStreamPosition* displays in a message box. A short delay in speaking may yield an *AudioStreamPosition* of 10,000, for example.

```
If Not RecoResult Is Nothing Then
    Dim rp As ISpeechPhraseInfo
    Set rp = RecoResult.PhraseInfo

    MsgBox "AudioTime: " & rp.AudioStreamPosition
End If
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

Elements Property

The **Elements** property returns information about the elements of the phrase.

Syntax

Set: (This property is read-only)

Get: [ISpeechPhraseElements](#) = *ISpeechPhraseInfo*.**Elements**

Parts

ISpeechPhraseInfo

The owning object.

ISpeechPhraseElements

Set: (This property is read-only)

Get: An *ISpeechPhraseElements* variable containing the phrase elements.

Example

The following code snippet assumes a valid recognition, *RecoResult*, although it still checks for validity. This prepares a string containing the number of elements from the recognized phrase and lists each word. Words are added in both the [DisplayText](#) and [LexicalForm](#).

```
Dim res As String
Dim i As Long
```

```
res = "Phrase Elements: Count=" & RecoResult.Elements.Count & "
For i = 0 To RecoResult.Elements.Count - 1
```

```
res = res & RecoResult.Elements(i).AudioStreamOffset  
" Text:" & RecoResult.Elements(i).DisplayText & " Le:
```

Next

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

EngineId Property

The **EngineId** property returns a string containing the GUID of the engine recognizing this phrase.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechPhraseInfo*.**EngineId**

Parts

ISpeechPhraseInfo

The owning object.

String

Set: (This property is read-only)

Get: A String variable that gets the property.

Example

The following code snippet assumes a valid recognition *RecoResult*, although it still checks for validity. The GUID of the speech recognition engine processing this recognition is returned in the String *srEngineID*.

```
Dim srEngineID As String
```

```
srEngineID = RecoResult.EngineId()
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

EnginePrivateData Property

The **EnginePrivateData** property returns the private data of the engine.

Private engine data represents a proprietary engine that returns non-standard information with the recognition. The format of the data is defined by the manufacturer. Not every recognition will contain private data. If there is no private data, the return value will be empty or NULL.

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechPhraseInfo*.**EnginePrivateData**

Parts

ISpeechPhraseInfo

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant variable containing the private data returned by the engine.

Example

The following code snippet assumes a valid recognition *RecoResult*, although it still checks for validity. A String is used as the return value but check the manufacturer's documentation for additional information.

```
Dim privateData As String
```

```
privateData = RecoResult.EnginePrivateData()
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechPhraseInfo](#)

GetDisplayAttributes Method

The **GetDisplayAttributes** method returns the display attribute for the text.

The display attribute is the padding of white spaces before or after each word as determined by the engine. The speech recognition engine determines this value for the language used. Western scripts commonly use spaces between words, although eastern languages may not. Default is [SDA_One_Trailing_Space](#).

```
ISpeechPhraseInfo.GetDisplayAttributes(  
    [StartElement As Long = 0],  
    [Elements As Long = -1],  
    [UseReplacements As Boolean = True]  
) As SpeechDisplayAttributes
```

Parameters

StartElement

[Optional] Specifies the word position from which to start. If omitted, the first word is used.

Elements

[Optional] Specifies the number of words retrieve to determine spacing. Default value is -1 indicating all words are retrieved.

UseReplacements

[Optional] Indicates if replacement text should be used. An example of a text replacement is speaking the sentence "write new check for twenty dollars." The retrieved

replacement text is "write new check for \$20." Default value is True. For more information on replacements, see the [SR Engine White Paper](#).

Return Value

The `GetDisplayAttributes` method returns a `SpeechDisplayAttributes` variable.

Example

In the following example, a successful recognition occurs. The variable *theAttributes* contains the white space attributes for the entire string. The attributes of a portion of the string could be chosen as shown in the subsequent example call. The code is inside a recognition event.

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal  
  
    Set RecoResult = Result  
    Dim theAttributes as SpeechDisplayAttributes  
  
    theAttributes = RecoResult.PhraseInfo.GetDisplayAttr.  
    theAttributes = RecoResult.PhraseInfo.GetDisplayAttr.  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechPhraseInfo](#)

GetText Method

The **GetText** method returns the text from a recognition as a single string.

```
ISpeechPhraseInfo.GetText(  
    [StartElement As Long = 0],  
    [Elements As Long = -1],  
    [UseReplacements As Boolean = True]  
) As String
```

Parameters

StartElement

[Optional] Specifies the word position from which to start. If omitted, the first word is used.

Elements

[Optional] Specifies the number of words to retrieve. Default value is -1 indicating all words are retrieved.

UseReplacements

[Optional] Indicates if replacement text should be used. An example of a text replacement is speaking the sentence "write new check for twenty dollars." The retrieved replacement text is "write new check for \$20." Default value is True. See the [SR Engine White Paper](#) for more information on replacements.

Return Value

The GetText method returns a String containing the words in the

phrase.

Remarks

GetText may be used only after a recognition attempt, whether successful ([SRERecognition](#)), or unsuccessful ([SREFalseRecognition](#)).

Example

In the following example, a successful recognition occurs and displays in a text box. The code is inside a recognition event.

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal  
    Set RecoResult = Result  
    TextBox1.Text = RecoResult.PhraseInfo.GetText  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

GrammarId Property

The **GrammarId** property returns the ID of the grammar that contains the top-level rule used to recognize the phrase.

The grammar ID is set with [ISpeechRecoContext.CreateGrammar](#).

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechPhraseInfo*.**GrammarId**

Parts

ISpeechPhraseInfo
The owning object.

Variant
Set: (This property is read-only)
Get: A Variant variable that gets the property.

Example

The following code snippet assumes a valid recognition *RecoResult*, although it still checks for validity. *GrammarId* is the file containing the current match, in this case "10."

```
Public g As ISpeechRecoGrammar  
Set g = RecoResult.CreateGrammar(10)
```

```
'Speech processing code here
```

```
If Not RecoResult Is Nothing Then  
    Dim rp As ISpeechPhraseInfo
```

```
    Set rp = RecoResult.PhraseInfo
    MsgBox "Grammar ID: " & rp.GrammarId
End If
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

LanguageId Property

The **LanguageId** property returns the language ID for the phrase elements.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseInfo*.**LanguageId**

Parts

ISpeechPhraseInfo

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the property.

Remarks

This is the same as the Win32 Language Identifier (LANGID).

Example

The following code snippet assumes a valid recognition, *RecoResult*, although it still checks for validity. Standard English has a LANGID decimal value of 1033.

```
If Not RecoResult Is Nothing Then
    Dim rp As ISpeechPhraseInfo
    Set rp = RecoResult.PhraseInfo
    MsgBox "LANGID: " & rp.LanguageId
```

End If

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

Properties Property

The **Properties** property returns the root property for the result.

Syntax

Set: (This property is read-only)

Get: [ISpeechPhraseProperties](#) = *ISpeechPhraseInfo*.**Properties**

Parts

ISpeechPhraseInfo

The owning object.

ISpeechPhraseProperties

Set: (This property is read-only)

Get: An ISpeechPhraseProperties interface for the properties.

Remarks

If the recognition result does not have an associated property, the Property will have a value of Nothing.

Example

The following code snippet assumes a valid recognition *RecoResult*, although it still checks for validity. The sample displays a message box for each Name and Value of available properties.

```
Dim rp As ISpeechPhraseInfo
```

```
Set rp = RecoResult.PhraseInfo

Dim i As Long
If Not rp.Properties Is Nothing Then
  For i = 0 To rp.Properties.Count
    MsgBox Result.PhraseInfo.Properties.Item(i).Name
    MsgBox Result.PhraseInfo.Properties.Item(i).Value
  Next i
End If
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

Replacements Property

The **Replacements** property returns possible text replacements.

Syntax

```
Set: (This property is read-only)  
Get: ISpeechPhraseReplacements =  
     ISpeechPhraseInfo.Replacements
```

Parts

ISpeechPhraseInfo
The owning object.

ISpeechPhraseReplacements
Set: (This property is read-only)
Get: An *ISpeechPhraseReplacements* interface for the replacements.

Example

The following code snippet assumes a valid recognition *RecoResult*, although it still checks for validity. The sample displays a message box for each item in the replacement list.

```
Dim pi As ISpeechPhraseInfo  
Dim rep As ISpeechPhraseReplacement  
  
Set pi = RecoResult.PhraseInfo  
  
For Each rep In pi.Replacements.Count
```

MsgBox rep.Text
Next

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

RetainedSizeBytes Property

The **RetainedSizeBytes** property returns the size in bytes of the retained audio data for the audio format specified.

It is possible that the application needs the retained audio in a format different from the input audio stream; therefore, `RetainedAudioSizeBytes` and `AudioSizeBytes` could be different as well. In this case, the scaled (or converted) retained audio will be `RetainedAudioSizeBytes` in length and `AudioSizeBytes` will be the size of the original stream length. `AudioSizeBytes` changes only if the quality of the original format of the stream changes, and in doing so, requiring more or fewer bytes.

Syntax

Set: (This property is read-only)

Get: `Long = ISpeechPhraseInfo.RetainedSizeBytes`

Parts

ISpeechPhraseInfo

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the property.

Example

The following code snippet assumes a valid recognition *RecoResult*, although it still checks for validity. In the sample

below, RetainedSizeBytes will be the same as AudioSizeBytes since no scaling takes place.

```
If Not RecoResult Is Nothing Then
    Dim rp As ISpeechPhraseInfo
    Set rp = RecoResult.PhraseInfo
    MsgBox "RetainedSizeBytes: " & rp.RetainedSizeBytes
End If
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

Rule Property

The **Rule** property retrieves information about the top-level rule that was used to recognize the phrase.

Syntax

Set: (This property is read-only)

Get: [ISpeechPhraseRule](#) = *ISpeechPhraseInfo*.**Rule**

Parts

ISpeechPhraseInfo

The owning object.

ISpeechPhraseRule

Set: (This property is read-only)

Get: An *ISpeechPhraseRule* variable that gets the property.

Example

The following code snippet assumes a valid recognition, *RecoResult*, although it still checks for validity. In the sample below, a few of the rule properties are retrieved.

```
If Not RecoResult Is Nothing Then
    Dim rp As ISpeechPhraseInfo
    Set rp = RecoResult.PhraseInfo

    Dim ruleName As String
    ruleName = rp.rule.Name

    Dim ruleID, firstElement, numberOfElements As Long
```

```
ruleID = rp.rule.Id  
firstElement = rp.rule.FirstElement  
numberOfElements = rp.rule.NumberOfElements  
End If
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechPhraseInfo](#)

SaveToMemory Method

The **SaveToMemory** method saves the phrase information from a recognition result to memory.

The phrase may be recalled at a later time. To retrieve the phrase information from memory use [SpPhraseInfoBuilder.RestorePhraseFromMemory](#).

ISpeechPhraseInfo.SaveToMemory() As Variant

Parameters

None

Return Value

The SaveToMemory method returns a Variant containing a pointer to saved phrase.

Example

The following example demonstrates storing and retrieving the phrase portion of a recognition result. An example of late binding for creating the PhraseBuilder object is also demonstrated.

The sample assumes a valid *RecoResult*.

```
'Save the phrase first
Dim thePhrase As Variant
thePhrase = RecoResult.PhraseInfo.SaveToMemory

'Retrieve the phrase
Dim PhraseBuilder As Object
Set PhraseBuilder = CreateObject("SAPI.SpPhraseInfoBuilder")
```

```
Dim PhraseInfo As ISpeechPhraseInfo
Set PhraseInfo = PhraseBuilder.RestorePhraseFromMemory(thePh
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseInfo](#)

StartTime Property

The **StartTime** property returns the start time for the start of phrase audio in absolute time.

The time is expressed in Universal Coordinated Time.

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechPhraseInfo*.**StartTime**

Parts

ISpeechPhraseInfo

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant variable that gets the property.

Example

The following code snippet assumes a valid recognition, *RecoResult*, although it still checks for validity.

```
If Not RecoResult Is Nothing Then
    Dim rp As ISpeechPhraseInfo
    Set rp = RecoResult.PhraseInfo
    MsgBox "StartTime: " & rp.StartTime
End If
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechPhraseProperties

The **ISpeechPhraseProperties** automation interface represents a collection of [ISpeechPhraseProperty](#) objects.

Automation Interface Elements

The ISpeechPhraseProperties automation interface contains the following elements:

Properties	Description
Count Property	Returns the number of objects in the collection.

Methods	Description
Item Method	Returns a member of the collection by its index.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseProperties](#)

Count Property

The **Count** property returns the number of `ISpeechPhraseProperty` objects in the `ISpeechPhraseProperties` object.

Syntax

Set: (This property is read-only)

Get: `Long = ISpeechPhraseProperties.Count`

Parts

ISpeechPhraseProperties

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the Count property.

Remarks

The `ISpeechPhraseProperties` object is a collection of `ISpeechPhraseProperty` objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The `Count` property is one of these common properties.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseProperties](#)

Item Method

The **Item** method returns a member of the ISpeechPhraseProperties collection by its index.

```
ISpeechPhraseProperties.Item(  
    Index As Long  
) As ISpeechPhraseProperty
```

Parameters

Index
Specifies the Index.

Return Value

The Item method returns an ISpeechPhraseProperty variable.

Remarks

The ISpeechPhraseProperties object is a collection of ISpeechPhraseProperty objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Item method is one of these common methods.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechPhraseProperty

The **ISpeechPhraseProperty** automation interface stores the information for a semantic property.

Automation Interface Elements

The ISpeechPhraseProperty automation interface contains the following elements:

Properties	Description
Children Property	Returns a collection of the property's child objects.
Confidence Property	Returns the confidence value for this semantic property computed by SAPI or the speech recognition engine.
EngineConfidence Property	Returns the confidence value for this semantic property computed by the speech recognition (SR) engine.
FirstElement Property	Returns the offset of the first spoken element spanned by this property.
Id Property	Returns the ID of the semantic property.
Name Property	Returns the name of the semantic property.
NumberOfElements Property	Returns the number of spoken elements spanned by this property.
Parent Property	Specifies the parent of the semantic property.
Value Property	Returns the value of the semantic property.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseProperty](#)

Children Property

The **Children** property returns a collection of the property's child objects.

A child is a rule within a rule. That is, within a command and control grammar, rules are explicitly defined with the Rule tag. To allow greater flexibility, it is possible to allow another rule to be used within one by declaring it with the RuleRef tag. In this case, the second rule referenced by the RuleRef tag could be a child. Child rules are defined with the RULE tag, and referenced with the RULEREf tag. Properties can be added to rulerefs and phrases within the rule with the PROPNAME attribute. If a child rule is present, the original rule containing the child is the parent. In this way, a rule can contain several levels of children and child rules can have several levels of parents.

An example of this is sol.xml for the card game solitaire. A user may play a card such as "move the red five on the black seven." Examine sol.xml; the rule is set up so that both the color of the card as well as the rank are referenced within the MoveCard rule. In this case, a successful recognition would have two top parent nodes: From and To. In turn, each parent node would have two children: color and rank. This way, an application could sort through the rules and know exactly which card moves (the From node) and which card receives it (the To node). The rule name would still be MoveCard.

In contrast, the more simple command "play the red ace," has only two nodes (color and rank). Neither node is considered to be either parent or child since the rule PlayCard has no PROPNAME tag. The nodes are peers of each other.

Syntax

Set: (This property is read-only)

```
Get: ISpeechPhraseProperties =  
    ISpeechPhraseProperty.Children
```

Parts

ISpeechPhraseProperty
The owning object.

ISpeechPhraseProperties
Set: (This property is read-only)
Get: An *ISpeechPhraseProperties* object returning the value of the property.

Example

The following code demonstrates the *Children* property from a command and control recognition.

To run this code, create a form with the following controls:

- Two labels called *Label1* and *Label2*

Paste this code into the Declarations section of the form.

The *Form_Load* procedure creates and activates a command and control grammar. The grammar file *sol.xml* is the *solitaire* grammar provided with the SDK. The path listed is for a standard SDK install and may be changed as needed.

The display indicates that the rules match. If any are child rules, then the child name is also displayed. For instance, if "move the red five on the black six," has been recognized, the application displays the recognized text in *Label1*. *Label2* displays the rule name and any associated children under it. If the rule has no children, *Label2* displays "No Children."

```
Option Explicit  
Public WithEvents RC As SpSharedRecoContext  
Public myGrammar As ISpeechRecoGrammar
```

```

Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.CmdLoadFromFile "C:\Program Files\Microsoft Sp
    myGrammar.CmdSetRuleIdState 0, SGDSActive
End Sub

Private Sub RC_FalseRecognition(ByVal StreamNumber As Long, l
    Beep
    Label1.Caption = "(no recognition)"
    Label2.Caption = ""
End Sub

Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Dim i, j, theFirstElement, theNumberOfElements As Long
    Dim theString As String

    Label1.Caption = Result.PhraseInfo.GetText & vbCrLf

    Label2.Caption = "Rule Properties Found : " & Result.Phr
    For i = 0 To Result.PhraseInfo.Properties.Count - 1

        'Property name used
        Label2.Caption = Label2.Caption & _
            "Rule " & i & ": " & Result.PhraseInfo.Propriet

        If Not Result.PhraseInfo.Properties.Item(i).Children
            Label2.Caption = Label2.Caption & _
                "    Children = " & Result.PhraseInfo.Propriet

            For j = 0 To Result.PhraseInfo.Properties.Item(i)
                Label2.Caption = Label2.Caption & _
                    "        Rule = " & Result.PhraseInfo.Propri
            Next
        Else
            Label2.Caption = Label2.Caption & "    No Childre
        End If

    Next
End Sub

```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseProperty](#)

Confidence Property

The **Confidence** property returns the confidence value for this semantic property computed by SAPI or the speech recognition engine.

It is an enumerated value of type [SpeechEngineConfidence](#). See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details.

Syntax

Set:	(This property is read-only)
Get:	SpeechEngineConfidence = <code>ISpeechPhraseProperty.Confidence</code>

Parts

ISpeechPhraseProperty
The owning object.

SpeechEngineConfidence
Set: (This property is read-only)
Get: A `SpeechEngineConfidence` variable that gets the property.

Remarks

It is possible to have different confidences for each rule name. This would be the result of pronunciation, background noise, or accent.

Example

The following code demonstrates getting the Confidence property from a command and control recognition. One label displays the recognized text and the other label displays the rule name activated as well as the confidence associated with that recognition.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a command and control grammar. The grammar file sol.xml is the solitaire grammar with the SDK. The path listed is for a standard SDK install and may be changed as needed.

If "play the red five" has been recognized, the application displays the recognized text in Label1. Label2 displays the rule name and the confidence for that recognition. In this case, the rules would be "color" and "rank" along with their confidence values.

```
Option Explicit
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar

Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.CmdLoadFromFile "C:\Program Files\Microsoft Sp
    myGrammar.CmdSetRuleIdState 0, SGDSActive
End Sub

Private Sub RC_FalseRecognition(ByVal StreamNumber As Long, l
    Beep
    Label1.Caption = "(no recognition)"
    Label2.Caption = ""
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal  
    Dim i As Long  
  
    Label1.Caption = Result.PhraseInfo.GetText & vbCrLf  
  
    Label2.Caption = "Rule Properties Found : " & Result.Phr  
    For i = 0 To Result.PhraseInfo.Properties.Count - 1  
        Label2.Caption = Label2.Caption & Result.PhraseInfo.l  
        Label2.Caption = Label2.Caption & " Confidence : " &  
    Next  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseProperty](#)

EngineConfidence Property

The **EngineConfidence** property returns the confidence value for this semantic property computed by the speech recognition (SR) engine.

The value range is specific to each SR engine and not standard across multiple SR engines. See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details.

Syntax

Set: (This property is read-only)

Get: *Single* = *ISpeechPhraseProperty*.**EngineConfidence**

Parts

ISpeechPhraseProperty

The owning object.

Single

Set: (This property is read-only)

Get: A *Single* variable that gets the property.

Example

The following code demonstrates retrieving the Engine Confidence property from a command and control recognition. One label displays the recognized text and the other label displays the rule name activated along with the engine confidence associated with that recognition.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a command and control grammar. The grammar file sol.xml is the solitaire grammar provided with the SDK. The path listed is for a standard SDK install and may be changed as needed.

If "play the red five" has been recognized, the application displays the recognized text in Label1. Label2 displays the rule name and the confidence for that recognition. In this case, the rules would be "color" and "rank" along with their Engine confidence values.

```
Option Explicit
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar

Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.CmdLoadFromFile "C:\Program Files\Microsoft Sp
    myGrammar.CmdSetRuleIdState 0, SGDSActive
End Sub

Private Sub RC_FalseRecognition(ByVal StreamNumber As Long, l
    Beep
    Label1.Caption = "(no recognition)"
    Label2.Caption = ""
End Sub

Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Dim i As Long

    Label1.Caption = Result.PhraseInfo.GetText & vbCrLf

    Label2.Caption = "Rule Properties Found : " & Result.Phr
    For i = 0 To Result.PhraseInfo.Properties.Count - 1
        Label2.Caption = Label2.Caption & Result.PhraseInfo.l
```

```
        Label2.Caption = Label2.Caption & " Engine Confidence  
    Next  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseProperty](#)

FirstElement Property

The **FirstElement** property returns the offset of the first spoken element spanned by this property.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseProperty*.**FirstElement**

Parts

ISpeechPhraseProperty

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the property.

Example

The following code demonstrates retrieving the Name property from a command and control recognition.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar. The grammar file sol.xml is the solitaire grammar provided with the SDK. The path listed is for a standard SDK

install and may be changed as needed.

If "play the red five" has been recognized, the application will display the recognized text in label1. Label2 displays the names of the two properties used from the grammar along with the actual word or phrase which activated the rule. In this case, the rules would be "color=red" and "rank=five." The number of elements in the phrase matching the property is displayed in parenthesis. It is possible for more than one word to activate a rule for a property. In the case of sol.xml, only single words are used. However, if the file were changed, for example, so that the ColorRed definition becomes "red card" instead of just "red", then the expression "play the red card five" would display two elements for the property.

```
Option Explicit
```

```
Public WithEvents RC As SpSharedRecoContext
```

```
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
```

```
    Set RC = New SpSharedRecoContext
```

```
    Set myGrammar = RC.CreateGrammar
```

```
    myGrammar.CmdLoadFromFile "C:\Program Files\Microsoft Sp
```

```
    myGrammar.CmdSetRuleIdState 0, SGDSActive
```

```
End Sub
```

```
Private Sub RC_FalseRecognition(ByVal StreamNumber As Long, l
```

```
    Beep
```

```
    Label1.Caption = "(no recognition)"
```

```
    Label2.Caption = ""
```

```
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
```

```
    Dim i, j, theFirstElement, theNumberOfElements As Long
```

```
    Dim theString As String
```

```
    Label1.Caption = Result.PhraseInfo.GetText & vbCrLf
```

```
    Label2.Caption = "Rule Properties Found : " & Result.Phr
```

```
    For i = 0 To Result.PhraseInfo.Properties.Count - 1
```

```
Label2.Caption = Label2.Caption & _
    "Rules Name: " & Result.PhraseInfo.Properties.Item(
theFirstElement = Result.PhraseInfo.Properties.Item(
theNumberOfElements = Result.PhraseInfo.Properties.I

theString = ""
For j = 0 To theNumberOfElements - 1
    theString = theString & Result.PhraseInfo.Elemen
    theString = theString & " "
Next

Label2.Caption = Label2.Caption & _
    " = " & theString & " (" & theNumberOfElements &

Next
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseProperty](#)

Id Property

The **Id** property returns the ID of the semantic property.

The ID is the numeric identifier associated with the [ISpeechPhraseProperty.Name](#) property. This property must be explicitly marked with the PROPID label.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseProperty.Id*

Parts

ISpeechPhraseProperty

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the property.

Remarks

Either Name or Id (if available) may be used to identify the rule invoked. Some languages, such as Visual Basic, can use strings in a Case Select statement. Therefore, the rule Name may be used directly in the Case Select statement. Other languages, such as C/C++ can only use numeric values in switch statements. In this case, the Id is more appropriate.

Example

See the example for [*ISpeechPhraseProperty.Name*](#) for more information.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseProperty](#)

Name Property

The **Name** property returns the name of the semantic property.

The Name is the name of the semantic property from the command and control grammar. This property must be explicitly marked with the PROPNAME label. This is often associated with an [ISpeechPhraseProperty.ID](#) which is a numeric identifier and is assigned by the PROPID label.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechPhraseProperty*.**Name**

Parts

ISpeechPhraseProperty

The owning object.

String

Set: (This property is read-only)

Get: A String variable that gets the property.

Remarks

Either Name or Id (if available) may be used to identify the rule invoked. Some languages, such as Visual Basic, can use strings in a Case Select statement. Therefore, the rule Name may be used directly in the Case Select statement. Other languages, such as C/C++ can only use numeric values in switch statements. In this case, the Id is more appropriate.

Example

The following code demonstrates getting the Name and Id property from a command and control recognition.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a command and control grammar. The grammar file sol.xml is the solitaire grammar provided with the SDK. The path listed is for a standard SDK install and may be changed as needed.

If "play the red five" has been recognized, the application displays the recognized text in Label1. Label2 displays the additional information about the grammar properties. In this case, the rules would be "color" and "rank" along with their numeric values.

```
Option Explicit
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar

Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.CmdLoadFromFile "C:\Program Files\Microsoft Sp
    myGrammar.CmdSetRuleIdState 0, SGDSActive
End Sub

Private Sub RC_FalseRecognition(ByVal StreamNumber As Long, l
    Beep
    Label1.Caption = "(no recognition)"
    Label2.Caption = ""
End Sub

Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Dim i, j, theFirstElement, theNumberOfElements As Long
```

```

Dim theString As String

Label1.Caption = Result.PhraseInfo.GetText & vbCrLf

Label2.Caption = "Rule Properties Found : " & Result.Phr
For i = 0 To Result.PhraseInfo.Properties.Count - 1

    'Property name used
    Label2.Caption = Label2.Caption & _
        "Property Name: " & Result.PhraseInfo.Properties

    'Property Id used
    Label2.Caption = Label2.Caption & _
        " (" & Result.PhraseInfo.Properties.Item(i).Id &

theFirstElement = Result.PhraseInfo.Properties.Item(
theNumberOfElements = Result.PhraseInfo.Properties.I

theString = ""
For j = 0 To theNumberOfElements - 1
    theString = theString & Result.PhraseInfo.Elemen
    theString = theString & " "
Next

Label2.Caption = Label2.Caption & _
    " = " & theString & " (" & theNumberOfElements &

Next
End Sub

```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseProperty](#)

NumberOfElements Property

The **NumberOfElements** property returns the number of spoken elements spanned by this property.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseProperty*.**NumberOfElements**

Parts

ISpeechPhraseProperty

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the property.

Example

See the example for [ISpeechPhraseProperty.FirstElement](#) for a complete example.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseProperty](#)

Parent Property

The **Parent** property specifies the parent of the semantic property.

A parent is a rule containing a child. A child is a rule within a rule. That is, within a command and control grammar, rules are explicitly defined with the Rule tag. To allow greater flexibility, it is possible to allow another rule to be used within one by declaring it with the RuleRef tag. In this case, the second rule being referenced by the RuleRef tag could be a child. Child rules are defined with the RULE tag, and referenced with the RULEREF tag. In this way, a rule can contain several levels of children and children rules can have several levels of parents.

An example of this is sol.xml for the card game solitaire. A user may play a card such as "move the red five on the black seven." Examine sol.xml; the rule is set up so that both the color of the card as well as the rank are referenced within the MoveCard rule. In this case, a successful recognition would have two top parent nodes: From and To. In turn, each parent node would have two children: color and rank. This way, an application could sort through the rules and know exactly which card moves (the From node) and which card receives it (the To node). The rule name would still be MoveCard.

In contrast, the more simple command "play the red ace," has only two nodes (color and rank). Neither node is considered a parent nor child since since the rule PlayCard does not have PROPNAME tag. Both nodes are peers of each other.

Syntax

Set: (This property is read-only)

Get: [ISpeechPhraseProperty](#) = *ISpeechPhraseProperty*.**Parent**

Parts

ISpeechPhraseProperty

The owning object.

ISpeechPhraseProperty

Set: (This property is read-only)

Get: An *ISpeechPhraseProperty* variable that gets the property.

Example

See [ISpeechPhraseProperty.Children](#) for a complete code sample.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseProperty](#)

Value Property

The **Value** property returns the value of the semantic property.

The Value property is the value of a semantic property set in either the VAL or VALSTR attributes in the grammar. Each terminal node can have a value associated with it assigned by the VAL (for numeric values) or VALSTR (for string) attributes. This value may be used for processing the rule.

For instance, in the sol.xml grammar for the card game solitaire used in the example below, the actual rank of the card (e.g., ace, five, king) is the terminal node since no other word or phrase is needed to complete the rule for rank. Each word has an associated value to assist with processing the rule. In the case of card rank, ace is assigned 1, two is assigned 2, and so on, through the king which is assigned a value of 13. Additionally, a common alternate for king is included as emperor which also has a value of 13.

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechPhraseProperty*.**Value**

Parts

ISpeechPhraseProperty

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant variable that gets the property.

Example

The following code demonstrates retrieving the value property from a command and control recognition. The application displays the recognized text and also makes a subjective statement about the worth of the card played based on rank. A low card (which includes the ace in this solitaire game), displays "You played a low card," for instance.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The grammar file sol.xml is the solitaire grammar provided with the SDK. The path listed is for a standard SDK install and may be changed as needed.

```
Option Explicit
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar

Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.CmdLoadFromFile "C:\Program Files\Microsoft Sp
    myGrammar.CmdSetRuleIdState 0, SGDSActive
End Sub

Private Sub RC_FalseRecognition(ByVal StreamNumber As Long, l
    Beep
    Label1.Caption = "(no recognition)"
    Label2.Caption = ""
End Sub

Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Dim i As Long
```

```
Label1.Caption = Result.PhraseInfo.GetText & vbCrLf

Label2.Caption = "Rule Properties Found : " & Result.Phr
For i = 0 To Result.PhraseInfo.Properties.Count - 1

    If Result.PhraseInfo.Properties.Item(i).Name = "rank

        Select Case Result.PhraseInfo.Properties.Item(i)
        Case Is < 5
            Label2.Caption = "You played a low card"
        Case Is < 9
            Label2.Caption = "You played a mediocre card"
        Case Else
            Label2.Caption = "You played a good card"
        End Select

    End If

Next
End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechPhraseReplacement

The **ISpeechPhraseReplacement** automation interface specifies a replacement, or text normalization, of one or more spoken words in a recognition result.

For example, the spoken words "twenty three" might be replaced by the replacement text "23."

Recognition results are returned in an [ISpeechRecoResult](#), which contains an [ISpeechPhraseInfo](#) object. The [ISpeechPhraseInfo](#) has two properties which are closely related. Its [Elements](#) property is a collection of [ISpeechPhraseElement](#) objects; its [Replacements](#) property is a collection of [ISpeechPhraseReplacement](#) objects. An [ISpeechPhraseReplacement](#) object is related to and dependent on the phrase elements in the recognition result which contains it.

Use of the [ISpeechPhraseReplacement](#) is demonstrated in a [code example](#) at the end of this section.

Automation Interface Elements

The [ISpeechPhraseReplacement](#) automation interface contains the following elements:

Properties	Description
DisplayAttributes Property	Returns a set of SpeechDisplayAttributes constants specifying information about the display of this word.
FirstElement Property	Returns the offset of the first spoken element to be replaced.
NumberOfElements Property	Returns the number of spoken elements to be replaced.

Text Property Returns the text of the replacement.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseReplacement](#)

DisplayAttributes Property

The **DisplayAttributes** property returns a set of `SpeechDisplayAttributes` constants specifying information about the display of this word.

Syntax

```
Set: (This property is read-only)  
Get: SpeechDisplayAttributes =  
     ISpeechPhraseReplacement.DisplayAttributes
```

Parts

ISpeechPhraseReplacement
The owning object.

SpeechDisplayAttributes
Set: (This property is read-only)
Get: One or more `SpeechDisplayAttributes` constants representing the value of the property.

Example

Use of the `DisplayAttributes` property is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseReplacement](#)

FirstElement Property

The **FirstElement** property returns the index of the first phrase element to be replaced.

The phrase elements replaced are contained in the same recognition result that contains the phrase replacement.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseReplacement*.**FirstElement**

Parts

ISpeechPhraseReplacement

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the property.

Example

Use of the FirstElement property is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseReplacement](#)

NumberOfElements Property

The **NumberOfElements** property returns the number of elements to replaced.

The phrase elements replaced are contained in the same recognition result that contains the phrase replacement.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseReplacement*.**NumberOfElements**

Parts

ISpeechPhraseReplacement

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the property.

Example

Use of the NumberOfElements property is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseReplacement](#)

Text Property

The **Text** property returns the text of the replacement.

The phrase elements replaced are contained in the same recognition result that contains the phrase replacement. The Text property is the replacement for the number of phrase elements in the [NumberOfElements](#) property, beginning with the element specified in the [FirstElement](#) property.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechPhraseReplacement*.**Text**

Parts

ISpeechPhraseReplacement

The owning object.

String

Set: (This property is read-only)

Get: A String variable representing the value of the property.

Example

Use of the Text property is demonstrated in a [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseReplacement](#)

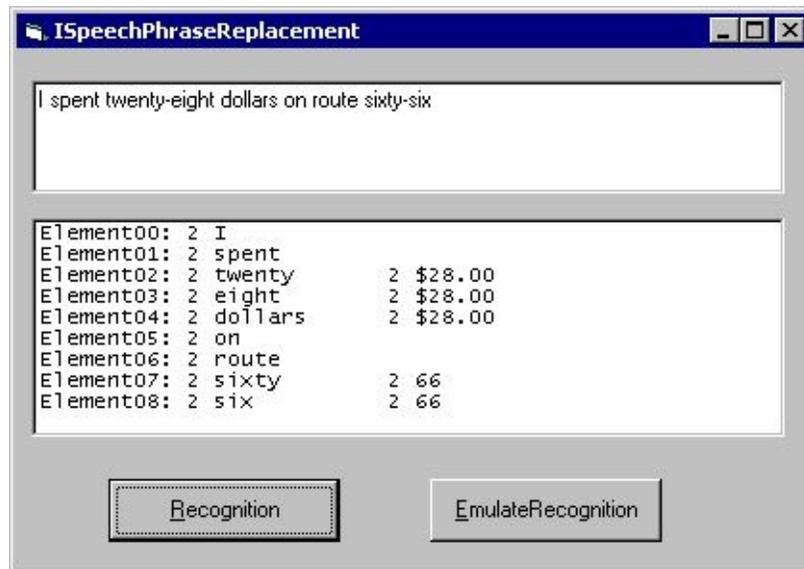
Example

The following Visual Basic form code demonstrates the ISpeechPhraseReplacement interface and its relation to the ISpeechPhraseElement interface. To run this code, create a form with the following controls:

- A text box called Text1
- A list box called List1
- Two command buttons called Command1 and Command2

Paste this code into the Declarations section of the form.

This example is based on the [ISpeechPhraseElement code example](#). Individual phrase elements from the recognition result are listed in the list box. The DisplayAttributes and DisplayText properties of each phrase element are shown. The DisplayAttributes and Text properties of the phrase replacements are displayed next to the elements which can be replaced by them.



Option Explicit

```
Const WAVEFILENAME = "C:\ISpeechPhraseReplacement.wav"
```

```
Dim R As SpeechLib.SpInprocRecognizer
Dim G As SpeechLib.ISpeechRecoGrammar
Dim F As SpeechLib.SpFileStream
Dim E As SpeechLib.ISpeechPhraseElement
```

```
Dim Rep As SpeechLib.ISpeechPhraseReplacement
Dim Replacements As SpeechLib.ISpeechPhraseReplacements
```

```
Dim V As SpeechLib.SpVoice
Dim V2 As SpeechLib.SpVoice 'Plays the wave file back
```

```
Dim WithEvents C As SpeechLib.SpInProcRecoContext
```

```
Private Sub Command1_Click()
```

```
    List1.Clear
    Call SpeakToFile(Text1.Text, WAVEFILENAME)
    F.Open WAVEFILENAME
    Set R.AudioInputStream = F
```

```
End Sub
```

```
Private Sub Command2_Click()
```

```

List1.Clear
C.Recognizer.EmulateRecognition Text1.Text

End Sub

Private Sub Form_Load()

    ' Create Recognizer, RecoContext, Grammar, and Voice
    Set R = New SpInprocRecognizer
    Set C = R.CreateRecoContext
    Set G = C.CreateGrammar(16)
    Set V = New SpVoice
    Set V.Voice = V.GetVoices("gender=male").Item(0)
    Set V2 = New SpVoice

    ' Load Grammar with solitaire XML, set active
    G.CmdLoadFromFile "c:\sol.xml", SLOStatic
    G.CmdSetRuleIdState 0, SGDSActive           'Set C &
    G.DictationSetState SGDSActive           'Set Dic

    Text1.Text = "I spent twenty-eight dollars on route sixt
    Command1.Caption = "&Recognition;"
    Command2.Caption = "&EmulateRecognition;"

End Sub

Private Sub SpeakToFile(ByVal strText As String, ByVal strFN:
    Set F = New SpFileStream           'Create stre
    F.Open strFName, SSFMCreatForWrite, True   'Open as the
    Set V.AudioOutputStream = F           'Set voice o
    V.Speak strText, SVSFIsXML           'Speak synch
    F.Close                               'Close file

End Sub

Private Sub C_Recognition(ByVal StreamNumber As Long, _
                        ByVal StreamPosition As Variant, _
                        ByVal RecognitionType As SpeechLib.Speech
                        ByVal Result As SpeechLib.ISpeechRecoRes

    Dim ECount As Integer   'Count of elements
    Dim ii As Integer

```

```

Dim nn As Integer
Dim R1 As Integer, R2 As Integer

'Arrays with an entry for every PhraseElement
Dim arrElements() As String
Dim arrReplaces() As String
ECount = Result.PhraseInfo.Elements.Count - 1
ReDim arrElements(ECount)
ReDim arrReplaces(ECount)

'Load PhraseElements into an array
nn = 0
For Each E In Result.PhraseInfo.Elements
    arrElements(nn) = E.DisplayAttributes & " " & E.Display
    nn = nn + 1
Next

'Load PhraseReplacements (if any) in an array
Set Repls = Result.PhraseInfo.Replacements
If Not Repls Is Nothing Then
    For Each Rep In Repls

        'Get the first and last elements
        'replaced by this Replacement
        R1 = Rep.FirstElement
        R2 = R1 + Rep.NumberOfElements - 1

        For ii = 0 To ECount
            'Is element within the range of this replacement
            If (ii >= R1) And (ii <= R2) Then
                arrReplaces(ii) = Rep.DisplayAttributes & " "
            End If
        Next ii
    Next Rep
End If

Dim X As String
For ii = 0 To ECount

    'Get PhraseElement and pad with blanks
    X = arrElements(ii) & String(15, " ")
    X = Left(X, 15)

```

```

        'Put element index in front
        X = "Element" & Format(ii, "00") & ": " & X

        'Put Replacement elements at the end
        X = X & arrReplaces(ii)

        List1.AddItem X
    Next ii

End Sub

Private Sub C_EndStream(ByVal StreamNumber As Long, _
                        ByVal StreamPosition As Variant, _
                        ByVal StreamReleased As Boolean)

    'Recognition uses the Filestream, EmulateReco does not
    If ActiveControl.Caption = "&Recognition;" Then
        F.Close
        DoEvents
        F.Open WAVEFILENAME
        V2.SpeakStream F
        F.Close
    End If
    List1.AddItem ""

End Sub

```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechPhraseReplacements

The **ISpeechPhraseReplacements** automation interface represents a collection of [ISpeechPhraseReplacement](#) objects.

Automation Interface Elements

The ISpeechPhraseReplacements automation interface contains the following elements:

Properties	Description
Count Property	Returns the number of objects in the collection.

Methods	Description
Item Method	Returns a member of the collection by its index.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseReplacements](#)

Count Property

The **Count** property returns the number of ISpeechPhraseReplacement objects in the ISpeechPhraseReplacements object.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseReplacements*.**Count**

Parts

ISpeechPhraseReplacements

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the Count property.

Remarks

The ISpeechPhraseReplacements object is a collection of ISpeechPhraseReplacement objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Count property is one of these common properties.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseReplacements](#)

Item Method

The **Item** method returns a member of the ISpeechPhraseReplacement collection by its index.

```
ISpeechPhraseReplacements.Item(  
    Index As Long  
) As ISpeechPhraseReplacement
```

Parameters

Index
Specifies the Index.

Return Value

The Item method returns an ISpeechPhraseReplacement variable.

Remarks

The ISpeechPhraseReplacements object is a collection of ISpeechPhraseReplacement objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Item method is one of these common methods.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechPhraseRule

The **ISpeechPhraseRule** automation interface represents the part of a recognition result that returns information about the grammar rule that produced the recognition.

An ISpeechPhraseRule object variable is returned by the [Rule](#) property of the ISpeechPhraseInfo object variable within a recognition result.

For an example of the use of the ISpeechPhraseRule interface, see the [code example](#) at the end of this section.

Automation Interface Elements

The ISpeechPhraseRule automation interface contains the following elements:

Properties	Description
Children Property	Returns an ISpeechPhraseRules collection of the rule's child rules.
Confidence Property	Returns the confidence for the rule computed by SAPI.
EngineConfidence Property	Returns the confidence score for the rule computed by the SR engine.
FirstElement Property	Returns the audio stream offset of the first phrase element in the recognition result matched by the rule.
Id Property	Returns the ID of the phrase rule.
Name Property	Returns the name of the phrase rule.
NumberOfElements Property	Returns the number of phrase elements spanned by this rule.

Parent Property

Returns the rule's parent rule.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseRule](#)

Children Property

The **Children** property returns an `ISpeechPhraseRules` collection of the rule's child rules.

If the rule has no children, its `Children` property will be `Nothing`.

Syntax

Set: (This property is read-only)

Get: [ISpeechPhraseRules](#) = `ISpeechPhraseRule.Children`

Parts

ISpeechPhraseRule

The owning object.

ISpeechPhraseRules

Set: (This property is read-only)

Get: An `ISpeechPhraseRules` object that gets the collection of child rules.

Example

For an example of the use of the `Children` property, see the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseRule](#)

Confidence Property

The **Confidence** property returns the confidence for this rule computed by SAPI.

It is an enumerated value of type [SpeechEngineConfidence](#). See [Confidence Scoring and Rejection](#) in [SAPI Speech Recognition Engine Guide](#) for additional details.

Syntax

Set:	(This property is read-only)
Get:	SpeechEngineConfidence = <code>ISpeechPhraseRule.Confidence</code>

Parts

ISpeechPhraseRule
The owning object.

SpeechEngineConfidence
Set: (This property is read-only)
Get: A `SpeechEngineConfidence` variable that gets the Confidence property.

Example

For an example of the use of the Confidence property, see the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseRule](#)

EngineConfidence Property

The **EngineConfidence** property returns the confidence score for the rule computed by the speech recognition (SR) engine.

The value is engine dependent and not standardized across multiple SR engines.

Syntax

Set: (This property is read-only)

Get: *Single* = *ISpeechPhraseRule*.**EngineConfidence**

Parts

ISpeechPhraseRule

The owning object.

Single

Set: (This property is read-only)

Get: A Single variable that gets the EngineConfidence.

Example

For an example of the use of the EngineConfidence property, see the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseRule](#)

FirstElement Property

The **FirstElement** property returns the audio stream offset of the first phrase element in the recognition result matched by the rule.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseRule*.**FirstElement**

Parts

ISpeechPhraseRule

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the audio stream offset of the first element.

Example

For an example of the use of the FirstElement property, see the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseRule](#)

Id Property

The **Id** property returns the ID of the phrase rule.

In the Speech Text Grammar Format, this data is specified in the ID attribute of the RULE element.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseRule*.**Id**

Parts

ISpeechPhraseRule

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the ID of the rule.

Example

For an example of the use of the Id property, see the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseRule](#)

Name Property

The **Name** property returns the name of the phrase rule.

In the Speech Text Grammar Format, this data is specified in the NAME attribute of the RULE element.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechPhraseRule*.**Name()**

Parts

ISpeechPhraseRule

The owning object.

String

Set: (This property is read-only)

Get: A String variable that gets the name of the rule.

Example

For an example of the use of the Name property, see the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseRule](#)

NumberOfElements Property

The **NumberOfElements** property returns the number of phrase elements spanned by this rule.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseRule*.**NumberOfElements**

Parts

ISpeechPhraseRule

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the number of elements.

Example

For an example of the use of the NumberOfElements property, see the [code example](#) at the end of this section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseRule](#)

Parent Property

The **Parent** property returns the rule's parent rule.

If the rule is a top-level rule, it has no parent, and its Parent property is Nothing.

Syntax

Set: (This property is read-only)

Get: [ISpeechPhraseRule](#) = *ISpeechPhraseRule*.**Parent**

Parts

ISpeechPhraseRule

The owning object.

ISpeechPhraseRule

Set: (This property is read-only)

Get: An *ISpeechPhraseRule* object that gets the parent rule.

Example

For an example of the use of the Parent property, see the [code example](#) at the end of this section.



ISpeechPhraseRule Code Example

The following Visual Basic form code demonstrates the use of the ISpeechPhraseRule interface. To run this code, create a form with the following controls:

- A text box called Text1
- A list box called List1
- A command button called Command1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a recognizer, a recognition context and a grammar object. It loads the grammar object with the Solitaire grammar sol.xml and activates it.

The Command1 procedure speaks the text in the text box into a file for recognition. The recognition context's Recognition event displays selected information from the recognition result. It displays the phrase elements first, and then the properties of the ISpeechPhraseRule object.

Note that both the Parent and Children properties may have a value of Nothing. Attempting to reference them in this state will cause a run-time error.

```
Option Explicit
```

```
Const WAVEFILENAME = "C:\ISpeechPhraseElement.wav"
```

```
Dim R As SpeechLib.SpInprocRecognizer  
Dim G As SpeechLib.ISpeechRecoGrammar  
Dim F As SpeechLib.SpFileStream  
Dim E As SpeechLib.ISpeechPhraseElement
```

```

Dim V As SpeechLib.SpVoice
Dim V2 As SpeechLib.SpVoice      'Plays the wave file back

Dim WithEvents C As SpeechLib.SpInProcRecoContext

Private Sub Command1_Click()

    List1.Clear
    Call SpeakToFile(Text1.Text, WAVEFILENAME)
    F.Open WAVEFILENAME
    Set R.AudioInputStream = F

End Sub

Private Sub Form_Load()

    ' Create Recognizer, RecoContext, Grammar, and Voice
    Set R = New SpInProcRecognizer
    Set C = R.CreateRecoContext
    Set G = C.CreateGrammar(16)
    Set V = New SpVoice
    Set V.Voice = V.GetVoices("gender=male").Item(0)
    Set V2 = New SpVoice

    ' Load Grammar with solitaire XML, set active
    G.CmdLoadFromFile "c:\sol.xml", SLODynamic
    G.CmdSetRuleIdState 0, SGDSActive           'Set C &
    G.DictationSetState SGDSActive             'Set Dic

    Text1.Text = "play the eight of clubs"

End Sub

Private Sub SpeakToFile(ByVal strText As String, ByVal strFN:
    Set F = New SpFileStream                   'Create stre:
    F.Open strFName, SSFMCreatForWrite, True  'Open as the
    Set V.AudioOutputStream = F              'Set voice o
    V.Speak strText, SVSFIsXML               'Speak synch
    F.Close                                   'Close file

End Sub

Private Sub C_Recognition(ByVal StreamNumber As Long, _

```

```

        ByVal StreamPosition As Variant, _
        ByVal RecognitionType As SpeechLib.SpeechRecognitionType, _
        ByVal Result As SpeechLib.ISpeechRecognitionResult) As String
Dim X As String
Dim ii As Integer
Dim PR As ISpeechPhraseRule
Dim PRs As ISpeechPhraseRules

ii = 0
For Each E In Result.PhraseInfo.Elements
    X = "element" & Str(ii) & ": " & E.DisplayText
    List1.AddItem X
    ii = ii + 1
Next

'This is the rule that recognition was based on
Set PR = Result.PhraseInfo.Rule

List1.AddItem ""
List1.AddItem "Id:                " & PR.Id
List1.AddItem "Rule:                " & PR.Name
List1.AddItem "NumberOfElements: " & PR.NumberOfElements
List1.AddItem "FirstElement:       " & PR.FirstElement
List1.AddItem "EngineConfidence:  " & PR.EngineConfidence
List1.AddItem "Confidence:        " & PR.Confidence
List1.AddItem ""

If PR.Parent Is Nothing Then
    List1.AddItem "Parent:                none"
Else
    List1.AddItem "Parent:                " & PR.Parent.Name
End If

If PR.Children Is Nothing Then
    List1.AddItem "Children:              none"
Else
    'This routine replaces the original PR object
    Set PRs = PR.Children
    ii = 0
    For Each PR In PRs
        List1.AddItem "Child" & Str(ii) & ": " & PR.DisplayText
        ii = ii + 1
    Next
End If

```

```
    Next  
End If
```

```
End Sub
```

```
Private Sub C_EndStream(ByVal StreamNumber As Long, _  
                        ByVal StreamPosition As Variant, _  
                        ByVal StreamReleased As Boolean)
```

```
    F.Close
```

```
    DoEvents
```

```
    F.Open WAVEFILENAME
```

```
    V2.SpeakStream F
```

```
    F.Close
```

```
End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechPhraseRules

The **ISpeechPhraseRules** automation interface represents a collection of [ISpeechPhraseRule](#) objects.

Automation Interface Elements

The ISpeechPhraseRules automation interface contains the following elements:

Properties	Description
Count Property	Returns the number of objects in the collection.

Methods	Description
Item Method	Returns a member of the collection by its index.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseRules](#)

Count Property

The **Count** property returns the number of ISpeechPhraseRule objects in the ISpeechPhraseRules object.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechPhraseRules.Count*

Parts

ISpeechPhraseRules

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that gets the Count property.

Remarks

The ISpeechPhraseRules object is a collection of ISpeechPhraseRule objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Count property is one of these common properties.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechPhraseRules](#)

Item Method

The **Item** method returns a member of the ISpeechPhraseRules collection by its index.

```
ISpeechPhraseRules.Item(  
    Index As Long  
) As ISpeechPhraseRule
```

Parameters

Index
Specifies the Index.

Return Value

The Item method returns an ISpeechPhraseRule variable.

Remarks

The ISpeechPhraseRules object is a collection of ISpeechPhraseRule objects. As a collection, it provides access to any or all of its members through certain methods and properties common to all collection objects. The Item method is one of these common methods.

Example

See the [SpObjectToken Example](#) for a complete example and additional details.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechRecognizerStatus

The **ISpeechRecognizerStatus** automation interface returns the status of the speech recognition (SR) engine represented by the recognizer object.

This method provides information for static elements about the SR engine such as the languages it supports. It also provides information for dynamic elements such as the engine's current stream position, and whether the stream is actively being sent to the engine.

These dynamic elements are equivalent to parameters returned by recognition context events. It may be advantageous for some applications to retrieve these elements by calling Status occasionally, rather than by receiving events constantly.

Automation Interface Elements

The ISpeechRecognizerStatus automation interface contains the following elements:

Properties	Description
AudioStatus Property	Returns the status of the recognizer's audio output.
ClsidEngine Property	Returns the unique identifier associated with the current engine.
CurrentStreamNumber Property	Returns the number of the current recognition stream.
CurrentStreamPosition Property	Returns the recognizer's position in the recognition stream, in bytes.
NumberOfActiveRules Property	Returns the current engine's number of active rules.
SupportedLanguages Property	Returns an array containing the languages the active engine

supports.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizerStatus](#)

AudioStatus Property

The **AudioStatus** property returns the status of the speech recognition (SR) engine's audio output.

Syntax

```
Set: (This property is read-only)  
Get: ISpeechAudioStatus =  
     ISpeechRecognizerStatus.AudioStatus
```

Parts

ISpeechRecognizerStatus

The owning object.

ISpeechAudioStatus

Set: (This property is read-only)

Get: An *ISpeechAudioStatus* variable which gets the property.

Example

For an example of the use of the `AudioStatus` property, see the code example in the `Recognizer Status` property section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizerStatus](#)

ClsidEngine Property

The **ClsidEngine** property returns the unique identifier associated with the current engine.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechRecognizerStatus*.**ClsidEngine**

Parts

ISpeechRecognizerStatus

The owning object.

String

Set: (This property is read-only)

Get: A String variable which gets the property.

Example

For an example of the use of the ClsidEngine property, see the code example in the Recognizer [Status](#) property section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizerStatus](#)

CurrentStreamNumber Property

The **CurrentStreamNumber** property returns the number of the current recognition stream.

Syntax

Set: (This property is read-only)

Get: *Long* =

ISpeechRecognizerStatus.**CurrentStreamNumber**

Parts

ISpeechRecognizerStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable which gets the property.

Example

For an example of the use of the CurrentStreamNumber property, see the code example in the Recognizer [Status](#) property section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizerStatus](#)

CurrentStreamPosition Property

The **CurrentStreamPosition** property returns the recognizer's position in the recognition stream, in bytes.

Syntax

Set: (This property is read-only)

Get: *Variant* =

ISpeechRecognizerStatus.**CurrentStreamPosition**

Parts

ISpeechRecognizerStatus

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant variable which gets the property.

Example

For an example of the use of the CurrentStreamPosition property, see the code example in the Recognizer [Status](#) property section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizerStatus](#)

NumberOfActiveRules Property

The **NumberOfActiveRules** property returns the current engine's number of active rules.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechRecognizerStatus*.**NumberOfActiveRules**

Parts

ISpeechRecognizerStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable which gets the property.

Example

For an example of the use of the NumberOfActiveRules property, see the code example in the Recognizer [Status](#) property section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizerStatus](#)

SupportedLanguages Property

The **SupportedLanguages** property returns an array containing the languages that the active engine supports.

The array contains the language ID in decimal format.

Syntax

Set: (This property is read-only)

Get: *Variant* =

ISpeechRecognizerStatus.**SupportedLanguages**

Parts

ISpeechRecognizerStatus

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant data type as an array with each element containing a language ID of a supported language.

Example

For an example of the use of the SupportedLanguages property, see the code example in the Recognizer [Status](#) property section.

Microsoft Speech SDK

Speech Automation 5.1



ISpeechRecoGrammar

The **ISpeechRecoGrammar** automation interface enables applications to manage the words and phrases for the SR engine.

A single SpRecognizer object can be associated with multiple SpRecoContext objects. Similarly, a single SpRecoContext object can be associated with multiple ISpRecoGrammar objects. This allows designers to create applications containing several grammars, each of which is specialized for a different type of recognition. Each ISpRecoGrammar object can contain both a context-free grammar (CFG) and a dictation grammar simultaneously.

Automation Interfaces

The ISpeechRecoGrammar automation interface contains the following elements:

Properties	Description
Id Property	Returns the Id assigned to the grammar when it was created.
RecoContext Property	Returns the RecoContext object that created this grammar.
Rules Property	Returns the collection of grammar rules in the RecoGrammar.
State Property	Gets and sets the operational status of the speech grammar.

Methods	Description
CmdLoadFromFile Method	Loads a command and control grammar from

	the specified file.
<u>CmdLoadFromMemory</u> Method	Loads a compiled speech grammar from memory.
<u>CmdLoadFromObject</u> Method	Loads a speech grammar from a COM object.
<u>CmdLoadFromProprietaryGrammar</u> Method	Loads a proprietary speech grammar.
<u>CmdLoadFromResource</u> Method	Loads a command and control grammar from a Win32 resource.
<u>CmdSetRuleIdState</u> Method	Activates or deactivates a rule by its rule ID.
<u>CmdSetRuleState</u> Method	Activates or deactivates a rule by its rule name.
<u>DictationLoad</u> Method	Loads a dictation topic into the grammar.
<u>DictationSetState</u> Method	Sets the dictation topic state.
<u>DictationUnload</u> Method	Unloads the active dictation topic from the grammar.
<u>IsPronounceable</u> Method	Determines if a word has a pronunciation.
<u>Reset</u> Method	Clears all grammar rules and resets the grammar's language to NewLanguage.
<u>SetTextSelection</u> Method	Sets the range of text selection information in a word sequence data buffer.

SetWordSequenceData Method

Defines a word sequence data buffer for use by the SR engine.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

CmdLoadFromFile Method

The **CmdLoadFromFile** method loads a command and control grammar from the specified file.

The grammar may be compiled or uncompiled, and it can be loaded for static or dynamic use, as specified in the LoadOption parameter.

```
ISpeechRecoGrammar.CmdLoadFromFile(  
    FileName As String,  
    [LoadOption As SpeechLoadOption = SLOStatic]  
)
```

Parameters

FileName

Specifies the file name. SAPI 5 supports loading of compiled static grammars through a URL.

LoadOption

[Optional] Specifies whether the grammar is to be loaded for static or dynamic use. The default is static.

Return Value

None.

Example

The following Visual Basic form code demonstrates the use of

the CmdLoadFromFile and the CmdLoadFromMemory methods. To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates two grammar objects and uses the CmdLoadFromFile method to load the Solitaire rules into the first grammar. The Command1_Click procedure calls a subroutine called GrammarToMemory, which creates a temporary grammar in a Variant variable, and returns this grammar to the caller. The Command1 procedure then reloads the first grammar with the temporary grammar.

Option Explicit

```
Dim C As SpeechLib.SpSharedRecoContext
Dim G1 As SpeechLib.ISpeechRecoGrammar
Dim G2 As SpeechLib.ISpeechRecoGrammar
```

```
Private Sub Command1_Click()
    Dim GT As Variant                'Temp grammar in Variant

    GT = GrammarToMemory(G1)        'GT is temp version of G1

    Call G2.CmdLoadFromMemory(GT, SLOStatic)
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set C = New SpSharedRecoContext
    Set G1 = C.CreateGrammar
    Set G2 = C.CreateGrammar
```

```
    Call G1.CmdLoadFromFile("c:\sol.xml", SLODynamic)
```

```
End Sub
```

```
Private Function GrammarToMemory(objGRM As SpeechLib.ISpeechRecoGrammar) As SpeechLib.ISpeechRecoGrammar
```

```
'Make changes to a standard grammar, and save it as temp  
'Add rules to the grammar, delete rules from the grammar  
  
'Return the variant from ISpeechGrammarRules.CommitAndSa  
GrammarToMemory = objGRM.Rules.CommitAndSave("")
```

End Function

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

CmdLoadFromMemory Method

The **CmdLoadFromMemory** method loads a compiled speech grammar from memory.

The

```
ISpeechRecoGrammar.CmdLoadFromMemory(  
    GrammarData As Variant,  
    [LoadOption As SpeechLoadOption = SL0Static]  
)
```

Parameters

GrammarData

Specifies the GrammarData.

LoadOption

[Optional] Specifies whether the grammar is to be loaded for static or dynamic use. The default is static.

Return Value

None.

Example

The following Visual Basic form code demonstrates use of the CmdLoadFromMemory method. To run this code, create a form and paste this code into the Declarations section.

The Form_Load procedure creates a recognition context and a CFG grammar object. It adds a simple rule to the grammar, and saves the grammar data to a Variant variable with the CommitAndSave method. It then uses the CmdLoadFromMemory method to load the second grammar object with the grammar data in the Variant variable. Finally, it activates the rule in the second grammar.

If the computer has a microphone, run this code and speak the phrase, "Hello, world" into the microphone. The recognition context's Recognition and FalseRecognition event procedures display messages indicating successful or unsuccessful recognition.

Option Explicit

```
Dim WithEvents recoCtx As SpSharedRecoContext
Dim grammar As ISpeechRecoGrammar
Dim grammar2 As ISpeechRecoGrammar
Dim gRules As ISpeechGrammarRules
Dim gRule As ISpeechGrammarRule
Dim state As ISpeechGrammarRuleState

Dim buffer As String      ' Any error information when committ.
Dim gData As Variant      ' The contents of the grammar held in

' Typically there is no reason to do something like this.
' This is just the simplest possible example to demonstrate
' ISpeechRecoGrammar.CmdLoadFromMemory.
```

```
Private Sub Form_Load()
```

```
    Set recoCtx = New SpSharedRecoContext
    Set grammar = recoCtx.CreateGrammar
    Set gRules = grammar.Rules

    ' Grammar has one rule, one state, and one transition
    Set gRule = gRules.Add("greeting", SRATopLevel, 1)
    Set state = gRule.InitialState
    state.AddWordTransition Nothing, "hello world", " "
```

```

' Save the grammar to memory (contents saved in gData)
gData = grammar.Rules.CommitAndSave(buffer)

Set grammar2 = recoCtx.CreateGrammar           'Sec
grammar2.CmdLoadFromMemory gData, SLOStatic   'Loa
grammar2.CmdSetRuleState "greeting", SGDSActive 'Set

End Sub

Private Sub recoCtx_FalseRecognition(ByVal StreamNumber As Long, ByVal Phrase As String)
' Gets here only when the phrase "hello world" is false!
MsgBox "(not recognized!)"
End Sub

Private Sub recoCtx_Recognition(ByVal StreamNumber As Long, ByVal Phrase As String, ByVal Result As RecognitionResult)
' Gets here only when the phrase "hello world" is recogn.
MsgBox Result.PhraseInfo.GetText
End Sub

```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

CmdLoadFromObject Method

The **CmdLoadFromObject** method loads a speech grammar from a component object model (COM) object.

The COM object must be created from a Windows dynamic link library (DLL).

```
ISpeechRecoGrammar.CmdLoadFromObject(  
    ClassId As String,  
    GrammarName As String,  
    [LoadOption As SpeechLoadOption = SLOStatic]  
)
```

Parameters

ClassId

Specifies The reference class ID of the object containing the command.

GrammarName

Specifies the GrammarName.

LoadOption

[Optional] Specifies whether the grammar is to be loaded for static or dynamic use. The default is static.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

CmdLoadFromProprietaryGrammar Method

The **CmdLoadFromProprietaryGrammar** method loads a proprietary speech grammar.

```
ISpeechRecoGrammar.CmdLoadFromProprietaryGrammar(  
    ProprietaryGuid As String,  
    ProprietaryString As String,  
    ProprietaryData As Variant,  
    [LoadOption As SpeechLoadOption = SLOStatic]  
)
```

Parameters

ProprietaryGuid

Specifies the GUID of the grammar, which is used by the application and the speech recognition engine to identify it for verifying support.

ProprietaryString

A command string used to send information about the grammar.

ProprietaryData

Additional information for the process.

LoadOption

[Optional] Specifies whether the grammar is to be loaded for static or dynamic use. The default is static. In the case of *CmdLoadFromProprietaryGrammar*, static loading must be

used.

Return Value

None.

Remarks

Using this method will unload the currently loaded context-free grammar or proprietary grammar.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

CmdLoadFromResource Method

The **CmdLoadFromResource** method loads a command and control grammar from a Win32 resource.

```
ISpeechRecoGrammar.CmdLoadFromResource(  
    hModule As Long,  
    ResourceName As Variant,  
    ResourceType As Variant,  
    LanguageId As Long,  
    [LoadOption As SpeechLoadOption = SL0Static]  
)
```

Parameters

hModule

Specifies the hModule.

ResourceName

Specifies the ResourceName.

ResourceType

Specifies the ResourceType.

LanguageId

Specifies the LanguageId.

LoadOption

[Optional] Specifies whether the grammar is to be loaded for static or dynamic use. The default is static.

Return Value

None.

Remarks

The grammar resource must be a compiled SAPI 5 binary version of a context-free grammar (CFG).

Using this method will unload the currently loaded CFG or proprietary grammar.

Example

The following Visual Basic form code demonstrates the use of the `CmdLoadFromResource` method. Before attempting to run this code, you must:

- Create a `SpeechDocs.dll` file as detailed in the [Sample DLL Code Example](#), and
- Make sure your microphone is connected and functional.

The `Form_Load` procedure creates a grammar object and loads it with the compiled Solitaire grammar contained in the DLL. There are no controls on this form; simply speak into your microphone. Phrases like "Move the red ten" or "Play the queen of hearts" will be recognized.

```
Option Explicit
```

```
Dim WithEvents C As SpSharedRecoContext  
Dim G As ISpeechRecoGrammar  
Dim hndRes As Long
```

```
' Change this constant to match the path  
' of the SpeechDocs DLL on your machine!
```

```

Const lib As String = "C:\SpeechDocs.dll"

Const resType As String = "CFGGRAMMAR"
Const resName As Long = 101
Const langID As Long = &H409;

Private Declare Function LoadLibrary Lib "kernel32" Alias "LoadLibraryA" ( _
    ByVal lpLibFileName As String) As Long

Private Sub Form_Load()
    Set C = New SpSharedRecoContext
    Set G = C.CreateGrammar

    ' Obtain a handle to the executable holding the grammar
    hndRes = LoadLibrary(lib)

    ' Load the grammar from the resource
    G.CmdLoadFromResource hndRes, resName, resType, langID, 0

    ' Set all DefaultToActive rules active
    ' NOTE: The solitaire grammar happens to have DefaultToActive
    ' which is why we can activate them this way.
    G.CmdSetRuleState "", SGDSActive

End Sub

Private Sub C_FalseRecognition( _
    ByVal StreamNumber As Long, _
    ByVal StreamPosition As Variant, _
    ByVal result As SpeechLib.ISpeechRecoResult _
)
    MsgBox "(not recognized!)"
End Sub

Private Sub C_Recognition( _
    ByVal StreamNumber As Long, _
    ByVal StreamPosition As Variant, _
    ByVal RecognitionType As SpeechLib.SpeechRecognitionType, _
    ByVal result As SpeechLib.ISpeechRecoResult _
)
    MsgBox result.PhraseInfo.GetText

```

End Sub

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

CmdSetRuleIdState Method

The **CmdSetRuleIdState** method activates or deactivates a rule by its rule ID.

```
ISpeechRecoGrammar.CmdSetRuleIdState(  
    RuleId As Long,  
    State As SpeechRuleState  
)
```

Parameters

RuleId

The Id of the rule to be changed. If Id is zero, all TopLevel and Active rules in the grammar will be changed.

State

The rule state to which the rule or rules will be changed.

Return Value

None.

Example

The following Visual Basic form code demonstrates the use of the Rules method, the CmdSetRuleState method and the CmdSetRuleIdState method. To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

The Form1_Load procedure creates a grammar and loads it with the solitaire grammar sol.xml, and uses the Rules method to create a collection of the rules contained in the grammar. The Command1_Click procedure creates an ISpeechGrammarRule object for the first rule contained in the grammar, and deactivates this rule using the CmdSetRuleState method and the rule's Name property. The procedure then creates an ISpeechGrammarRule object for the second rule, and deactivates that rule using the CmdSetRuleIdState method and the rule's ID property.

Another example of the use of CmdSetRuleIdState can be found in the code example for the [CmdLoadFromMemory](#) method.

Option Explicit

```
Dim C As SpeechLib.SpSharedRecoContext
Dim G As SpeechLib.ISpeechRecoGrammar
Dim R As SpeechLib.ISpeechGrammarRules
```

```
Dim Rule As SpeechLib.ISpeechGrammarRule
```

```
Private Sub Command1_Click()
```

```
    'Get first rule in rules collection and set it inactive
    'Use CmdSetRuleState method and the rule NAME
```

```
    Set Rule = R.Item(0)
    G.CmdSetRuleState Rule.Name, SGDSInactive
```

```
    'Get next rule in rules collection and set it inactive,
    'Use CmdSetRuleIdState method and the rule ID
```

```
    Set Rule = R.Item(1)
    G.CmdSetRuleIdState Rule.Id, SGDSInactive
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
Set C = New SpSharedRecoContext
Set G = C.CreateGrammar

'Load the Solitaire grammar so it can be changed
Call G.CmdLoadFromFile("c:\sol.xml", SLODynamic)

Set R = G.Rules      'Get the collection of rules

End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

CmdSetRuleState Method

The **CmdSetRuleState** method activates or deactivates a rule by its name.

```
ISpeechRecoGrammar.CmdSetRuleState(  
    Name As String,  
    State As SpeechRuleState  
)
```

Parameters

Name

The name of the rule to be changed. If Name is an empty string (""), all TopLevel and Active rules in the grammar will be changed.

State

The rule state to which the rule or rules will be changed.

Return Value

None.

Example

The following Visual Basic form code demonstrates the use of the Rules method, the CmdSetRuleState method and the CmdSetRuleIdState method. To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

The Form1_Load procedure creates a grammar and loads it with the solitaire grammar sol.xml, and uses the Rules method to create a collection of the rules contained in the grammar. The Command1_Click procedure creates an ISpeechGrammarRule object for the first rule contained in the grammar, and inactivates this rule using the CmdSetRuleState method and the rule's Name property. The procedure then creates an ISpeechGrammarRule object for the second rule, and inactivates that rule using the CmdSetRuleIdState method and the rule's ID property.

```
Option Explicit
```

```
Dim C As SpeechLib.SpSharedRecoContext  
Dim G As SpeechLib.ISpeechRecoGrammar  
Dim R As SpeechLib.ISpeechGrammarRules
```

```
Dim Rule As SpeechLib.ISpeechGrammarRule
```

```
Private Sub Command1_Click()
```

```
    'Get first rule in rules collection and set it inactive  
    'Use CmdSetRuleState method and the rule NAME
```

```
    Set Rule = R.Item(0)  
    G.CmdSetRuleState Rule.Name, SGDSInactive
```

```
    'Get next rule in rules collection and set it inactive,  
    'Use CmdSetRuleIdState method and the rule ID
```

```
    Set Rule = R.Item(1)  
    G.CmdSetRuleIdState Rule.Id, SGDSInactive
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set C = New SpSharedRecoContext
```

```
Set G = C.CreateGrammar
'Load the Solitaire grammar so it can be changed
Call G.CmdLoadFromFile("c:\sol.xml", SLODynamic)
Set R = G.Rules      'Get the collection of rules
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

DictationLoad Method

The **DictationLoad** method loads a dictation topic into the grammar.

```
ISpeechRecoGrammar.DictationLoad(  
    [TopicName As String = ""],  
    [LoadOption As SpeechLoadOption = SLOStatic]  
)
```

Parameters

TopicName

[Optional] Specifies a dictation topic. The default value is the empty string.

LoadOption

[Optional] Specifies whether the grammar is to be loaded for static or dynamic use. The default is static.

Return Value

None.

Remarks

SAPI currently defines one specialized dictation topic: SPTOPIC_SPELLING. SR engines are not required to support specialized dictation topics (including spelling). When using another manufacturer's SR engine, consult its documentation

for details.

Example

The following Visual Basic form code demonstrates the use of the DictationLoad, DictationSetState, and DictationUnload methods. It creates a grammar, configures the grammar to perform both dictation and command and control (C and C) recognition, and toggles between the two types of recognition.

To run this code, create a form with a command button called Command1 and paste this code into the Declarations section of the form. The Form_Load procedure creates a grammar object, associates it with the system dictation lexicon and the Solitaire C and C grammar, and begins recognition in dictation mode. The Command1_Click procedure toggles the recognition mode between dictation and C and C. The Form_Unload procedure unloads the dictation grammar and inactivates the C and C grammar.

```
Option Explicit
```

```
Dim C As SpeechLib.SpSharedRecoContext  
Dim G As SpeechLib.ISpeechRecoGrammar
```

```
Private Sub Command1_Click()
```

```
    If Command1.Caption = "&Dictation;" Then  
        G.CmdSetRuleIdState 0, SGDSInactive      'C&C; off  
        G.DictationSetState SGDSActive          'Dictation on  
        Command1.Caption = "&C; and C"  
    Else  
        G.DictationSetState SGDSInactive        'Dictation off  
        G.CmdSetRuleIdState 0, SGDSInactive    'C&C; on  
        Command1.Caption = "&Dictation;"  
    End If
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
'Create a RecoContext and its Grammar
Set C = New SpSharedRecoContext
Set G = C.CreateGrammar

'Get dictation grammar and set it inactive
G.DictationLoad "", SLOStatic
G.DictationSetState SGDSInactive

'Get Command & Control grammar, and set it inactive
G.CmdLoadFromFile "C:\SOL.XML", SLOStatic
G.CmdSetRuleIdState 0, SGDSInactive

'Set dictation active and set up Command1.Caption
G.DictationSetState SGDSActive
Command1.Caption = "&C; and C"

End Sub

Private Sub Form_Unload(Cancel As Integer)
    G.DictationUnload
    G.CmdSetRuleIdState 0, SGDSInactive
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

DictationSetState Method

The **DictationSetState** method sets the dictation topic state.

```
ISpeechRecoGrammar.DictationSetState(  
    State As SpeechRuleState  
)
```

Parameters

State

A `SpeechRuleState` constant that specifies the dictation topic state.

Return Value

None.

Example

The following Visual Basic form code demonstrates the use of the `DictationLoad`, `DictationSetState`, and `DictationUnload` methods. It creates a grammar, configures the grammar to perform both dictation and command and control (C and C) recognition, and toggles between the two types of recognition.

To run this code, create a form with the following control:

- A command button called `Command1`

Paste this code into the Declarations section of the form.

The `Form_Load` procedure creates a grammar object, associates

it with the system dictation lexicon and the Solitaire C and C grammar, and begins recognition in dictation mode. The Command1_Click procedure toggles the recognition mode between dictation and C and C. The Form_Unload procedure unloads the dictation grammar and deactivates the C and C grammar.

Option Explicit

```
Dim C As SpeechLib.SpSharedRecoContext
Dim G As SpeechLib.ISpeechRecoGrammar
```

```
Private Sub Command1_Click()
```

```
    If Command1.Caption = "&Dictation;" Then
        G.CmdSetRuleIdState 0, SGDSInactive      'C&C; off
        G.DictationSetState SGDSActive          'Dictation on
        Command1.Caption = "&C; and C"
    Else
        G.DictationSetState SGDSInactive        'Dictation off
        G.CmdSetRuleIdState 0, SGDSInactive      'C&C; on
        Command1.Caption = "&Dictation;"
    End If
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    'Create a RecoContext and its Grammar
    Set C = New SpSharedRecoContext
    Set G = C.CreateGrammar

    'Get dictation grammar and set it inactive
    G.DictationLoad "", SLOStatic
    G.DictationSetState SGDSInactive

    'Get Command & Control grammar, and set it inactive
    G.CmdLoadFromFile "C:\SOL.XML", SLOStatic
    G.CmdSetRuleIdState 0, SGDSInactive

    'Set dictation active and set up Command1.Caption
```

```
G.DictationSetState SGDSActive  
Command1.Caption = "&C; and C"
```

```
End Sub
```

```
Private Sub Form_Unload(Cancel As Integer)  
    G.DictationUnload  
    G.CmdSetRuleIdState 0, SGDSInactive  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

DictationUnload Method

The **DictationUnload** method unloads the active dictation topic from the grammar.

ISpeechRecoGrammar.**DictationUnload()**

Parameters

None.

Return Value

None.

Example

The following Visual Basic form code demonstrates the use of the DictationLoad, DictationSetState, and DictationUnload methods. It creates a grammar, configures the grammar to perform both dictation and command and control (C and C) recognition, and toggles between the two types of recognition.

To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a grammar object, associates it with the system dictation lexicon and the Solitaire C and C grammar, and begins recognition in dictation mode. The Command1_Click procedure toggles the recognition mode

between dictation and C and C. The Form_Unload procedure unloads the dictation grammar and deactivates the C and C grammar.

Option Explicit

```
Dim C As SpeechLib.SpSharedRecoContext
Dim G As SpeechLib.ISpeechRecoGrammar
```

```
Private Sub Command1_Click()
```

```
    If Command1.Caption = "&Dictation;" Then
        G.CmdSetRuleIdState 0, SGDSInactive      'C&C; off
        G.DictationSetState SGDSActive          'Dictation on
        Command1.Caption = "&C; and C"
    Else
        G.DictationSetState SGDSInactive        'Dictation off
        G.CmdSetRuleIdState 0, SGDSInactive     'C&C; on
        Command1.Caption = "&Dictation;"
    End If
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    'Create a RecoContext and its Grammar
    Set C = New SpSharedRecoContext
    Set G = C.CreateGrammar

    'Get dictation grammar and set it inactive
    G.DictationLoad "", SLOStatic
    G.DictationSetState SGDSInactive

    'Get Command & Control grammar, and set it inactive
    G.CmdLoadFromFile "C:\SOL.XML", SLOStatic
    G.CmdSetRuleIdState 0, SGDSInactive

    'Set dictation active and set up Command1.Caption
    G.DictationSetState SGDSActive
    Command1.Caption = "&C; and C"
```

End Sub

Private Sub Form_Unload(Cancel As Integer)

 G.DictationUnload

 G.CmdSetRuleIdState 0, SGDSInactive

End Sub

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

Id Property

The **Id** property returns the ID assigned to the grammar when it was created.

Syntax

Set: Not available.

Get: *Variant* = *ISpeechRecoGrammar.Id*

Parts

ISpeechRecoGrammar
The owning object.

Variant

Set: (This property is read-only).

Get: A Variant variable that gets the property value.

Example

The following Visual Basic form code demonstrates the use of the ID and RecoContext properties. To run this code, create a form with the following controls:

- Two command buttons called Command1 and Command2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a recognition context with two ISpeechRecoGrammar objects. The ISpeechRecoGrammar objects are created with ID properties of 6 and 7. The Command1 and Command2 procedures each send one of the grammar objects to the GrammarSub procedure as parameters.

The GrammarSub procedure displays the ID of the grammar object parameter, and uses the parameter's RecoContext property to pause and resume the recognition context that owns the grammar.

There is no special significance to the Id property values of 6 or 7; these values are arbitrary. Additionally, the Pause and Resume methods in the GrammarSub procedure are intended simply to show how the grammar's RecoContext property provides access to the methods and properties of the recognition context which owns the grammar.

Option Explicit

```
Dim RecoContext As SpeechLib.SpSharedRecoContext
Dim objGR1 As SpeechLib.ISpeechRecoGrammar
Dim objGR2 As SpeechLib.ISpeechRecoGrammar
```

```
Private Sub Command1_Click()
    Call GrammarSub(objGR1)
End Sub
```

```
Private Sub Command2_Click()
    Call GrammarSub(objGR2)
End Sub
```

```
Private Sub Form_Load()
```

```
    Set RecoContext = New SpSharedRecoContext
    Set objGR1 = RecoContext.CreateGrammar(6)           'Create Grammar 6
    Set objGR2 = RecoContext.CreateGrammar("7")       'Create Grammar 7
```

```
End Sub
```

```
Private Sub GrammarSub(G As SpeechLib.ISpeechRecoGrammar)
```

```
    G.RecoContext.Pause           'Pause the RecoContext that owns
    MsgBox G.Id                    'Display the Id
    G.RecoContext.Resume          'Resume the RecoContext
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

IsPronounceable Method

The **IsPronounceable** method determines if a word has a pronunciation.

Additionally, the `SpeechWordPronounceable` constant returned by this method indicates whether the word exists in the grammar object's lexicon. Words are likely to be pronounceable even if they are not found in the lexicon.

```
ISpeechRecoGrammar.IsPronounceable(  
    Word As String  
) As SpeechWordPronounceable
```

Parameters

Word

Specifies the Word.

Return Value

A `SpeechWordPronounceable` constant.

Example

The following Visual Basic form code demonstrates the use of the `IsPronounceable` method.

To run this code, create a form with the following controls:

- A command button called `Command1`
- A text box called `Text1`

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a grammar and loads the general dictation topic. The Command1_Click procedure passes the word or words in Text1 to the IsPronounceable method and displays the resulting SpeechWordPronounceable constant.

Most correctly spelled words will be known and pronounceable; most incorrectly spelled words will be unknown and pronounceable. The example begins with a word which is pronounceable, even though it is misspelled.

```
Option Explicit
```

```
Dim C As SpeechLib.SpSharedRecoContext  
Dim G As SpeechLib.ISpeechRecoGrammar
```

```
Private Sub Command1_Click()  
    Dim strTemp As String
```

```
    Select Case G.IsPronounceable(Text1.Text)  
    Case SWPKnownWordPronounceable  
        strTemp = "KnownWordPronounceable"  
    Case SWPUnknownWordPronounceable  
        strTemp = "UnknownWordPronounceable"  
    Case SWPUnknownWordUnpronounceable  
        strTemp = "UnknownWordUnpronounceable"  
    End Select
```

```
    MsgBox "The word "" & Text1.Text & "" is " & strTemp
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set C = New SpSharedRecoContext  
    Set G = C.CreateGrammar
```

```
    G.DictationLoad ""
```

```
    Text1.Text = "missspeled"
```

End Sub

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

RecoContext Property

The **RecoContext** property returns the RecoContext object that created the grammar.

Syntax

Set: Not available.

Get: ***ISpeechRecoContext*** =
ISpeechRecoGrammar.RecoContext

Parts

ISpeechRecoGrammar
The owning object.

ISpeechRecoContext

Set: (This property is read-only).

Get: An *ISpeechRecoContext* object which instantiates the RecoContext object that created the grammar.

Example

The following Visual Basic form code demonstrates the use of the *Id* and *RecoContext* properties. To run this code, create a form with the following controls:

- Two command buttons called *Command1* and *Command2*

Paste this code into the Declarations section of the form.

The *Form_Load* procedure creates a recognition context with two *ISpeechRecoGrammar* objects. The *ISpeechRecoGrammar* objects are created with *Id* properties of 6 and 7. The

Command1 and Command2 procedures each send one of the grammar objects to the GrammarSub procedure as parameters. The GrammarSub procedure displays the ID of the grammar object parameter, and uses the parameter's RecoContext property to pause and resume the recognition context that owns the grammar.

There is no special significance to the Id property values of 6 or 7; these values are arbitrary. Additionally, the Pause and Resume methods in the GrammarSub procedure are intended simply to show how the grammar's RecoContext property provides access to the methods and properties of the recognition context that created the grammar.

Option Explicit

```
Dim RecoContext As SpeechLib.SpSharedRecoContext
Dim objGR1 As SpeechLib.ISpeechRecoGrammar
Dim objGR2 As SpeechLib.ISpeechRecoGrammar
```

```
Private Sub Command1_Click()
    Call GrammarSub(objGR1)
End Sub
```

```
Private Sub Command2_Click()
    Call GrammarSub(objGR2)
End Sub
```

```
Private Sub Form_Load()
```

```
    Set RecoContext = New SpSharedRecoContext
    Set objGR1 = RecoContext.CreateGrammar(6)           'Create (
    Set objGR2 = RecoContext.CreateGrammar("7")       'Create (
```

```
End Sub
```

```
Private Sub GrammarSub(G As SpeechLib.ISpeechRecoGrammar)
```

```
    G.RecoContext.Pause           'Pause the RecoContext that owns
    MsgBox G.Id                    'Display the Id
```

G.RecoContext.Resume 'Resume the RecoContext

End Sub

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

Reset Method

The **Reset** method clears all grammar rules and resets the grammar's language to *NewLanguage*.

```
ISpeechRecoGrammar.ResetGrammar(  
    [NewLanguage As Long = 0]  
)
```

Parameters

NewLanguage

[Optional] Specifies the ID of the new language. The default value is zero.

Return Value

None.

Example

For an example of the use of the Reset method, see the [CmdLoadFromMemory](#) method.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

Rules Property

The **Rules** property returns the collection of grammar rules contained in the RecoGrammar.

Syntax

Set: Not available.

Get: ***ISpeechGrammarRules*** = *ISpeechRecoGrammar*.**Rules**

Parts

ISpeechRecoGrammar

The owning object.

ISpeechGrammarRules

Set: (This property is read-only).

Get: An *ISpeechGrammarRules* variable which contains the grammar's rules.

Example

The following Visual Basic form code demonstrates the use of the Rules method, the CmdSetRuleState method and the CmdSetRuleIdState method. To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

The Form1_Load procedure creates a grammar and loads it with the solitaire grammar sol.xml, and uses the Rules method to create a collection of the rules contained in the grammar. The

Command1_Click procedure creates an ISpeechGrammarRule object for the first rule contained in the grammar, and deactivates this rule using the CmdSetRuleState method and the rule's Name property. The procedure then creates an ISpeechGrammarRule object for the second rule, and deactivates that rule using the CmdSetRuleIdState method and the rule's Id property.

Option Explicit

```
Dim C As SpeechLib.SpSharedRecoContext
Dim G As SpeechLib.ISpeechRecoGrammar
Dim R As SpeechLib.ISpeechGrammarRules
```

```
Dim Rule As SpeechLib.ISpeechGrammarRule
```

```
Private Sub Command1_Click()
```

```
    'Get first rule in rules collection and set it inactive
    'Use CmdSetRuleState method and the rule NAME
```

```
    Set Rule = R.Item(0)
    G.CmdSetRuleState Rule.Name, SGDSInactive
```

```
    'Get next rule in rules collection and set it inactive,
    'Use CmdSetRuleIdState method and the rule ID
```

```
    Set Rule = R.Item(1)
    G.CmdSetRuleIdState Rule.Id, SGDSInactive
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set C = New SpSharedRecoContext
    Set G = C.CreateGrammar
```

```
    'Load the Solitaire grammar so it can be changed
```

```
    Call G.CmdLoadFromFile("c:\sol.xml", SLODynamic)
```

```
    Set R = G.Rules      'Get the collection of rules
```

End Sub

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

SetTextSelection Method

The **SetTextSelection** method sets the range of text selection information in a WordSequenceData buffer.

The [SetWordSequenceData](#) method sends application-specific data to the speech recognition (SR) engine for recognition. The SetTextSelection method describes the part of that text which has been selected by the user with a mouse or keyboard.

```
ISpeechRecoGrammar.SetTextSelection(  
    Info As SpTextSelectionInformation  
)
```

Parameters

Info

An SpTextSelectionInformation object which specifies the text selection range.

Return Value

None.

Example

For an example of the use of the SetTextSelection method, see the [SetWordSequenceData](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

SetWordSequenceData Method

The **SetWordSequenceData** method defines a word sequence buffer for use by the speech recognition (SR) engine.

Some recognition grammars specify every word and phrase that they are capable of recognizing. The Solitaire grammar is one of these; it contains only a small amount of data, and all its data elements were known to the grammar designer. But it would be impractical to create such a grammar for a speech-enabled order-entry system handling several hundred thousand inventory items.

In order to eliminate the need to specify every recognizable word in a grammar, yet still maintain the high quality of rule-based recognition, SAPI provides applications with a means to link the grammar to application-specific and user-specific data. An order-entry application could display the text for several inventory items on a user's monitor, and send this text to the engine for recognition. When a user spoke a part number, the engine would more easily recognize it from the few words on the user's screen than from thousands of part numbers specified in a grammar.

To accomplish this, the SR engine maintains a text buffer which is associated with the XML grammar tag <TEXTBUFFER>. When the recognition process arrives at a <TEXTBUFFER> tag, it expects that the application has placed text in this buffer, and that the user's speech represents one word or phrase out of the words and phrases in the buffer. If recognition is successful, the words or phrases selected from the buffer are recognized as if they had been specified in the grammar rule literally.

The SetWordSequenceData method sends this text to the engine's buffer. It is associated with the [SetTextSelection](#) method, which describes the range of text the user has selected with the mouse. The SetWordSequenceData method always

sends selection data with the text data, because changing the text in the buffer invalidates the previous selection data. The `SetTextSelection` method sends selection data only.

```
ISpeechRecoGrammar.SetWordSequenceData(  
    Text As String,  
    TextLength As Long,  
    Info As SpTextSelectionInformation  
)
```

Parameters

Text

Specifies the text.

TextLength

Specifies the length of the text.

Info

An `SpTextSelectionInformation` object which specifies the text selection range.

Return Value

None.

Example

The following Visual Basic form code demonstrates the use of the `SetWordSequenceData` and `SetTextSelection` methods. To run this code, create a form with the following controls:

- A text box called Text1
- A list box called List1
- A command button called Command1

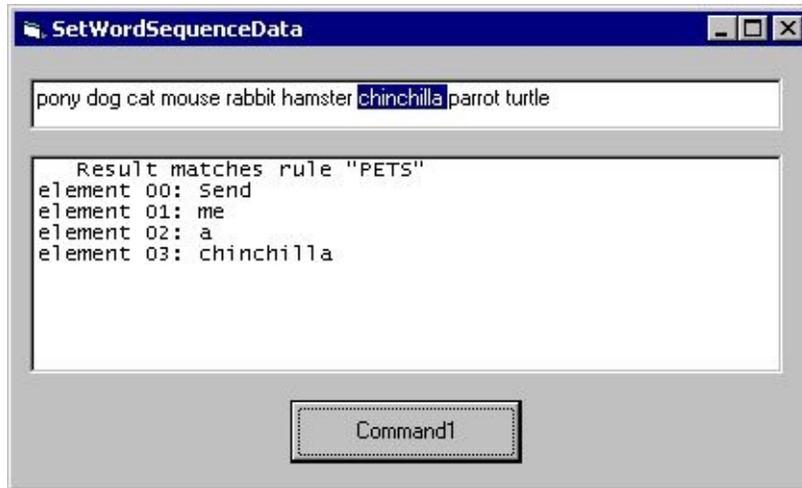
Paste this code into the Declarations section of the form.

Because selecting data in the text box is an important visual part of this example, the HideSelection property of the text box should be set to False.

The Form_Load procedure creates a recognizer, a recognition context, and a grammar object. It fills the text box with the names of several types of animals. It writes an XML grammar file which contains a grammar rule using a text tag. The rule expects a sentence like "Send me a *," where "*" is a type of animal listed in the text box.

Double-click on an animal type in the text box to select it, and then click Command1. The Command1 procedure gets the selected animal type and builds a sentence such as, "Send me a hamster," or "Send me a chinchilla." If no animal type is selected in the text box, the sample uses the first animal listed. The procedure then speaks the sentence into a wave file and sends the file to the recognition context for recognition.

The Recognition event procedure displays some of the recognition data in the list box. It first determines if the recognition result satisfied a grammar rule; if so, it displays the name of the rule. In this case, PETS is the only rule that can be satisfied. The procedure then displays the individual phrase elements of the sentence.



Option Explicit

```
Const WAVEFILENAME = "C:\SetWordSequenceData.wav"
Const XMLFILENAME = "c:\texttag.xml"
```

```
Dim R As SpeechLib.SpInprocRecognizer
Dim G As SpeechLib.ISpeechRecoGrammar
Dim F As SpeechLib.SpFileStream
Dim E As SpeechLib.ISpeechPhraseElement
Dim V As SpeechLib.SpVoice
Dim V2 As SpeechLib.SpVoice      'Plays the wave file back
```

```
Dim WithEvents C As SpeechLib.SpInProcRecoContext
Dim TSI As SpeechLib.SpTextSelectionInformation
```

```
Private Sub WriteGrammar(strFName)
    Open strFName For Output As #1
    Print #1, "<GRAMMAR>"
    Print #1, " <RULE NAME=""PETS"" TOPLEVEL=""ACTIVE"">"
    Print #1, "   <O>please</O>"
    Print #1, "   <P>send me a</P>"
    Print #1, "   <TEXTBUFFER/>"      '<-- Calls for WordSequel
    Print #1, " </RULE>"
    Print #1, "</GRAMMAR>"
    Close #1
End Sub
```

```
Private Sub SetTextSelection(T As Control)
    TSI.ActiveOffset = 0                'Start of text
```

```

        TSI.ActiveLength = Len(T.Text)           'Length of text
        TSI.SelectionOffset = T.SelStart + 1    'Start of selection
        TSI.SelectionOffset = T.SelLength      'Length of selection
        G.SetTextSelection TSI                 'Send text-selection data
    End Sub

```

```

Private Sub SetWordSequenceData(T As Control)
    Call SetTextSelection(T)                   'Set up selection
    G.SetWordSequenceData T.Text, Len(T.Text), TSI 'Send the data
End Sub

```

```

Private Sub SpeakToFile(ByVal strText As String, ByVal strFileName As String)
    Set F = New SpFileStream                   'Create stream
    F.Open strFileName, SSFMCreatForWrite, True 'Open as the writer
    Set V.AudioOutputStream = F               'Set voice output stream
    V.Speak strText, SVSFIisXML                'Speak synchronously
    F.Close                                     'Close file
End Sub

```

```

Private Sub Command1_Click()
    Dim W As String

    'Get selected word for voice to speak
    If Text1.SelLength Then
        W = Mid(Text1.Text, Text1.SelStart + 1, Text1.SelLength)
    Else
        W = "pony"
    End If
    W = "send me a " & W 'Voice speaks this string

    'Send buffer text and selection data to engine
    Call SetWordSequenceData(Text1)

    List1.Clear
    Call SpeakToFile(W, WAVEFILENAME)
    F.Open WAVEFILENAME
    Set R.AudioInputStream = F

End Sub

```

```

Private Sub Form_Load()

```

```

' Create Recognizer, RecoContext, Grammar, and Voice
Set R = New SpInprocRecognizer
Set C = R.CreateRecoContext
Set G = C.CreateGrammar()
Set V = New SpVoice
Set V.Voice = V.GetVoices("gender=male").Item(0)
Set V2 = New SpVoice
Set TSI = New SpeechLib.SpTextSelectionInformation

'Write a grammar with a <TEXTTAG> transition, then use it
Call WriteGrammar(XMLFILENAME)
G.CmdLoadFromFile XMLFILENAME, SLODynamic
G.CmdSetRuleIdState 0, SGDSActive           'Set C &
G.DictationSetState SGDSActive             'Set Dic

Text1.Text = "pony dog cat mouse rabbit hamster chinchil.
End Sub

Private Sub C_Recognition(ByVal StreamNumber As Long, _
                        ByVal StreamPosition As Variant, _
                        ByVal RecognitionType As SpeechLib.Speech
                        ByVal Result As SpeechLib.ISpeechRecoRes

Dim X As String
Dim ii As Integer

If Not Result.PhraseInfo.Rule.Name = "" Then
    List1.AddItem "    Result matches rule "" & Result.Pl
End If

ii = 0
For Each E In Result.PhraseInfo.Elements
    X = "element " & Format(ii, "00") & ": " & E.Display
    List1.AddItem X
    ii = ii + 1
Next

End Sub

Private Sub C_EndStream(ByVal StreamNumber As Long, _
                        ByVal StreamPosition As Variant, _
                        ByVal StreamReleased As Boolean)

F.Close

```

```
DoEvents
F.Open WAVEFILENAME
V2.SpeakStream F
F.Close
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoGrammar](#)

State Property

The **State** property gets and sets the operational status of the speech grammar.

The status consists of a `SpeechGrammarState` constant; its three states can be summarized as enabled, disabled, and exclusive.

Syntax

```
Set: ISpeechRecoGrammar.State = SpeechGrammarState
```

```
Get: SpeechGrammarState = ISpeechRecoGrammar.State
```

Parts

ISpeechRecoGrammar
The owning object.

SpeechGrammarState

Set: A `SpeechGrammarState` constant that sets the property.

Get: A `SpeechGrammarState` constant that gets the property.

Example

The following code snippet demonstrates the use of the `State` property. The grammar is disabled while the `CmdLoadFromFile` method loads the grammar, and enabled when the load has completed.

```
Dim C As SpeechLib.SpSharedRecoContext  
Dim G As SpeechLib.ISpeechRecoGrammar
```

```
Set C = New SpSharedRecoContext  
Set G = C.CreateGrammar()
```

```
'  
'  
'
```

```
G.State = SGSDisabled      'Disable grammar while loading
```

```
G.CmdLoadFromFile ("c:\sol.xml")
```

```
G.State = SGSEnabled      'Re-enable when done loading
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechRecoResult

The **ISpeechRecoResult** automation interface returns information about a recognition attempt.

A recognition result is returned by a recognition context in the three following cases:

- A successful recognition
- An intermediate recognition (also called a hypothesis)
- An unsuccessful recognition (or a false recognition)

A successful recognition is a word or phrase that surpasses a predetermined confidence rating. It is considered to be accurate enough to be passed back to the user as the text that was actually spoken. A hypothesis is an intermediate step toward recognition. The text has been parsed and examined and is available to the user for closer examination. Any number of hypotheses may be produced during a recognition attempt. A hypothesis may not reflect the final recognition and should not be used to predict it. A false recognition is a recognized word or phrase that does not meet or exceed a predetermined confidence rating. The false recognition will still contain a valid recognition result including text representing the speech. However, the text was not able to meet confidence criteria. Any of the following can contribute to a false recognition:

- Background noise
- Inexact pronunciation
- Uncommon words
- Unusual sequence of words

Any one of the three recognition types above is treated the same for a recognition result. A valid recognition result is returned by SAPI and its content may be examined. Information includes the phrase itself, the owning recognition context, the

audio format (if the audio was retained) and other properties in this class.

Automation Interface Elements

The ISpeechRecoResult automation interface contains the following elements:

Properties	Description
AudioFormat Property	Gets or sets the audio stream format.
PhraseInfo Property	Returns an ISpeechPhraseInfo structure containing detailed information about the last recognized phrase.
RecoContext Property	Retrieves the current ISpeechRecoContext for the recognizer.
Times Property	Retrieves the time information associated with the result.

Methods	Description
Alternates Method	Returns a list of alternative words.
Audio Method	Creates an audio stream from the audio data in the result object.
DiscardResultInfo Method	Discards the requested data from a phrase object.
SaveToMemory Method	Saves the phrase portion of the recognition result to memory.
SpeakAudio Method	Plays the audio sequence containing the recognized phrase.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResult](#)

Alternates Method

The **Alternates** method returns a list of alternate words.

Many recognitions, successful or not, frequently return several words or phrases that closely match the spoken sequence. The one most nearly matching the sequence with a high confidence level is returned as a successful recognition. The other words and phrases are returned as alternates that are available for examination.

```
ISpeechRecoResult.Alternates(  
    RequestCount AS Long,  
    [StartElement AS Long = 0],  
    [Elements AS Long = -1]  
) AS ISpeechPhraseAlternates
```

Parameters

RequestCount

Specifies the maximum number of alternates to retrieve, which must be greater than zero. Any number of alternates may be chosen and will be returned in descending order of confidence. That is, the first alternate returned has the highest confidence and will most likely be the word or phrase chosen by the successful recognition. The second alternate returned will be the next most likely choice, and the last alternate returned the least likely match.

StartElement

[Optional] Specifies which element to use as a starting point. If omitted, zero is used and indicates the first alternate as the starting point. Because it is zero based, the second element would be one.

Elements

[Optional] Specifies the number of elements to retrieve. Default is -1, which specifies all alternate elements are retrieved.

Return Value

The Alternates method returns an *ISpeechPhraseAlternates* variable.

Remarks

ISpeechRecoResult.Alternates applies only to dictation grammar. Command and control alternates are handled separately and independently. See [ISpeechRecoContext.CmdMaxAlternates](#) for command and control alternates.

Example

The following example returns the 30 most likely matches for a recognition. It assumes a successful recognition in *RecoResult*. The example also only uses one parameter, the number of alternates to return; the other parameters are assigned by default and return all the available words and phrases.

```
Dim phraseAlternate As ISpeechPhraseAlternates
Dim i As Long
Dim theString As String

Set phraseAlternate = RecoResult.Alternates(30)
For i = 0 To phraseAlternate.Count - 1
    theString = phraseAlternate.Item(i).GetText
    TextField.Text = TextField.Text + "Alternates #" + s
Next i
```

If the recognition was "we the people," then the first six alternates might look like the following. Notice that the first alternate is the recognized word or phrase.

Alternates #0: we the people
Alternates #1: we have people
Alternates #2: we people
Alternates #3: we do people
Alternates #4: we had people
Alternates #5: we can people

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResult](#)

Audio Method

The **Audio** method creates an audio stream from the audio data in the result object.

The resulting stream can be used as input for *SPVoice.SpeakStream*. However, a single call to *ISpeechRecoResults.SpeakAudio* performs the same action of speaking the recognized result.

The audio portion of the recognition is not automatically available. By default, the recognition context does not retain audio. To retain it, call the *RetainedAudio* property and set it to *SRAORetainAudio*. While *RetainedAudio* can be toggled at anytime, audio for a specific phrase must be retained before recognition attempts begin. Therefore, *RetainedAudio* must be called before the phrase is spoken.

```
ISpeechRecoResult.Audio(  
    [StartElement As Long = 0],  
    [Elements As Long = -1]  
) As ISpeechBaseStream
```

Parameters

StartElement

[Optional] Specifies the starting element. The default value is zero, indicating the first element is used.

Elements

[Optional] Specifies the number of elements to speak. Default value is -1, indicating all elements are to be played.

Return Value

The Audio method returns an ISpeechBaseStream stream.

Remarks

Even if there are no elements, that is, *StartElement* = 0 and *Elements* = 0, the audio will still be played. Unrecognized results not having elements will still have audio.

Example

The following code snippet assumes a valid *RecoResult*. The stream is returned and passed to *SPVoice.SpeakStream* to hear. The code also shows support for RetainedAudio in the current recognizer. As a note, the voice object is created using a late binding method.

```
Dim WithEvents RecoResult As SpSharedRecoContext
Set RecoResult = New SpSharedRecoContext
RecoResult.RetainedAudio = SRA0RetainAudio

'Demonstrates retrieving the RetainedAudio value
Dim audioOption As SpeechRetainedAudioOptions
audioOption = RecoResult.RetainedAudio

'Rest of the code to process recognition goes here
...

'Gets the stream to speak
Dim stream As ISpeechBaseStream
Set stream = RecoResult.Audio

'Creates the voice
Dim Voice As SpVoice
Set Voice = CreateObject("SAPI.SpVoice")
Voice.SpeakStream stream
```



Object: [ISpeechRecoResult](#)

AudioFormat Property

The **AudioFormat** property gets or sets the audio stream format.

The controlling recognition context must retain the audio portion of the recognition. By default, a recognition context does not retain audio; that is, *RecoContext.RetainedAudio* is set to [SRAONone](#). Attempts to access this retained audio stream, including references to *AudioFormat*, cause an SPERR_NO_AUDIO_DATA error. To retain the audio, use [ISpeechRecoContext.RetainedAudio](#) passing [SRAORetainAudio](#) as the parameter.

Syntax

Get:	SpAudioFormat = <i>ISpeechRecoResult</i> . AudioFormat
Set:	<i>ISpeechRecoResult</i> . AudioFormat = SpAudioFormat

Parts

ISpeechRecoResult

The owning object.

SpAudioFormat

An object variable representing an audio output device.

Get: The token represents the current audio output device of the voice.

Set: The token represents the audio output device assigned to the voice.

In either case, the format for *SpAudioFormat.Type* is of type [SpeechAudioFormatType](#).

Example

The following code snippet demonstrates retrieving and setting the audio format.

The following code snippets assume a valid *RecoResult*. The following code demonstrates setting the recognition context.

```
Set RC = New SpSharedRecoContext  
RC.RetainedAudio = SRAORetainAudio
```

```
'Get audio format  
Dim GetFormat as SpAudioFormat  
Set Format = RecoResult.AudioFormat
```

```
'Set audio format  
Dim SetFormat as SpAudioFormat  
Set SetFormat = CreateObject("SAPI.SpAudioFormat")
```

```
SetFormat.Type = SAFT11kHz16BitMono  
Set RecoResult.AudioFormat = SetFormat
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResult](#)

DiscardResultInfo Method

The **DiscardResultInfo** method discards the requested data from a phrase object.

Applications that have no use for certain types of retained data, and will be persisting or serializing the phrase or result objects, may discard unnecessary data.

For example, an application performing offline transcription may need to retain only the audio and the final result. It can remove the alternates with *object.DiscardResultInfo*(SPDF_ALTERNATES) to eliminate the alternate data (possibly including a large amount of private engine data). Once the result information is discarded, all attempts to access that data will be unsuccessful. For example, once retained audio has been discarded, a call to [ISpeechRecoResult.Audio](#) will fail.

```
ISpeechRecoResult.DiscardResultInfo(  
    ValueTypes As SpeechDiscardType  
)
```

Parameters

ValueTypes

Flags indicating elements to discard. Multiple values may be combined with logical operands.

Return Value

None.

Example

The following snippet demonstrates discarding the audio portion of the recognition. Retaining audio is explicitly set in the second

line. After a successful recognition and a valid *RecoResult*, the snippet speaks back the recognition. The retained audio is discarded immediately and the next attempt at speaking that audio fails with an SPERR_NO_AUDIO_DATA.

```
Set RecoResult = New SpSharedRecoContext  
RecoResult.RetainedAudio = SRAORetainAudio
```

'Speech processing code goes here

```
RecoResult.SpeakAudio  
RecoResult.DiscardResultInfo (SDTAudio)  
RecoResult.SpeakAudio
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResult](#)

PhraseInfo Property

The **PhraseInfo** property returns an `ISpeechPhraseInfo` structure containing detailed information about the last recognized phrase.

`ISpeechPhraseInfo` elements contains read-only data about timing for the phrase and audio stream, elements (words and phrases) in the recognized phrase, grammar and grammar rules information. The structure is used to examine the recognition information. It is also used by other calls such as `ISpeechRecoResult.PhraseInfo.GetText` to retrieve the phrase text.

Syntax

Set: (This property is read-only)

Get: [*ISpeechPhraseInfo*](#) = `ISpeechRecoResult.PhraseInfo`

Parts

ISpeechRecoResult

The owning object.

ISpeechPhraseInfo

Set: (This property is read-only)

Get: An `ISpeechPhraseInfo` variable that gets the property.

Example

The following code snippet assumes a previously defined `RecoResult`. If a completed recognition occurs, the code parses out selected information about stream times and element times.

Although the preferred method of retrieving the text uses *ISpeechRecoResult.GetText*, this snippet not only retrieves the text on an element by element basis, but also retrieves the stream time associated with the word.

```
Dim rString As String
Dim i As Integer
Dim rp As ISpeechPhraseInfo
Set rp = RecoResult.PhraseInfo

If Not RecoResult Is Nothing Then
    rString = rString + "LangID= " & rp.LanguageId & vbCrLf
    rString = rString + "AudioBytes= " & rp.AudioSizeBytes & vbCrLf
    rString = rString + "AudioTime= " & rp.AudioSizeTime & vbCrLf

    For i = 0 To rp.Elements.Count - 1
        rString = rString + "Stream offset:" & rp.Elements(i).StreamOffset & vbCrLf
        rString = rString + "Text form: " & rp.Elements(i).Text & vbCrLf
        rString = rString + "Lex form: " & rp.Elements(i).LexForm & vbCrLf
    Next
End If
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResult](#)

RecoContext Property

The **RecoContext** property retrieves the current *ISpeechRecoContext* for the recognizer.

Applications may have multiple recognition contexts open at the same time. *RecoContext* provides a means to determine which context owns the recognition. For example, in situations with more than recognition context, a recognition result may call *RecoContext* to get the context associated with it. The returned context may be changed afterward and will not affect other contexts.

Syntax

Set: (This property is read-only)

Get: *ISpeechRecoContext* = *ISpeechRecoResult.RecoContext*

Parts

ISpeechRecoResult

The owning object.

SpSharedRecoContext

Set: (This property is read-only)

Get: An *SpSharedRecoContext* variable that gets the property.

Example

The following snippet assumes a valid recognition *RecoResult*. If more than one recognition context exists, the one owning *RecoResult* is retrieved and the event interest is changed. No other recognition contexts are affected.

```
Dim myContext As ISpeechRecoContext
```

```
Set myContext = RecoResult.RecoContext()  
myContext.EventInterests = SREFalseRecognition
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResult](#)

SaveToMemory Method

The **SaveToMemory** method saves the entire recognition result to memory.

The phrase may be recalled at a later time. To retrieve the recognition result from memory use [SpSharedRecoContext.CreateResultFromMemory](#) or [SpInProcRecoContext.CreateResultFromMemory](#) depending on the recognition context used.

ISpeechRecoResult.**SaveToMemory()** As Variant

Parameters

None

Return Value

The SaveToMemory method returns a Variant containing a pointer to saved phrase.

Example

See [ISpeechRecoContext.CreateResultFromMemory.htm](#) for a complete code sample.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResult](#)

SpeakAudio Method

The **SpeakAudio** method plays the audio sequence containing the recognized phrase.

The audio portion of the recognition is not automatically available. By default, the recognition context does not retain audio. Call `RetainedAudio` and set it to `SRAORetainAudio` to retain the audio. Although `RetainedAudio` can be toggled at anytime, audio for a specific phrase must be retained before recognition attempts begin. Therefore, `RetainedAudio` must be called before the phrase is spoken.

`SpeakAudio` combines two other methods. The first, [ISpeechRecoResult.Audio](#), which retrieves the audio. The second call is [SpVoice.SpeakStream](#), which plays it back.

```
ISpeechRecoResult.SpeakAudio(  
    [StartElement As Long = 0],  
    [Elements As Long = -1],  
    [Flags As SpeechVoiceSpeakFlags = SVSFDefault]  
) As Long
```

Parameters

StartElement

[Optional] Specifies the starting element. The default value is zero, indicating that the first element is used.

Elements

[Optional] Specifies the number of elements to speak. Default value is -1, indicating that all elements are to be played.

Flags

[Optional] Specifies the Flags. Default value is SVSFDefault indicating that no special speak restrictions are imposed.

Return Value

The SpeakAudio method returns a Long variable indicating the stream number.

Remarks

Even if there are no elements, that is, StartElement = 0 and Elements = 0, the audio will still be played. Unrecognized results not having elements will still have audio.

Also see [ISpeechRecoContext.RetainedAudio](#) for an additional code sample.

Example

The following code snippet assumes a valid *RecoResult* and that *RetainedAudio* is set for the current recognizer. The first sample plays back the entire recognized phrase.

```
RecoResult.SpeakAudio
```

The next example demonstrates *SpeakAudio* speaking every other word in the phrase and skipping the others. The code also shows support for *RetainedAudio* in the current recognizer.

```
Dim WithEvents RecoResult As SpSharedRecoContext  
Dim theCount, i As Long
```

```
Set RecoResult = New SpSharedRecoContext  
RecoResult.RetainedAudio = SRAORetainAudio
```

```
'Rest of the code to process recognition goes here
```

```
'Get the number of phrase elements (words in the phrase) and  
'step through every other one.
```

```
theCount = RecoResult.PhraseInfo.Elements.Count
For i = 0 To theCount - 1 Step 2
    streamNumber = RecoResult.SpeakAudio(i, 1, SVSFDefault)
Next i
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResult](#)

Times Property

The **Times** property retrieves the time information associated with the result.

Syntax

Set: (This property is read-only)

Get: [ISpeechRecoResultTimes](#) = ISpeechRecoResult.**Times**

Parts

ISpeechRecoResult

The owning object.

ISpeechRecoResultTimes

Set: (This property is read-only)

Get: An ISpeechRecoResultTimes variable containing the time information for the recognition.

Example

The following snippet assumes a valid recognition *RecoResult*. *ISpeechRecoResultTimes* is retrieved for the current recognition. Since all the values are read-only, they are added to a string for possible display.

```
Dim myTimes As ISpeechRecoResultTimes
Dim rString As String
```

```
Set myTimes = RecoResult.Times()
rString = "Len= " & myTimes.Length & vbCrLf
rString = rString & "Start= " & myTimes.OffsetFromStart & vbCrLf
rString = rString & "Time= " & myTimes.StreamTime & vbCrLf
rString = rString & "TickCount= " & myTimes.TickCount
```

Alternatively, the same information may be retrieved directly from *RecoResult*.

```
Dim rString As String
```

```
rString = "Len= " & RecoResult.Times.Length & vbCrLf  
rString = rString & "Start= " & RecoResult.Times.OffsetFromS  
rString = rString & "Time= " & RecoResult.Times.StreamTime &  
rString = rString & "TickCount= " & RecoResult.Times.TickCou
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechRecoResultTimes

The **ISpeechRecoResultTimes** automation interface contains the time information for speech recognition results.

The [ISpeechRecoResultTimes](#) property returns an ISpeechRecoResultTimes object.

Automation Interface Elements

The ISpeechRecoResultTimes automation interface contains the following elements:

Properties	Description
Length Property	Returns the time length of the last recognition.
OffsetFromStart Property	Returns the time from the start of the stream to the start of the phrase.
StreamTime Property	Returns the time of the stream in Universal Coordinated Time.
TickCount Property	Returns the elapsed time from the start of the system to the start of the current result.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResultTimes](#)

Length Property

The **Length** property returns the time length of the last recognition.

It is specified in 100 nanosecond units.

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechRecoResultTimes*.**Length**

Parts

ISpeechRecoResultTimes

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant type containing the length of the recognition.

Example

When a recognition occurs with a valid *RecoResult*, the times can be extracted for later use to allocate memory or to select data from the audio stream. For example, using the phrase "variety is the spice of life," the Length could be 32,200,000 nanoseconds or 3.2 seconds.

The first example retrieves the recognition length using the Length property.

```
Dim myTimes As ISpeechRecoResultTimes  
Set myTimes = RecoResult.Times
```

```
Dim recoLength As Variant  
recoLength = myTimes.Length
```

The second example retrieves the recognition length by directly accessing the *RecoResult* structure.

```
Dim recoLength As Variant  
recoLength = RecoResult.Times.Length
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResultTimes](#)

OffsetFromStart Property

The **OffsetFromStart** property returns the time from the start of the stream to the start of the phrase.

It is specified in 100 nanosecond units.

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechRecoResultTimes*.**OffsetFromStart**

Parts

ISpeechRecoResultTimes

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant type containing time offset from the start of the stream.

Example

When a recognition occurs with a valid *RecoResult*, the times can be extracted in order to select data at a later time. For example, using the phrase "variety is the spice of life," the **OffsetFromStart** position could be 21120 if the user started speaking relatively quickly once the stream began.

The first example retrieves the recognition offset using the **OffsetFromStart** property.

```
Dim myTimes As ISpeechRecoResultTimes
Set myTimes = RecoResult.Times
```

```
Dim offsetPosition As Variant  
offsetPosition = myTimes.OffsetFromStart
```

The second example retrieves the recognition offset by directly accessing the *RecoResult* structure.

```
Dim offsetPosition As Variant  
offsetPosition = RecoResult.Times.OffsetFromStart
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResultTimes](#)

StreamTime Property

The **StreamTime** property returns the time of the stream in Universal Coordinated Time.

This value is zero if a wave file input is used.

Syntax

Set: (This property is read-only)

Get: *Variant* = *ISpeechRecoResultTimes*.**StreamTime**

Parts

ISpeechRecoResultTimes

The owning object.

Variant

Set: (This property is read-only)

Get: A Variant type containing the Universal Coordinated Time time of the phrase start for the stream.

Example

When a recognition occurs with a valid *RecoResult*, the StreamTime can be extracted.

The first example retrieves the recognition stream time using the StreamTime property.

```
Dim myTimes As ISpeechRecoResultTimes  
Set myTimes = RecoResult.Times
```

```
Dim streamTime As Variant  
streamTime = myTimes.StreamTime
```

The second example retrieves the recognition stream time by directly accessing the *RecoResult* structure.

```
Dim streamTime As Variant  
streamTime = RecoResult.Times.StreamTime
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoResultTimes](#)

TickCount Property

The **TickCount** property returns the elapsed time from the start of the system to the start of the current result.

It is specified in millisecond units. The TickCount returns zero if the wave file input is used.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechRecoResultTimes*.**TickCount**

Parts

ISpeechRecoResultTimes

The owning object.

Long

Set: (This property is read-only)

Get: A Variant type containing the elapsed time

Example

When a recognition occurs with *RecoResult*, the TickCount can be extracted to determine the absolute time for the computer system. It is more accurate than retrieving the clock values.

The first example retrieves the recognition tick count using the TickCount property.

```
Dim myTimes As ISpeechRecoResultTimes  
Set myTimes = RecoResult.Times
```

```
Dim tickCount As Variant
```

```
tickCount = myTimes.TickCount
```

The second example retrieves the recognition tick count by directly accessing the *RecoResult* structure.

```
Dim tickCount As Variant  
tickCount = RecoResult.Times.TickCount
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechVoiceStatus

The **ISpeechVoiceStatus** automation interface defines the types of information returned by the [SpVoice.Status](#) method.

Most ISpeechVoiceStatus properties consist of real-time feedback from the text-to-speech (TTS) engine. These properties are equivalent to parameters returned by [voice events](#), and like voice events, they are used only with asynchronous speech. It may be advantageous for some applications to retrieve these elements by calling Status occasionally, rather than by receiving events constantly.

It should be noted that voice status and voice events are closely associated with the status of the audio output device. A voice speaking to a file stream produces no audio output and has no audio output status. As a result, ISpeechVoiceStatus always displays the voice as inactive.

Use of the ISpeechVoiceStatus is demonstrated in a [code example](#) at the end of this section.

Automation Interface Elements

The ISpeechVoiceStatus automation interface contains the following elements:

Properties	Description
CurrentStreamNumber Property	Retrieves the number of the text input stream being spoken by the TTS engine.
InputSentenceLength Property	Retrieves the length of the sentence currently being spoken by the TTS engine.
InputSentencePosition	Retrieves the position one

Property	character prior to the beginning of the sentence currently being spoken by the TTS engine.
<u>InputWordLength</u> Property	Retrieves the length of the word currently being spoken by the TTS engine.
<u>InputWordPosition</u> Property	Retrieves the position one character prior to the beginning of the word currently being spoken by the TTS engine.
<u>LastBookmark</u> Property	Retrieves the string value of the last bookmark encountered by the TTS engine.
<u>LastBookmarkId</u> Property	Retrieves the ID of the last bookmark encountered by the TTS engine.
<u>LastHResult</u> Property	Retrieves the HRESULT, or internal status code, from the last Speak or SpeakStream operation performed by the TTS engine.
<u>LastStreamNumberQueued</u> Property	Retrieves the number of the last audio stream spoken by the TTS engine.
<u>PhonemeId</u> Property	Retrieves the ID of the current phoneme being spoken by the TTS engine.
<u>RunningState</u> Property	Retrieves the run state of the voice, which indicates whether the voice is speaking or inactive.
<u>VisemeId</u> Property	Gets the ID of the current viseme being spoken by the TTS engine.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

CurrentStreamNumber Property

The **CurrentStreamNumber** property retrieves the number of the text input stream being spoken by the text-to-speech (TTS) engine.

The CurrentStreamNumber property of an ISpeechVoiceStatus object is valid only when its RunningState property is SRSEIsSpeaking.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechVoiceStatus*.**CurrentStreamNumber**

Parts

ISpeechVoiceStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable that returns the stream number.

Example

The following Visual Basic form code demonstrates the use of the CurrentStreamNumber and RunningState properties. To run this code, create a form with the following controls:

- A command button called Command1
- A list box called List1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice. The Command1_Click procedure speaks three text streams asynchronously, putting each in the list box, and then performs a loop until the voice finishes speaking. Inside this loop, the code uses the CurrentStreamNumber property to highlight each line of text in the list box while it is spoken by the TTS engine. A RunningState property of SRSEDone indicates that the voice has finished speaking.

Option Explicit

```
Private V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Dim ii As Integer
```

```
    List1.Clear
```

```
    'Place text strings in the List box, and speak them
```

```
    List1.AddItem "This is stream number one."
```

```
    V.Speak "This is stream number one.", SVSFlagsAsync
```

```
    List1.AddItem "A second stream, now."
```

```
    V.Speak "A second stream, now.", SVSFlagsAsync
```

```
    List1.AddItem "The third stream is next."
```

```
    V.Speak "The third stream is next.", SVSFlagsAsync
```

```
    'Check status periodically
```

```
    Do
```

```
        For ii = 0 To 1000
```

```
            DoEvents
```

```
        Next ii
```

```
        'Highlight the stream being spoken
```

```
        List1.ListIndex = V.Status.CurrentStreamNumber - 1
```

```
    Loop Until V.Status.RunningState = SRSEDone 'Exit when v
```

```
    List1.ListIndex = -1    'Turn highlight off
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpVoice
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

InputSentenceLength Property

The **InputSentenceLength** property retrieves the length of the sentence currently being spoken by the text-to-speech (TTS) engine.

The InputSentenceLength property of an ISpeechVoiceStatus object is valid only when its RunningState property is SRSEIsSpeaking.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechVoiceStatus*.**InputSentenceLength**

Parts

ISpeechVoiceStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the sentence length.

Example

The following Visual Basic form code demonstrates the use of the InputSentenceLength and InputSentencePosition properties of an ISpeechVoiceStatus object. To run this code, create a form with the following controls:

- A command button called Command1

- A text box called Text1
- Set the HideSelection property of Text1 to False

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object and places sample sentences in the text box. The Command1_Click procedure speaks the contents of the text box asynchronously and loops until the voice finishes speaking. In this loop, the code uses InputSentencePosition and InputSentenceLength properties to highlight each sentence in the text box as it is spoken by the TTS engine. A RunningState property of SRSEDone indicates that the voice has finished speaking.

```
Option Explicit
```

```
Dim V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Dim ii As Integer
```

```
    Dim S As SpeechLib.ISpeechVoiceStatus
```

```
    V.Speak Text1.Text, SVSFlagsAsync    'Speak the user-edit
```

```
    'Check status periodically
```

```
    Do
```

```
        For ii = 0 To 5000
```

```
            DoEvents
```

```
        Next ii
```

```
        Set S = V.Status    'Get status in an ISpeechVoiceSt
```

```
        'Text1.HideSelection must be False for this selection
```

```
        Text1.SelStart = S.InputSentencePosition
```

```
        Text1.SelLength = S.InputSentenceLength
```

```
    Loop Until V.Status.RunningState = SRSEDone 'Exit when v
```

```
Text1.SelLength = 0
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
Set V = New SpVoice
```

```
Text1.Text = "One sentence. Another sentence. Still one i
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

InputSentencePosition Property

The **InputSentencePosition** property retrieves the position one character prior to the beginning of the sentence currently being spoken by the text-to-speech (TTS) engine.

The InputSentencePosition property of an ISpeechVoiceStatus object is valid only when its RunningState property is SRSEIsSpeaking.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechVoiceStatus*.**InputSentencePosition**

Parts

ISpeechVoiceStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the character position.

Example

The following Visual Basic form code demonstrates the use of the InputSentenceLength and InputSentencePosition properties of an ISpeechVoiceStatus object. To run this code, create a form with the following controls:

- A command button called Command1

- A text box called Text1
- Set the HideSelection property of Text1 to False

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object and places a few sentences in the text box. The Command1_Click procedure speaks the contents of the text box asynchronously and loops until the voice finishes speaking. In this loop, the code uses InputSentencePosition and InputSentenceLength properties to highlight each sentence in the text box as it is spoken by the TTS engine. A RunningState property of SRSEDone indicates that the voice has finished speaking.

```
Option Explicit
```

```
Dim V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Dim ii As Integer
```

```
    Dim S As SpeechLib.ISpeechVoiceStatus
```

```
    V.Speak Text1.Text, SVSFlagsAsync    'Speak the user-edit
```

```
    'Check status periodically
```

```
    Do
```

```
        For ii = 0 To 5000
```

```
            DoEvents
```

```
        Next ii
```

```
        Set S = V.Status    'Get status in an ISpeechVoiceSt
```

```
        'Text1.HideSelection must be False for this selection
```

```
        Text1.SelStart = S.InputSentencePosition
```

```
        Text1.SelLength = S.InputSentenceLength
```

```
    Loop Until V.Status.RunningState = SRSEDone 'Exit when v
```

```
Text1.SelLength = 0
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
Set V = New SpVoice
```

```
Text1.Text = "One sentence. Another sentence. Still one i
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

InputWordLength Property

The **InputWordLength** property retrieves the length of the word currently being spoken by the text-to-speech (TTS) engine.

The InputWordLength property of an ISpeechVoiceStatus object is valid only when its RunningState property is SRSEIsSpeaking.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechVoiceStatus*.**InputWordLength**

Parts

ISpeechVoiceStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the word length.

Example

The following Visual Basic form code demonstrates the use of the InputWordLength and InputWordPosition properties of an ISpeechVoiceStatus object. To run this code, create a form with the following controls:

- A command button called Command1
- A text box called Text1

- Set the HideSelection property of Text1 to False

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object and places a sentence in the text box. The Command1_Click procedure speaks the contents of the text box asynchronously and loops until the voice finishes speaking. In this loop, the code uses InputSentencePosition and InputSentenceLength properties to highlight each word in the text box as it is spoken by the TTS engine. A RunningState property of SRSEDone indicates that the voice has finished speaking.

```
Option Explicit
```

```
Dim V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Dim ii As Integer
```

```
    Dim S As SpeechLib.ISpeechVoiceStatus
```

```
    V.Speak Text1.Text, SVSFlagsAsync    'Speak the user-editi
```

```
    'Check status periodically
```

```
    Do
```

```
        For ii = 0 To 5000
```

```
            DoEvents
```

```
        Next ii
```

```
        Set S = V.Status    'Get status in an ISpeechVoiceSt
```

```
        'Text1.HideSelection must be False for this selection
```

```
        Text1.SelStart = S.InputWordPosition
```

```
        Text1.SelLength = S.InputWordLength
```

```
    Loop Until V.Status.RunningState = SRSEDone 'Exit when v
```

```
    Text1.SelLength = 0
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpVoice
```

```
    Text1.Text = "This is a sentence containing several word:
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

InputWordPosition Property

The **InputWordPosition** property retrieves the position one character prior to the beginning of the word currently being spoken by the text-to-speech (TTS) engine.

The InputWordPosition property of an ISpeechVoiceStatus object is valid only when its RunningState property is SRSEIsSpeaking.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechVoiceStatus*.**InputWordPosition**

Parts

ISpeechVoiceStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the character position.

Example

The following Visual Basic form code demonstrates the use of the InputWordLength and InputWordPosition properties of an ISpeechVoiceStatus object. To run this code, create a form with the following controls:

- A command button called Command1
- A text box called Text1

- Set the HideSelection property of Text1 to False

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object and places a sentence in the text box. The Command1_Click procedure speaks the contents of the text box asynchronously and loops until the voice finishes speaking. In this loop, the code uses InputSentencePosition and InputSentenceLength properties to highlight each word in the text box as it is being spoken by the TTS engine. A RunningState property of SRSEDone indicates that the voice has finished speaking.

```
Option Explicit
```

```
Dim V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Dim ii As Integer
```

```
    Dim S As SpeechLib.ISpeechVoiceStatus
```

```
    V.Speak Text1.Text, SVSFlagsAsync    'Speak the user-edit
```

```
    'Check status periodically
```

```
    Do
```

```
        For ii = 0 To 5000
```

```
            DoEvents
```

```
        Next ii
```

```
        Set S = V.Status    'Get status in an ISpeechVoiceSt
```

```
        'Text1.HideSelection must be False for this selection
```

```
        Text1.SelStart = S.InputWordPosition
```

```
        Text1.SelLength = S.InputWordLength
```

```
    Loop Until V.Status.RunningState = SRSEDone 'Exit when v
```

```
    Text1.SelLength = 0
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpVoice
```

```
    Text1.Text = "This is a sentence containing several word:
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

LastBookmark Property

The **LastBookmark** property retrieves the text value of the last bookmark encountered by the text-to-speech (TTS) engine.

The text value of a bookmark is enclosed in an XML attribute called Mark.

Syntax

Set: (This property is read-only)

Get: *String* = *ISpeechVoiceStatus*.**LastBookmark**

Parts

ISpeechVoiceStatus

The owning object.

String

Set: (This property is read-only)

Get: A String variable returning the string value of the last bookmark.

Example

The following code snippet demonstrates the use of the LastBookmark and LastBookmarkId properties. The code creates a voice that speaks a text stream containing several bookmarks, and then calls the Status method to get an ISpeechVoiceStatus object. This example demonstrates how the LastBookmark and the LastBookmarkId properties return the bookmark text and also displays the format of a bookmark.

```
Dim objVOICE As SpeechLib.SpVoice
Dim objSTATUS As SpeechLib.ISpeechVoiceStatus

Set objVOICE = New SpVoice

objVOICE.Speak "<BOOKMARK MARK='1. Monday' /> monday " _
               & "<bookmark mark='2. Tuesday' /> tuesday ", SVSF:

Set objSTATUS = objVOICE.Status

MsgBox "LastBookmark is " & objSTATUS.LastBookmark      ' "2
MsgBox "LastBookmarkId is " & objSTATUS.LastBookmarkId  ' "2
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

LastBookmarkId Property

The **LastBookmarkId** property retrieves the ID of the last bookmark encountered by the text-to-speech (TTS) engine.

The text value of a bookmark is enclosed in an XML attribute called Mark. The bookmark ID consists of the numeric integer value of the leading characters of the bookmark text. For example, in a bookmark with text "17.53.01 Section 53," the bookmark ID would be "17." A bookmark with text that begins with a non-numeric character will always have a bookmark ID of zero.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechVoiceStatus*.**LastBookmarkId**

Parts

ISpeechVoiceStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the ID of the last bookmark.

Example

The following code snippet demonstrates the use of the LastBookmark and LastBookmarkId properties. The code creates a voice which speaks a text stream containing several bookmarks. The code then calls the Status method to get an

ISpeechVoiceStatus object. This example demonstrates how the LastBookmark and the LastBookmarkId properties return the bookmark text and also displays the format of a bookmark.

```
Dim objVOICE As SpeechLib.SpVoice
Dim objSTATUS As SpeechLib.ISpeechVoiceStatus

Set objVOICE = New SpVoice

objVOICE.Speak "<BOOKMARK MARK='1. Monday' /> monday " _
               & "<bookmark mark='2. Tuesday' /> tuesday ", SVSF:

Set objSTATUS = objVOICE.Status

MsgBox "LastBookmark is " & objSTATUS.LastBookmark      ' "2
MsgBox "LastBookmarkId is " & objSTATUS.LastBookmarkId  ' "2
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

LastHResult Property

The **LastHResult** property retrieves the HRESULT, or internal status code, from the last Speak or SpeakStream operation performed by the SpVoice object.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechVoiceStatus*.**LastHResult**

Parts

ISpeechVoiceStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the HRESULT.

Example

The following Visual Basic form code demonstrates the use of the LastHResult property of an ISpeechVoiceStatus object. To run this code, create a form with the following controls:

- A command button called Command1
- A text box called Text1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object. The

Command1_Click procedure speaks a text stream asynchronously. The LastHResult property value is displayed in the text box. A value of 0 indicates completion with no error.

```
Option Explicit
```

```
Dim V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    V.Speak "This is a text stream", SVSFlagsAsync
```

```
    'wait for maximum 10 seconds to finish speaking  
    V.WaitUntilDone 10000
```

```
    Text1.Text = Format(V.Status.LastHResult)
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpVoice
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

LastStreamNumberQueued Property

The **LastStreamNumberQueued** property retrieves the number of the last audio stream enqueued by the voice.

Syntax

Set: (This property is read-only)

Get: *Long* = *ISpeechVoiceStatus*.**LastStreamNumberQueued**

Parts

ISpeechVoiceStatus

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the stream number of the last audio stream spoken.

Example

The following code snippet demonstrates the use of the LastStreamNumberQueued property. To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object. The Command1_Click procedure speaks three streams asynchronously, creates an ISpeechVoiceStatus object and

prints the LastStreamNumberQueued property value. The WaitUntilDone method then blocks execution until the voice finishes speaking the three streams, and the LastStreamNumberQueued property value is printed again. The value of the LastStreamNumberQueued property in both cases is 3.

Option Explicit

```
Private V As SpeechLib.SpVoice
Private S As SpeechLib.ISpeechVoiceStatus

Private Sub Command1_Click()

    'Enqueue three streams

    V.Speak "this is stream number one.", SVSFlagsAsync
    V.Speak "a second stream, now.", SVSFlagsAsync
    V.Speak "the third stream is next", SVSFlagsAsync

    'Get status while voice is speaking

    Set S = V.Status 'Get status thru ISpeechVoiceStatus ob.

    Print "Voice is speaking and LastStreamNumberQueued is "
        & S.LastStreamNumberQueued
    DoEvents 'Let Print results be seen immediately

    V.WaitUntilDone (99999) 'Wait until voice finishes

    'Get status thru "Voice.Status.Property" syntax

    Print "Voice is finished and LastStreamNumberQueued is "
        & V.Status.LastStreamNumberQueued

End Sub

Private Sub Form_Load()

    Set V = New SpVoice
```

End Sub

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

Phonemeld Property

The **Phonemeld** property retrieves the ID of the current phoneme being spoken by the voice.

Syntax

Set: (This property is read-only)

Get: *Integer* = *ISpeechVoiceStatus*.**Phonemeld**

Parts

ISpeechVoiceStatus

The owning object.

Integer

Set: (This property is read-only)

Get: An Integer variable returning the Phoneme ID.

Example

The following Visual Basic form code demonstrates the use of the Phonemeld property of an ISpeechVoiceStatus object. To run this code, create a form with the following controls:

- A command button called Command1
- Two text boxes called Text1 and Text2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object and places a sentence in the text box. The Command1_Click procedure speaks the contents of the text box asynchronously, and loops until the voice has finished speaking. Inside the loop, the code

checks Phonemeld property periodically and displays it in Text2.

```
Option Explicit
```

```
Private V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Dim ii As Integer
```

```
    Text2.Text = ""
```

```
    V.Speak Text1.Text, SVSFlagsAsync
```

```
    Do
```

```
        For ii = 0 To 20000
```

```
            DoEvents
```

```
        Next ii
```

```
        Text2.Text = Text2.Text & V.Status.PhonemeId & " "
```

```
    Loop While V.Status.RunningState = SRSEIsSpeaking
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpVoice
```

```
    Text1.Text = "way, we, why, woe, woo."
```

```
    Text2.Text = ""
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

RunningState Property

The **RunningState** property retrieves the run state of the voice, which indicates whether the voice is speaking or inactive.

The values of the RunningState property are contained in the SpeechRunState enumeration.

Syntax

Set: (This property is read-only)

Get: [SpeechRunState](#) = *ISpeechVoiceStatus*.**RunningState**

Parts

ISpeechVoiceStatus

The owning object.

SpeechRunState

Set: (This property is read-only)

Get: A SpeechRunState constant returning the run state of the voice.

Example

The following Visual Basic form code demonstrates the use of the RunningState property of an ISpeechVoiceStatus object. To run this code, create a form with the following controls:

- A command button called Command1
- A text box called Text1

- A timer called Timer1

Paste this code into the Declarations section of the form.

The Form_Load procedure places a sentence in the text box and creates two voice objects, one with an alert Priority setting. In the Command1_Click procedure, the timer is activated, and the normal Priority voice enqueues the contents of the text box, and waits a tenth of a second. Then the alert Priority voice speaks a short stream which interrupts the normal voice.

The timer procedure changes the color of the text box depending on the run state of the normal voice. When the normal voice is speaking, the text color is red; when it is done speaking, the text color is blue; when the voice is not speaking, the text color is black.

The text color change in this example has no significance other than indicating the running state of the normal voice.

Option Explicit

```
Private V As SpeechLib.SpVoice  
Private VHim As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Timer1.Interval = 250  
    Timer1.Enabled = True
```

```
    'Make sure normal voice starts first  
    V.Speak Text1.Text, SVSFlagsAsync  
    V.WaitUntilDone (100)  
    VHim.Speak "Alert voice!", SVSFlagsAsync
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpVoice  
    Text1.Text = "turn this text red while the voice is speaki
```

```
Set V.Voice = V.GetVoices("Gender=Female").Item(0)

Set VHim = New SpVoice
VHim.Priority = SVPAlert
Set VHim.Voice = VHim.GetVoices("Gender=Male").Item(0)
End Sub

Private Sub Timer1_Timer()

    Select Case V.Status.RunningState
    Case SRSEIsSpeaking
        Text1.ForeColor = vbRed
    Case SRSEDone
        Text1.ForeColor = vbBlue
    Case Else
        Text1.ForeColor = vbBlack
    End Select

End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

Visemeld Property

The **Visemeld** property retrieves the ID of the current viseme being spoken by the voice.

Syntax

Set: (This property is read-only)

Get: *Integer* = *ISpeechVoiceStatus*.**Visemeld**

Parts

ISpeechVoiceStatus

The owning object.

Integer

Set: (This property is read-only)

Get: An Integer variable returning the Visemeld.

Example

The following Visual Basic form code demonstrates the use of the Visemeld property of an ISpeechVoiceStatus object. To run this code, create a form with the following controls:

- A command button called Command1
- Two text boxes called Text1 and Text2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object and places a sentence in the text box. The Command1_Click procedure speaks the contents of the text box asynchronously, and loops until the voice has finished speaking. Inside the loop, the code

checks Visemeld property periodically and displays it in Text2.

```
Option Explicit
```

```
Private V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Dim ii As Integer
```

```
    Text2.Text = ""
```

```
    V.Speak Text1.Text, SVSFlagsAsync
```

```
    Do
```

```
        For ii = 0 To 20000
```

```
            DoEvents
```

```
        Next ii
```

```
        Text2.Text = Text2.Text & V.Status.VisemeId & " "
```

```
    Loop While V.Status.RunningState = SRSEIsSpeaking
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpVoice
```

```
    Text1.Text = "say, see, sigh, so, sue."
```

```
    Text2.Text = ""
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechVoiceStatus](#)

ISpeechVoiceStatus

Example

The following code snippet demonstrates the use of all ISpeechVoiceStatus properties. To run this code, create a form with the following controls:

- A command button called Command1
- A list box called List1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object. The Command1_Click procedure speaks two streams asynchronously, and adds status information into the list box every one-half second until both streams have been spoken.

```
Option Explicit
```

```
Private V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    'Enqueue two streams containing bookmarks
```

```
    List1.Clear
```

```
    V.Speak "this is stream number<bookmark mark='1one' /> one"
    List1.AddItem "LastStreamNumberQueued is " & V.Status.LastStreamNumberQueued
    DoEvents
```

```
    V.Speak "this is stream number<bookmark mark='2two' /> two"
    List1.AddItem "LastStreamNumberQueued is " & V.Status.LastStreamNumberQueued
    DoEvents
```

```
Do
```

```
V.WaitUntilDone (500) 'Wait for 0.5 second so we won
```

```
List1.AddItem ""  
List1.AddItem "LastStreamNumberQueued is " & V.Status.  
List1.AddItem "CurrentStreamNumber is " & V.Status.C  
List1.AddItem "InputSentenceLength is " & V.Status.I  
List1.AddItem "InputSentencePosition is " & V.Status  
List1.AddItem "InputWordLength is " & V.Status.Input  
List1.AddItem "InputWordPosition is " & V.Status.Inp  
List1.AddItem "RunningState is " & V.Status.RunningS  
List1.AddItem "LastBookmark is " & V.Status.LastBook  
List1.AddItem "LastBookmarkId is " & V.Status.LastBo  
List1.AddItem "VisemeId is " & V.Status.VisemeId  
List1.AddItem "PhonemeId is " & V.Status.PhonemeId  
List1.AddItem "LastHResult is " & V.Status.LastHResu.  
DoEvents
```

```
Loop Until V.Status.RunningState = SRSEDone 'Exit when v
```

```
End Sub
```

```
Private Sub Form_Load()  
    Set V = New SpVoice  
End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



SpAudioFormat

The **SpAudioFormat** automation object represents an audio format.

Most applications using standard audio formats will use the **Type** property to set and retrieve formats. Non-standard formats using wav files will use **SetWavFormatEx** and **GetWaveFormatEx** to set and retrieve formats, respectively. Non-standard formats using sources other than wav files use **Guid**.

Automation Interface Elements

The SpAudioFormat automation object has the following elements:

Properties	Description
Guid Property	Returns the GUID of the default format.
Type Property	Gets and sets the speech audio format as a SpeechAudioFormatType .

Methods	Description
GetWaveFormatEx Method	Gets the audio format as an SpWaveFormatEx object.
SetWaveFormatEx Method	Sets the audio format with an SpWaveFormatEx object.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpAudioFormat](#)

Type: Hidden

GetWaveFormatEx Method

The **GetWaveFormatEx** method gets the audio format as an SpWaveFormatEx object.

Non-standard formats using wav files should use GetWaveFormatEx to retrieve formats.

SpAudioFormat.GetWaveFormatEx() As [SpWaveFormatEx](#)

Parameters

None.

Return Value

The GetWaveFormatEx method returns an SpWaveFormatEx variable.

Example

The following Visual Basic form code demonstrates the use of the GetWaveFormatEx and SetWaveFormatEx properties. To run this code, create a form with the following controls:

- Two command buttons called Command1 and Command2

Paste this code into the Declarations section of the form.

The Command1 procedure creates an SpAudioFormat object and sets it to the audio format SAFT22kHz16BitStereo. It then gets the format object's SpWaveFormatEx object and displays the properties. The code then changes the format of the SpAudioFormat object to SAFT11kHz16BitMono, gets a new SpWaveFormatEx object and displays its properties again. Note

that the SpWaveFormatEx properties have changed to reflect the new audio format.

The Command2 procedure creates an SpAudioFormat object and sets it to the audio format SAFT22kHz16BitStereo. It then gets the format object's SpWaveFormatEx object and displays the properties. The code then changes the properties of the SpWaveFormatEx object to match the SAFT11kHz16BitMono format and sets the format of the SpAudioFormat object with the SetWaveFormatEx method. Note that the SpAudioFormat object's Type property has changed to SAFT11kHz16BitMono to reflect the new SpWaveFormatEx properties.

Option Explicit

```
Dim F As SpeechLib.SpAudioFormat
Dim W As SpeechLib.SpWaveFormatEx
```

```
Private Sub Command1_Click()
```

```
    'Create an empty SpAudioFormat object
    'Set it to the default format
    'Get its format in an SpWaveFormatEx object
    Set F = New SpAudioFormat
    F.Type = SAFT22kHz16BitStereo
    Set W = F.GetWaveFormatEx

    Debug.Print
    Debug.Print "Default SpAudioFormat and SpWaveFormatEx"
    Debug.Print "Format:          SAFT22kHz16BitStereo"
    Debug.Print "Format code:      " & F.Type
    Debug.Print "AvgBytesPerSec   " & W.AvgBytesPerSec
    Debug.Print "BitsPerSample    " & W.BitsPerSample
    Debug.Print "BlockAlign       " & W.BlockAlign
    Debug.Print "Channels          " & W.Channels
    Debug.Print "ExtraData        " & W.ExtraData
    Debug.Print "FormatTag        " & W.FormatTag
    Debug.Print "SamplesPerSec    " & W.SamplesPerSec

    'Give the SpAudioFormat object an audio type
    'Get its format in an SpWaveFormatEx object
```

```
F.Type = SAFT11kHz16BitMono
Set W = F.GetWaveFormatEx
```

```
Debug.Print
Debug.Print "Changing SpAudioFormat changes SpWaveFormatEx"
Debug.Print "Format:          SAFT11kHz16BitMono"
Debug.Print "Format code:     " & F.Type
Debug.Print "AvgBytesPerSec " & W.AvgBytesPerSec
Debug.Print "BitsPerSample  " & W.BitsPerSample
Debug.Print "BlockAlign    " & W.BlockAlign
Debug.Print "Channels      " & W.Channels
Debug.Print "ExtraData     " & W.ExtraData
Debug.Print "FormatTag     " & W.FormatTag
Debug.Print "SamplesPerSec " & W.SamplesPerSec
```

```
End Sub
```

```
Private Sub Command2_Click()
```

```
'Create an empty SpAudioFormat object
'Set it to the default format
'Get its format in an SpWaveFormatEx object
```

```
Set F = New SpAudioFormat
F.Type = SAFT22kHz16BitStereo
Set W = F.GetWaveFormatEx
```

```
Debug.Print
Debug.Print "Default SpAudioFormat and SpWaveFormatEx:"
Debug.Print "Format:          SAFT22kHz16BitStereo"
Debug.Print "Format code:     " & F.Type
Debug.Print "AvgBytesPerSec " & W.AvgBytesPerSec
Debug.Print "BitsPerSample  " & W.BitsPerSample
Debug.Print "BlockAlign    " & W.BlockAlign
Debug.Print "Channels      " & W.Channels
Debug.Print "ExtraData     " & W.ExtraData
Debug.Print "FormatTag     " & W.FormatTag
Debug.Print "SamplesPerSec " & W.SamplesPerSec
```

```
'Set SpWaveFormatEx properties as in SAFT11kHz16BitMono
'this will reset the SpAudioFormat Type.
```

```
Debug.Print
Debug.Print "Changing SpWaveFormatEx properties changes :
W.AvgBytesPerSec = 22050
W.BitsPerSample = 16
W.BlockAlign = 2
W.Channels = 1
W.SamplesPerSec = 11025

Call F.SetWaveFormatEx(W)
Debug.Print "Format code: " & F.Type
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpAudioFormat](#)

Guid Property

The **Guid** property returns the GUID of the default audio format.

Non-standard formats using sources other than wav files should use Guid to set and retrieve formats.

Syntax

```
Set: SpAudioFormat.Guid = String
```

```
Get: String = SpAudioFormat.Guid
```

Parts

SpAudioFormat

The owning object.

String

Set: A String variable that sets the property.

Get: A String variable that gets the property.

Example

For an example of the use of the Guid property, see the code example in the [Type](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpAudioFormat](#)

Type: Hidden

SetWaveFormatEx Method

The **SetWaveFormatEx** method sets the audio format with an `SpWaveFormatEx` object.

Non-standard formats using wav files should use `SetWavFormatEx` to set formats.

```
SpAudioFormat.SetWaveFormatEx(  
    WaveFormatEx As SpWaveFormatEx  
)
```

Parameters

WaveFormatEx
Specifies the `WaveFormatEx`.

Return Value

None.

Example

For an example of the use of the `SetWaveFormatEx` method, see the [GetWaveFormatEx](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpAudioFormat](#)

Type Property

The **Type** property gets and sets the speech audio format as a `SpeechAudioFormatType`.

Most applications using standard audio formats should use `Type` to set and retrieve formats.

Syntax

```
Set: SpAudioFormat.Type = SpeechAudioFormatType
```

```
Get: SpeechAudioFormatType = SpAudioFormat.Type
```

Parts

SpAudioFormat

The owning object.

SpeechAudioFormatType

Set: A `SpeechAudioFormatType` object that sets the property.

Get: A `SpeechAudioFormatType` object that gets the property.

Example

The following Visual Basic form code demonstrates the use of the `Type` and `Guid` properties. To run this code, create a form with the following controls:

- Two command buttons called `Command1` and `Command2`

Paste this code into the Declarations section of the form.

The `Command1` procedure creates a token category object and

sets it to the category of audio inputs, selects a token for the first MMSys resource, and instantiates an SpMMAudioIn object with the token's [CreateInstance](#) method. The code then creates an SpAudioFormat object from the SpMMAudioIn object, and changes the Type property of the SpAudioFormat object. Finally, the code sets the Format property of the SpMMAudioIn object with the SpAudioFormat object.

The Command2 procedure performs the same series of operations with audio outputs instead of audio inputs.

Option Explicit

```
Dim C As SpeechLib.SpObjectTokenCategory
Dim T As SpeechLib.SpObjectToken
```

```
Dim I As SpeechLib.SpMMAudioIn
Dim O As SpeechLib.SpMMAudioOut
```

```
Dim F As SpeechLib.SpAudioFormat
```

```
Private Sub Command1_Click()
```

```
    Debug.Print
    Debug.Print "MMSys AudioIn"
    Debug.Print
```

```
    'Set category object to audio input resources
    Set C = New SpObjectTokenCategory
    C.SetId SpeechCategoryAudioIn
```

```
    'Set token object to first MMSys input resource
    Set T = C.EnumerateTokens("Technology=MMSys").Item(0)
    Debug.Print "First device: " & T.GetDescription
```

```
    'Create an SpMMAudioIn object from the token,
    'and show some of its properties
    Set I = T.CreateInstance()
    Debug.Print "DeviceId: " & I.DeviceId
    Debug.Print "original Audio Format:" & I.Format.Type
```

```
'Create an Audio Format object from resource
'If Audio Format's Type is standard, then change it
Set F = I.Format
If F.Type = SAFT22kHz16BitMono Then
    F.Type = SAFT11kHz16BitMono
End If
```

```
'Set Audioinput's format with changed format object
Set I.Format = F
Debug.Print " changed Audio Format:" & I.Format.Type
Debug.Print "Guid:" & F.Guid
```

End Sub

```
Private Sub Command2_Click()
```

```
    Debug.Print
    Debug.Print "MMSys AudioOut"
    Debug.Print
    Set C = New SpObjectTokenCategory
    C.SetId SpeechCategoryAudioOut
    Set T = C.EnumerateTokens("Technology=MMSys").Item(0)
    Debug.Print "First device: " & T.GetDescription
    Set O = T.CreateInstance()
    Debug.Print "DeviceId: " & O.DeviceId
    Debug.Print "original Audio Format:" & O.Format.Type
    Set F = O.Format
    If F.Type = SAFT22kHz16BitMono Then
        F.Type = SAFT11kHz16BitMono
    End If
    Set O.Format = F
    Debug.Print " changed Audio Format:" & O.Format.Type
    Debug.Print "Guid:" & F.Guid
```

End Sub

Microsoft Speech SDK

Speech Automation 5.1



SpCustomStream

The **SpCustomStream** automation object supports the use of existing IStream objects in SAPI.

The Format property and the Read, Write and Seek methods are inherited from the [ISpeechBaseStream](#) interface.

Automation Interface Elements

The SpCustomStream automation object has the following elements:

Properties	Description
BaseStream Property	Gets and sets the base stream object in a custom stream.
Format Property	Gets and sets the cached wave format of the stream as an SpAudioFormat object.
Methods	Description
Read Method	Reads data from an audio stream.
Seek Method	Returns the current read position of the audio stream in bytes.
Write Method	Writes data to the audio stream.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpCustomStream](#)

BaseStream Property

The **BaseStream** property gets and sets the base stream object in a custom stream.

Syntax

```
Set: SpCustomStream.BaseStream = IUnknown
```

```
Get: IUnknown = SpCustomStream.BaseStream
```

Parts

SpCustomStream

The owning object.

IUnknown

Set: An Unknown variable that sets the base stream.

Get: An Unknown variable that gets the base stream.

Example

The following Visual Basic form code demonstrates the use of the BaseStream property. To run this code, create a form with following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

The From_Load procedure creates two voice objects. The Command1 procedure creates a new custom stream, uses the CreateStreamOnHGlobal API to create a new genetic Istream object, and sets the Istream as the custom stream's BaseStream property. Finally, it uses the first voice to speak a phrase into the

custom stream, and then plays back the custom stream audio with the second voice.

Option Explicit

```
Dim V1 As SpeechLib.SpVoice
Dim V2 As SpeechLib.SpVoice
Dim C As SpeechLib.SpCustomStream
```

```
Private Declare Function CreateStreamOnHGlobal Lib "Ole32.dll"
    ByVal hGlobal As Any, ByVal fDeleteOnRelease As Boolean,
    ByVal ppStream As IStream) As Long
```

```
Private Sub Command1_Click()
    Dim GeneticIstream As SpeechLib.IStream

    'Create a genetic Istream object,
    'Use as custom stream's base stream
    CreateStreamOnHGlobal 0&, True, GeneticIstream
    Set C = New SpCustomStream
    Set C.BaseStream = GeneticIstream

    'Set custom stream as voice's output stream
    'Make voice speak into the stream
    Set V1.AudioOutputStream = C
    V1.Speak "hello world"

    'Seek to beginning of stream, and speak stream
    C.Seek 0, SSSPTRelativeToStart
    V2.SpeakStream C
End Sub
```

End Sub

```
Private Sub Form_Load()
    Set V1 = New SpVoice      'Creates a custom stream
    Set V2 = New SpVoice      'Speaks the custom stream
End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



SpFileStream

The **SpFileStream** automation object enables data streams to be read and written as files.

SpFileStream objects normally contain audio data, but may also be used for text data.

The Format property and the Read, Write and Seek methods are inherited from the [ISpeechBaseStream](#) interface.

Automation Interface Elements

The SpFileStream automation object has the following elements:

Properties	Description
Format Property	Gets and sets the cached wave format of the stream as an SpAudioFormat object.

Methods	Description
Close Method	Closes the filestream object.
Open Method	Opens a filestream object for reading or writing.
Read Method	Reads data from an audio stream.
Seek Method	Returns the current read position of the audio stream in bytes.
Write Method	Writes data to the audio stream.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpFileStream](#)

Close Method

The **Close** method closes the filestream object.

SpFileStream.Close()

Parameters

None.

Return Value

None.

Example

For a simple example using the SpFileStream.Close method, see the [SpFileStream.Open](#) method.

The ISpeechPhraseElement [code example](#) demonstrates further use of the SPFileStream object. This example uses a text-to-speech voice to speak into an SPFileStream object, and uses the resulting file as the input for speech recognition.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpFileStream](#)

Open Method

The **Open** method opens a filestream object for reading or writing.

```
SpFileStream.Open(  
    FileName As String,  
    [FileMode As SpeechStreamFileMode = SSFMOpenForRead],  
    [DoEvents As Boolean = False]  
)
```

Parameters

FileName

Specifies the FileName.

FileMode

[Optional] Specifies the FileMode. Default value is SSFMOpenForRead.

DoEvents

[Optional] When FileMode is SSFMCreateForWrite, DoEvents specifies whether playback of the resulting sound file will generate voice events. Default value is False.

Return Value

None.

Remarks

When the SpFileStream object is used with audio data, the FileMode parameter controls access strictly. That is, the FileMode parameter SFMOpenForRead prevents write access, and the FileMode parameter SSFMCreateForWrite prevents read access. When the SpFileStream object is used with text data, only the SFMOpenForRead FileMode parameter controls access strictly. The FileMode parameter SFMOpenForRead prevents write access, but the SSFMCreateForWrite parameter allows text data to be read as well as written.

When an SpVoice object creates an SPFileStream object, the engine may embed event data in the stream. In order to embed these events in the file stream, it must be opened for writing with the DoEvents parameter set to True. In order to receive these events when the stream is played back, it must be opened for reading with the DoEvents parameter set to True. Several other factors are involved. Please see [SpVoice events](#) for further details.

Example

The following code snippet demonstrates the use of an SpFileStream object to capture the output of a voice in a file.

The ISpeechPhraseElement [code example](#) demonstrates further use of the SpFileStream object. This example uses a text-to-speech voice to speak into an SpFileStream object, and uses the resulting file as the input for speech recognition.

```
Dim objVOICE As SpeechLib.SpVoice
Dim objFSTRM As SpeechLib.SpFileStream

Set objVOICE = New SpVoice
Set objFSTRM = New SpFileStream

'Open file path as a stream
Call objFSTRM.Open("c:\VoiceToFile.wav", SSFMCreateForWrite,

'Set voice output to the stream and speak
```

```
Set objVOICE.AudioOutputStream = objFSTRM
objVOICE.Speak "cee : \ voice to file dot wave", SVSFNLPSpea

Call objFSTRM.Close
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechRecoContext

The **ISpeechRecoContext** automation interface defines a recognition context.

For a list of available methods and properties, see [Method/Property List](#).

What is a Recognition Context?

A recognition context is the primary means by which an application interacts with SAPI for speech recognition. It is an object that allows an application to start and stop recognition, receive recognition results and other events. It also controls which words and phrases are available for the user to speak. An application may have several recognition contexts open at the same time, each controlling a different part of the application. A specific recognition context controls the collection of available words and is associated with a specific part of the application. In a more general sense, that word collection is the confine to which speech recognition attempts are restricted and will poll within to match words. Words not contained in the collection or context, will not be used for that speech recognition attempt. By setting recognition contexts, applications limit or expand the scope of the words needed for a particular aspect of the application. This granularity for speech recognition improves the quality of recognition by removing words not needed at that moment. Conversely, the granularity also allows words to be added to the application if needed.

For example, an application may have only one recognition context: that of all the words in the dictionary and those words are available all the time. If that application were purely dictation, the one-context model would work well. The user could say any word at any time to the application and it would probably be successfully recognized. However, if the application had a new requirement of exiting when the user said "close," that one-context model breaks down. The user would be disappointed if, in the course of dictation, the word "close" were spoken and the application suddenly stopped and closed.

Clearly, there are two uses (or contexts) for the word "close." The first is a part of speech ("please close the door," "that was too close for comfort," "we'll close in on the criminal"). The second context is that of a specific command. There must be a

method to differentiate the two. A recognition context permits applications to do that.

Applications may have more than one recognition context. In fact, it is recommended to have as many as makes sense. For example, one recognition context may be assigned to the menu bar, another to the dictation screen, yet another to dialog boxes, even if only temporarily such as a Yes/No/Cancel dialog box. Programmers need to decide the scope of the recognition context. The menu system for an application may even have multiple recognition contexts, perhaps one for each menu bar item. This granularity grants applications the ability to concentrate resources robustly. For example, a small menu may only have 12 items associated with it. Not only that, but it would be 12 very specific words. It makes little sense, therefore, to have the entire dictation collection, some 65,000 to 100,000 words, available when in fact only 12 words are needed. The larger-than-needed vocabulary would not only take up more processing time, but could result in more mismatched words. By the same reasoning, in the "close" example above, a dictation model should treat the word "close" no differently than any other word. Two recognition contexts could be used to separate the differences.

Using Recognition Contexts

Creating a recognition context is done using a two-step process. The context must be declared and then created. The following code sample creates an instance of a recognition context named *RC*. The keyword *New* creates a reference to a new object of the specified class.

```
Public WithEvents RC As SpSharedRecoContext  
Set RC = New SpSharedRecoContext
```

Recognition context types

Recognition contexts may be one of two types: shared or in process (InProc). A shared context allows resources to be used by other recognition contexts or applications. All applications on the machine using shared recognition contexts are sharing a single audio input, SR engine, and grammars. When the user speaks, the SR engine will do recognition, and SAPI decides which context to send the recognition result to, based on which grammar the result best matches. In general, most applications should use shared contexts. The following code snippet declares a shared recognition context.

```
Public WithEvents RC As SpSharedRecoContext
```

InProc contexts restrict available resources to one context or application. That is, an SR engine or microphone used by an InProc recognition context may not be used by any other applications. In situations requiring the highest performance standards, response time or exacting recognition quality, use the InProc context. InProc contexts are important to embedded systems in other hardware platforms. InProc contexts are also used for non-microphone recognition such as recognizing from a file. However, InProc should be used sparingly since it excludes other applications from the speech recognition resources. The following code snippet declares an InProc recognition context.

Public WithEvents RC As SpInProcRecoContext

In either case, the two types are based on `ISpeechRecoContext`. Any declaration should include the keyword *WithEvents* so that recognition context also supports events.

Defaults

Recognition context is created with intelligent defaults using the defaults of the computer system. These defaults are assigned using Speech properties in Control Panel. While applications may override default values for specific reasons, applications should not manually set or change default values directly. These defaults include:

- [Recognizer](#) to determine the speech recognition engine
- [EventInterests](#) to determine which events the speech recognition engine generates
- [RetainedAudio](#) to persist the actual audio for the speech
- [RetainedAudioFormat](#) to determine the retained audio format
- [Voice](#) to speak the text

Grammars

The only resource that must be explicitly created is the grammar using [CreateGrammar](#). The grammar defines the set of words for the recognition context. Grammars also may be of two types: dictation and command and control (C and C). Dictation grammars are usually an unrestricted word list designed to encompass the full range of words in a language. Dictation allows any word or phrase to be spoken and it is used in the traditional sense to dictate a letter or paper, for example. The following code snippet declares a dictation grammar. It assumes a valid *RC* recognition context.

```
Set myGrammar = RC.CreateGrammar
```

```
myGrammar.DictationSetState SGDSActive
```

A command and control grammar is a limited word list restricting the speaker to a small set of words. In this way, users can speak a command, usually a single word, with greater chance of recognition. The smaller scope of words disallows words not on a specific list. A grammar is useful for speech-enabling menus, for example. Menu grammars are typically smaller with exact word or phrase commands such as "New," "Exit," or "Open." The following code snippet declares a command and control grammar. It assumes a valid *RC* recognition context.

```
Set myGrammar = RC.CreateGrammar  
myGrammar.CmdLoadFromFile "sol.xml", SLODynamic  
myGrammar.CmdSetRuleIdState 101, SGDSActive
```

Because the word list is limited, an explicit list is used. In this case, the command file `sol.xml` is used. In addition, the code sample activates one rule; the rule has an identification value of 101. For a more thorough discussion of grammars and designing grammars see [Text grammar format](#).

States

While individual grammar rules may be activated or deactivated as conditions change, all grammars in the recognition context may also be activated or deactivated with the [State](#) property. Grammars can be turned off, for example, if the window is no longer the current focus and likewise turned back on when the window becomes foremost again. In addition the recognition context may be momentarily stopped and then restarted. The [Pause](#) method halts the speech recognition temporarily for the engine to synchronize with the grammars. After a pause, [Resume](#) resumes the recognition process. While paused, the engine will continue to accept sound input and speech processing, provided the pause is not excessive; by default, this

is not more than 30 seconds.

The `ISpeechRecoContext` object is always associated with a single speech recognition engine (also called a recognizer). However, a single recognizer may have many recognition contexts.

Events

As a result of interactions with the recognition context, the SR engine sends back certain information to the application using the Events mechanism. An event is a specific occurrence that might be of interest to the user or application. Examples of events include notifying the application of a successful recognition or indicating that a designated position in the stream has been reached. Regardless, the application is free to process events or ignore them.

In addition, events may be filtered, allowing the engine to return some or all events, or to prevent an event from being generated in the first place if it has no significance to the application. Filtering is controlled by [EventInterests](#).

A complete list of events is described in [ISpeechRecoContext events](#).

Automation Interface Elements

The `ISpeechRecoContext` automation interface contains the following elements:

Properties	Description
AllowVoiceFormatMatchingOnNextSet Property	Determines if the recognition context can change the voice format to

	match that of the engine.
<u>AudioInputInterferenceStatus</u> Property	Returns information about interference with the recognition context's audio input.
<u>CmdMaxAlternates</u> Property	Specifies the maximum number of alternates that will be generated for command and control grammars.
<u>EventInterests</u> Property	Specifies the types of events raised by the object.
<u>Recognizer</u> Property	Identifies the recognizer associated with the recognition context.
<u>RequestedUIType</u> Property	Specifies the UIType of the last UI requested from the engine.
<u>RetainedAudio</u> Property	Gets and sets the audio retention status of the recognition context.
<u>RetainedAudioFormat</u> Property	Gets and sets the format of audio retained by the recognition context.
<u>State</u> Property	Gets or sets the active state of the recognition context.

Voice Property	Specifies the SpVoice object associated with the recognition context.
VoicePurgeEvent Property	Gets and sets the collection of SpeechRecoEvents which will stop the voice and purge the voice queue.

Methods	Description
Bookmark Method	Sets a bookmark within the current recognition stream.
CreateGrammar Method	Creates an SpGrammar object.
CreateResultFromMemory Method	Creates a recognition result object from a phrase that has been saved to memory.
Pause Method	Pauses the engine object to synchronize with the SR engine.
Resume Method	Releases the SR engine from the paused state and restarts the recognition process.
SetAdaptationData Method	Passes the SR engine a string of adaptation data.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

Type: Hidden

AllowVoiceFormatMatchingOnNextSet Property

The **AllowVoiceFormatMatchingOnNextSet** property determines if the recognition context can change the audio format of the output voice to match the audio format of the input stream.

AllowVoiceFormatMatchingOnNextSet can be used only if a voice has been created through the recognition context. If this property is set to True, the voice's output format will be set to the same format as the associated SR engine's audio input format. This conversion takes place only during the next setting of the [SpVoice.Voice](#). However, if this voice object has already been bound to a stream which has specific format, the voice's format will not be changed to the SR engine's audio input format even if set to True. If False, the conversion is not made.

Using the same audio format for input and output source is useful for sound cards that do not support full-duplex audio (i.e., input format must match output format). If the input format quality is lower than the output format quality, the output format quality will be reduced to equal the input quality.

By default, AllowVoiceFormatMatchingOnNextSet is set to True.

Syntax

```
Set: ISpeechRecoContext.AllowVoiceFormatMatchingOnNextSet  
    = Boolean  
Get: Boolean =  
    ISpeechRecoContext.AllowVoiceFormatMatchingOnNextSet
```

Parts

SpeechRecoContext
The owning object.

Boolean

Set: A Boolean variable that sets the property.

Get: A Boolean variable that gets the property.

Example

The following snippet demonstrates retrieving and setting `AllowVoiceFormatMatchingOnNextSet`.

```
Public WithEvents RecognitionContext As SpSharedRecoContext  
Set RecognitionContext = New SpSharedRecoContext
```

```
Dim voiceChange As Boolean  
voiceChange = RecognitionContext.AllowVoiceFormatMatchingOnNextSet
```

```
RecognitionContext.AllowVoiceFormatMatchingOnNextSet = True  
voiceChange = RecognitionContext.AllowVoiceFormatMatchingOnNextSet
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

AudioInputInterferenceStatus Property

The **AudioInputInterferenceStatus** property returns information about interference with the audio input of the recognition context.

This information is usually returned by the [ISpeechRecoContext Interference](#) event.

Syntax

Set:	(This property is read-only)
Get:	SpeechInterference = <code>SpeechRecoContext.AudioInputInterferenceStatus</code>

Parts

SpeechRecoContext
The owning object.

SpeechInterference
Set: (This property is read-only)
Get: A *SpeechInterference* constant reflecting the interference type.

Example

The following code snippet demonstrates retrieving the interference status for the last recognition. It assumes a valid *RecognitionContext* and speech recognition code in place.

```
Public WithEvents RecognitionContext As SpSharedRecoContext
```

```
'Speech processing code here
```

```
Dim interference As SpeechInterference  
interference = RecognitionContext.AudioInputInterferenceStat
```

A more realistic example returns the Interference event. In this example, the event displays a message in the Label1 label item of the application's Form1.

```
Private Sub RC_Interference(ByVal StreamNumber As Long, ByVa  
    Form1.Label1.Caption = "Interference detected: " & inter  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

Bookmark Method

The **Bookmark** method sets a bookmark within the current recognition stream.

A bookmark relates an audio stream position with an occurrence significant to the application. The speech recognition (SR) engine has a latency period between the speech and the recognition. This latency may be caused by a long speech period (the engine must have all the phrase before processing it) or other events may already be queued and the engine must finishing process those first. However, during that latent period, the user or application may continue operations (such as moving the mouse or speaking). As a result, the condition of the application may be different when the recognition comes back than it was when recognition was initiated. A bookmark allows the application to mark or record a condition of the application to a particular recognition attempt.

Bookmark is a convenient alternative to polling the engine for stream position (see [ISpeechRecognizer.Status](#)). A Bookmark event is returned when the bookmark has been processed by the engine. See [ISpeechRecoContext.Bookmark](#) for further information.

```
SpeechRecoContext.Bookmark(  
    Options As SpeechBookmarkOptions,  
    StreamPos As Variant,  
    BookmarkId As Variant  
)
```

Parameters

Options

Specifies whether the recognition context will pause when

encountering bookmarks. This must be value of type `SpeechBookmarkOptions`.

StreamPos

Specifies the stream position. This value may be anywhere in the stream and will send a `Bookmark` event when that position is reached. Additionally it may be any one of two special values: [Speech_StreamPos_Asap](#) or [Speech_StreamPos_RealTime](#).

BookmarkId

Specifies the `BookmarkId`. `BookmarkId` is additional information provided by the application and is unique to the application. As a `Variant` data type, it may be numeric, `String`, or any other format and is used to pass information within the method or event. If *BookmarkId* is a `String`, the value must be able to convert to a numeric value. This information is returned back using the `Bookmark` event.

Return Value

None.

Remarks

An application that wants to display a recognition progress meter, for example could use `ISpeechRecoContext.Bookmark` and update the UI when each bookmark is received.

Example

This code snippet demonstrates initiation of a `Bookmark` event at the start of the each speech recognition stream by setting the

StreamPos to zero. Each event is identified with *BookmarkId* of "10" and that value may be used inside the event for additional processing. Additional events may be indicated by adding new Bookmarks. After the grammar is closed, a Bookmark event with *BookmarkId* of "99" is sent.

This example assumes valid speech processing support and that *FileName* points to a valid grammar file.

```
Dim SharedRecognizer As Object
Dim WithEvents SharedReco As SpSharedRecoContext

Dim RecoGrammar As Object
Set RecoGrammar = Context.CreateGrammar(123)
RecoGrammar.CmdLoadFromFile FileName

'This bookmark is initiated before speech recognition
SharedReco.Bookmark SBONone, 0, "10"

'Speech processing code here

RecoGrammar.CmdSetRuleState "", SGDSInactive
SharedReco.Bookmark SBONone, 0, "99"
```

The corresponding Bookmark could be defined as this. This allows further refinement of the processing after the event occurs. Notice *BookmarkId* is converted to a numeric value.

```
Private Sub InProcRecoContext_Bookmark(ByVal StreamNumber As
    If BookmarkId = 10 Then
        'Some processing code goes here
    ElseIf BookmarkId = 99 Then
        'Some processing code goes here
    End If
End Sub
```

BookmarkId can also be one of the two predefined constants: *Speech_StreamPos_Asap* and *Speech_StreamPos_RealTime*, which are 0 and -1. *Speech_StreamPos_Asap* means Bookmark event will occur when the SR engine reaches a synchronization

point, and be fired as soon as possible.

Speech_StreamPos_RealTime means Bookmark event will occur when the SR engine reaches the current audio device position.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

CmdMaxAlternates Property

The **CmdMaxAlternates** property specifies the maximum number of alternates that will be generated for command and control grammars.

By default, the maximum alternates value is zero, so an application must call this method before attempting to retrieve or depend on alternates for command and control. Not all speech recognition engines support command and control or proprietary grammar alternates. If the particular engine does not support alternates, this method will indicate it has succeeded but the number of alternates returned will always be zero.

CmdMaxAlternates has no effect on dictation alternates. See [ISpeechRecoResult.Alternates](#) for information regarding dictation alternates.

Syntax

Set: *SpeechRecoContext*.**CmdMaxAlternates** = *Long*

Get: *Long* = *SpeechRecoContext*.**CmdMaxAlternates**

Parts

SpeechRecoContext
The owning object.

Long
Set: A Long variable setting the maximum number of alternates.

Get: A Long variable retrieving the maximum number of alternates.

Remarks

The current version of the Microsoft speech recognition engine supplied with SAPI 5 does not support command and control alternates. However, other manufacturer's engines may.

Example

The following snippet demonstrates retrieving the current number of alternates. However, because not all engines support the alternates feature, a test is performed beforehand. If the attribute does not exist, the application receives a run-time error of SPERR_NOT_FOUND. The On Error statement provides a graceful handling in that case. If the attribute does exist, no error will occur and *cfgAttribute* will be valid, even if only as Empty.

The samples assumes a valid *RecoResult*.

```
Dim objToken As Object
Set objToken = RecoResult.RecoContext.Recognizer.Recognizer

On Error GoTo ErrorHandler

Dim cfgAttribute As String
cfgAttribute = objToken.GetAttribute("CGFAlternates")

Dim numAlternates As Long
numAlternates = RecoResult.RecoContext.CmdMaxAlternates
Exit Sub

ErrorHandler:
    'Error handling code here
    Debug.Print Err.Number
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

CreateGrammar Method

The **CreateGrammar** method creates an object based on ISpeechRecoGrammar.

Before speech recognition takes place, a speech recognition (SR) engine requires two things: grammar creation and grammar activation. A recognizer may have more than one grammar associated with it although they are usually limited to one each of two types: dictation and context free grammar (CFG). CFG is used for command and control. The recognizer can have more than one active grammar of the same type open at one time. In this case, the recognition is assigned to the grammar in which a unique match is made. If more than one grammar can match the recognition, the earliest opened grammar will receive the recognition.

A grammar must be activated prior to use. Call [ISpeechRecoGrammar.DictationSetState](#) to activate or deactivate a dictation grammar. [ISpeechRecoGrammar.CmdSetRuleState](#) is used to activate or deactivate a command and control rule which also controls the associated grammar. By controlling the state of grammars, different grammars may be used at different times.

```
SpeechRecoContext.CreateGrammar(  
    [GrammarId As Variant = 0]  
) As ISpeechRecoGrammar
```

Parameters

GrammarId

[Optional] Specifies the GrammarId. The GrammarId identifies each grammar. The values do not have to be

unique although each grammar instance can only have one identifier. The default value is zero.

Return Value

The CreateGrammar method returns an ISpeechRecoGrammar variable.

Example

The following snippet demonstrates creating and activating a dictation grammar with a GrammarId of zero. The zero for the GrammarId is not required as it is the default. Activate the grammar by calling [ISpeechRecoGrammar.DictationSetState](#).

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar

Set RC = New SpSharedRecoContext
Set myGrammar = RC.CreateGrammar(0)
myGrammar.DictationLoad
myGrammar.DictationSetState SGDSActive
```

The next example demonstrates creating and activating a command and control grammar. The grammar still has a GrammarId of zero (the default value), as the example above does. In addition to creating and activating the grammar, the grammar file must also be described by calling one of the command load methods. See the [ISpeechRecoGrammar interface](#) for more loading options. In this case, [ISpeechRecoGrammar.CmdLoadFromFile](#) is used to load the command file *sol.xml*.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Set RC = New SpSharedRecoContext  
Set myGrammar = RC.CreateGrammar
```

```
myGrammar.CmdLoadFromFile "sol.xml", SLODynamic  
myGrammar.CmdSetRuleIdState 0, SGDSActive
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

CreateResultFromMemory Method

The **CreateResultFromMemory** method creates a recognition result object from a saved recognition result.

The result must have been created with [ISpeechRecoResult.SaveToMemory](#).

```
SpeechRecoContext.CreateResultFromMemory(  
    ResultBlock As Variant  
) As ISpeechRecoResult
```

Parameters

ResultBlock

A Variant variable containing a saved recognition result.

Return Value

An ISpeechRecoResult object.

Example

The following Visual Basic form code demonstrates the use of the *SpeechRecoContext.CreateResultFromMemory* and [ISpeechRecoResult.SaveToMemory](#). The application displays the text of the current recognition as well as the previous one. This application also plays the the retained audio associated with the last recognition.

To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
Dim gLastPhrase As Variant

Private Sub Form_Load()
    Set RC = New SpSharedRecoContext
    RC.RetainedAudio = SRAORetainAudio
    Set myGrammar = RC.CreateGrammar

    myGrammar.DictationSetState SGDSActive
    gLastPhrase = Empty
End Sub

Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    If IsEmpty(gLastPhrase) = False Then
        Dim GetRecoResult As ISpeechRecoResult
        Set GetRecoResult = RC.CreateResultFromMemory(gLastPh

        savedText = GetRecoResult.PhraseInfo.GetText()
        Label1.Caption = "Last phrase: " & savedText
        GetRecoResult.SpeakAudio
    End If

    Label1.Caption = Label1.Caption & vbCrLf & "New phrase:

    Dim thePhrase As Variant
    gLastPhrase = Result.SaveToMemory
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

EventInterests Property

The **EventInterests** property specifies the types of events accepted by the `SpeechRecoContext` object.

An event interest is a filtering mechanism for each recognition context. By setting `EventInterests`, the recognition context allows or denies speech recognition engine events to reach the application. All, none or selected types of events may be filtered. By default, speech recognition allows all events except `SREAudioLevel` (a change in audio level).

Syntax

```
Set: SpeechRecoContext.EventInterests =  
SpeechRecoEvents
```

```
Get: SpeechRecoEvents =  
SpeechRecoContext.EventInterests
```

Parts

SpeechRecoContext
The owning object.

SpeechRecoEvents

Set: One or more `SpeechRecoEvents` constants which set the property.

Get: A `SpeechRecoEvents` variable which gets the value of the property.

Remarks

The set of event constants defined for `EventInterests` is located in the [SpeechRecoEvents](#) enumeration list. Each type of event is represented by its own enumerated value. The value passed in to `EventInterests` is the total of each of the events required for the application. For example,

```
SRESoundStart + SRERecognition
```

could be passed as the value. Alternatively the same two interests could be passed as the single value of 18; `SRESoundStart` (value 2) plus `SRERecognition` (value 16). Conversely, if the application retrieved the current event setting and the value was 1064, it indicates that three events are active. Only `SREPhraseStart` (value 8), `SREHypothesis` (value 32), and `SREInterference` (value 1024) add up to that 1064 value.

See [SpVoice.EventInterests](#) for additions details.

`SpVoice.EventInterests` is a similar function but affects text-to-speech events.

Example

The first snippet demonstrates retrieving the current event interest level. This sample assumes a valid *RecognitionContext*. If this value was not changed, *myInterests* is 327,679 (all `SpeechRecoEvents` values except `SREAudioLevel`). Both samples assume a valid *RecognitionContext*.

```
Dim myInterests As SpeechRecoEvents  
myInterests = RecognitionContext.EventInterests
```

The next snippet demonstrates setting the event interest to three events. Instead of naming each value, a single value of 578 may be used: the sum of the three event values. The last two lines perform the same function and while displayed here for demonstration purposes, need not be used together.

```
Dim myInterests As SpeechRecoEvents
```

```
RecognitionContext.EventInterests = SRESoundStart + SREBookm  
RecognitionContext.EventInterests = 578
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

Pause Method

The **Pause** method pauses the engine object to synchronize with the speech recognition (SR) engine.

Pause stops the SR engine at a synchronization point to change grammars and rule states. A synchronization point occurs after any Recognition event, Bookmark event or as soon as possible after explicitly calling Pause. After synchronizing grammars and rule states, the engine continues recognizing if Resume is called.

After the application has changed the state or grammar, it should call [ISpeechRecoContext.Resume](#). A call to Resume must be made for every call made to Pause. SAPI will automatically feed the buffered audio data into the SR engine, ensuring that no real-time audio data is lost and that the user experience is not interrupted. SAPI will restart the SR engine once Resume has been called.

During the pause, SAPI continues to collect and store audio input in an audio buffer. The SAPI audio buffer has a static limit to prevent SAPI applications or SR engines from consuming large amounts of system memory. If the speech recognition engine pauses too long, and the audio buffer fills, then a buffer overflow (SPERR_AUDIO_BUFFER_OVERFLOW) occurs. This would result in interruptions of other applications running SAPI. The buffer is set to 30 times the average bytes per second or approximately 30 seconds. Consequently, the audio data collected between the point when the buffer overflow occurred, and when the stream was reactivated, will be completely lost. Use Pause only for very short periods and call Resume immediately once grammars and rules states have changed.

SpeechRecoContext.**Pause()**

Parameters

None.

Return Value

None.

Remarks

Grammar and rule state changes can be requested while the SR engine is running. However, the changes will not take place until the engine stops and synchronizes. Since a Recognition event is a common synchronization point, in many situations, it may not be necessary to call Pause. However, if the grammar or state change needs to be implemented immediately, call Pause, make the change and then call Resume.

The SAPI 5 SR engine synchronizes close to every 60 seconds, which aids in timely shutdowns and avoids problems with loud and continuous background noises. There is no requirement for other manufacturer's engines to also synchronize like this although it is encouraged.

Example

The following code snippet demonstrates uses Pause and Resume. The example code is Paused allowing the user to change the state or grammar.

```
Public WithEvents RC As SpSharedRecoContext  
Set RC = New SpSharedRecoContext
```

```
'setup the recognition context  
'...
```

'pause the context so that event notifications are not received'
RC.Pause

'[quickly] perform the processing, as stated above'
'...'

RC.Resume
'applications will start receiving event notifications again'

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

Recognizer Property

The **Recognizer** property identifies the recognizer associated with the recognition context.

Syntax

Set: (This property is read-only)

Get: [ISpeechRecognizer](#) = *SpeechRecoContext*.**Recognizer**

Parts

SpeechRecoContext
The owning object.

SpeechRecognizer
Set: (This property is read-only)
Get: A *SpeechRecognizer* object that gets the value of the property.

Remarks

Though it is usually acceptable to declare an object with the exact class name, *ISpeechRecognizer* is most often implemented as either *SpSharedRecognizer* or *SpInprocRecognizer*. Since it is not always known ahead of time which type of recognizer will be used, it is safer to declare the object as *Object*.

Example

The following code snippets demonstrates retrieving the recognizer associated with recognition context. The first

example is a shared recognizer.

```
Public WithEvents RecognitionContext As SpSharedRecoContext  
  
Set RecognitionContext = New SpSharedRecoContext  
Dim theRecognizer As Object  
Set theRecognizer = RecognitionContext.Recognizer
```

The second example is an InProc recognizer.

```
Public WithEvents RecognitionContext As SpInprocRecoContext  
  
Set RecognitionContext = New SpInprocRecoContext  
Dim theRecognizer As Object  
Set theRecognizer = RecognitionContext.Recognizer
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

RequestedUIType Property

The **RequestedUIType** property specifies the UIType of the last UI requested from the engine.

After a speech recognition (SR) engine sends a RequestUI event, the UIType persists until the next RequestUI event. This way the application can check for the last requested UI type. If no UI has been requested, the UIType string will be Empty.

Syntax

Set: (This property is read-only)

Get: *String* = *SpeechRecoContext*.**RequestedUIType**

Parts

SpeechRecoContext
The owning object.

String

Set: (This property is read-only)

Get: A String variable specifying the UIType. The UIType is a String corresponding to the UI requested. For a list of available SAPI 5 UI, see [Engine User Interfaces](#).

Remarks

See [RequestUI](#) event, [ISpeechRecognizer.DisplayUI](#), and [Engine User Interfaces](#) for more information.

Example

The following code snippet demonstrates retrieving the UI last requested from the engine. Due to the complexity of replicating a RequestUI, this is not a complete code sample.

This sample assumes a valid *RC* as the recognition context.

```
Dim theUI As String  
theUI = RC.RequestedUIType
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

Resume Method

The **Resume** method releases the speech recognition (SR) engine from the paused state and restarts the recognition process.

See [ISpeechRecoContext.Pause](#) for details.

SpeechRecoContext.Resume()

Parameters

None.

Return Value

None.

Example

See [ISpeechRecoContext.Pause](#) for details.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

RetainedAudio Property

The **RetainedAudio** property gets and sets the audio retention status of the recognition context.

By default, a recognition context does not retain audio and is initially set to `SRAONone`. Calls attempting to access non-existent audio result in an `SPERR_NO_AUDIO_DATA` error. The calls [ISpeechRecoResult.Audio](#) and [ISpeechRecoResult.SpeakAudio](#) result in this error. The error can also occur when setting an `SpAudioFormat` instance.

To retain the audio, set this property to `SRAORetainAudio`.

Syntax

```
Set: SpeechRecoContext.RetainedAudio =  
    SpeechRetainedAudioOptions  
Get: SpeechRetainedAudioOptions =  
    SpeechRecoContext.RetainedAudio
```

Parts

SpeechRecoContext
The owning object.

SpeechRetainedAudioOptions
Set: A `SpeechRetainedAudioOptions` constant that sets the property.
Get: A `SpeechRetainedAudioOptions` constant that gets the property.

Example

The following Visual Basic form code demonstrates the use of the *ISpeechRecoContext.RetainedAudio*. The application displays the text of the recognition along with the actual spoken part.

To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext
    RC.RetainedAudio = SRAORetainAudio
    Set myGrammar = RC.CreateGrammar

    myGrammar.DictationSetState SGDSActive
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Label1.Caption = Result.PhraseInfo.GetText
    Result.SpeakAudio
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

RetainedAudioFormat Property

The **RetainedAudioFormat** property gets and sets the format of audio retained by the recognition context.

By default, the retained audio format will be the same as the input audio format. The input audio format is retrieved by calling [Recognizer.GetFormat](#) with SFTInput as the parameter. The audio format may be set, or reset if the format has previously changed, to the same format as the engine uses by passing Nothing in as the parameter.

Syntax

```
Set: SpeechRecoContext.RetainedAudioFormat =  
    SpAudioFormat
```

```
Get: SpAudioFormat =  
    SpeechRecoContext.RetainedAudioFormat
```

Parts

SpeechRecoContext
The owning object.

SpAudioFormat
Set: An SpAudioFormat object that sets the audio format.
Get: An SpAudioFormat object that returns the audio format.

Remarks

The recognition context's RetainedAudio need not be set to SRAORetainAudio for RetainedAudioFormat to be called

successfully. `RetainedAudioFormat` indicates the format in which the audio would be retained, regardless of whether it is currently retained.

Example

The following snippet demonstrates retrieving the audio format of the audio input stream, through the `RetainedAudioFormat` reference.

```
Set RC = New SpSharedRecoContext
```

```
Dim audioFormat As SpAudioFormat
```

```
Set audioFormat = RC.RetainedAudioFormat
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

SetAdaptationData Method

The **SetAdaptationData** method passes the speech recognition (SR) engine a string of adaptation data.

An application can improve recognition accuracy for dictating uncommon words, or uncommon word groupings by training the SR engine for the new words or word groupings. An application creates or obtains typical text and sends the results to the engine using the SetAdaptationData method.

```
SpeechRecoContext.SetAdaptationData(  
    AdaptationString As String  
)
```

Parameters

AdaptationString
Specifies the AdaptationString.

Return Value

None.

Remarks

Applications using adaptation data should break the data into small sections (1KB or less) and submit the sections individually. First, specify interest in adaptation events using the [ISpeechRecoContext.EventInterests](#) method. The Adaptation event interest is on by default. Then, send a small data section

to the SR engine using the `SetAdaptationData` method, and wait for an [*ISpeechRecoContext.Adaptation*](#) event, which indicates that the adaptation data has been processed. Send all successive data sections in this way. Finally, use the `EventInterests` method to turn off Adaptation events. Since this event is returned only after an explicit `SetAdaptationData` call, the event interest does not need to be removed unless an excessive number of words is added and the processing time becomes unacceptable. The Adaptation event indicates that the engine has processed the `AdaptationString` and that it is ready to accept another `SetAdaptationData` call.

Example

The following code snippet demonstrates adding an uncommon word. After the word is adapted by the recognizer, this sample posts a message to a hypothetical `Label1` in the application. The sample assumes a valid *RC* (as the recognition context).

```
RC.SetAdaptationData ("Simmiting")
```

```
'Speech processing code here
```

```
Private Sub RC_Adaptation(ByVal StreamNumber As Long, ByVal :  
    Label1.Caption = "Word added"  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

State Property

The **State** property gets or sets the active state of the recognition context.

The entire grammar associated with the recognition context can be disabled or enabled. The individual rule states of the grammar are unaffected otherwise. These conditions allow the application to control the state of the grammars at a high level. For example, if the window loses the current focus, the recognition context (s) associated with that window may be disabled if recognitions are not needed. Likewise, when the window regains the focus, all the recognition contexts may be enabled.

Syntax

Set: `SpeechRecoContext.State` = [SpeechRecoContextState](#)

Get: [SpeechRecoContextState](#) = `SpeechRecoContext.State`

Parts

SpeechRecoContext
The owning object.

SpeechRecoContextState
Set: A `SpeechRecoContextState` variable that sets the property.
Get: A `SpeechRecoContextState` variable that gets the property.

Example

The following code snippet demonstrates disabling and enabling a command and control grammar. The sample opens a configuration file, sol.xml, and sets one rule, identified as 101, as active. The application could continue and disable the entire grammar so that no recognitions are possible based on that grammar. Later, when the application enables the grammar, the same single rule (101) would still be active.

```
Set RC = New SpSharedRecoContext
```

```
Set myGrammar = RC.CreateGrammar  
myGrammar.CmdLoadFromFile "sol.xml", SLODynamic  
myGrammar.CmdSetRuleIdState 101, SGDSActive
```

```
'Possible program execution could go here  
RC.State = SRCS_Disabled
```

```
'Possible program execution could go here  
RC.State = SRCS_Enabled
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

Voice Property

The **Voice** property specifies the SpVoice object associated with the recognition context.

Syntax

```
Set: SpeechRecoContext.Voice = SpVoice
```

```
Get: SpVoice = SpeechRecoContext.Voice
```

Parts

SpeechRecoContext
The owning object.

SpVoice
Set: An SpVoice object that sets the Voice property.
Get: An SpVoice object that gets the Voice property.

Remarks

ISpeechRecoContext.Voice allows the voice to be changed temporarily and for limited contexts. Change the voice using Speech properties in Control Panel.

Example

The first code snippet demonstrates speaking a successful recognition. The code represented is from the recognition event. Because SAPI makes extensive use of defaults, each recognition context will use the default system voice, which is specified using Speech properties in Control Panel.

```
Public WithEvents RC As SpSharedRecoContext
```

```
'Speech processing code goes here
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal  
    Label1.Caption = Result.PhraseInfo.GetText  
    RC.Voice.Speak Label1.Caption  
End Sub
```

The next code sample changes the current voice to "Microsoft Sam" if it is available. The sample then speaks the new name. The sample assumes a valid *RC* at the time of the voice change.

```
Public WithEvents RC As SpSharedRecoContext
```

```
Set RC.Voice.Voice = RC.Voice.GetVoices("name=Microsoft Sam"  
RC.Voice.Speak "I have changed to " & RC.Voice.Voice.GetDesc
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext](#)

VoicePurgeEvent Property

The **VoicePurgeEvent** property gets and sets the RecoContext event which will stop the RecoContext's voice and purge the voice queue.

Applications can use the Voice property of a RecoContext object to prompt a user for spoken input. Setting the VoicePurgeEvent to the SRESoundStart event will cause the RecoContext's voice to stop speaking when the user begins speaking.

Syntax

```
Set: SpeechRecoContext.VoicePurgeEvent =  
    SpeechRecoEvents  
Get: SpeechRecoEvents =  
    SpeechRecoContext.VoicePurgeEvent
```

Parts

SpeechRecoContext
The owning object.

SpeechRecoEvents
Set: A SpeechRecoEvents constant that sets the VoicePurgeEvent.
Get: A SpeechRecoEvents constant that gets the VoicePurgeEvent

Example

The following Visual Basic form code demonstrates the use of the VoicePurgeEvent property. To run this code, create a form

with the following controls:

- Two command buttons, called Command1 and Command2

Paste this code into the Declarations section of the form.

The code creates a RecoContext with a grammar and a voice. The grammar is loaded with a set of rules so that the RecoContext can begin recognition as soon as the grammar is activated. Both command button procedures speak text which contains a bookmark. The RecoContext's bookmark event activates the grammar, which initiates recognition, and the activated RecoContext receives an SRESoundStart event.

In the Command1_Click procedure, the voice continues speaking after recognition has begun, because the VoicePurgeProperty has been set to zero, and the SRESoundStart event does not effect either the RecoContext or its Voice. In the Command2_Click procedure, the VoicePurgeProperty causes the SRESoundStart event to stop the voice.

```
Dim WithEvents Voice As SpVoice
Dim Context As SpSharedRecoContext
Dim RecoGrammar As ISpeechRecoGrammar
```

```
Const SpeakStr1 = "Recognition is started by the next bookma
Const SpeakStr2 = "<bookmark mark='first'/> but the voice ke
```

```
Private Sub Command1_Click()
```

```
    'Speak with no VoicePurgeEvent
```

```
    Context.VoicePurgeEvent = 0
    Voice.Speak SpeakStr1, SVSFIsXML + SVSFlagsAsync
    Voice.Speak SpeakStr2, SVSFIsXML + SVSFlagsAsync
    Voice.WaitUntilDone (999)
    RecoGrammar.CmdSetRuleState "", SGDSInactive
```

```
End Sub
```

```
Private Sub Command2_Click()
```

```
'Speak with VoicePurgeEvent on SRESoundStart
```

```
Context.VoicePurgeEvent = SRESoundStart  
Voice.Speak SpeakStr1, SVSFIsXML + SVSFlagsAsync  
Voice.Speak SpeakStr2, SVSFIsXML + SVSFlagsAsync  
Voice.WaitUntilDone (999)  
RecoGrammar.CmdSetRuleState "", SGDSInactive
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
Set Context = New SpSharedRecoContext  
Set RecoGrammar = Context.CreateGrammar(123)  
RecoGrammar.CmdLoadFromFile "c:\sol.xml"
```

```
'The voice must be  
Set Voice = Context.Voice  
Voice.EventInterests = SVEBookmark + SVEEndInputStream +
```

```
End Sub
```

```
Private Sub Voice_Bookmark(ByVal StreamNumber As Long, ByVal
```

```
'after the first bookmark, we activate the grammar,  
'and the SR sound start event should pause TTS voice.  
RecoGrammar.CmdSetRuleState "", SGDSActive 'active the
```

```
End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechRecoContext (Events)

The **ISpeechRecoContext (Events)** automation interface defines the types of events that a recognition context can receive.

Automation Interfaces

The ISpeechRecoContext (Events) automation interface contains the following elements:

Events	Description
Adaptation Event	Occurs when the SR engine has finished processing a block of adaptation data.
AudioLevel Event	Occurs when the SAPI audio object detects a change in audio level.
Bookmark Event	Occurs when the SR engine encounters a bookmark within the current recognition stream.
EndStream Event	Occurs when the SR engine encounters the end of an input audio stream.
EnginePrivate Event	Occurs when a private SR engine raises a private event.
FalseRecognition Event	Occurs when the SR engine produces a false recognition.
Hypothesis Event	Occurs when the SR engine produces a hypothesis.
Interference Event	Occurs when the SR engine encounters interference in the input audio stream.

<u>PhraseStart</u> Event	Occurs when the SR engine identifies the start of a phrase.
<u>PropertyNumberChange</u> Event	Occurs when the speech recognition engine detects a change in a property number value.
<u>PropertyStringChange</u> Event	Occurs when the speech recognition engine detects a change in a property String value.
<u>Recognition</u> Event	Occurs when the SR engine produces a recognition.
<u>RecognitionForOtherContext</u> Event	Occurs when the recognition context encounters a recognition result that belongs to another recognition context.
<u>RecognizerStateChange</u> Event	Occurs when the SR engine changes state.
<u>RequestUI</u> Event	Occurs when the SR engine requests additional information from the user.
<u>SoundEnd</u> Event	Occurs when the SR engine encounters an end of sound in the audio input stream.
<u>SoundStart</u> Event	Occurs when the SR engine encounters the start of sound in the audio input stream.
<u>StartStream</u> Event	Occurs when the SR engine encounters the start of an audio input stream.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

Adaptation Event

The **Adaptation** event occurs when the SR engine has finished processing a block of adaptation data.

The Adaptation event not only indicates the engine has processed the word but also that the engine is ready to accept another `SetAdaptationData` call.

See [ISpeechRecoContext.SetAdaptationData](#) for additional details.

```
SpeechRecoContext.Adaptation(  
    StreamNumber As Long,  
    StreamPosition As Variant  
)
```

Parameters

StreamNumber
Specifies the stream number.

StreamPosition
Specifies the position within the stream.

Example

See [ISpeechRecoContext.SetAdaptationData](#) for a code example.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

AudioLevel Event

The **AudioLevel** event occurs when the SAPI audio object detects a change in audio level.

The AudioLevel event is the only speech recognition event in SAPI automation that is not set by default. If this event is needed, it must be explicitly set with EventInterests.

```
SpeechRecoContext.AudioLevel(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    AudioLevel As Long  
)
```

Parameters

StreamNumber
Specifies the stream number.

StreamPosition
Specifies the position within the stream.

AudioLevel
Specifies the AudioLevel.

Example

The following Visual Basic form code demonstrates the use of the AudioLevel event. The application displays the audio level of the speaker's voice as well as the text of a successful

recognition. The value of the speaker's voice is shown as both a numeric value as well as a histogram.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar. Because the AudioLevel event is not set by default, it must be explicitly set with EventInterests. In this case, EventInterests is reset for only two events, with a Recognition as the second one.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive

    RC.EventInterests = SRERecognition + SREAudioLevel
End Sub
```

```
Private Sub RC_AudioLevel(ByVal StreamNumber As Long, ByVal AudioLevel As Integer)
    Label2.Caption = Val(AudioLevel)

    Label2.Caption = Label2.Caption & vbCrLf

    For i = 1 To AudioLevel
        Label2.Caption = Label2.Caption & "*"
    Next
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal Recognition As String)
    Label1.Caption = Result.PhraseInfo.GetText
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

Bookmark Event

The **Bookmark** event occurs when the speech recognition (SR) engine encounters a bookmark within the current recognition stream.

See [ISpeechRecoContext.Bookmark](#) for further information.

```
SpeechRecoContext.Bookmark(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    BookmarkId As Variant,  
    Options As SpeechBookmarkOptions  
)
```

Parameters

StreamNumber

Specifies the stream number.

StreamPosition

Specifies the stream position to initiate the Bookmark event. Due to the latency period of the engines, there may be a delay between the cause of the event and the initiation of the event.

BookmarkId

Specifies the BookmarkId. BookmarkId is additional information provided by the application and is unique to the application. As a Variant data type, it may be numeric, String, or any other format and is used to pass information within the method or event. If BookmarkId is a String, the value must be able to convert to a numeric value. This information

is returned back using the Bookmark event.

Options

Specifies the options, specifically whether the recognition pauses during the bookmark processing.

Example

See [ISpeechRecoContext.Bookmark](#) for further information.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

EndStream Event

The **EndStream** event occurs when the speech recognition engine encounters the end of an input audio stream.

```
SpeechRecoContext.EndStream(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    StreamReleased As Boolean  
)
```

Parameters

StreamNumber

Specifies the stream number.

StreamPosition

Specifies the position within the stream.

StreamReleased

Indicates that the stream was released. Usually, stopping a stream does not release it and *StreamReleased* should be False. However, if the stream is not associated with an audio device or if the stream ran out of data while in the run state, the stream will be both stopped and released; in this case, *StreamReleased* will be True.

Example

The following Visual Basic form code demonstrates the use of the StartStream and EndStream events. The application displays

the status of the stream and a stream number. It also displays a successful recognition if a stream is active.

To run this code, create a form with the following controls:

- A command button called Command1
- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form. The Form_Load procedure creates and activates a dictation grammar.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
Public fRecoEnabled As Boolean
```

```
Private Sub Command1_Click()
    If fRecoEnabled = True Then
        myGrammar.DictationSetState SGDSInactive
        fRecoEnabled = False
    Else
        myGrammar.DictationSetState SGDSActive
        fRecoEnabled = True
    End If
End Sub
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext
    Set myGrammar = RC.CreateGrammar

    myGrammar.DictationSetState SGDSActive
    fRecoEnabled = True

    Command1.Caption = "Start Recognition"
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Label1.Caption = Result.PhraseInfo.GetText
End Sub
```

```
Private Sub RC_EndStream(ByVal StreamNumber As Long, ByVal S
    Label2.Caption = "Stream stopped at position: " & Stream
```

```
End Sub
```

```
Private Sub RC_StartStream(ByVal StreamNumber As Long, ByVal  
    Label12.Caption = "Stream number = " & Val(StreamNumber)  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

EnginePrivate Event

The **EnginePrivate** event occurs when a speech recognition engine (SR) raises a private event.

A private event is a custom event defined by the SR engine. This event allows engines to define a specialized event beyond the standard suite of events. Engines are not required support this event. The SAPI 5 Microsoft engines do not use the EnginePrivate event. If using another manufacturer's engine, check their documentation for possible implementation of this event.

```
SpeechRecoContext.EnginePrivate(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    EngineData As Variant  
)
```

Parameters

StreamNumber

Specifies the stream number.

StreamPosition

Specifies the position within the stream.

EngineData

Specifies the private EngineData. This is a Variant data type and is specific to the manufacturer's design. Check the manufacturer's documentation for complete details.

Example

No sample code is available. The event is unique to manufacturer's engines and will vary among engines. The SAPI 5 Microsoft engines do not use the EnginePrivate event.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

FalseRecognition Event

The **FalseRecognition** event occurs when the speech recognition (SR) engine produces a false recognition.

A false recognition is the result of a recognition attempt in which either the word or phrase does not exist in the grammar, or that the speech does not adequately meet the confidence score for the application. Although applicable to dictation grammars, it is more common with command and control instances, because the available words are significantly restricted.

The recognition result returned with a FalseRecognition is still valid and contains all the information a Recognition event does, including the text. Although the text is not necessarily representative of the intended speech, it represents the best estimate of that speech. If using alternates (automatically chosen alternate phrase selections for a recognition), the first alternate returned will likely be the same as the FalseRecognition result.

```
SpeechRecoContext.FalseRecognition(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    Result As ISpeechRecoResult  
)
```

Parameters

StreamNumber
Specifies the StreamNumber.

StreamPosition
Specifies the StreamPosition.

Result

An ISpeechRecoResult object containing the recognition results.

Remarks

There are three possible results from a recognition attempt:

1. The first, a [Recognition](#) event is the result of a successful recognition. This event indicates that the word or phrase was matched to elements in an open grammar, and that the match had a sufficiently high confidence rating.
2. The second possible result is a [FalseRecognition](#) event. This event indicates that speech was detected but it either did not match words in an open grammar, or the match did not merit a high enough confidence rating. The recognition result returned with a FalseRecognition is still valid and contains all the information of a Recognition event, including the text.
3. The third possible result is [RecognitionForOtherContext](#) event. This event indicates a successful recognition result, but a different recognition context was used to match words. SAPI attempts to match the word or phrase in the current recognition context. However, if no match is possible (perhaps the recognition context is currently inactive, the rule is inactive, or the word or phrase is simply not included in the grammar), SAPI then attempts to find a match in other recognition contexts. If a match is found in another application whose grammar is available, the Recognition event is sent to that application instead and the current application receives a RecognitionForOtherContext event.

Example

The following Visual Basic form code demonstrates the use of the FalseRecognition event. The application displays the text of a successful recognition.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar. The grammar file sol.xml also needs to be created and use the XML code from the [RecoCC sample application](#).

The sol.xml file has only one phrase that can be recognized. To use this application speak the phrase "new game" and it should be successfully recognized and displayed in Label1. Any other word or phrase should not be recognized and will display in Label2.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.CmdLoadFromFile "sol.xml", SLODynamic
    myGrammar.CmdSetRuleIdState 0, SGDSActive
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Label1.Caption = Result.PhraseInfo.GetText
    Label2.Caption = ""
End Sub
```

```
Private Sub RC_FalseRecognition(ByVal StreamNumber As Long,
    Label2.Caption = Result.PhraseInfo.GetText
    Label1.Caption = ""
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

Hypothesis Event

The **Hypothesis** event occurs when the SR engine produces a hypothesis.

A hypothesis is an interim recognition result. Each time the engine attempts a recognition it generates an interim results and Hypothesis events are sent out. A hypothesis may or may not be close to the final version of the recognition. In fact, a hypothesis may not bear any likeness to the final result due to the sound quality, idiomatic phrasing, or uncommon word or phrase usage.

The member *Result* is a valid recognition result and may be used in the same way as a [Recognition](#) event. However, the values are interim and could change for the next Hypothesis event.

```
SpeechRecoContext.Hypothesis(  
    StreamNumber AS Long,  
    StreamPosition AS Variant,  
    Result AS ISpeechRecoResult  
)
```

Parameters

StreamNumber
Specifies the stream number.

StreamPosition
Specifies the position within the stream.

Result

An ISpeechRecoResult object containing the recognition results.

Example

The following Visual Basic form code demonstrates the use of the Hypothesis event. The application displays all the hypotheses for the current recognition attempt and then displays the final recognition.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar. The PhraseStart event clears the display each time so the current hypothesis is displayed. For longer recognitions, a larger Label1 may be used or changed to a scrolling text box.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext
```

```
    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive
End Sub
```

```
Private Sub RC_Hypothesis(ByVal StreamNumber As Long, ByVal Result As ISpeechRecoResult)
    Label1.Caption = Label1.Caption & Result.PhraseInfo.GetText
End Sub
```

```
Private Sub RC_PhraseStart(ByVal StreamNumber As Long, ByVal Result As ISpeechRecoResult)
    Label1.Caption = ""
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal Result As ISpeechRecoResult)
    Label2.Caption = Result.PhraseInfo.GetText
End Sub
```

End Sub

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

Interference Event

The **Interference** event occurs when the speech recognition (SR) engine encounters interference in the input audio stream.

Interference may be caused by several factors of which six common ones are listed in the enumeration `SpeechInterference`. The Interference event indicates that the engine detected a condition which could prevent the optimal recognition process. Interference does not prevent the completion of the recognition. However, applications receiving this event may experience a lower quality recognition (the final text does not accurately reflect the spoken text) as the sound quality prevented a successful recognition.

```
SpeechRecoContext.Interference(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    Interference As SpeechInterference  
)
```

Parameters

StreamNumber
Specifies the stream number.

StreamPosition
Specifies the position within the stream.

Interference
A `SpeechInterference` constant that specifies the type of interference.

Example

The following Visual Basic form code demonstrates the use of the Interference event. The application displays the text of a successful recognition and also one of two common interferences results.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar. The interference may be intentional caused by either making a sudden loud noise (such as a sharp increase of your voice or by clapping your hands near the microphone) or talking very quickly.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext
```

```
    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive
End Sub
```

```
Private Sub RC_Interference(ByVal StreamNumber As Long, ByVal Interference As Integer)
    Select Case Interference
        Case SITooFast
            RC.Voice.Speak "Too fast. Speak more slowly."
        Case SITooloud
            RC.Voice.Speak "Too loud. Speak softly."
            Label2.Caption = "SITooloud detected at stream p"
    End Select
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
```

```
Label1.Caption = Result.PhraseInfo.GetText  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

PhraseStart Event

The **PhraseStart** event occurs when the speech recognition (SR) engine identifies the start of a phrase.

```
SpeechRecoContext.PhraseStart(  
    StreamNumber As Long,  
    StreamPosition As Variant  
)
```

Parameters

StreamNumber
Specifies the stream number.

StreamPosition
Specifies the position within the stream.

Remarks

For speech processing, the SR engine must perform the following sequence: Stream start, sound start and phrase start. A stream start indicates a valid stream is ready for audio input. The stream persists unless the recognition context is disabled or the associated grammar is deactivated. The sound start indicates a sound level has been detected. However, it is possible the SR engine could stop that recognition attempt if the input sound were questionable. For example, if the sound were a constant level or if above or below pre-determined sound levels. If the sound level is acceptable and variable, a phrase start is initiated and it is assumed to be the beginning of a

recognition attempt.

Example

The following Visual Basic form code demonstrates the use of the PhraseStart event but also StartStream, SoundStart, SoundEnd, Hypothesis, and Recognition since they can be related. The application displays all the hypotheses for the current recognition attempt in one window and the other events in a second window.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar. The SoundStart event clears the displays each time. For longer recognitions, a larger Label2 may be used or change it to a scrolling text box.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive
End Sub
```

```
Private Sub RC_Hypothesis(ByVal StreamNumber As Long, ByVal :
    Label2.Caption = Label2.Caption & Result.PhraseInfo.GetT
End Sub
```

```
Private Sub RC_PhraseStart(ByVal StreamNumber As Long, ByVal
    Label1.Caption = Label1.Caption & "      Phrase start de
    Label2.Caption = ""
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
```

```
Label1.Caption = Label1.Caption & Result.PhraseInfo.GetT  
End Sub
```

```
Private Sub RC_SoundEnd(ByVal StreamNumber As Long, ByVal St  
Label1.Caption = Label1.Caption & "    Sound end" & vbCrLf  
End Sub
```

```
Private Sub RC_SoundStart(ByVal StreamNumber As Long, ByVal :  
Label1.Caption = "    Sound begin" & vbCrLf  
End Sub
```

```
Private Sub RC_StartStream(ByVal StreamNumber As Long, ByVal  
Label1.Caption = "Stream Number: " & StreamNumber & vbCrLf  
Label2.Caption = ""  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

PropertyNumberChange Event

The **PropertyNumberChange** event occurs when the speech recognition engine detects a change in a property number value.

See [ISpeechRecognizer.GetPropertyNumber](#) for complete details and code sample.

```
SpeechRecoContext.PropertyNumberChange(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    PropertyName As String,  
    NewNumberValue As Long  
)
```

Parameters

StreamNumber

The stream number.

StreamPosition

The stream position.

PropertyName

Specifies the value of property Name.

NewNumberValue

The new numeric value of the property.

Remarks

For a complete list of SAPI 5 supported properties see the [SAPI 5 SR Properties White Paper](#).

Example

See [ISpeechRecognizer.GetPropertyNumber](#) for a complete code sample.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

PropertyStringChange Event

The **PropertyStringChange** event occurs when the speech recognition (SR) engine detects a change in a property String value.

The SAPI 5 SR engine does not support any properties with associated string values. However, other manufacturer's engines could. See [ISpeechRecognizer.GetPropertyNumber](#) for a related and similar feature.

```
SpeechRecoContext.PropertyStringChange(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    PropertyName As String,  
    NewStringValue As String  
)
```

Parameters

StreamNumber

The stream number.

StreamPosition

The stream position.

PropertyName

Specifies the value of property Name.

NewStringValue

The new String value of the property.

Example

See [ISpeechRecognizer.GetPropertyNumber](#) for a related code sample.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

Recognition Event

The **Recognition** event occurs when the speech recognition (SR) engine produces a recognition.

This could be considered the most important event for speech recognition because it returns the result of a successful recognition. A successful recognition is recognized a word or phrase that is matched in an open grammar for that recognition context and whose quality of speech meets a minimum confidence score. If neither criteria is met, the engine returns a `FalseRecognition` event. Spoken content may not meet the confidence score for several reasons including background interference, inarticulate speech or an uncommon word or phrase.

The member *Result* contains the recognition result object and from that may derive much of the information about the speech.

```
SpeechRecoContext.Recognition(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    RecognitionType As SpeechRecognitionType,  
    Result As ISpeechRecoResult  
)
```

Parameters

StreamNumber

Specifies the stream number owning the recognition.

StreamPosition

Specifies the position within the stream.

RecognitionType

A `SpeechRecognitionType` constant that specifies the `RecognitionType` or the recognition state of the engine.

Result

An `ISpeechRecoResult` object containing the recognition results.

Remarks

There are three possible results from a recognition attempt:

1. The first, a [Recognition](#) event is the result of a successful recognition. This event indicates that the word or phrase was matched to elements in an open grammar, and that the match had a sufficiently high confidence rating.
2. The second possible result is a [FalseRecognition](#) event. This event indicates that speech was detected but it either did not match words in an open grammar, or the match did not merit a high enough confidence rating. The recognition result returned with a `FalseRecognition` is still valid and contains all the information of a `Recognition` event, including the text.
3. The third possible result is [RecognitionForOtherContext](#) event. This event indicates a successful recognition result, but a different recognition context was used to match words. SAPI attempts to match the word or phrase in the current recognition context. However, if no match is possible (perhaps the recognition context is currently inactive, the rule is inactive, or the word or phrase is simply not included in the grammar), SAPI then attempts to find a match in other recognition contexts. If a match is found in another application

whose grammar is available, the Recognition event is sent to that application instead and the current application receives a RecognitionForOtherContext event.

Example

The following Visual Basic form code demonstrates the use of the Recognition event. The application displays the text of a successful recognition.

To run this code, create a form with the following control:

- A label called Label1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext
    Set myGrammar = RC.CreateGrammar

    myGrammar.DictationSetState SGDSActive
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Label1.Caption = Result.PhraseInfo.GetText
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

RecognitionForOtherContext Event

The **RecognitionForOtherContext** event occurs when the recognition context encounters a recognition result that belongs to another recognition context.

A RecognitionForOtherContext event indicates a successful recognition but that another application currently running claims it. This is a useful event if multiple shared instances are running at the same time. If the owning application cannot use or claim the recognition, the recognition event is sent to another context that can claim it. In that case, the recognition context claiming it receives the Recognition event and all other contexts receive a RecognitionForOtherContext event.

This event is more applicable to command and control grammars than to dictation. For example, the current recognition context returns a recognition result such as "file." If that recognition context does not have the word "file" in its active grammar, the speech recognition engine searches all the other open and active grammars, even if they belong to other applications. If another context has the word "file" available, the recognition event is sent there instead.

If other recognition contexts should not receive recognition events, because the window or application is not the current one for example, change state of those contexts using [*ISpeechRecoContext.State*](#).

```
SpeechRecoContext.RecognitionForOtherContext(  
    StreamNumber As Long,  
    StreamPosition As Variant  
)
```

Parameters

StreamNumber

Specifies the stream number.

StreamPosition

Specifies the position within the stream.

Remarks

There are three possible results from a recognition attempt:

1. The first, a [Recognition](#) event is the result of a successful recognition. This event indicates that the word or phrase was matched to elements in an open grammar, and that the match had a sufficiently high confidence rating.
2. The second possible result is a [FalseRecognition](#) event. This event indicates that speech was detected but it either did not match words in an open grammar, or the match did not merit a high enough confidence rating. The recognition result returned with a `FalseRecognition` is still valid and contains all the information of a `Recognition` event, including the text.
3. The third possible result is [RecognitionForOtherContext](#) event. This event indicates a successful recognition result, but a different recognition context was used to match words. SAPI attempts to match the word or phrase in the current recognition context. However, if no match is possible (perhaps the recognition context is currently inactive, the rule is inactive, or the word or phrase is simply not included in the grammar), SAPI then attempts to find a match in other recognition contexts. If a match is found in another application whose grammar is available, the `Recognition` event is sent to that application instead and the current

application receives a RecognitionForOtherContext event.

Example

The following Visual Basic form code demonstrates the use of the RecognitionForOtherContext event. The application displays the text of a successful recognition or indicates that the recognition belongs to another application.

The setup for this scenario is somewhat complex. To duplicate two applications running command and control instances, the following code must be compiled into two different applications. Create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar. Create the grammar file sol.xml and use the XML code from the [RecoCC sample application](#). The second application should open *sol2.xml*. This will be the only modification.

The xml files have only one phrase each that can be recognized. Run both applications. Now speak, "new game." The first application gets the recognition while the second application displays "For another context." Now speak "file game" and the situation reverses. These applications will assign and display commands properly regardless of the front-most application.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.CmdLoadFromFile "sol.xml", SLODynamic
    myGrammar.CmdSetRuleIdState 0, SGDSActive
```

```
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal  
    Label1.Caption = Result.PhraseInfo.GetText  
    Label4.Caption = ""
```

```
End Sub
```

```
Private Sub RC_RecognitionForOtherContext(ByVal StreamNumber  
    Label4.Caption = "For another context"  
    Label1.Caption = ""
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

RecognizerStateChange Event

The **RecognizerStateChange** event occurs when the SR engine changes state.

```
SpeechRecoContext.RecognizerStateChange(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    NewState As SpeechRecognizerState  
)
```

Parameters

StreamNumber

Specifies the stream number.

StreamPosition

Specifies the position within the stream.

NewState

A `SpeechRecognizerState` constant specifying the new state of the speech recognition (SR) engine.

Example

The following Visual Basic form code demonstrates the use of the `RecognizerStateChange` event. The application displays the recognition text but also has a button to control the SR engine's state (also called the recognizer). Click this button to toggle the recognizer state between Active and Inactive. The application also beeps after a state change.

To run this code, create a form with the following controls:

- A command button called Command1
- A label called Label1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Command1_Click()
    If RC.Recognizer.State = SRSActive Then
        RC.Recognizer.State = SRSInactive
        Command1.Caption = "SRSInactive"
    Else
        RC.Recognizer.State = SRSActive
        Command1.Caption = "SRSActive"
    End If
End Sub
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive

    Command1.Caption = "SRSActive"
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Label1.Caption = Result.PhraseInfo.GetText
End Sub
```

```
Private Sub RC_RecognizerStateChange(ByVal StreamNumber As L
    Beep
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

RequestUI Event

The **RequestUI** event occurs when the speech recognition (SR) engine requests additional information from the user.

Though not required, SR engines may employ a process improvement procedure and request additional information from the user. For example, if the recognition attempts are consistently poor or if the engine detects consistent background interference, the SR engine could request that the application use the training or microphone wizard. This event is a suggestion by the SR engine to run the particular UI. The application may choose to initiate the UI or may ignore the request.

```
SpeechRecoContext.RequestUI(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    UIType As String  
)
```

Parameters

StreamNumber

Specifies the stream number.

StreamPosition

Specifies the position within the stream.

UIType

Specifies the UIType. The UIType is a String corresponding to the UI requested. For a list of available SAPI 5 UI, see [Engine User Interfaces](#).

Remarks

See [ISpeechRecognizer.DisplayUI](#) and [ISpeechRecognizer.IsUISupported](#) for additional information.

Example

The following Visual Basic code shows a typical RequestUI event. Since the SR engine initiates the call, reproducing the event is difficult. However, if the application receives this event, the code displays the user training wizard. It is the same training wizard that is available through the Speech properties in Control Panel.

```
Private Sub RC_RequestUI(ByVal StreamNumber As Long, ByVal S
    If UIType = SpeechUserTraining Then
        RC.Recognizer.DisplayUI Form1.hwnd, "My User Trainin
    End If
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

SoundEnd Event

The **SoundEnd** event occurs when the speech recognition (SR) engine encounters an end of sound in the audio input stream.

SoundStart indicates a sound level significant enough to be a voice. When that sound stops, a SoundEnd event is generated. A recognition attempt occurs only after a SoundEnd event; hence, long continuous speaking periods may take an equally long time to process.

Light background noise will not register as an input sound. Likewise a loud noise will be considered the start of an input sound. If the sound is constant, a time-out occurs sending a SoundEnd event.

```
SpeechRecoContext.SoundEnd(  
    StreamNumber As Long,  
    StreamPosition As Variant  
)
```

Parameters

StreamNumber
Specifies the StreamNumber.

StreamPosition
Specifies the StreamPosition.

Example

The following Visual Basic form code demonstrates the use of

the SoundStart and SoundEnd events. The application displays a stream number and notifications that a sound has begun or ended. It also displays a successful recognition result.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext
```

```
    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive
```

```
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Label1.Caption = Result.PhraseInfo.GetText
End Sub
```

```
Private Sub RC_SoundEnd(ByVal StreamNumber As Long, ByVal St
    Label2.Caption = "Sound end at position: " & StreamPosit.
End Sub
```

```
Private Sub RC_SoundStart(ByVal StreamNumber As Long, ByVal :
    Label2.Caption = "Sound start"
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

SoundStart Event

The **SoundStart** event occurs when the SR engine encounters the start of sound in the audio input stream.

SoundStart indicates a sound level significant enough to be a voice. When that sound stops, a SoundEnd event is generated. A recognition attempt occurs only after a SoundEnd event; hence, long continuous speaking periods may take an equally long time to process.

Light background noise will not register as an input sound. Likewise a loud noise will be considered the start of an input sound. If the sound is constant, a time-out occurs sending a SoundEnd event.

```
SpeechRecoContext.SoundStart(  
    StreamNumber As Long,  
    StreamPosition As Variant  
)
```

Parameters

StreamNumber
Specifies the stream number.

StreamPosition
Specifies the position within the stream. If downsampling an audio stream, *StreamPosition* will be the byte position within the converted stream.

Remarks

For speech processing, the SR engine must perform the following sequence: Stream start, sound start and phrase start. A stream start indicates a valid stream is ready for audio input. The stream persists unless the recognition context is disabled or the associated grammar is deactivated. The sound start indicates a sound level has been detected. However, it is possible the SR engine could stop that recognition attempt if the input sound were questionable. For example, if the sound were a constant level or if above or below pre-determined sound levels. If the sound level is acceptable and variable, a phrase start is initiated and it is assumed to be the beginning of a recognition attempt.

Example

The following Visual Basic form code demonstrates the use of the SoundStart and SoundEnd events. The application displays a stream number and notifications that a sound has begun or ended. It also displays a successful recognition

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar.

```
Public WithEvents RC As SpSharedRecoContext
```

```
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
```

```
    Set RC = New SpSharedRecoContext
```

```
    Set myGrammar = RC.CreateGrammar
```

```
    myGrammar.DictationSetState SGDSActive
```

```
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
```

```
    Label1.Caption = Result.PhraseInfo.GetText
```

End Sub

```
Private Sub RC_SoundEnd(ByVal StreamNumber As Long, ByVal StreamPosition As Long)
    Label2.Caption = "Sound end at position: " & StreamPosition
End Sub
```

```
Private Sub RC_SoundStart(ByVal StreamNumber As Long, ByVal StreamPosition As Long)
    Label2.Caption = "Sound start"
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecoContext Events](#)

StartStream Event

The **StartStream** event occurs when the speech recognition (SR) engine encounters the start of an audio input stream.

The stream number of each new stream will be incremented from the last stream number, so that each value is unique to the application's lifespan. Stream numbers may be used to track input sources. In the case of recognizing wav files in a batch environment, for instance, stream numbers may be used to uniquely identify the source. In other cases, a StreamStart event can indicate the beginning of a new recognition attempt.

```
SpeechRecoContext.StartStream(  
    StreamNumber As Long,  
    StreamPosition As Variant  
)
```

Parameters

StreamNumber
Specifies the stream number.

StreamPosition
Specifies the position within the stream.

Remarks

For speech processing, the SR engine must perform the following sequence: Stream start, sound start and phrase start. A stream start indicates a valid stream is ready for audio input. The stream persists unless the recognition context is disabled or

the associated grammar is deactivated. The sound start indicates a sound level has been detected. However, it is possible the SR engine could stop that recognition attempt if the input sound were questionable. For example, if the sound were a constant level or if above or below pre-determined sound levels. If the sound level is acceptable and variable, a phrase start is initiated and it is assumed to be the beginning of a recognition attempt.

Example

The following Visual Basic form code demonstrates the use of the StartStream and EndStream events. The application displays the status of the stream and a stream number. It also displays a successful recognition if a stream is active.

To run this code, create a form with the following controls:

- A command button called Command1
- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
Public fRecoEnabled As Boolean

Private Sub Command1_Click()
    If fRecoEnabled = True Then
        myGrammar.DictationSetState SGDSInactive
        fRecoEnabled = False
    Else
        myGrammar.DictationSetState SGDSActive
        fRecoEnabled = True
    End If
End Sub

Private Sub Form_Load()
```

```
Set RC = New SpSharedRecoContext
Set myGrammar = RC.CreateGrammar

myGrammar.DictationSetState SGDSActive
fRecoEnabled = True

Command1.Caption = "Start Recognition"
End Sub

Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Label1.Caption = Result.PhraseInfo.GetText
End Sub

Private Sub RC_EndStream(ByVal StreamNumber As Long, ByVal S
    Label2.Caption = "Stream stopped at position: " & Stream
End Sub

Private Sub RC_StartStream(ByVal StreamNumber As Long, ByVal
    Label2.Caption = "Stream number = " & Val(StreamNumber)
End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



ISpeechRecognizer

The **ISpeechRecognizer** automation interface represents a speech recognition engine.

A recognizer is a speech recognition (SR) engine. Engines are generally categorized by two major characteristics. The first is the language of the engine. The language is provided by the manufacturer and may not be changed. A computer may have several types of engines installed at the same time. However, the use of the engine is limited and is dependent of the recognition type. The second major characteristic is the recognition type, that is, whether the instance of the engine is created as in-process (also known as InProc) or shared. See Recognition Types for more information.

A default SR engine is selected using Speech properties in Control Panel. This is provided as a convenience to users and the default engine will be used if no other type is explicitly specific. In many cases, users requirements can be met by a single engine. However, applications are not restricted to the default engine and may use other engines as needed. In contrast, an active engine is one that is instantiated and being used by at least one recognition context.

More than one instance of an engine may used. Each application may have its own instance of a recognizer and in some situations, each recognition context may have its won engine instance. Application using InProc recognizers must have their own instance. If greater granularity is needed for recognition, it is recommended to use different recognition contexts rather than using multiple recognizers.

The following code snippet declares a shared recognition context and an implicit shared recognizer.

```
Public WithEvents RC As SpSharedRecoContext  
Set RC = New SpSharedRecoContext
```

However, there are cases when a recognizer may be needed prior to declaring a recognition context. To do so, declare the recognizer in the standard fashion.

```
Dim SharedRecognizer As SpSharedRecognizer  
Set SharedRecognizer = CreateObject("SAPI.SpSharedRecognizer")
```

Recognition Types

A recognition engine can be created with one of two types: InProc and shared. The first, InProc, is within the same process as the application. An InProc recognizer restricts access to only that one application. For example, an InProc recognizer would prohibit other applications from using the system microphone. While other applications could run their own instance of an InProc recognizer, no resource could be common among them. An InProc engine may be used, for example when recognizing from a wav file. In fact, a shared engine may not use wav files for input.

The second type of recognizer is the shared recognizer. It is run as a separate process from the application. As a result, other applications may use the engine's resources at the same time. For instance, the same system microphone will be used by all open applications. In turn, the resulting recognition from the engine may be used by any of the applications. In fact, shared engines go so far as to actually inform applications when a recognition result is *not* applicable to them.

For shared engines, a recognizer instance is created automatically when a recognition context is created. In this case, the type of recognizer will be the same as the type of recognition context. That is, a shared recognition context will create a shared recognizer of the same type as the active engine. However, all applications using a shared engine must create instances of that engine type. One application may not use one shared engine and a second application use another engine type.

InProc engines are similarly created in that an InProc engine instance is created when an InProc recognition context is created. InProc engine instances may be created separately. Each application may have only one InProc engine instance active at time. However, while an application can only have one

InProc engine instance at a time, different applications, even if open at the same time, may each have a different engine active. For example, one application may be using an InProc English engine and another application may be recognizing from a Chinese engine.

Regardless, recognizers and recognition contexts must be the same type. If a recognizer is created as a shared resource, resulting recognition contexts associated with that recognizer must also be shared. The same is true for InProc recognizers and recognition contexts. See [Recognition Event](#) for an example. In addition, the recognition context is declared and created without having to declare the recognizer explicitly.

Automation Interface Elements

The ISpeechRecognizer automation interface contains the following elements:

Properties	Description
AllowAudioInputFormatChangesOnNextSet Property	Specifies whether the recognizer can change audio input formats on subsequent audio streams.
AudioInput Property	Gets and sets the recognizer's audio input device.
AudioInputStream Property	Gets and sets the

	recognizer's audio input stream.
<u>IsShared</u> Property	Indicates whether a recognition engine is shared or InProc.
<u>Profile</u> Property	Specifies the recognizer's current recognition profile.
<u>Recognizer</u> Property	Specifies characteristics about the active recognizer.
<u>State</u> Property	Returns the current state of the recognition engine.
<u>Status</u> Property	Returns an object representing the status of the recognizer.

Methods	Description
<u>CreateRecoContext</u> Method	Creates a recognition context object from the recognizer.
<u>DisplayUI</u> Method	Initiates the display of the specified UI.
<u>EmulateRecognition</u>	Emulates recognition from a textual

Method	source rather than from a spoken source.
<u>GetAudioInputs</u> Method	Returns a selection of the available audio input devices.
<u>GetFormat</u> Method	Returns the current input audio format.
<u>GetProfiles</u> Method	Returns a selection of the available user speech profiles.
<u>GetPropertyNumber</u> Method	Returns a numeric value specified by the named key.
<u>GetPropertyString</u> Method	Returns the string value corresponding to the specified key name.
<u>GetRecognizers</u> Method	Returns a selection of SpeechRecognizer objects in the speech configuration database.
<u>IsUISupported</u> Method	Determines if the specified UI is supported.
<u>SetPropertyNumber</u> Method	Sets a numeric property corresponding to the specified name.
<u>SetPropertyString</u> Method	Sets a text property corresponding to the specified name.

Microsoft Speech SDK



Speech Automation 5.1

Object: [ISpeechRecognizer](#)

Type: Hidden

AllowAudioInputFormatChangesOnNextSet Property

The **AllowAudioInputFormatChangesOnNextSet** property specifies whether the recognizer can change audio input formats on subsequent audio streams.

When this property is True, recognizer's input stream format is reset to match the speech recognition engine's preferred format. When this property is False, no changes to the audio format takes place. The default value of this property is True.

The format will not actually be changed until the next time the input is set. Calls to [ISpeechRecognizer.AudiInput](#) and [ISpeechRecognizer.AudiInputStream](#) set the input.

Syntax

```
Set: SpeechRecognizer.AllowAudioInputFormatChangesOnN  
    = Boolean  
Get: Boolean =  
     SpeechRecognizer.AllowAudioInputFormatChangesOnN
```

Parts

SpeechRecognizer
The owning object.

Boolean
Set: A Boolean variable that sets the property.
Get: A Boolean variable that gets the property.

Example

The following snippet assumes a valid recognizer. The sample retrieves the current state of `AllowAudioInputFormatChangesOnNextSet`.

```
Dim Recognizer As SpSharedRecognizer
Dim fAllowFormatChanges As Boolean

fAllowFormatChanges = Recognizer.AllowAudioInputFormatChange
```

This sample sets the current state of `AllowAudioInputFormatChangesOnNextSet` to `False`.

```
Dim Recognizer As SpSharedRecognizer

Recognizer.AllowAudioInputFormatChangesOnNextSet = False
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

AudioInput Property

The **AudioInput** property gets and sets the recognizer's audio input device.

Syntax

Set: *SpeechRecognizer*.**AudioInput** = [SpObjectToken](#)

Get: [SpObjectToken](#) = *SpeechRecognizer*.**AudioInput**

Parts

SpeechRecognizer

The owning object.

SpObjectToken

Set: An SpObjectToken object that sets the property. If this parameter is Nothing, the default audio input device will be used.

Get: An SpObjectToken object that sets the property.

Example

The following Visual Basic form code demonstrates the use of the AudioInput property. The current audio input device (commonly a sound card) is displayed. To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

The Command1 procedure creates a new recognizer object and

displays the default audio input device. It then sets the recognizer's AudioInput property to Nothing and shows the results. Finally, the procedure lists the names of all available audio input devices.

Option Explicit

```
Dim R As SpeechLib.SpSharedRecognizer
Dim T As SpeechLib.SpObjectToken
```

```
Private Sub Command1_Click()
```

```
    Set R = New SpSharedRecognizer
```

```
    Debug.Print "New SpSharedRecognizer"
```

```
    Debug.Print "    AudioInput: " & R.AudioInput.GetDescription
```

```
    Debug.Print
```

```
    Set R.AudioInput = Nothing
```

```
    Debug.Print "Set to Nothing"
```

```
    Debug.Print "    AudioInput: " & R.AudioInput.GetDescription
```

```
    Debug.Print
```

```
    Debug.Print "Show all available inputs"
```

```
    For Each T In R.GetAudioInputs
```

```
        Debug.Print "    AudioInput: " & T.GetDescription
```

```
    Next
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

AudiInputStream Property

The **AudiInputStream** property gets and sets the recognizer's audio input stream.

Using the AudiInputStream enables an application to use file streams or other stream objects for input rather than audio devices.

AudiInputStream is used with InProc recognizers only. Attempting to use it in a shared environment will result in an SPERR_NOT_SUPPORTED_FOR_SHARED_RECOGNIZER error.

Syntax

```
Set: SpeechRecognizer.AudiInputStream =  
    ISpeechBaseStream
```

```
Get: ISpeechBaseStream =  
    SpeechRecognizer.AudiInputStream
```

Parts

SpeechRecognizer
The owning object.

ISpeechBaseStream
Set: An ISpeechBaseStream variable that sets the property. If no value is stated, the default of Nothing will be passed in.
Get: An ISpeechBaseStream variable that gets the property.

Example

The following Visual Basic form code demonstrates the use of AudiInputStream. The application gets a wav file and uses that

as the input source.

To run this code, create a form without any controls. The file is found in the SAPI 5.1 SDK and is a small, one word file. The location is assumed to be on the C: drive, although the string may be changed to accommodate other locations.

```
Dim WithEvents InProcRecoContext As SpInProcRecoContext
Dim InProcRecognizer As Object

Private Sub Form_Load()
    Set InProcRecognizer = CreateObject("SAPI.SpInprocRecogn.
    Set InProcRecoContext = InProcRecognizer.CreateRecoConte:

    Dim FileName As String
    FileName = "C:\Program Files\Microsoft Speech SDK 5.1\Sa
    Dim FileStream As ISpeechFileStream
    Set FileStream = CreateObject("SAPI.SpFileStream")
    FileStream.Open FileName

    Set InProcRecognizer.AudioInputStream = FileStream
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

CreateRecoContext Method

The **CreateRecoContext** method creates a recognition context object from the recognizer.

SpeechRecognizer.CreateRecoContext() As [ISpeechRecoContext](#)

Parameters

None.

Return Value

The CreateRecoContext method returns an ISpeechRecoContext object.

Example

The following Visual Basic form code demonstrates the use of CreateRecoContext.

To run this code, create a form without any controls. Copy this code and paste it into the Declarations section of the form.

```
Private Sub Form_Load()  
    Dim SharedRecognizer As SpSharedRecognizer  
    Set SharedRecognizer = CreateObject("SAPI.SpSharedRecogn.  
  
    Dim myContext As Object  
    Set myContext = SharedRecognizer.CreateRecoContext  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

DisplayUI Method

The **DisplayUI** method initiates the display of the specified UI.

The speech recognition (SR) and text-to-speech (TTS) engines are capable of displaying and running various user interfaces (UI). These displays assist with different aspects of the speech environment:

- User training
- Microphone training wizards
- Adding and removing words
- Setting controls for the engine

Many of these UIs are available using Speech properties in Control Panel. In addition, engines are capable of requesting specific UIs be run to improve a situation. For example, the SR could request more user training if the recognitions are consistently poor.

Engines are not required to support UIs and not all engines will have the same UIs. Consult the manufacturer's engine documentation for specific details. An application may call [*ISpeechRecognizer.IsUISupported*](#) before attempting to invoke a particular UI to see if the engine supports it. Invoking unsupported UIs will cause a run-time error. If the UI is available, use *ISpeechRecognizer.DisplayUI* to invoke the display.

```
SpeechRecognizer.DisplayUI(  
    hWndParent As Long,  
    Title As String,  
    TypeOfUI As String,  
    [ExtraData As Variant = Nothing]  
)
```

Parameters

hWndParent

Specifies the window handle of the owning window.

Title

Specifies the caption used for the UI window.

TypeOfUI

A String specifying the name of the UI to display. For a list of available SAPI 5 UI, see [Engine User Interfaces](#).

ExtraData

[Optional] Specifies the ExtraData. This information is unique to the application and may be used to provide additional or more specific information to the UI. By default, the Nothing value is used and indicates the UI does not use any additional information provided by this method.

Return Value

None.

Remarks

See [ISpeechRecognizer.IsUISupported](#) and [ISpeechRecoContext.RequestUI](#) for additional information.

Example

The following Visual Basic form code demonstrates the use of

the DisplayUI event. The application runs the training wizard, the same one available using Speech properties in Control Panel.

To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar

Private Sub Command1_Click()
    Dim theRecognizer As ISpeechRecognizer
    Set theRecognizer = RC.Recognizer

    theRecognizer.DisplayUI Form1.hWnd, "My App's Additional
End Sub

Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

EmulateRecognition Method

The **EmulateRecognition** method emulates recognition from a textual source rather than from a spoken source.

Using `EmulateRecognition`, applications can accept input from either speech or a textual source. All the events are fired back to the application exactly as if a normal recognition had taken place. The result phrase will have the semantic properties set in the same way as a spoken result. A recognition event will be produced only if the text actually parses through the active rules (if dictation is active, any text will parse).

Since the recognition attempt will not use an audio data, certain events such [Interference](#), [Hypothesis](#), and [AudioLevel](#) cannot occur.

```
SpeechRecognizer.EmulateRecognition(  
    TextElements As Variant,  
    [ElementDisplayAttributes As Variant = SDA_No_Trailing_  
    [LanguageId As Long = 0]  
)
```

Parameters

TextElements

Specifies the elements of the phrase to to recognize. It must be one of two cases.

- If *TextElements* is a BSTR string then it is assumed that the elements in *TextElements* are assumed to be space delimited and *DisplayAttributes* parameter is ignored.
- If *TextElements* is an array of BSTR words then this parameter specifically lists each element in the

phrase. *ElementDisplayAttributes* can be optionally specified as appropriate to the phrase's need.

In either case, additional information may be specified for each element by using the following syntax on each *TextElement*: `"/display_text/lexical_form/pronunciation;"`. This syntax can be used in both the *BSTR* and the array of *BSTRs* case.

ElementDisplayAttributes

[Optional] Specifies the [SpeechDisplayAttributes](#) value for each word element. This value is specific to the language and usually determined by the speech recognition engine. By default the value is *SDA_No_Trailing_Space* and is considered standard for English languages. This parameter is only valid if an array of *BSTRs* for the *TextElements* parameter is specified. It must be one of three cases.

- If *ElementDisplayAttributes* is a NULL pointer, *VT_NULL*, or *VT_EMPTY* then *SDA_No_Trailing_Space* is assumed (which is the default).
- If it is a *BSTR* then it can be "" (empty string), " " (space), or " " (double space) and SAPI matches the *SpeechDisplayAttribute* uses it for all text elements. If an integer value (*VT_I1* to *VT_I4*) is specified, then this value is the *SpeechDisplayAttribute* value and will be used for each element in the words array.
- If it is an array of integer values (*VT_I1* to *VT_I4*) SAPI uses those values for the *SpeechDisplayAttribute*.

LanguageId

[Optional] Specifies the *LanguageId*. This is the same as the Win32 Language Identifier (*LANGID*). By default the value is zero, indicating the system default is used.

Return Value

None.

Remarks

Use this method (simulating speech) to test applications that use speech recognition. Also the restrictions in the parameters *TextElements* and *ElementDisplayAttributes* accommodate languages not using spaces to separate words.

Example

The following Visual Basic form code demonstrates the use of `EmulateRecognition`. The application displays the successful recognition result of dictation. It also emulates speech by clicking the button.

To run this code, create a form with the following controls:

- A label called `Label1`
- A command button called `Command1`

Paste this code into the Declarations section of the form.

The `Form_Load` procedure creates and activates a dictation grammar. Click `Command1` to start emulated speech and display results.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Command1_Click()
    RC.Recognizer.EmulateRecognition ("We the people")
End Sub
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext
    Set myGrammar = RC.CreateGrammar
```

```
    myGrammar.DictationSetState SGDSActive  
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal  
    Label1.Caption = Result.PhraseInfo.GetText  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

GetAudioInputs Method

The **GetAudioInputs** method returns a selection of the available audio input devices.

Audio input devices (sound cards, for example) are stored in the speech configuration database as a series of tokens, with each token representing one audio input device. GetAudioInputs retrieves all available audio tokens. The returned list is an ISpeechObjectTokens object. Additional or more detailed information about the tokens is available in methods associated with ISpeechObjectTokens.

The token search may be further refined using the RequiredAttributes and OptionalAttributes search attributes. Only tokens matching the specified RequiredAttributes search attributes are returned. Of those tokens matching the RequiredAttributes key, OptionalAttributes lists devices in the order matching OptionalAttributes. If no search attributes are offered, all tokens are returned. If no audio devices match the criteria, GetAudioInputs returns an empty selection, that is, an ISpeechObjectTokens collection with an [ISpeechObjectTokens::Count](#) property of zero.

See [Object Tokens and Registry Settings White Paper](#) for a list of SAPI 5-defined attributes.

```
SpeechRecognizer.GetAudioInputs(  
    [RequiredAttributes As String = ""],  
    [OptionalAttributes As String = ""]  
) As ISpeechObjectTokens
```

Parameters

RequiredAttributes

[Optional] Specifies the RequiredAttributes. To be returned by GetAudioInputs, audio input tokens must contain all of the specific required attributes. If no profiles match the selection, the selection returned will not contain any elements. By default, no attributes are required and so returns all the tokens discovered.

OptionalAttributes

[Optional] Specifies the OptionalAttributes. Returned tokens containing the RequiredAttributes are sorted by OptionalAttributes. If OptionalAttributes is specified, the tokens are listed with the OptionalAttributes first. By default, no attribute is specified and the list returned from the speech configuration database is in the order that attributes were discovered.

Return Value

An ISpeechObjectTokens collection containing the selected audio input tokens.

Remarks

The format of selection criteria may either be *Value* or "*Attribute = Value*". Values may be excluded by "*Attribute != Value*".

Example

This code sample demonstrates the GetAudioInputs method. After creating an instance for a recognizer, GetAudioInputs polls the computer for available audio input tokens and displays the results.

To run this code, create a form with the following control:

- A label called Label1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates the recognizer.

```
Private Sub Form_Load()  
    Dim SharedRecognizer As SpSharedRecognizer  
    Set SharedRecognizer = CreateObject("SAPI.SpSharedRecogn.  
  
    Dim theRecognizers As ISpeechObjectTokens  
    Set theRecognizers = SharedRecognizer.GetAudioInputs  
  
    Dim i As Long  
    Dim tokenObject As SpObjectToken  
  
    Label1.Caption = ""  
    For i = 0 To theRecognizers.Count - 1  
        Set tokenObject = theRecognizers.Item(i)  
        Label1.Caption = Label1.Caption & tokenObject.GetDes  
    Next i  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

GetFormat Method

The **GetFormat** method returns the current input audio format.

The sound format may be different at different times or at different points during processing. For example, the audio format at the time the input reaches the sound device (the audio card for instance) may differ from the audio format by the time it reaches the speech recognition (SR) engine. GetFormat specifies which location should be polled and returns the audio format for that location.

```
SpeechRecognizer.GetFormat(  
    Type As SpeechFormatType  
) As SpAudioFormat
```

Parameters

Type

Request for the audio format at entering the sound device or SR engine.

Return Value

The GetFormat method returns an SpAudioFormat variable.

Example

This code sample demonstrates the GetFormat method. After a successful recognition, two GetFormat calls poll SAPI for the audio formats of the sound device and the SR engine and

display results in a label.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates the recognizer.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext
    Set myGrammar = RC.CreateGrammar

    myGrammar.DictationSetState SGDSActive
```

```
End Sub
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal
    Dim audioFormat As SpAudioFormat
```

```
    Label1.Caption = Result.PhraseInfo.GetText
```

```
    Set audioFormat = RC.Recognizer.GetFormat(SFTInput)
    audioFormatType = audioFormat.Type
    Label2.Caption = "Audio input type: " & audioFormat.Type
```

```
    Set audioFormat = RC.Recognizer.GetFormat(SFTSREngine)
    audioFormatType = audioFormat.Type
    Label2.Caption = Label2.Caption & "SR engine input type:
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

GetProfiles Method

The **GetProfiles** method returns a selection of the available user speech profiles.

Profiles are stored in the speech configuration database as a series of tokens, with each token representing one profile. GetProfiles retrieves all available profile tokens. The returned list is an ISpeechObjectTokens object. Additional or more detailed information about the tokens is available in methods associated with ISpeechObjectTokens.

The token search may be further refined using the RequiredAttributes and OptionalAttributes search attributes. Only tokens matching the specified RequiredAttributes search attributes are returned. Of those tokens matching the RequiredAttributes key, OptionalAttributes lists devices in the order matching OptionalAttributes. If no search attributes are offered, all tokens are returned. If no audio devices match the criteria, GetAudioInputs returns an empty selection, that is, an ISpeechObjectTokens collection with an [ISpeechObjectTokens::Count](#) property of zero.

See [Object Tokens and Registry Settings White Paper](#) for a list of SAPI 5-defined attributes.

```
SpeechRecognizer.GetProfiles(  
    [RequiredAttributes As String = ""],  
    [OptionalAttributes As String = ""]  
) As ISpeechObjectTokens
```

Parameters

RequiredAttributes

[Optional] Specifies the RequiredAttributes. To be returned by

GetProfiles, profile tokens must contain all of the specific required attributes. If no profiles match the selection, the selection returned will not contain any elements. By default, no attributes are required and so returns all the tokens discovered.

OptionalAttributes

[Optional] Specifies the OptionalAttributes. Returned tokens containing the RequiredAttributes are sorted by OptionalAttributes. If OptionalAttributes is specified, the tokens are listed with the OptionalAttributes first. By default, no attribute is specified and the list returned from the speech configuration database is in the order that attributes were discovered.

Return Value

An ISpeechObjectTokens object.

Remarks

The format of selection criteria may either be *Value* or "*Attribute = Value*". Values may be excluded by "*Attribute != Value*".

See [ISpeechRecognizer.Profile](#) for related details.

Example

This code sample demonstrates the GetProfiles and Profile method. After creating an instance for a recognizer, GetProfiles polls the computer for available profile tokens. The results are displayed. The first one listed is the current profile. Clicking the button will change the current profile to the first different profile found.

To run this code, create a form with the following controls:

- A label called Label1
- A command button called Command1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates the recognizer. The string functions in the middle of the Form_Load loop lets the application display only the ID portion of the ID string; the complete ID string can be long, exceeding a reasonably sized label.

It is possible for your computer to have only one profile. To demonstrate this application, additional profiles should be added. Add new profiles through Speech properties of Control Panel.

```
Public SharedRecognizer As SpSharedRecognizer
Public theRecognizers As ISpeechObjectTokens

Private Sub Command1_Click()
    Label1.Caption = ""

    Dim currentProfile As SpObjectToken
    Set currentProfile = SharedRecognizer.Profile

    For i = 0 To theRecognizers.Count - 1
        Set tokenObject = theRecognizers.Item(i)
        If tokenObject.Id <> currentProfile.Id Then
            Set SharedRecognizer.Profile = tokenObject
            Label1.Caption = Label1.Caption & "New Profile i
            Exit For
        End If
    Next i
End Sub

Private Sub Form_Load()
    Set SharedRecognizer = CreateObject("SAPI.SpSharedRecogn.

    Set theRecognizers = SharedRecognizer.GetProfiles
```

```
Dim i, idPosition As Long
Dim tokenObject As SpObjectToken

Label1.Caption = ""
For i = 0 To theRecognizers.Count - 1
    Set tokenObject = theRecognizers.Item(i)
    Label1.Caption = Label1.Caption & tokenObject.GetDesi

    idPosition = InStrRev(tokenObject.Id, "\")
    Label1.Caption = Label1.Caption & Mid(tokenObject.Id,

    Label1.Caption = Label1.Caption & vbCrLf
Next i
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

Type: Hidden

GetPropertyNumber Method

The **GetPropertyNumber** method returns a numeric value specified by the named key.

The speech recognition (SR) engine maintains several changeable values for setting the characteristics of the SR process. These controls, or properties, are listed in pairs with the control name and an associated value. For example, the SAPI 5 SR engine defines a property allocating CPU usage for the SR engine. This property is named ResourceUsage, and has a range of 0 to 100 percent. The default value is 50. This number is the percentage of the CPU time allocated for SR processing. This value may be changed to increase or decrease the processing time for SR features.

To get the current value, call [ISpeechRecognizer.GetPropertyNumber](#). To set a new value, call [ISpeechRecognizer.SetPropertyNumber](#). In addition, a [PropertyNumberChange](#) event is sent after the change is made. This event is broadcast to all applications set to receive it. Since multiple applications, or even recognition contexts within the same application can use the same SR engine, the applications may need to react to the new settings appropriately. Applications should restrict making changes unless there is a compelling reason. Changes to the engine properties will take effect at the next synchronization point. The changes persist in the engine until properties are changed again.

Persisting the changes on a permanent basis is an engine design issue. There is no requirement to do so and each engine manufacturer may implement changing characteristics differently. See the manufacturer's engine documentation for specific details. The SAPI 5 engine changes are not permanent beyond the life span of the recognizer object. That is, when all the applications using the same recognizer object (essentially

an SR engine) quit, the values will return to the default state.

If properties need to be changed in a SAPI 5 engine, use Speech properties in Control Panel. For instance, the AdaptationOn property controlling background and continuous adaptation of speech recognition is the same property as Background Adaptation in Settings for the Recognition Profile on the Speech Recognition tab.

Properties are not required of SR engines and each manufacturer's engine may be different. Consult the manufacturer's documentation for specific information.

```
SpeechRecognizer.GetPropertyNumber(  
    Name As String,  
    Value As Long  
) As Boolean
```

Parameters

Name

Specifies the string name of the property.

Value

Specifies the value associated with the property *Name*. This value is passed back upon successful completion of the call. If the return value is False, *Value* will not be updated.

Return Value

A Boolean variable of True if the property is supported, or False if not supported.

Remarks

For a complete list of SAPI 5 supported properties, see the [SAPI 5 SR Properties White Paper](#).

Example

The following Visual Basic form code demonstrates the use of the SetPropertyNumber, GetPropertyNumber, and PropertyNumberChange event. The application displays the current value of the property, in this case ResourceUsage in a label. The first button displays the current value. The second button increments the value by one. The new value displays as a result of the PropertyNumberChange event. The original or starting value will be the same as the Accuracy vs. Recognition Response Time slider in the Recognition Profile Settings. This is set on the SR tab of Speech properties in Control Panel.

To run this code, create a form with the following controls:

- Two command buttons called Command1 and Command2
- Two labels called Label1 and Label2

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
Const THE_PROPERTY = "ResourceUsage"

Private Sub Command1_Click()
    Dim lActualValue As Long
    Dim fSupported As Boolean

    Label2.Caption = ""
    fSupported = RC.Recognizer.GetPropertyNumber(THE_PROPERTY)
    Label1.Caption = lActualValue
End Sub
```

```
Private Sub Command2_Click()  
    Dim lActualValue As Long  
    Dim fSupported As Boolean  
  
    Label1.Caption = ""  
    fSupported = RC.Recognizer.GetPropertyNumber(THE_PROPERTY)  
    fSupported = RC.Recognizer.SetPropertyNumber(THE_PROPERTY)  
End Sub  
  
Private Sub Form_Load()  
    Set RC = New SpSharedRecoContext  
  
    Set myGrammar = RC.CreateGrammar  
    myGrammar.DictationSetState SGDSActive  
End Sub  
  
Private Sub RC_PropertyNumberChange(ByVal StreamNumber As Long)  
    Label2.Caption = NewNumberValue  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

Type: Hidden

GetPropertyString Method

The **GetPropertyString** method returns the string value corresponding to the specified key name.

The SAPI 5 speech recognition (SR) engine does not support any properties with associated string values. However, other manufacturer's engines could. See [ISpeechRecognizer.GetPropertyNumber](#) for a related and similar feature.

```
SpeechRecognizer.GetPropertyString(  
    Name As String,  
    Value As String  
) As Boolean
```

Parameters

Name

Specifies the string name of the property.

Value

Specifies the String value associated with the property *Name*. This value is passed back upon successful completion of the call. If the return value is False, *Value* will not be updated.

Return Value

A Boolean variable of True if the property is supported, or False if not supported.

Example

See [ISpeechRecognizer.GetPropertyNumber](#) for a related code sample.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

GetRecognizers Method

The **GetRecognizers** method returns a selection of SpeechRecognizer objects in the speech configuration database.

Recognizers are stored in the speech configuration database as a series of tokens, with each token representing one recognizer (also called a speech recognition engine). GetRecognizers retrieves all available recognizer tokens and returns them in a list as an ISpeechObjectTokens object. Additional or more detailed information about the tokens is available in methods associated with ISpeechObjectTokens. If no recognizers match the criteria, GetRecognizers returns an empty selection, that is, an ISpeechObjectTokens collection with an [ISpeechObjectTokens::Count](#) property of zero.

The recognizer token search may be further refined by using the RequiredAttributes and OptionalAttributes search attributes. Only token matching the specified search attributes are returned. If no search attributes are offered, all tokens are returned.

See [Object Tokens and Registry Settings White Paper](#) for a list of SAPI 5-defined attributes.

```
ISpeechRecognizer.GetRecognizers(  
    [RequiredAttributes As String = ""],  
    [OptionalAttributes As String = ""]  
) As ISpeechObjectTokens
```

Parameters

RequiredAttributes

[Optional] Specifies the RequiredAttributes. To be returned by GetRecognizers, recognizer tokens must contain all of the

specific required attributes. If no recognizers match the selection, the selection returned will not contain any elements. By default no attributes are required and the method returns all the token discovered.

OptionalAttributes

[Optional] Specifies the OptionalAttributes. Returned tokens containing the RequiredAttributes are sorted by OptionalAttributes. If OptionalAttributes is specified, the tokens are listed with the OptionalAttributes first. By default, no attribute is specified and the list returned from the speech configuration database is in the order that attributes were discovered.

Return Value

A ISpeechObjectTokens collection containing tokens for the selected recognizers.

Remarks

The format of selection criteria may either be *Value* or "*Attribute = Value*". Values may be excluded by "*Attribute != Value*".

Example

This code sample demonstrates the GetRecognizers method. After creating an instance for a recognizer, GetRecognizers polls the computer for available recognizer tokens, which represent individual engines. The results are displayed.

To run this code, create a form with the following control:

- A label called Label1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates the recognizer.

```
Private Sub Form_Load()  
    Dim SharedRecognizer As SpSharedRecognizer  
    Set SharedRecognizer = CreateObject("SAPI.SpSharedRecogn.  
  
    Dim theRecognizers As ISpeechObjectTokens  
    Set theRecognizers = SharedRecognizer.GetRecognizers  
  
    Dim i As Long  
    Dim recoObject As SpObjectToken  
  
    Label1.Caption = ""  
    For i = 0 To theRecognizers.Count - 1  
        Set recoObject = theRecognizers.Item(i)  
        Label1.Caption = Label1.Caption & recoObject.GetDesc  
    Next i  
End Sub
```

The next example is similar to the first except that the "Telephony" attribute is required. Any recognizer token returned must be able to support telephony.

```
Private Sub Form_Load()  
    Dim SharedRecognizer As SpSharedRecognizer  
    Set SharedRecognizer = CreateObject("SAPI.SpSharedRecogn.  
  
    Dim theRecognizers As ISpeechObjectTokens  
    Set theRecognizers = SharedRecognizer.GetRecognizers("Te.  
  
    Dim i As Long  
    Dim recoObject As SpObjectToken  
  
    Label1.Caption = ""  
    For i = 0 To theRecognizers.Count - 1  
        Set recoObject = theRecognizers.Item(i)  
        Label1.Caption = Label1.Caption & recoObject.GetDesc  
    Next i  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

IsShared Property

The **IsShared** property indicates whether a recognition engine is shared or in process (InProc).

Syntax

Set: (This property is read-only)

Get: *Boolean* = *SpeechRecognizer*.**IsShared**

Parts

SpeechRecognizer

The owning object.

Boolean

Set: (This property is read-only)

Get: A Boolean variable ISpeechRecognizer the property. True indicates the recognizer is shared; False indicates that it is an InProc recognizer.

Example

The following Visual Basic form code demonstrates the use of `isShared`. The application displays whether the recognizer is shared or InProc.

To run this code, create a form with the following control:

- A label called `Label1`

Paste this code into the Declarations section of the form.

The Form_Load procedure creates and activates a dictation grammar.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar

Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive

    Dim procShared As Boolean
    procShared = RC.Recognizer.isShared
    If procShared = True Then
        Label1.Caption = "Recognizer is shared."
    Else
        Label1.Caption = "Recognizer is inproc."
    End If
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

IsUISupported Method

The **IsUISupported** method determines if the specified UI is supported.

The speech recognition (SR) and text-to-speech (TTS) engines are capable of displaying and running various user interfaces (UI). These displays assist with different aspects of the speech environment such as user training, microphone wizards, adding and removing words, or setting controls for the engine. Many of these UIs are available using Speech properties in Control Panel. In addition, the engines are capable of requesting that specific UIs are run to improve a situation. For example, the SR could request more user training if the recognitions are consistently poor.

Engines are not required to support UI and not all engines will have the same UI. Consult the manufacturer's engine documentation for specific details. An application may call *ISpeechRecognizer.IsUISupported* before attempting to invoke a particular UI to see if the engine supports it. Invoking unsupported UIs will cause a run-time error. If the UI is available, use [ISpeechRecognizer.DisplayUI](#) to invoke the display.

```
SpeechRecognizer.IsUISupported(  
    TypeOfUI As String,  
    [ExtraData As Variant = Nothing]  
) As Boolean
```

Parameters

TypeOfUI

A String specifying the name of the UI to display. For a list of available SAPI 5 UI, see [Engine User Interfaces](#).

ExtraData

[Optional] Specifies the *ExtraData*. This information is unique to the application and may be used to provide additional or more specific information to the UI. By default the *Nothing* value is used and indicates the UI does not use any additional information provided by this method.

Return Value

The *IsUISupported* method returns a Boolean variable.

Remarks

See [ISpeechRecognizer.DisplayUI](#) and [ISpeechRecoContext.RequestUI](#) for additional information.

Example

The following Visual Basic form code demonstrates the use of the *IsUISupported* event. The application attempts to run two UIs. The first is the training wizard (the same one available using *Speech* properties in *Control Panel*) and the second UI is a nonexistent one called *MyAppUI*.

To run this code, create a form with the following control:

- A command button called *Command1*

Paste this code into the *Declarations* section of the form.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar

Private Sub Command1_Click()
    RunUI SpeechUserTraining
    RunUI "MyAppUI"
End Sub
```

```
Private Sub Form_Load()  
    Set RC = New SpSharedRecoContext  
  
    Set myGrammar = RC.CreateGrammar  
    myGrammar.DictationSetState SGDSActive  
End Sub  
  
Private Function RunUI(theUI As String)  
    If RC.Recognizer.IsUISupported(theUI) = True Then  
        RC.Recognizer.DisplayUI Form1.hwnd, "My App's Additi  
    Else  
        MsgBox theUI & " UI not supported"  
    End If  
End Function
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

Profile Property

The **Profile** property specifies the speech recognition engine's current recognition profile.

A recognition profile represents a single user and training sessions on the system. The user can manually create, delete, and set the current profile using Speech properties in Control Panel.

A profile should not be set while the engine is active. Doing so while a recognition engine is active can cause unexpected results, depending on how and when the speech recognition engine reads the profile information.

The newly-installed profile is not a permanent change but is valid only for the life span of the recognizer. To set it as the default, use [SpObjectTokenCategory.Default](#).

Syntax

```
Set: SpeechRecognizer.Profile = SpObjectToken
```

```
Get: SpObjectToken = SpeechRecognizer.Profile
```

Parts

SpeechRecognizer

The owning object.

SpObjectToken

Set: An SpObjectToken variable that sets the profile. If no value is stated, the default of Nothing will be passed in.

Get: An SpObjectToken variable that gets the current profile.

Example

See [ISpeechRecognizer.GetProfiles](#) for a complete example.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

Recognizer Property

The **Recognizer** property specifies characteristics about the active recognizer.

Recognizers and the attributes associated with them are stored in the speech configuration database as a series of tokens, with each token representing one attribute. Recognizer retrieves an object (*SpObjectToken*) which is capable of accessing the attributes for the recognizer. Additional or more detailed information about the tokens is available in methods associated with *SpObjectToken*.

See [Object Tokens and Registry Settings White Paper](#) for a list of SAPI 5-defined engine attributes.

Syntax

```
Set: SpeechRecognizer.Recognizer = SpObjectToken
```

```
Get: SpObjectToken = SpeechRecognizer.Recognizer
```

Parts

SpeechRecognizer
The owning object.

SpObjectToken
Set: An *SpObjectToken* variable that sets the property.
Get: An *SpObjectToken* variable that gets the property.

Example

This code sample demonstrates the Recognizer property. After creating an instance for a recognizer, the Recognizer property can retrieve attribute information about the active recognizer. The engine ID is displayed. Also two attributes are attempted to be displayed. The first is "SpeakingStyle" (an attribute for the SAPI 5 SR engine. The other is "MyEngineAttribute," and should not be present. Tokens not found will cause a run-time error and as a result, require the error handling code.

To run this code, create a form with the following control:

- A label called Label1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates the recognizer.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    On Error GoTo TokenNotFound
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive

    Label1.Caption = RC.Recognizer.Recognizer.Id & vbCrLf

    Dim objectToken As SpObjectToken
    Set objectToken = RC.Recognizer.Recognizer

    Dim tokenName As String
    tokenName = "SpeakingStyle"
    Label1.Caption = Label1.Caption & tokenName & " : " & ob

    tokenName = "MyEngineAttribute"
    Label1.Caption = Label1.Caption & tokenName & " : " & ob
    Exit Sub
```

```
TokenNotFound:
    Label1.Caption = Label1.Caption & tokenName & " : " & "T
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

Type: Hidden

SetPropertyNumber Method

The **SetPropertyNumber** method sets a numeric property corresponding to the specified name.

See [ISpeechRecognizer.GetPropertyNumber](#) for complete details and code sample.

```
SpeechRecognizer.SetPropertyNumber(  
    Name As String,  
    Value As Long  
) As Boolean
```

Parameters

Name

Specifies the string name of the property.

Value

Specifies the new value of property *Name*. If the return value is False, *Name* will not be changed.

Return Value

A Boolean variable of True if the property is supported, or False if not supported.

Remarks

For a complete list of SAPI 5 supported properties see the [SAPI 5 SR Properties White Paper](#).

Example

See [ISpeechRecognizer.GetPropertyNumber](#) for a complete code sample.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

Type: Hidden

SetPropertyString Method

The **SetPropertyString** method sets a text property corresponding to the specified name.

The SAPI 5 speech recognition (SR) engine does not support any properties with associated string values. However, other manufacturer's engines could. See [ISpeechRecognizer.GetPropertyNumber](#) for a related and similar feature.

```
SpeechRecognizer.SetPropertyString(  
    Name As String,  
    Value As String  
) As Boolean
```

Parameters

Name

Specifies the string name of the property.

Value

Specifies the new String value of property *Name*. If the return value is False, *Name* will not be changed.

Return Value

A Boolean variable of True if the property is supported, or False if not supported.

Example

See [ISpeechRecognizer.GetPropertyNumber](#) for a related code sample.

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

State Property

The **State** property returns the current state of the recognition engine.

The recognition engine may be in one of several states of audio processing. If active, the engine receives audio data and processes it. Likewise, if inactive, neither audio data nor additional processing takes place. See the enumeration `SpeechRecognizerState` for complete details of the states.

Syntax

```
Set: SpeechRecognizer.State = SpeechRecognizerState
```

```
Get: SpeechRecognizerState = SpeechRecognizer.State
```

Parts

SpeechRecognizer

The owning object.

SpeechRecognizerState

Set: A `SpeechRecognizerState` variable that sets the property.

Get: A `SpeechRecognizerState` variable that gets the property.

Example

This code sample demonstrates the `State` method. After creating an instance for a recognizer, `State` retrieves the processing state about the recognizer. The recognizer may be

turned on and off with the button and the current state will be displayed afterward. Speech will be recognized and displayed but only when the engine is active.

To run this code, create a form with the following controls:

- Two labels called Label1 and Label2
- A command button called Command1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates the recognizer. Click the button to toggle the engine on and off. Speaking while the engine is active will display the text in the second label.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Command1_Click()
    If Label1.Caption = 1 Then
        RC.Recognizer.State = SRSInactive
    Else
        RC.Recognizer.State = SRSActive
    End If

    ShowState
    RenameButton
End Sub

Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive

    Dim recoState As SpeechRecognizerState
    recoState = RC.Recognizer.State

    ShowState
    RenameButton
End Sub
```

```
Private Function RenameButton()  
    If Label1.Caption = 1 Then  
        Command1.Caption = "Turn Off Engine"  
    Else  
        Command1.Caption = "Turn On Engine"  
    End If  
End Function
```

```
Private Function ShowState()  
    Dim engineState As SpeechRecognizerState  
    Label1.Caption = RC.Recognizer.State  
End Function
```

```
Private Sub RC_Recognition(ByVal StreamNumber As Long, ByVal  
    Label2.Caption = Result.PhraseInfo.GetText  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Interface: [ISpeechRecognizer](#)

Status Property

The **Status** property returns an object representing the status of the recognizer.

This method provides information for static features about the speech recognition (SR) engine such as the languages it supports. It also provides information for dynamic features such as current stream position the engine has recognized up to, and if the stream is actively being sent to the engine.

Syntax

Set: (This property is read-only)

Get: [ISpeechRecognizerStatus](#) = *SpeechRecognizer*.**Status**

Parts

SpeechRecognizer

The owning object.

ISpeechRecognizerStatus

Set: (This property is read-only)

Get: An *ISpeechRecognizerStatus* that gets the property.

Example

This code sample demonstrates the Status method. After creating an instance for a recognizer, Status retrieves information about the recognizer. The class ID that created the engine, the supported languages, (in decimal format) and the current device position is displayed.

To run this code, create a form with the following control:

- A label called Label1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates the recognizer.

```
Public WithEvents RC As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar
```

```
Private Sub Form_Load()
    Set RC = New SpSharedRecoContext

    Set myGrammar = RC.CreateGrammar
    myGrammar.DictationSetState SGDSActive

    Dim recoStatus As ISpeechRecognizerStatus
    Set recoStatus = RC.Recognizer.Status

    'Display engine CLSID
    Label1.Caption = recoStatus.ClsidEngine & vbCrLf

    'Display supported languages
    Dim i As Long
    Dim x As Variant
    For i = 0 To UBound(recoStatus.SupportedLanguages)
        Label1.Caption = Label1.Caption & recoStatus.Support
    Next i

    'display audio position
    Label1.Caption = Label1.Caption & recoStatus.AudioStatus
End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



SpLexicon

The **SpLexicon** automation object provides access to lexicons. Lexicons contain information about words that can be recognized or spoken.

SAPI defines two types of lexicons. The first is the application lexicon. Application lexicons are the words that all applications can use. They are installed by speech-enabled applications and are read-only. As a result, the application lexicons may vary slightly among computers. The second type is the user lexicon. User lexicons store words specific to a speech user.

An SpLexicon object includes the user lexicon and all application lexicons available on the computer. Calls to SpLexicon methods may return data from several different lexicons.

The SpUnCompressedLexicon object represents a single application lexicon.

Automation Interface Elements

The SpLexicon automation object contains the following elements:

Properties	Description
GenerationId Property	Gets the generation ID of the current application lexicon.

Methods	Description
AddPronunciation Method	Adds a pronunciation, specified in phone symbols, to the current user lexicon.
AddPronunciationByPhonelds	Adds a pronunciation,

Method	specified in phone IDs, to the current user lexicon.
GetGenerationChange Method	Gets a list of words in the current user lexicon that have changed since the specified generation.
GetPronunciations Method	Gets the pronunciations and parts of speech for a word from the user and application lexicons.
GetWords Method	Gets a list of all words in the current user and application lexicons.
RemovePronunciation Method	Removes a word and/or its pronunciations, specified in phone symbols, from the user lexicon.
RemovePronunciationByPhoneIds Method	Removes a word and/or its pronunciations, specified in phone IDs, from the user lexicon.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpLexicon](#)

AddPronunciation Method

The **AddPronunciation** method adds a pronunciation, specified in phone symbols, to the current user lexicon.

```
SpLexicon.AddPronunciation(  
    bstrWord As String,  
    LangId As Long,  
    [PartOfSpeech As SpeechPartOfSpeech = SPSUnknown],  
    [bstrPronunciation As String = ""]  
)
```

Parameters

bstrWord

The word to add.

LangId

The language Id of the word.

PartOfSpeech

[Optional] The PartOfSpeech. Default value is SPSUnknown.

bstrPronunciation

[Optional] The pronunciation, in phones.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpLexicon](#)

Type: Hidden

AddPronunciationByPhoneIds Method

The **AddPronunciationByPhoneIds** method adds a pronunciation, specified in phone IDs, to the current user lexicon.

```
SpLexicon.AddPronunciationByPhoneIds(  
    bstrWord As String,  
    LangId As Long,  
    [PartOfSpeech As SpeechPartOfSpeech = SPSUnknown],  
    [PhoneIds As Variant = Nothing]  
)
```

Parameters

bstrWord

The word to add.

LangId

The language Id of the word.

PartOfSpeech

[Optional] The PartOfSpeech. Default value is SPSUnknown.

PhoneIds

[Optional] The pronunciation, in phone IDs. By default the Nothing value is used.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpLexicon](#)

Type: Hidden

GenerationId Property

The **GenerationId** property gets the generation ID of the current user lexicon.

The GenerationId functions acts as a version number, making it possible to roll back, cancel or undo additions to the lexicon.

Syntax

Set: (This property is read-only)

Get: *Long* = *SpLexicon*.**GenerationId**

Parts

SpLexicon

The owning object.

Long

Set: (This property is read-only)

Get: A Long variable returning the generation ID.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpLexicon](#)

GetGenerationChange Method

The **GetGenerationChange** method gets a list of words in the current user lexicon that have changed since the specified generation.

```
SpLexicon.GetGenerationChange(  
    GenerationID As Long  
) As ISpeechLexiconWords
```

Parameters

GenerationID
Specifies the GenerationID.

Return Value

The GetGenerationChange method returns an ISpeechLexiconWords variable.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpLexicon](#)

GetPronunciations Method

The **GetPronunciations** method gets the pronunciations and parts of speech for a word from the user and application lexicons.

An [ISpeechLexiconPronunciation](#) object contains a word's pronunciations, part of speech and phone ids. Because a word may have more than one pronunciation and more than one part of speech, the GetPronunciations method returns a collection of these objects.

```
SpLexicon.GetPronunciations(  
    bstrWord As String,  
    [LangId As Long = 0],  
    [TypeFlags As SpeechLexiconType = SLTUser | SLTApp]  
) As ISpeechLexiconPronunciations
```

Parameters

bstrWord

The target lexicon word.

LangId

[Optional] The language Id. By default the value is zero which indicates the system LangId is used.

TypeFlags

[Optional] TypeFlags.

Return Value

An ISpeechLexiconPronunciations object, which is a collection of one or more ISpeechLexiconPronunciation objects.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpLexicon](#)

GetWords Method

The **GetWords** method gets a list of all words in the current user and application lexicons.

```
SpLexicon.GetWords(  
    [Flags As SpeechLexiconType = SLTUser | SLTApp],  
    [GenerationID As Long = 0]  
) As ISpeechLexiconWords
```

Parameters

Flags

[Optional] If Flags is SLTUser, user lexicon words are returned; SLTApp returns application lexicon words. By default both types of words are returned.

GenerationID

[Optional] The GenerationID. By default the value is zero.

Return Value

An [ISpeechLexiconWords](#) object, which is a collection of one or more [ISpeechLexiconWord](#) objects.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpLexicon](#)

RemovePronunciation Method

The **RemovePronunciation** method removes a word and/or its pronunciations, specified in phone symbols, from the current user lexicon.

```
SpLexicon.RemovePronunciation(  
    bstrWord As String,  
    LangId As Long,  
    [PartOfSpeech As SpeechPartOfSpeech = SPSUnknown],  
    [bstrPronunciation As String = ""]  
)
```

Parameters

bstrWord

The lexicon word to be removed.

LangId

The language Id.

PartOfSpeech

[Optional] The PartOfSpeech. Default value is SPSUnknown.

bstrPronunciation

[Optional] The pronunciation, in phones, to be removed. If this parameter is not specified, all pronunciations of the word will be removed.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpLexicon](#)

Type: Hidden

RemovePronunciationByPhoneIds Method

The **RemovePronunciationByPhoneIds** method removes a word and/or its pronunciations, specified in phone IDs, from the current user lexicon.

```
SpLexicon.RemovePronunciationByPhoneIds(  
    bstrWord As String,  
    LangId As Long,  
    [PartOfSpeech As SpeechPartOfSpeech = SPSUnknown],  
    [PhoneIds As Variant = Nothing]  
)
```

Parameters

bstrWord

The lexicon word to be removed.

LangId

The language Id.

PartOfSpeech

[Optional] The PartOfSpeech. Default value is SPSUnknown.

PhoneIds

[Optional] The pronunciation, in phone ids, to be removed. If this parameter is not specified, all pronunciations of a lexicon word will be removed.

Return Value

None.

Microsoft Speech SDK

Speech Automation 5.1



SpMemoryStream

The **SpMemoryStream** automation object supports audio stream operations in memory.

The Format property and the Read, Write and Seek methods are inherited from the [ISpeechBaseStream](#) interface.

Automation Interface Elements

The SpMemoryStream automation object has the following elements:

Properties	Description
Format Property	Gets and sets the cached wave format of the stream as an SpAudioFormat object.

Methods	Description
GetData Method	Gets the contents of the stream.
Read Method	Reads data from an audio stream.
Seek Method	Returns the current read position of the audio stream in bytes.
SetData Method	Sets the contents of the stream.
Write Method	Writes data to the audio stream.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpMemoryStream](#)

GetData Method

The **GetData** method gets the entire contents of the stream.

The GetData method does not change the stream's Seek pointer.

```
SpMemoryStream.GetData() As Variant
```

Parameters

None.

Return Value

A Variant variable containing the stream data.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpMemoryStream](#)

SetData Method

The **SetData** method sets the contents of the stream.

The SetData method writes stream data into the memory stream, and sets the Seek pointer to zero, so that the next SetData call to the stream will overwrite the data just written.

```
SpMemoryStream.SetData(  
    Data As Variant  
)
```

Parameters

Data
Specifies the Data.

Return Value

None.

Microsoft Speech SDK

Speech Automation 5.1



SpMMAudioIn

The **SpMMAudioIn** automation object supports audio implementation for the standard Windows wave-in multimedia layer.

Automation Interfaces

The SpMMAudioIn automation object contains the following elements:

Properties	Description
BufferInfo Property	Returns the audio buffer information as an ISpeechAudioBufferInfo object.
BufferNotifySize Property	Gets and sets the audio stream buffer size information.
DefaultFormat Property	Returns the default audio format as an SpAudioFormat object.
DeviceId Property	Gets and sets the multimedia device ID that is used by the audio object.
EventHandle Property	Returns a Win32 event handle that applications can use to wait for status changes in the I/O stream.
Format Property	Gets and sets the cached wave format of the stream as an SpAudioFormat object.
LineId Property	Gets and sets the current line identifier associated with the multimedia device.
MMHandle Property	Returns the handle of the multimedia audio device stream.
Status Property	Returns the audio status as an ISpeechAudioStatus object.

Volume Property Gets and sets the volume level.

Methods	Description
<u>Read</u> Method	Reads data from the audio stream.
<u>Seek</u> Method	Returns the current read position of the audio stream in bytes.
<u>SetState</u> Method	Sets the audio state with a <code>SpeechAudioState</code> constant.
<u>Write</u> Method	Writes data to the audio stream.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpMMAudiIn](#)

Deviceld Property

The **Deviceld** property gets and sets the multimedia device ID that is used by the audio object.

Syntax

Set: *SpMMAudioIn*.**Deviceld** = Long

Get: Long = *SpMMAudioIn*.**Deviceld**

Parts

SpMMAudioIn

The owning object.

Long

Set: A Long variable that sets the device ID.

Get: A Long variable that gets the device ID.

Example

See [Speech Telephony Application Guide](#) for examples with *SpMMAudioIn.Deviceld* property.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpMMAudiIn](#)

LineId Property

The **LineId** property gets and sets the current line identifier associated with the multimedia device.

Syntax

Set: *SpMMAudioIn*.**LineId** = *Long*

Get: *Long* = *SpMMAudioIn*.**LineId**

Parts

SpMMAudioIn

The owning object.

Long

Set: A Long variable that sets the line ID.

Get: A Long variable that gets the line ID.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpMMAudioIn](#)

Type: Hidden

MMHandle Property

The **MMHandle** property returns the handle of the multimedia audio device stream.

Syntax

Set: Not available.

Get: *Long* = *SpMMAudioIn*.**MMHandle**

Parts

SpMMAudioIn

The owning object.

Long

Set: (This property is read-only).

Get: A Long variable that gets the device handle.

Microsoft Speech SDK

Speech Automation 5.1



SpMMAudioOut

The **SpMMAudioOut** automation object supports audio implementation for the standard Windows wave-out multimedia layer.

Automation Interfaces

The SpMMAudioOut automation object has the following elements:

Properties	Description
BufferInfo Property	Returns the audio buffer information as an ISpeechAudioBufferInfo object.
BufferNotifySize Property	Gets and sets the audio stream buffer size information.
DefaultFormat Property	Returns the default audio format as an SpAudioFormat object.
DeviceId Property	Gets and sets the multimedia device ID being used by the audio object.
EventHandle Property	Returns a Win32 event handle that applications can use to wait for status changes in the I/O stream.
Format Property	Gets and sets the cached wave format of the stream as an SpAudioFormat object.
LineId Property	Gets and sets the current line identifier associated with the multimedia device.
MMHandle Property	Returns the handle of the multimedia audio device stream.
Status Property	Returns the audio status as an ISpeechAudioStatus object.

Volume Property Gets and sets the volume level.

Methods	Description
<u>Read</u> Method	Reads data from the audio stream.
<u>Seek</u> Method	Returns the current read position of the audio stream in bytes.
<u>SetState</u> Method	Sets the audio state with a <code>SpeechAudioState</code> constant.
<u>Write</u> Method	Writes data to the audio stream.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpMMAudioOut](#)

Deviceld Property

The **Deviceld** property gets and sets the multimedia device ID being used by the audio object.

Syntax

```
Set: SpMMAudioOut.Deviceld = Long
```

```
Get: Long = SpMMAudioOut.Deviceld
```

Parts

SpMMAudioOut

The owning object.

Long

Set: A Long variable that sets the device ID.

Get: A Long variable that gets the device ID.

Example

See [Speech Telephony Application Guide](#) for examples with *SpMMAudioOut.Deviceld* property.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpMMAudioOut](#)

LineId Property

The **LineId** property gets and sets the current line identifier associated with the multimedia device.

Syntax

```
Set: SpMMAudioOut.LineId = Long
```

```
Get: Long = SpMMAudioOut.LineId
```

Parts

SpMMAudioOut

The owning object.

Long

Set: A Long variable that sets the line ID.

Get: A Long variable that gets the line ID.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpMMAudioOut](#)

Type: Hidden

MMHandle Property

The **MMHandle** property returns the handle of the multimedia audio device stream.

Syntax

Set: Not available.

Get: *Long* = *SpMMAudioOut*.**MMHandle**

Parts

SpMMAudioOut

The owning object.

Long

Set: (This property is read-only).

Get: A Long variable that gets the device handle.

Microsoft Speech SDK

Speech Automation 5.1



SpObjectToken

The SpObjectToken automation object represents an available resource of a type used by SAPI. The default interface for this object is [ISpeechObjectTokens](#).

The Speech configuration database contains folders representing the resources on a computer that are used by SAPI 5.1 speech recognition (SR) and text-to-speech (TTS). These folders are organized into resource categories, such as voices, lexicons, and audio input devices. The [SpObjectTokenCategory](#) object provides access to a category of resources, and the SpObjectToken object provides access to a single resource.

Several Speech Automation objects support methods that return collections of resources from a specific category of available resources. Examples are [SpVoice.GetAudioOutputs](#), [SpVoice.GetVoices](#) and [SpSharedRecognizer.GetProfiles](#), as well as the SpObjectToken object's [MatchesAttributes](#) method. Each of these operations returns an [ISpeechObjectTokens](#) object variable containing a collection of SpObjectToken objects.

The read-only [Id](#) property of an SpObjectToken object is the path to the folder of the resource with which it is associated. The read-only [DataKey](#) property is a data key object providing read and write access to this folder. An SpObjectToken created with the New keyword has an empty Id property, and is therefore not associated with a resource. Before it can be used, a new SpObjectToken must be associated with a resource by means of its [SetId](#) method.

The SpObjectToken object also provides the ability to create and access storage files associated with a resource. The paths of data storage files created by an engine or by applications for a specific resource are stored in its object token.

See the [SpObjectToken Example](#) for a complete example and additional details.

Automation Interface Elements

The SpObjectToken automation interface contains the following elements:

Properties	Description
Category Property	Returns the category of the object token as an SpObjectTokenCategory object.
DataKey Property	Returns the data key of the object token as an ISpeechDataKey object.
Id Property	Returns the ID of the token.

Methods	Description
CreateInstance Method	Creates an instance of the object represented by the token.
DisplayUI Method	Displays the specified UI.
GetAttribute Method	Returns the value of the specified attribute.
GetDescription Method	Returns the name of the resource represented by the object token.
GetStorageFileName Method	Creates a storage file for data associated with the object token.
IsUISupported Method	Determines if the specified UI is supported.
MatchesAttributes Method	Indicates whether the token matches specified attributes.
Remove Method	Removes the token from the speech configuration database.
RemoveStorageFileName Method	Removes a storage file associated with the object token.
SetId Method	Associates a new object token

with a resource by setting its ID property.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

Category Property

The **Category** property returns the category of the object token as an SpObjectTokenCategory object.

If the object token has not been initialized, an attempt to reference this property will generate an SPERR_UNINITIALIZED error.

Syntax

Set: (This property is read-only)

Get: [SpObjectTokenCategory](#) = SpObjectToken.**Category**

Parts

SpObjectToken

The owning object.

SpObjectTokenCategory

Set: (This property is read-only)

Get: An SpObjectTokenCategory object returning the category.

Example

The following Visual Basic form code demonstrates the use of the SpObjectToken's Category and Id properties and its GetDescription method. It also demonstrates the handling of uninitialized object tokens. To run this code, create a form with the following controls:

- A list box called List1

- A command button called Command1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object and a recognizer object. The Command1_Click procedure creates several object tokens and sends them to a subroutine which displays information about them in the list box. The first token sent is a newly-created token with no Category. The subroutine code generates an error when it tries to get the Category property of this token. The other tokens passed to the subroutine are properly initialized, and for each token, the subroutine displays the Description of the token, the ID of the token, and the ID of the token's category.

Option Explicit

```
Dim T As SpeechLib.SpObjectToken
Dim C As SpeechLib.SpObjectTokenCategory
```

```
Dim V As SpeechLib.SpVoice
Dim R As SpeechLib.SpSharedRecognizer
```

```
Const SPERR_UNINITIALIZED = &H80045001;
```

```
Private Sub Command1_Click()
```

```
    List1.Clear
```

```
    'Create new token -- uninitialized
    Set T = New SpObjectToken
    Call TokenInfo("new token", T)
```

```
    'Voice object furnishes Voice tokens and AudioOutput tokens
    Set T = V.GetVoices().Item(0)
    Call TokenInfo("voice token", T)
```

```
    Set T = V.GetAudioOutputs().Item(0)
    Call TokenInfo("audioout token", T)
```

```
    'Recognizer object furnishes recognizer tokens and AudioOutput tokens
```

```

    Set T = R.GetRecognizers().Item(0)
    Call TokenInfo("recognizer token", T)

    Set T = R.GetAudioInputs().Item(0)
    Call TokenInfo("audioin token", T)

End Sub

Private Sub TokenInfo(text, token As SpObjectToken)

    On Error GoTo TokenInfoExit
    List1.AddItem text
    Set C = token.Category

TokenInfoExit:
    Select Case Err.Number

        Case 0
            List1.AddItem " Token.GetDescription:"
            List1.AddItem " " & token.GetDescription
            List1.AddItem " Token.Category.Id:"
            List1.AddItem " " & token.Category.Id
            List1.AddItem " Token.Id:"
            List1.AddItem " " & token.Id

        Case SPERR_UNINITIALIZED
            List1.AddItem " SPERR_UNINITIALIZED"

    End Select
    List1.AddItem ""

End Sub

Private Sub Form_Load()
    'Voice object furnishes Voice tokens and AudioOutput tokens
    'Recognizer object furnishes recognizer tokens and AudioOutput tokens
    Set V = New SpVoice
    Set R = New SpSharedRecognizer
End Sub

```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

CreateInstance Method

The **CreateInstance** method creates an instance of the object represented by the token.

```
SpObjectToken.CreateInstance(  
    [pUnkOuter As IUnknown = Nothing],  
    [ClsContext As SpeechTokenContext = STCALL]  
) As IUnknown
```

Parameters

pUnkOuter

[Optional] Specifies the pUnkOuter. By default, the Nothing value is used.

ClsContext

[Optional] Specifies the ClsContext. By default STCALL is used.

Return Value

An IUnknown object, representing the object instantiated.

Example

For an example of the use of the CreateInstance method, see the code example in the SpAudioFormat [Type](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

Type: Hidden

DataKey Property

The **DataKey** property returns the data key of the object token as an *ISpeechDataKey* object.

A data key object provides read and write access to the contents of a particular folder in the Speech configuration database. The data key of an object token accesses the folder referenced by its *Id* property.

Syntax

Set: (This property is read-only)

Get: [*ISpeechDataKey*](#) = *SpObjectToken*.**DataKey**

Parts

SpObjectToken

The owning object.

ISpeechDataKey

Set: (This property is read-only)

Get: An *ISpeechDataKey* object returning the data key.

Example

Use of the *RemoveStorageFileName* method is demonstrated in the code example for the [GetStorageFileName](#) method.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

Type: Hidden

DisplayUI Method

The **DisplayUI** method displays the specified UI.

```
SpObjectToken.DisplayUI(  
    hWnd As Long,  
    Title As String,  
    TypeOfUI As String,  
    [ExtraData As Variant = Nothing],  
    [Object As IUnknown = Nothing]  
)
```

Parameters

hWnd

Specifies the hWnd.

Title

Specifies the Title.

TypeOfUI

Specifies the TypeOfUI.

ExtraData

[Optional] Specifies the ExtraData. By default, the Nothing value is used.

Object

[Optional] Specifies the Object. By default, the Nothing value is used.

Return Value

None.

Example

For an example of the use of the DisplayUI method, see the example in the Recognizer [DisplayUI](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

GetAttribute Method

The **GetAttribute** method returns the value of the specified attribute.

The String returned contains the value for the Attribute. If the Attribute is present but does not contain additional information, the String will be Empty. If the Attribute is not present, SPERR_NOT_FOUND is returned. Not all engines support all attributes and it is possible to customize attributes for each engine.

```
SpObjectToken.GetAttribute(  
    AttributeName As String  
) As String
```

Parameters

AttributeName
Specifies the AttributeName.

Return Value

The GetAttribute method returns a String variable.

Remarks

In Visual Basic, attempting to access a nonexistent Attribute will cause a run-time error. Therefore, it is recommended to include an On Error statement trapping such cases.

Example

The following code snippet retrieves the information associated with requested attribute. While the actual values will vary among engines. For the Microsoft speech engine, Microsoft Mary is returned for the nameAttribute. VendorAttribute will be Empty since the VendorPreferred attribute has no additional information associated with it. Even so, VendorPreferred has significance merely if it is present or not. However, an SPERR_NOT_FOUND will occur for FakeAttribute since it should not be present.

```
Dim objVoice As SpeechLib.SpVoice
Set objVoice = New SpeechLib.SpVoice

Dim objToken As SpeechLib.SpObjectToken
Set objToken = objVoice.Voice

On Error GoTo ErrorHandler

Dim nameAttribute, vendorAttribute, fakeAttribute As String
nameAttribute = objToken.GetAttribute("Name")
vendorAttribute = objToken.GetAttribute("VendorPreferred")
fakeAttribute = objToken.GetAttribute("FakeAttribute")
Exit Sub

ErrorHandler:
    'Error handling code here
    Debug.Print Err.Number
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

GetDescription Method

The **GetDescription** method returns the name of the resource represented by the object token.

```
SpObjectToken.GetDescription(  
    [Locale As Long = 0]  
) As String
```

Parameters

Locale

[Optional] Specifies the Locale. By default, zero is used.

Return Value

The GetDescription method returns a String variable.

Example

Use of the GetDescription method is demonstrated in the code example in the [Category](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

Type: Hidden

GetStorageFileName Method

The **GetStorageFileName** method creates a storage file for data associated with the object token.

```
SpObjectToken.GetStorageFileName(  
    ObjectStorageCLSID As String,  
    KeyName As String,  
    FileName As String,  
    Folder As SpeechTokenShellFolder  
) As String
```

Parameters

ObjectStorageCLSID

Globally unique identifier (GUID) of the calling object. The method searches the registry for an entry key name of *ObjectStorageCLSID*, and then a corresponding Files subkey. If the registry entry is not present, one is created.

KeyName

The name of the attribute file for the registry entry of *clsidCaller*. This attribute stores the location of the resource file.

FileName

A specifier that is either "" or a path/file name for storage file.

- If this starts with "X:\" or "\\\" it is assumed to be a full path; otherwise it is assumed to be relative to special folders given in the *nFolder* parameter.

- If it ends with a "\", or is NULL, a unique file name will be created. The file name will be something like: "SP_7454901D23334AAF87707147726EC235.dat". "SP_" and ".dat" are the default prefix name and file extension name. The numbers in between are generated guid number to make sure the file name is unique.
- If the name contains a %d the %d is replaced by a number to give a unique file name. The default file extension is .dat, the user can specify anything else. Intermediate directories are created.
- If a relative file is used, the value stored in the registry includes the *nFolder* value as %nFolder% before the rest of the path.

Folder

One or more SpeechTokenShellFolder constants specifying the Folder.

Return Value

A String variable containing the path of the storage file.

Example

The following Visual Basic form code demonstrates the GetStorageFileName and RemoveStorageFileName methods. To run this code, create a form with the following controls:

- Four command buttons, called Command1, Command2, Command3, and Command4

Paste this code into the Declarations section of the form.

The operations performed in this example can best be viewed with REGEDIT.EXE. For a discussion of Registry issues, please see the [ISpeechDataKey interface](#).

The Form_Load procedure creates a new SpObjectToken object, and sets its ID property to a new subfolder called Demo within the Voices\Tokens folder. The True parameter of the SetId call forces the creation of this folder, if it does not already exist.

The Command1 procedure creates a data key object which references the new Demo folder, and uses the data key to write a value into the folder.

The Command2 procedure calls the GetStorageFileName method. After this call, the Demo folder contains a subfolder called {CDD1141B-82FB-405c-99BE-69A793A92D87}. This is the CLSID used as a parameter of the GetStorageFileName method. Within this is a folder called Files, which contains a the storage file path in a value called Demo.

The Command3 procedure calls the RemoveStorageFileName method. This deletes the storage file, and removes the Demo value from the Files folder.

The Command4 procedure uses the ISpeechDataKey.Remove method to delete the {CDD1141B-82FB-405c-99BE-69A793A92D87} folder and Demo folders.

Option Explicit

```
Dim T As SpeechLib.SpObjectToken           'one object token
Dim C As SpeechLib.SpObjectTokenCategory   'one object token
Dim K As SpeechLib.ISpeechDataKey         '
Dim ID As String
Dim SF As String                           'Storage file name
Dim TestCLSID As String
```

```
Private Sub Command1_Click()
```

```
    'Set data key object to the demo voice's folder,
    'Write a CLSID value
```

```

    Set K = T.DataKey
    K.SetStringValue "CLSID", TestCLSID

End Sub

Private Sub Command2_Click()

    'Create a storage file for the token
    SF = T.GetStorageFileName(TestCLSID, "Demo", vbNullString,
                               STSF_FlagCreate + STSF_AppData)
    MsgBox SF

End Sub

Private Sub Command3_Click()

    'Remove the storage file
    Call T.RemoveStorageFileName(TestCLSID, "Demo", True)

End Sub

Private Sub Command4_Click()

    'Remove the "{CDD1141B-82FB-405c-99BE-69A793A92D87}" folder
    'and the "Demo" folder
    T.Remove TestCLSID
    T.Remove ""

End Sub

Private Sub Form_Load()

    TestCLSID = "{CDD1141B-82FB-405c-99BE-69A793A92D87}"

    'Create new category object, set it to Voices category
    Set C = New SpObjectTokenCategory
    C.SetId SpeechCategoryVoices

    'Create new token object, and set its ID
    'to the path of a folder which does not exist
    '"True" parameter creates the folder

```

```
Set T = New SpObjectToken
ID = SpeechCategoryVoices & "\Tokens\Demo"
T.SetId ID, , True
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

Id Property

The **Id** property returns the ID of the token.

The ID of the object is the path to its folder within the Speech configuration database.

Syntax

Set: (This property is read-only)

Get: *String* = *SpObjectToken.Id*

Parts

SpObjectToken

The owning object.

String

Set: (This property is read-only)

Get: A String variable returning the ID.

Example

Use of the Id property is demonstrated in a code example in the [Category](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

Type: Hidden

IsUISupported Method

The **IsUISupported** method determines if the specified UI is supported.

```
SpObjectToken.IsUISupported(  
    TypeOfUI As String,  
    [ExtraData As Variant = Nothing],  
    [Object As IUnknown = Nothing]  
) As Boolean
```

Parameters

TypeOfUI

Specifies the TypeOfUI.

ExtraData

[Optional] Specifies the ExtraData. By default, the Nothing value is used.

Object

[Optional] Specifies the Object. By default, the Nothing value is used.

Return Value

The IsUISupported method returns a Boolean variable. If True, the specified UI is supported; if False, it is not supported.

Example

For an example of the use of the `IsUISupported` method, see the example in the Recognizer [IsUISupported](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

MatchesAttributes Method

The **MatchesAttributes** method indicates whether the token matches specified attributes.

```
SpObjectToken.MatchesAttributes(  
    Attributes As String  
) As Boolean
```

Parameters

Attributes
Specifies the Attributes.

Return Value

A Boolean variable. If True, the token matches the specified attributes; if False, it does not match.

Example

The following Visual Basic form code demonstrates the use of the MatchesAttributes method. To run this code, create a form with the following controls:

- A combo box control called Combo1
- A command button called Command1
- A list box called List1

Paste this code into the Declarations section of the form.

The Form_Load procedure initializes a new

SpObjectTokenCategory object to the voices category, selects all voices into an ISpeechObjectTokens collection, and loads Combo1 with several attribute declarations. These attributes are intentionally irregular with regard to capitalization, spacing and spelling.

The Command1 procedure contains a loop that performs a MatchesAttributes call on each available voice. The name of each voice and the Boolean result of the MatchesAttributes method is displayed in the list box.

Option Explicit

```
Dim C As SpeechLib.SpObjectTokenCategory      'a category of objects
Dim E As SpeechLib.ISpeechObjectTokens        'an enumeration of objects
Dim T As SpeechLib.SpObjectToken              'one object token
```

```
Private Sub Command1_Click()
```

```
    Dim Vname, Vmatch
```

```
    List1.Clear
```

```
    List1.AddItem "Test for voices matching "" " & Combo1.text
```

```
    List1.AddItem ""
```

```
    For Each T In E
```

```
        Vname = T.GetDescription
```

```
        Vmatch = T.MatchesAttributes(Combo1.text)
```

```
        List1.AddItem " " & Vname & " " & Vmatch
```

```
    Next
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Combo1.AddItem "vendor=microsoft"
```

```
    Combo1.AddItem "gender=female"
```

```
    Combo1.AddItem "Gender = Male"
```

```
    Combo1.AddItem "gender!=male"
```

```
    Combo1.AddItem "name=Microsoft sam"
```

```
Combo1.AddItem "Name != microsoft Sam"
Combo1.AddItem "name=MICROSOFT SAM,gender=female" 'no |
Combo1.AddItem "gedner = male" 'no |
Combo1.ListIndex = 0
```

```
Set C = New SpObjectTokenCategory 'create new token ca
C.SetId SpeechCategoryVoices 'init ID of voices c
Set E = C.EnumerateTokens() 'no parameters -- ge
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

Type: Hidden

Remove Method

The **Remove** method removes the token from the speech configuration database.

```
SpObjectToken.Remove(  
    ObjectStorageCLSID As String  
)
```

Parameters

ObjectStorageCLSID

Specifies the CLSID associated with the object token to remove. If *ObjectStorageCLSID* is an empty string ("") or vbNullString, the entire token is removed; otherwise, only the specified section is removed.

Return Value

None.

Example

Use of the RemoveStorageFileName method is demonstrated in the code example for the [GetStorageFileName](#) method.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

Type: Hidden

RemoveStorageFileName Method

The **RemoveStorageFileName** method removes a storage file associated with the object token.

```
SpObjectToken.RemoveStorageFileName(  
    ObjectStorageCLSID As String,  
    KeyName As String,  
    DeleteFile As Boolean  
)
```

Parameters

ObjectStorageCLSID

The globally unique identifier (GUID) of the calling object.

KeyName

The KeyName.

DeleteFile

If True, the storage file will be deleted after removal.

Return Value

None.

Example

Use of the RemoveStorageFileName method is demonstrated in the code example for the [GetStorageFileName](#) method.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectToken](#)

Type: Hidden

SetId Method

The **SetId** method associates a new object token with a resource by setting its ID property.

The ID of an SpObjectToken object is the path to its folder within the Speech configuration database.

The Id property of an SpObjectToken can be set once. Subsequent SetId calls will return the error message, SPERR_ALREADY_INITIALIZED. Attempting to use an SpObjectToken object before setting its Id property will return SPERR_UNINITIALIZED.

```
SpObjectToken.SetId(  
    Id As String,  
    [CategoryID As String],  
    [CreateIfNotExist As Boolean = False]  
)
```

Parameters

Id

The ID of the token.

CategoryID

[Optional] The category ID of the token. By default the value is the empty string value of "".

CreateIfNotExist

[Optional] Specifies creating the token. If True, the folder is created if one does not already exist. By default the value is False, and no folder is created.

Return Value

None.

Example

Use of the RemoveStorageFileName method is demonstrated in the code example for the [GetStorageFileName](#) method.

Microsoft Speech SDK

Speech Automation 5.1



SpObjectToken Example

The following code example demonstrates the use of the Count property and the Item method. The sample displays all available recognizers on the computer. The available audio input devices (sound cards) or profiles may also be displayed by remarking out the other two lines and removing the remarks from the desired line.

The example works for all objects returned as ISpeechObjectTokens. ISpeechObjectTokens is actually a collection of objects. The Count property returns the number of items in the collection. The Item method returns an individual member at the given index.

The type of the members in the collection will be different based on the creating call. In this example code, the collection is a list of recognizers.

To run this code, create a form with the following control:

- A label called Label1

Paste this code into the Declarations section of the form.

```
Private Sub Form_Load()  
    Dim SharedRecognizer As SpSharedRecognizer  
    Set SharedRecognizer = New SpSharedRecognizer  
  
    Dim theResources As ISpeechObjectTokens  
    Set theResources = SharedRecognizer.GetRecognizers  
    'Set theResources = SharedRecognizer.GetAudioInputs  
    'Set theResources = SharedRecognizer.GetProfiles  
  
    Dim i As Long  
    Dim recoObject As SpObjectToken  
  
    Label1.Caption = ""  
    For i = 0 To theResources.Count - 1  
        Set recoObject = theResources.Item(i)  
        Label1.Caption = Label1.Caption & recoObject.GetDesc
```

```
Next i
End Sub
```

Count and Item may also be used for other object collections. The following code snippet demonstrates the use of the Count property and the Item method for ISpeechLexiconWords and ISpeechLexiconWord. The collection itself is of type ISpeechLexiconWords and the individual members of are type ISpeechLexiconWord. Other collections may use *Count* and *Item* in the same manner. For instance ISpeechGrammarRules and ISpeechGrammarRule may be substituted respectively for the collection and member type.

For the sake of brevity, CreateCollection is assumed to be a function that creates the collection, again for this example of ISpeechLexiconWords. The Count property returns the count of member items and the Item method returns a specific member of the collection.

```
Dim C As ISpeechLexiconWords           'The collection
Dim M As ISpeechLexiconWord           'An item in the collecti

Set C = CreateCollection                'Create the collection

'Get last member of the collection in object "M"

lngCount = C.Count                      'How many items
If lngCount Then
    Set M = C.Item(lngCount - 1)       'Get the last one
End If
```

Microsoft Speech SDK

Speech Automation 5.1



SpObjectTokenCategory

The **SpObjectTokenCategory** automation object represents a class of object tokens. Object tokens are associated with specific folders in the Speech configuration database, and these folders are organized into categories, such as Recognizers, AudioInputs and Voices. An SpObjectTokenCategory object represents a single category of object tokens, and provides access to all the tokens within that category.

Applications can derive the category of an initialized SpObjectToken object from its [Category](#) property, or they can create a new SpObjectTokenCategory object and use the [SetId](#) method to associate it with a particular category.

Automation Interface Elements

The SpObjectTokenCategory automation object has the following elements:

Properties	Description
Default Property	Gets and sets the ID of the default token in the category.
Id Property	Returns the name of the object token category.

Methods	Description
EnumerateTokens Method	Returns a selection of SpObjectToken objects.
GetDataKey Method	Returns the data key of the category in the speech configuration database.
SetId Method	Sets the ID of the category.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectTokenCategory](#)

Default Property

The **Default** property gets and sets the ID of the default token in the category.

The ID of the object is the path to its folder within the Speech configuration database.

Each category of object tokens has a default token, which represents the default resource of that category. The Default property of the voice category, for example, is the ID of the default system voice.

Syntax

```
Set: SpObjectTokenCategory.Default = String
```

```
Get: String = SpObjectTokenCategory.Default
```

Parts

SpObjectTokenCategory

The owning object.

const

Set: A String variable that sets the property.

Get: A String variable that gets the property.

Example

Please see the code example for the [SetId](#) method. This code creates a new *SpObjectTokenCategory* object and associates it with the category of voices. The Default property then returns the ID of the default system voice.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectTokenCategory](#)

EnumerateTokens Method

The **EnumerateTokens** method returns a selection of SpObjectToken objects.

Selection criteria may be applied optionally.

```
SpObjectTokenCategory.EnumerateTokens(  
    [RequiredAttributes As String = ""],  
    [OptionalAttributes As String = ""]  
) As ISpeechObjectTokens
```

Parameters

RequiredAttributes

[Optional] Specifies the RequiredAttributes. To be returned by EnumerateTokens, the searched tokens must contain all of the specific required attributes. If no profiles match the selection, the selection returned will not contain any elements. By default no attributes are required and so returns all the tokens discovered.

OptionalAttributes

[Optional] Specifies the OptionalAttributes. Returned tokens containing the RequiredAttributes are sorted by OptionalAttributes. If OptionalAttributes is specified, the tokens are listed with the OptionalAttributes first. By default no attribute is specified so the list returned is in the order discovered from the speech configuration database.

Return Value

The EnumerateTokens method returns an ISpeechObjectTokens

variable.

Example

The following Visual Basic form code demonstrates a simple use of the EnumerateTokens method. To run this code, create a form with the following controls:

- A list box called List1
- Two command buttons called Command1 and Command2

Paste this code into the Declarations section of the form.

The Command1 procedure creates an SpVoice object, and uses the voice's GetVoices method to get an ISpeechObjectTokens collection containing an object token for each voice on the computer. A "For Each" loop lists each voice token's Description property in the list box.

The Command2 procedure creates a new SpObjectTokenCategory object, uses the [SetId](#) method to associate it with the category of voices, and the EnumerateTokens method to get an ISpeechObjectTokens collection containing an object token for each voice on the computer. A "For Each" loop lists each voice token's Description property in the list box.

The list of voices displayed by the two command buttons will be identical.

Option Explicit

```
Dim V As SpeechLib.SpVoice           'voice object
Dim T As SpeechLib.SpObjectToken     'object token
Dim E As SpeechLib.ISpeechObjectTokens 'an enumeration
Dim C As SpeechLib.SpObjectTokenCategory 'a category of ol
```

```
Private Sub Command1_Click()
```

```

List1.Clear
List1.AddItem "Enumerate voices with SpVoice.GetVoices"
List1.AddItem ""

Set V = New SpVoice           'create new voice
Set E = V.GetVoices()         'no parameters -- ge

For Each T In E
    List1.AddItem "    " & T.GetDescription
Next

End Sub

Private Sub Command2_Click()

    List1.Clear
    List1.AddItem "Enumerate voices with SpObjectToken.EnumerateTokens"
    List1.AddItem ""

    Set C = New SpObjectTokenCategory 'create new token category
    C.SetId SpeechCategoryVoices      'init ID of voices category
    Set E = C.EnumerateTokens()       'no parameters -- ge

    For Each T In E
        List1.AddItem "    " & T.GetDescription
    Next

End Sub

```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectTokenCategory](#)

Type: Hidden

GetDataKey Method

The **GetDataKey** method returns the data key of the SpObjectTokenCategory object.

A data key object provides read and write access to the contents of a particular folder in the Speech configuration database. The data key of an SpObjectTokenCategory object accesses the folder referenced by its [Id](#) property.

```
SpObjectTokenCategory.GetDataKey(  
    [Location As SpeechDataKeyLocation = SDKLDefaultLocation  
) As ISpeechDataKey
```

Parameters

Location

[Optional] Specifies the location in the speech configuration database. Default value is SDKLDefaultLocation.

Return Value

An ISpeechDataKey object.

Example

The following Visual Basic form code demonstrates the use of the GetDataKey method to enumerate the list of voices in the voices category. To run this code, create a form with the following controls:

- A list box called List1

- A command button called Command1

Paste this code into the Declarations section of the form.

The Command1 procedure creates a new SpObjectTokenCategory object, and uses the SetId method to associate the object with the category of voices. It retrieves the data key of the category, and then the data key of the category's Tokens subfolder. It then uses the data key object's EnumKeys method to enumerate the tokens in the Tokens subfolder. The result displayed will be identical to the lists of voices displayed by the code sample in the [EnumerateTokens](#) method.

Option Explicit

```
Dim C As SpeechLib.SpObjectTokenCategory      'a category of ol
Dim K As SpeechLib.ISpeechDataKey            'data key object
Dim E As String                              'gets names of s
Dim ii As Integer

Private Sub Command1_Click()

    List1.Clear
    List1.AddItem "Enumerate voice tokens with SpObjectToken
    List1.AddItem ""

    Set C = New SpObjectTokenCategory        'create new token ca
    C.SetId SpeechCategoryVoices            'init with ID of voi

    Set K = C.GetDataKey                    'set to key of voice
    Set K = K.OpenKey("Tokens")            'reset to key of its

    On Error Resume Next
    For ii = 0 To 9999                      'enumerate subkeys w.
        E = K.EnumKeys(ii)                 'next subkey
        If Err.Number Then Exit For        'this will be used!
        List1.AddItem "    " & E
    Next
    Err.Clear
```

End Sub

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectTokenCategory](#)

Id Property

The **Id** property returns the ID of the object token category.

The ID of the object is the path to its folder within the Speech configuration database.

Syntax

Set: (This property is read-only)

Get: *String* = *SpObjectTokenCategory*.**Id**

Parts

SpObjectTokenCategory

The owning object.

String

Set: (This property is read-only)

Get: A String variable that gets the property.

Example

Please see the code example for the [SetId](#) method. This code creates a new *SpObjectTokenCategory* object and associates it with the category of voices. The *Id* property then returns the ID of the voices category.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpObjectTokenCategory](#)

SetId Method

The **SetId** method sets the ID of the SpObjectTokenCategory object.

The ID of the object is the path to its folder within the Speech configuration database.

An SpObjectTokenCategory object is created with its Id property in an uninitialized state; attempting to reference the Id of a new SpObjectTokenCategory object will result in an SPERR_UNINITIALIZED error. An SpObjectTokenCategory object in this state is not associated with any of the seven SAPI object token categories.

The SpeechStringConstants module contains constants that specify the paths of SAPI's object token categories within the configuration database. The SetId method typically sets one of those constants as the category object's Id property. This has the effect of setting the object to one of the SAPI object token categories.

This table shows the SpeechStringConstants constant for each of the seven object token categories:

Object Token Category	SpeechStringConstants member
Voices	SpeechCategoryVoices
Recognizers	SpeechCategoryRecognizers
AppLexicons	SpeechCategoryAppLexicons
AudioInput	SpeechCategoryAudioIn
AudioOutput	SpeechCategoryAudioOut
PhoneConverters	SpeechCategoryPhoneConverters
Recoprofiles	SpeechCategoryRecoProfiles

```
SpObjectTokenCategory.SetId(  
    Id As String,
```

```
    [CreateIfNotExist As Boolean = False]  
)
```

Parameters

Id

Specifies the Id, usually with a member of `SpeechStringConstants`.

CreateIfNotExist

[Optional] Specifies creating the token. If True, an entry is created if one does not already exist. By default the value is False and no file is created.

Return Value

None.

Example

The following Visual Basic form code demonstrates the use of the `SetId` method. To run this code, create a form with the following controls:

- A list box called `List1`
- A command button called `Command1`

Paste this code into the Declarations section of the form.

The `Command1` procedure creates a new `SpObjectTokenCategory` object, generates an error by attempting to reference the `Id` property of the uninitialized object, and then performs a `SetId` call which assigns the object

to the category of TTS voices. The code then displays the token category's default token, which is the default TTS voice, and finally demonstrates the error which occurs from attempting to perform a second SetId on the SpObjectTokenCategory object.

Option Explicit

```
Dim C As SpeechLib.SpObjectTokenCategory
```

```
Const SPERR_UNINITIALIZED = &H80045001;
```

```
Const SPERR_ALREADY_INITIALIZED = &H80045002;
```

```
Private Sub Command1_Click()
```

```
    On Error Resume Next
```

```
    'Uninitialized object
```

```
    List1.AddItem "New SpObjectTokenCategory object:"
```

```
    Set C = New SpObjectTokenCategory
```

```
    If Err = 0 Then
```

```
        List1.AddItem "    SUCCEEDED"
```

```
    End If
```

```
    'Show the Id of a new object
```

```
    List1.AddItem "Id of New SpObjectTokenCategory object:"
```

```
    List1.AddItem C.Id
```

```
    If Err.Number = SPERR_UNINITIALIZED Then
```

```
        List1.AddItem "    SPERR_UNINITIALIZED"
```

```
        Err.Clear
```

```
    End If
```

```
    'Show the SpeechCategoryVoices constant
```

```
    List1.AddItem "SpeechCategoryVoices constant:"
```

```
    List1.AddItem "    " & SpeechCategoryVoices
```

```
    'Perform the SetId, which assigns it to the Voices category
```

```
    List1.AddItem "Perform SetId on New SpObjectTokenCategory:"
```

```
    C.SetId SpeechCategoryVoices
```

```
    List1.AddItem "    " & C.Id
```

```
    'Show the default item of the new category
```

```
    List1.AddItem "Show SpObjectTokenCategory's default item"
```

```
List1.AddItem " " & C.Default

'Show error on second SetId
List1.AddItem "Try a second SetId on SpObjectTokenCatego
C.SetId SpeechCategoryRecognizers
If Err.Number = SPERR_ALREADY_INITIALIZED Then
    List1.AddItem " SPERR_ALREADY_INITIALIZED"
    Err.Clear
End If

End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



SpPhoneConverter

The **SpPhoneConverter** automation object supports conversion between phoneme symbols and phoneme IDs.

Each language supported by SAPI uses a set of phonemes which represent all the meaningful sounds in that language. Each phoneme has a symbolic representation of its sound, and a numeric ID; in most languages, phoneme IDs are assigned sequentially. Please see the [American English Phoneme Representation](#) page for further details.

The following table shows a few English words transcribed into phoneme symbols and phoneme ID's.

Words	Phoneme Symbols	Phoneme IDs
one	w, ah, n	46, 12, 33
two	t, uw	41, 44
three	th, r, iy	42, 38, 28
four	f, ao, r	24, 13, 38
five	f, ay, v	24, 16, 45
six	s, ih, k, s	39, 27, 30, 39

Automation Interface Elements

The SpPhoneConverter automation object has the following elements:

Properties	Description
LanguageId Property	Gets and sets the language id of the converter.

Methods	Description
---------	-------------

<u>IdToPhone</u> Method	Converts an array of phoneme IDs to a string of phoneme symbols.
<u>PhoneToId</u> Method	Converts a string of phoneme symbols to an array of phoneme IDs.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpPhoneConverter](#)

IdToPhone Method

The **IdToPhone** method converts an array of phoneme IDs to a string of phoneme symbols.

If the IdToPhone method is called before setting the SpPhoneConverter's [LanguageId](#) property, an SPERR_UNINITIALIZED error will occur.

```
SpPhoneConverter.IdToPhone(  
    IdArray As Variant  
) As String
```

Parameters

IdArray

An array of phoneme IDs or a single phoneme ID.

Return Value

A String variable.

Example

The following Visual Basic form code demonstrates the use of the IdToPhone. The application assigns values to an array and then displays the resulting phoneme string.

To run this code, create a form with the following control:

- A label called Label1

Paste this code into the Declarations section of the form.

```
Dim objPhoneConverter As New SpPhoneConverter

Private Sub Form_Load()
' US English
objPhoneConverter.LanguageId = 1033

' Get the phoneme symbols of the phoneme id 1
objPhoneConverter.IdToPhone (1)

Dim ids(2) As Integer
ids(0) = 1
ids(1) = 2
ids(2) = 3

' Get a string of phoneme symbols for the specified phoneme :
Label1.Caption = "" & objPhoneConverter.IdToPhone(ids) & "
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpPhoneConverter](#)

LanguageId Property

The **LanguageId** property gets and sets the language ID of the converter.

Syntax

```
Set: SpPhoneConverter.LanguageId = Long
```

```
Get: Long = SpPhoneConverter.LanguageId
```

Parts

SpPhoneConverter

The owning object.

SpeechLanguageId

Set: A Long variable that sets the language ID.

Get: A Long variable that gets the language ID.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpPhoneConverter](#)

PhoneTold Method

The **PhoneTold** method converts a string of phoneme symbols to an array of phoneme IDs.

If the PhoneTold method is called before setting the SpPhoneConverter's [LanguageId](#) property, an SPERR_UNINITIALIZED error will occur.

```
SpPhoneConverter.PhoneToId(  
    Phonemes As String  
) As Variant
```

Parameters

Phonemes
A string of phoneme symbols.

Return Value

A Variant array.

Microsoft Speech SDK

Speech Automation 5.1



SpPhraseInfoBuilder

The **SpPhraseInfoBuilder** automation object provides the ability to rebuild phrase information from audio data saved to memory.

Automation Interface Elements

The SpPhraseInfoBuilder automation object has the following element:

Methods	Description
<u>RestorePhraseFromMemory</u> Method	Recreates phrase information from a phrase that has been saved to memory.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpPhraseInfoBuilder](#)

RestorePhraseFromMemory Method

The **RestorePhraseFromMemory** method recreates phrase information from a phrase that has been saved to memory.

The [ISpeechPhraseInfo.SaveToMemory](#) method saves phrase information as a Variant variable. RestorePhraseFromMemory method uses this variable to recreate an object based on ISpeechPhraseInfo.

```
SpPhraseInfoBuilder.RestorePhraseFromMemory(  
    PhraseInMemory As Variant  
) As ISpeechPhraseInfo
```

Parameters

PhraseInMemory

A Variant variable containing a phrase saved to memory.

Return Value

A ISpeechPhraseInfo object returning the phrase information.

Example

The following example demonstrates storing and retrieving the phrase portion of a recognition result. An example of late binding for creating the PhraseBuilder object is also demonstrated.

The sample assumes a valid *RecoResult*.

```
'Save the phrase first
Dim thePhrase As Variant
thePhrase = RecoResult.PhraseInfo.SaveToMemory

'Retrieve the phrase
Dim PhraseBuilder As Object
Set PhraseBuilder = CreateObject("SAPI.SpPhraseInfoBuilder")

Dim PhraseInfo As ISpeechPhraseInfo
Set PhraseInfo = PhraseBuilder.RestorePhraseFromMemory(thePh
```

Microsoft Speech SDK

Speech Automation 5.1



SpTextSelectionInformation

The **SpTextSelectionInformation** automation object provides access to the text selection information pertaining to a word sequence buffer.

For an example of the use of the SpTextSelectionInformation object, see the [SetWordSequenceData](#) section of the ISpeechRecoGrammar interface.

Automation Interface Elements

The SpTextSelectionInformation automation object has the following elements:

Properties	Description
ActiveLength Property	Gets and sets the count of characters for the active range of the text selection buffer.
ActiveOffset Property	Gets and sets the offset of the active text selection buffer from the beginning of the word sequence data buffer.
SelectionLength Property	Gets and sets the count of characters in the selected text within the word sequence data buffer.
SelectionOffset Property	Gets and sets the offset of the selected text within the word sequence buffer.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpTextSelectionInformation](#)

ActiveLength Property

The **ActiveLength** property gets and sets the count of characters for the active range of the text selection buffer.

Syntax

Set: *SpTextSelectionInformation*.**ActiveLength** = *Long*

Get: *Long* = *SpTextSelectionInformation*.**ActiveLength**

Parts

SpTextSelectionInformation

The owning object.

Long

Set: A Long variable that sets the property.

Get: A Long variable that gets the property.

Example

For an example of the use of the ActiveLength property, see the [ISpeechRecoGrammar.SetWordSequenceData](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpTextSelectionInformation](#)

ActiveOffset Property

The **ActiveOffset** property gets and sets the offset of the active text selection buffer from the beginning of the WordSequenceData buffer.

Syntax

Set: *SpTextSelectionInformation*.**ActiveOffset** = *Long*

Get: *Long* = *SpTextSelectionInformation*.**ActiveOffset**

Parts

SpTextSelectionInformation

The owning object.

Long

Set: A Long variable that sets the property.

Get: A Long variable that gets the property.

Example

For an example of the use of the ActiveOffset property, see the [ISpeechRecoGrammar.SetWordSequenceData](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpTextSelectionInformation](#)

SelectionLength Property

The **SelectionLength** property gets and sets the count of characters in the selected text within the word sequence data buffer.

Syntax

Set: *SpTextSelectionInformation*.**SelectionLength** = *Long*

Get: *Long* = *SpTextSelectionInformation*.**SelectionLength**

Parts

SpTextSelectionInformation

The owning object.

Long

Set: A Long variable that sets the property.

Get: A Long variable that gets the property.

Example

For an example of the use of the SelectionLength property, see the [ISpeechRecoGrammar.SetWordSequenceData](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpTextSelectionInformation](#)

SelectionOffset Property

The **SelectionOffset** property gets and sets the offset of the selected text within the word sequence buffer.

Syntax

Set: *SpTextSelectionInformation*.**SelectionOffset** = *Long*

Get: *Long* = *SpTextSelectionInformation*.**SelectionOffset**

Parts

SpTextSelectionInformation

The owning object.

Long

Set: A Long variable that sets the property.

Get: A Long variable that gets the property.

Example

For an example of the use of the SelectionOffset property, see the [ISpeechRecoGrammar.SetWordSequenceData](#) section.

Microsoft Speech SDK

Speech Automation 5.1



SpUnCompressedLexicon

The **SpUnCompressedLexicon** automation object provides access to lexicons, which contain information about words that can be recognized or spoken.

The SpUnCompressedLexicon object represents a single application lexicon.

Automation Interfaces

The SpUnCompressedLexicon automation object contains the following elements:

Properties	Description
GenerationId Property	Gets the generation ID of the current application lexicon.

Methods	Description
AddPronunciation Method	Adds a pronunciation, specified in phone symbols, to the current application lexicon.
AddPronunciationByPhoneIds Method	Adds a pronunciation, specified in phone IDs, to the current application lexicon.
GetGenerationChange Method	Gets a list of words in the current application lexicon that have changed since the specified generation.
GetPronunciations Method	Gets the pronunciations and parts of speech for a word from the user or

	application lexicons.
<u>GetWords</u> Method	Gets a list of all words in the user or application lexicons.
<u>RemovePronunciation</u> Method	Removes a word and/or its pronunciations, specified in phone symbols, from the current application lexicon.
<u>RemovePronunciationByPhoneIds</u> Method	Removes a word and/or its pronunciations, specified in phone IDs, from the current application lexicon.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpUnCompressedLexicon](#)

AddPronunciation Method

The **AddPronunciation** method adds a pronunciation, specified in phonemes, to the application lexicon.

```
SpUnCompressedLexicon.AddPronunciation(  
    bstrWord As String,  
    LangId As Long,  
    [PartOfSpeech As SpeechPartOfSpeech = SPSUnknown],  
    [bstrPronunciation As String]  
)
```

Parameters

bstrWord

The word to add.

LangId

The language ID of the lexicon word.

PartOfSpeech

[Optional] Specifies the PartOfSpeech. Default value is SPSUnknown.

bstrPronunciation

[Optional] The pronunciation, in phonemes.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpUnCompressedLexicon](#)

Type: Hidden

AddPronunciationByPhoneIds Method

The **AddPronunciationByPhoneIds** method adds a pronunciation, specified in phone ids, to the application lexicon.

```
SpUnCompressedLexicon.AddPronunciationByPhoneIds(  
    bstrWord As String,  
    LangId As Long,  
    [PartOfSpeech As SpeechPartOfSpeech = SPSUnknown],  
    [PhoneIds As Variant]  
)
```

Parameters

bstrWord

The word to add.

LangId

The language Id of the word.

PartOfSpeech

[Optional] The PartOfSpeech. Default value is SPSUnknown.

PhoneIds

[Optional] The pronunciation, in phone ids.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpUnCompressedLexicon](#)

Type: Hidden

GenerationId Property

The **GenerationId** property gets the generation ID of the current application lexicon.

The GenerationId function acts as a version number, making it possible to roll back, cancel or undo additions to the lexicon.

Syntax

Set: Not available.

Get: *Long* = *SpUnCompressedLexicon*.**GenerationId**

Parts

SpUnCompressedLexicon

The owning object.

Long

Set: (This property is read-only).

Get: A Long variable returning the generation ID.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpUnCompressedLexicon](#)

Type: Hidden

GetGenerationChange Method

The **GetGenerationChange** method gets a list of words in the current application lexicon that have changed since the specified generation.

```
SpUnCompressedLexicon.GetGenerationChange(  
    GenerationID As Long  
) As ISpeechLexiconWords
```

Parameters

GenerationId
Specifies the GenerationId.

Return Value

The GetGenerationChange method returns an ISpeechLexiconWords object.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpUnCompressedLexicon](#)

GetPronunciations Method

The **GetPronunciations** method gets the pronunciations and parts of speech for a word in the user or application lexicons.

An [ISpeechLexiconPronunciation](#) object contains a word's pronunciations, part of speech and phone IDs. Because a word may have more than one pronunciation and more than one part of speech, the GetPronunciations method returns a collection of these objects.

```
SpUnCompressedLexicon.GetPronunciations(  
    bstrWord As String,  
    [LangId As Long],  
    [TypeFlags As SpeechLexiconType]  
) As ISpeechLexiconPronunciations
```

Parameters

bstrWord

The target lexicon word.

LangId

[Optional] The language ID.

TypeFlags

[Optional] A [SpeechLexiconType](#) constant. If Flags is SLTUser, only user lexicon words are returned; if Flags is SLTApp, only application lexicon words are returned.

Return Value

An ISpeechLexiconPronunciations object, which is a collection of one or more ISpeechLexiconPronunciation objects.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpUnCompressedLexicon](#)

GetWords Method

The **GetWords** method gets a list of all words in the current the user or application lexicon.

An [ISpeechLexiconWord](#) object contains a word's pronunciation and type. The GetWords method returns a collection of these objects.

```
SpUnCompressedLexicon.GetWords(  
    [Flags As SpeechLexiconType],  
    [GenerationId As Long]  
) As ISpeechLexiconWords
```

Parameters

Flags

[Optional] A [SpeechLexiconType](#) constant. If Flags is SLTUser, only user lexicon words are returned; if Flags is SLTApp, only application lexicon words are returned.

GenerationID

[Optional] The GenerationId.

Return Value

An [ISpeechLexiconWords](#) object, which is a collection of one or more [ISpeechLexiconWord](#) objects.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpUnCompressedLexicon](#)

RemovePronunciation Method

The **RemovePronunciation** method removes a word and/or its pronunciations, specified in phone symbols, from the current application lexicon.

```
SpUnCompressedLexicon.RemovePronunciation(  
    bstrWord As String,  
    LangId As Long,  
    [PartOfSpeech As SpeechPartOfSpeech = SPSUnknown],  
    [bstrPronunciation As String]  
)
```

Parameters

bstrWord

The lexicon word to be removed.

LangId

The language ID.

PartOfSpeech

[Optional] The PartOfSpeech. Default value is SPSUnknown.

bstrPronunciation

[Optional] The pronunciation, in phones, to be removed. If this parameter is not specified, all pronunciations of the word will be removed.

Return Value

None.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpUnCompressedLexicon](#)

Type: Hidden

RemovePronunciationByPhoneIds Method

The **RemovePronunciationByPhoneIds** method removes a word and/or its pronunciations, specified in phone IDs, from the current application lexicon.

```
SpUnCompressedLexicon.RemovePronunciationByPhoneIds(  
    bstrWord As String,  
    LangId As Long,  
    [PartOfSpeech As SpeechPartOfSpeech = SPSUnknown],  
    [PhoneIds As Variant]  
)
```

Parameters

bstrWord

The lexicon word to be removed.

LangId

The language ID.

PartOfSpeech

[Optional] The PartOfSpeech. Default value is SPSUnknown.

PhoneIds

[Optional][Optional] The pronunciation, in phone IDs, to be removed. If this parameter is not specified, all pronunciations of a lexicon word will be removed.

Return Value

None.

Microsoft Speech SDK

Speech Automation 5.1



SpVoice

The SpVoice object brings the text-to-speech (TTS) engine capabilities to applications using SAPI automation. An application can create numerous SpVoice objects, each independent of and capable of interacting with the others. An SpVoice object, usually referred to simply as a voice, is created with default property settings so that it is ready to speak immediately.

Voice Characteristics and UI Support

The fundamental characteristics of the voice are the [Voice](#) property, which can be thought of as the person of the voice, the [Rate](#) property, and the [Volume](#) property. "Microsoft Mary" and "Microsoft Mike" are examples of Voices. Use the [GetVoices](#) method to determine what other voices are available to the voice object.

These properties can be modified with a User Interface (UI). The [IsUISupported](#) method determines if a specific UI is supported. Use the [DisplayUI](#) method to display a supported UI. The TTS tab of Speech properties in Control Panel, which enables users to modify the characteristics of the default system voice, is an example of a voice UI.

Speaking and Queueing

The [Speak](#) method places a text stream in the TTS engine's input queue and returns a stream number. It can be called synchronously or asynchronously. When called synchronously, the Speak method does not return until the text has been spoken; when called asynchronously, it returns immediately, and the voice speaks as a background process.

When synchronous speech is used in an application, the application's execution is blocked while the voice speaks, and the user is effectively locked out. This may be acceptable for

simple applications, or those with no graphical user interface (GUI), but when sophisticated user interaction is intended, asynchronous speaking will generally be more appropriate.

Asynchronous speaking can place numerous text streams into the input queue. These streams are also referred to as speech requests. The stream number returned by an asynchronous `Speak` call is the stream's index in the voice queue. The [WaitUntilDone](#) method blocks execution until the voice finishes speaking, enabling an application to speak a text stream asynchronously and determine when it finishes. The hidden [SpeakCompleteEvent](#) method is similar to `WaitUntilDone`, except that it returns an event handle for the background speaking process, and does not block application execution.

The [SpeakStream](#) method operates like the `Speak` method, except that it speaks sound files instead of text.

Voice Output

An `SpVoice` object is created with its audio output set to the system default audio output. Use the [GetAudioOutputs](#) method to determine what other outputs are available to the voice, and use the [AudioOutput](#) property to set its audio output to one of them.

Use the [AudioOutputStream](#) property with other Speech automation objects to store audio output in memory (see [SpMemoryStream](#)) or in files (see [SpFileStream](#)).

Voice Events

As a voice speaks text, it can generate events when it detects certain conditions in the input stream. These events are contained in the [SpeechVoiceEvents](#) enumeration. Examples of these events are completion of phonemes, words, or sentences, as well as changes of voice or the presence of bookmarks. The range of conditions which can be reported by `SpeechVoiceEvents` is wide enough that most applications will

use only a few of them. To prevent the TTS engine from generating events that will be ignored by the application, use the [EventInterests](#) property to specify the events of interest. Only these events will be raised.

The point in the input text stream at which a potential event has been completed is referred to as an event boundary. At each event boundary, the event type is compared with the current [EventInterests](#). If the event type is of interest, an event of that type is raised. Voice events return the input stream number in order to associate them with the appropriate stream.

Voice Priorities and Alerts

Application error handling has traditionally interrupted a UI with message boxes or alert boxes describing error states. Because a TTS application might operate with no graphical UI at all, it is able to implement error handling with a TTS voice. This voice is referred to as an alert, because its purpose is identical to that of an alert box or message box. To create an alert voice, create a new `SpVoice` object and set its [Priority](#) property appropriately. The alert voice should also use a different `Voice` property from the normal voice, so that users can easily distinguish the two.

When a speaking voice detects a pending alert, it continues speaking until it arrives at a specific application-defined stopping point, such as a sentence or a word. This stopping point is called the alert boundary because it is an event boundary at which alerts can be processed. When the alert has finished speaking, the interrupted voice resumes. Get and set the alert boundary with the [AlertBoundary](#) property.

Status and Control

The [Status](#) method may return an [ISpeechVoiceStatus](#) object, which contains several types of information about the state of the voice. Some `ISpeechVoiceStatus` properties are equivalent to parameters returned by voice events; it may be advantageous for some applications to get these elements by calling `Status`

occasionally, rather than by receiving events constantly.

Voice status and voice events are closely associated with the status of the audio output device. A voice speaking to a file stream produces no audio output, generates no events, and has no audio output status. As a result, the `ISpeechVoiceStatus` data returned by that voice will always show it to be inactive.

A speaking voice can be paused at the next alert boundary with the [Pause](#) method. A paused voice can be resumed with the [Resume](#) method. The [Skip](#) method causes the voice to skip forward or backward in the input stream.

Automation Interface Elements

The `SpVoice` automation object has the following elements:

Properties	Description
AlertBoundary Property	Gets and sets alert boundary, which specifies how a speaking voice pauses for alerts.
AllowAudioOutputFormatChangesOnNextSet Property	Gets and sets flag that specifies whether the voice is allowed to change its audio output format automatically.
AudioOutput Property	Gets and sets current audio output object by the voice.
AudioOutputStream Property	Gets and sets

	current audio stream object by the voice.
<u>EventInterests</u> Property	Gets and sets types of events received by the voice.
<u>Priority</u> Property	Gets and sets priority level of voice.
<u>Rate</u> Property	Gets and sets speaking rate of the voice.
<u>Status</u> Property	Returns the current speaking and event status of voice in an ISpeechVoice object.
<u>SynchronousSpeakTimeout</u> Property	Gets and sets interval, in milliseconds, at which the voice synchronous Speak and SpeakStream calls will time out when its output device is unavailable.
<u>Voice</u> Property	Gets and sets currently active member of the Voices collection.
<u>Volume</u> Property	Gets and sets base volume (loudness) level.

the voice.

Methods	Description
DisplayUI Method	Initiates the display of the specified UI.
GetAudioOutputs Method	Returns a selection of available audio output tokens.
GetVoices Method	Returns a selection of voices available to the voice.
IsUISupported Method	Determines if the specified UI is supported.
Pause Method	Pauses the voice at the nearest alert boundary and closes the output device, allowing it to be used by other voices.
Resume Method	Causes the voice to resume speaking when paused.
Skip Method	Causes the voice to skip forward or backward by the specified number of items within the current input text stream.
Speak Method	Initiates the speaking of a text string, text file or wave file by the voice.
SpeakCompleteEvent Method	Gets an event handle from the voice that will be signaled when the voice finishes speaking.
SpeakStream Method	Initiates the speaking of a text stream or sound file by the voice.
WaitUntilDone Method	Blocks the caller until either the voice has finished speaking or the specified time interval has elapsed.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

AlertBoundary Property

The **AlertBoundary** property gets and sets the alert boundary, which specifies how a speaking voice pauses for alerts.

Because a TTS application might operate without a graphical user interface (UI), it must be able to implement error handling with a TTS voice. The only capability needed for voice error messaging is the ability to interrupt another voice; this capability is assigned to a voice by setting the [Priority](#) property to `SVPAlert`. A voice with a priority setting of `SVPAlert` is referred to as an alert voice, or simply as an alert, because its purpose is identical to that of an alert box or message box.

When a speaking voice detects a pending alert interruption, it continues speaking until it arrives at a specific application-defined stopping point, such as a sentence or a word. This stopping point is called the alert boundary because it is the event boundary at which alerts can be processed.

The `SpeechVoiceEvents` enumeration contains the types of events that a voice object can receive. The `AlertBoundary` property consists of one of these constants. The default setting is `SVEWordBoundary`.

Syntax

```
Set: SpVoice.AlertBoundary = SpeechVoiceEvents
```

```
Get: SpeechVoiceEvents = SpVoice.AlertBoundary
```

Parts

SpVoice

The owning object.

SpeechVoiceEvents

Set: A *SpeechVoiceEvents* constant that sets the alert boundary.

Get: A *SpeechVoiceEvents* constant that gets the alert boundary.

Example

The following code demonstrates the use of the *Priority*, *EventInterests* and *AlertBoundary* properties. The code uses the *GetVoices* method to select a male voice and a female voice, and sets the *Priority* property of the female voice to *SVPAlert*, making it an alert voice. The *Priority* of the male voice remains *SVPNormal*.

The code sets the *EventInterests* of the normal voice to *SVEBookmark*, so that it can receive bookmark events, and then speak a text string containing bookmarks. The normal voice's *Bookmark* event uses the alert voice to speak the bookmark data. Because the alert voice interrupts the normal voice, the normal voice is essentially using its own events to interrupt itself.

Note that the interruption does not occur immediately; the alert voice must first enqueue its text stream, and then the normal voice must detect the pending alert and stop speaking at the next alert boundary. The normal voice might speak several words or phonemes past the bookmark before it pauses for the alert voice to speak.

In this example, the normal voice can be interrupted on a phoneme boundary, which may divide a word. Changing the *AlertBoundary* to word or sentence boundaries will noticeably change the interaction of the two voices.

Option Explicit

```
Dim WithEvents objHIM As SpeechLib.SpVoice 'Normal voice wil
Dim objHER As SpeechLib.SpVoice 'Alert voice wil
```

```
Private Sub Command1_Click()
```

```
    Dim strSpeak As String
```

```
    'Create an Alert Priority voice - female
```

```
    Set objHER = New SpVoice
```

```
    objHER.Priority = SVPAAlert
```

```
    Set objHER.Voice = objHER.GetVoices("gender=female").Item(0)
```

```
    'Create a Normal Priority voice - male
```

```
    Set objHIM = New SpVoice
```

```
    Set objHIM.Voice = objHIM.GetVoices("gender=male").Item(0)
```

```
    objHER.Speak "the priority of this voice is S V P alert"
```

```
    objHIM.Speak "the priority of this voice is S V P normal"
```

```
    objHIM.EventInterests = SVEBookmark 'Receive bookmark events
```

```
    objHIM.AlertBoundary = SVEPhoneme 'Let alert voice speak
```

```
    'Normal voice speaks text which generates events.
```

```
    strSpeak = "This is text <BOOKMARK mark='first' /> that is  
               & "for the <BOOKMARK mark='second' /> purpose"
```

```
    objHIM.Speak strSpeak, SVSFIsXML + SVSFlagsAsync
```

```
End Sub
```

```
Private Sub objHIM_Bookmark(ByVal StreamNumber As Long, ByVal  
                             ByVal Bookmark As String, ByVal l
```

```
    objHER.Speak Bookmark, SVSFlagsAsync
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

Type: Hidden

AllowAudioOutputFormatChangesOnNextProperty

The **AllowOutputFormatChangesOnNextSet** property gets and sets the flag that specifies whether SAPI will adjust the format of a voice object's new audio output device automatically.

By default, when an application sets a voice object's [AudioOutput](#) property to an audio device, SAPI will change the format of that device to match the engine's preferred format. In cases where a specific audio format is required, such as telephony applications, the **AllowOutputFormatChangesOnNextSet** property can be used to prevent this format change.

When this property is true, SAPI adjusts the format of the audio output object to the engine's preferred format. When it is false, SAPI uses the audio output object's format. If the output is set to a stream object, SAPI will convert the output to the format of the stream.

Syntax

```
Set: SpVoice.AllowAudioOutputFormatChangesOnNextSet  
    = Boolean  
Get: Boolean =  
     SpVoice.AllowAudioOutputFormatChangesOnNextSet
```

Parts

SpVoice

The owning object.

Boolean

Set: A Boolean variable that sets the property value.

Get: A Boolean variable that gets the property value.

Remarks

Using the same audio format for input and output source is useful for sound cards that do not support full-duplex audio (i.e., input format must match output format). If the input format quality is lower than the output format quality, the output format quality will be reduced to equal the input quality.

Example

The following Visual Basic form code demonstrates the use of the `AllowAudioOutputFormatChangesOnNextSet` property. To run this code, create a form with the following controls:

- Two command buttons called `Command1` and `Command2`.

Paste this code into the Declarations section of the form.

The `Form_Load` procedure creates a voice object, an audio output object, and a `SpeechAudioFormatType` variable. Both command button procedures set the format of the audio output object to `SAFT22kHz8BitMono`, then set the `AudioOutputStream` of the voice to the audio output object, and then test if the voice's audio format has been changed.

In the `Command1` procedure, the `AllowAudioOutputFormatChangesOnNextSet` is set to `True`, and the voice's format is changed. In the `Command2` procedure, this property is set to `False`, and the voice's format is not changed.

`Option Explicit`

```
Dim V As SpeechLib.SpVoice  
Dim O As SpMMAudioOut
```

```

Dim S As ISpeechBaseStream

Dim f As SpeechLib.SpeechAudioFormatType 'This is an Enum

Private Sub Command1_Click()

    V.AllowAudioOutputFormatChangesOnNextSet = True

    O.Format.Type = f 'AudioOut obj gets SAFT2:
    Set V.AudioOutputStream = O 'The "Next Set"
    V.Speak "Adjust my format" 'Speak
    Set S = V.AudioOutputStream 'Stream object gets voice

    If S.Format.Type = f Then
        MsgBox "format not adjusted"
    Else
        MsgBox "format adjusted"
    End If
End Sub

Private Sub Command2_Click()

    V.AllowAudioOutputFormatChangesOnNextSet = False

    O.Format.Type = f 'AudioOut obj gets SAFT2:
    Set V.AudioOutputStream = O 'The "Next Set"
    V.Speak "Leave my format alone" 'Speak
    Set S = V.AudioOutputStream 'Stream object gets voice

    If S.Format.Type = f Then
        MsgBox "format not adjusted"
    Else
        MsgBox "format adjusted"
    End If
End Sub

Private Sub Form_Load()

    Set V = New SpVoice
    Set O = New SpMMAudioOut
    f = SAFT22kHz8BitMono 'The test audio output format
End Sub

```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

AudioOutput Property

The **AudioOutput** property gets and sets the current audio output object used by the voice.

The AudioOutput property can be set with the object token for a standard Windows multi-media device. To use other types of devices, please see the [Speech Telephony Application Guide](#).

Syntax

```
Set: SpVoice.AudioOutput = SpObjectToken
```

```
Get: SpObjectToken = SpVoice.AudioOutput
```

Parts

SpVoice

The owning object.

SpObjectToken

Get: An SpObjectToken object that gets the current audio output.

Set: An SpObjectToken object that sets the audio output.

Remarks

When setting the value, if the object is Nothing then the default audio device will be used.

Example

The following Visual Basic form code demonstrates the use of

the GetAudioOutputs method and the AudioOutput property. To run this code, create a form with the following controls:

- A command button called Command1
- A list box called List1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates an SpVoice object and calls a subroutine called ShowAudioOutputs. This subroutine uses the GetAudioOutputs method to select all available output devices, and adds each device to the list box. The string (CURRENT) is appended to the current audio device name.

Select a device in the list box, and then click Command1. The Command1_Click procedure resets the voice's AudioOutput property to the device selected in the list box, and calls the ShowAudioOutputs subroutine, which will display the device selected as the current audio output.

Option Explicit

```
Private V As SpeechLib.SpVoice
Private T As SpeechLib.SpObjectToken
```

```
Private Sub Command1_Click()
    If List1.ListIndex > -1 Then
        Set V.AudioOutput = V.GetAudioOutputs().Item(List1.L
        Call ShowAudioOutputs
    End If
End Sub
```

```
Private Sub Form_Load()
    Set V = New SpVoice
    Call ShowAudioOutputs
End Sub
```

```
Private Sub ShowAudioOutputs()
    Dim strAudio As String
    Dim strCurrentAudio As String
```

```
List1.Clear
Set T = V.AudioOutput           'Token for current a
strCurrentAudio = T.GetDescription 'Get description from

'Show all available outputs; highlight the one in use

For Each T In V.GetAudioOutputs
    strAudio = T.GetDescription    'Get description from
    If strAudio = strCurrentAudio Then
        strAudio = strAudio & " (CURRENT)" 'Show current
    End If
    List1.AddItem strAudio         'Add description to .
Next
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

AudioOutputStream Property

The **AudioOutputStream** property gets and sets the current audio stream object used by the voice.

Setting the voice's AudioOutputStream property may cause its audio output format to be automatically changed to match the text-to-speech (TTS) engine's preferred audio output format. If the voice's [AllowAudioOutputFormatChangesOnNextSet](#) property is True, the format change takes place; if False, the format remains unchanged. In order to set the AudioOutputStream property of a voice to a specific format, its AllowOutputFormatChangesOnNextSet should be False.

Syntax

Set: *SpVoice*.**AudioOutputStream** = [*ISpeechBaseStream*](#)

Get: [*ISpeechBaseStream*](#) = *SpVoice*.**AudioOutputStream**

Parts

SpVoice

The owning object.

ISpeechBaseStream

Get: An ISpeechBaseStream object that gets the current audio output stream.

Set: An ISpeechBaseStream object that sets the audio output stream.

Remarks

Voice status and voice events are closely associated with the status of the audio output device. A voice speaking to a file stream produces no audio output, generates no events, and has no audio output status. As a result, the `ISpeechVoiceStatus` data returned by that voice will always indicate that it is inactive.

Example

The following Visual Basic form code demonstrates the use of the `AudioOutputStream` property. To run this code, create a form with the following controls:

- A textbox called `Text1`
- Two command buttons called `Command1` and `Command2`

Paste the this code into the Declarations section of the form.

The `Command1_Click` procedure sets the `AudioOutputStream` property of the voice to a file called `AudioOutputStream.wav` and speaks the contents of the text box into a wave file. It then sets the voice's `AudioOutputStream` property to `Nothing`, so that subsequent voice output will be directed to the audio system rather than to a file.

The `Command2_Click` procedure plays back the wave file that created by the `Command1_Click` procedure.

```
Option Explicit
```

```
Dim objVOICE As SpeechLib.SpVoice  
Dim objFSTRM As SpeechLib.SpFileStream
```

```
Const strFName = "C:\AudioOutputStream.wav"
```

```
Private Sub Command1_Click()
```

```
    'Build a local file path and open it as a stream  
    Call objFSTRM.Open(strFName, SSFMCreatForWrite, False)
```

```
'Set voice AudioOutputStream to the stream and speak
Set objVOICE.AudioOutputStream = objFSTRM
objVOICE.Speak Text1.Text
```

```
'Close the stream and set voice back to speaking
Call objFSTRM.Close
Set objVOICE.AudioOutputStream = Nothing
```

```
Command2.Enabled = True
```

```
End Sub
```

```
Private Sub Command2_Click()
    objVOICE.Speak Text1.Text, SVSFIsXML
End Sub
```

```
Private Sub Form_Load()
    Set objVOICE = New SpVoice
    Set objFSTRM = New SpFileStream
    Command2.Enabled = False 'Force create file before pl
    Text1.Text = "The TTS voice will speak this text into a
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

DisplayUI Method

The **DisplayUI** method initiates the display of the specified UI.

The speech recognition (SR) and text-to-speech (TTS) engines are capable of displaying and running various user interfaces (UI). These displays assist with different aspects of the speech environment such as user training, microphone wizards, adding and removing words, or setting controls for the engine. Many of these UIs are available using Speech properties in Control Panel. In addition, the engines are capable of requesting that the user run specific UIs to improve recognition. For example, the SR engine could request more user training if recognition is consistently poor.

Engines are not required to support UI and not all engines will have the same UI. Consult the manufacturer's engine documentation for specific details. An application may call `IsUISupported` before attempting to invoke a particular UI to see if the engine supports it. Invoking unsupported UIs will cause a run-time error. If the UI is available, use `DisplayUI` to invoke the display.

```
SpVoice.DisplayUI(  
    hWndParent As Long,  
    Title As String,  
    TypeOfUI As String,  
    [ExtraData As Variant = Nothing]  
)
```

Parameters

hWndParent

Specifies the window handle of the owning window.

Title

Specifies the caption used for the UI window.

TypeOfUI

A String specifying the name of the UI to display. For a list of available SAPI 5 UI, see [Engine User Interfaces](#).

ExtraData

[Optional] Specifies the ExtraData. This information is unique to the application and may be used to provide additional or more specific information to the UI. By default, the Nothing value is used and indicates that the UI does not use any additional information provided by this method.

Return Value

None.

Remarks

See [SPVoice.IsUISupported](#) for additional information.

Example

The following Visual Basic form code demonstrates the use of the DisplayUI and IsUISupported methods. The application runs the audio sound panel, the same sound panel that is available using Speech properties in Control Panel.

To run this code, create a single form without any items on it. Paste this code into the Declarations section of the form.

```
Option Explicit
Public WithEvents vox As SpeechLib.SpVoice
```

```
Private Sub Form_Load()  
    Set vox = New SpVoice  
    RunUI SpeechAudioVolume  
End Sub
```

```
Private Function RunUI(theUI As String)  
    Dim x As Boolean  
  
    If vox.IsUISupported(theUI) = True Then  
        vox.DisplayUI Form1.hwnd, "My App's Sound Levels", tl  
    Else  
        MsgBox theUI & " UI not supported"  
    End If  
End Function
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

EventInterests Property

The **EventInterests** property gets and sets the types of events received by the SpVoice object.

When a text-to-speech (TTS) engine speaks a stream of text, it is constantly detecting certain conditions in the stream, such as the completion of phonemes, words and sentences. When it detects one of these conditions, the engine is able to generate a component object model (COM) event that will be received by the voice object that enqueued the stream.

When the engine detects a potential event condition in a stream, it checks the EventInterests property of the voice that enqueued the stream. If that event type is included in the voice object's event interests, the engine will generate an event of that type.

In Visual Basic, it is necessary to use the WithEvents keyword when dimensioning an SpVoice object intended to receive events. The default setting of the EventInterests property is 33278, or 0x081FE, which represents the sum of all SpeechVoiceEvents constants except SVEAudioLevel (a change in audio level).

Syntax

Set: *SpVoice*.**EventInterests** = [**SpeechVoiceEvents**](#)

Get: [**SpeechVoiceEvents**](#) = *SpVoice*.**EventInterests**

Parts

SpVoice

The owning object.

SpeechVoiceEvents

Set: One or more SpeechVoiceEvents setting the EventInterests.

Get: A number equivalent to the SpeechVoiceEvents in the EventInterests.

Remarks

The values assigned to SpeechVoiceEvents constants are single-bit values, like 1, 2, 4, 8, 16, etc. Use a logical Or function to add them to EventInterests, and a logical XOr function to remove them. It should be noted that a logical Xor function does not zero a bit value, but toggles the value. Because of this, it is necessary to ensure that the bit value is set before attempting to zero it with an Xor.

Recognition contexts support an [EventInterests](#) property, which uses a similar syntax to specify interest in speech recognition events.

Example

The following code snippet demonstrates the syntax of the EventInterests property. Interest in individual events is set and reset using logical Or and Xor statements.

```
Option Explicit
```

```
Dim WithEvents objVoice As SpeechLib.SpVoice
```

```
Private Sub Form_Load()  
    Set objVoice = New SpVoice  
    Call EventInterests  
End Sub
```

```
Private Sub EventInterests()
```

```
    'Add the SVEPhoneme constant to Event Interests  
    'Setting bit with logical 'Or' doesn't require testing.  
    objVoice.EventInterests = objVoice.EventInterests Or SVEI
```

```
    'Remove the SVEViseme constant from Event Interests  
    'Zeroing bit with logical 'Xor' requires testing!
```

```
    If (objVoice.EventInterests And SVEViseme) = SVEViseme Then  
        objVoice.EventInterests = objVoice.EventInterests Xor SVEViseme  
    End If
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

GetAudioOutputs Method

The **GetAudioOutputs** method returns a selection of available audio output tokens.

See [Object Tokens and Registry Settings White Paper](#) for a list of SAPI 5-defined attributes.

```
SpVoice.GetAudioOutputs(  
    [RequiredAttributes As String = ""],  
    [OptionalAttributes As String = ""]  
) As ISpeechObjectTokens
```

Parameters

RequiredAttributes

[Optional] Specifies the RequiredAttributes. To be returned by GetAudioOutputs, audio output tokens must contain all of the specific required attributes. If no tokens match the selection, the selection returned will not contain any elements. By default, no attributes are required and so the method returns all the tokens discovered.

OptionalAttributes

[Optional] Specifies the OptionalAttributes. Returned tokens containing the RequiredAttributes are sorted by OptionalAttributes. If OptionalAttributes is specified, the tokens are listed with the OptionalAttributes first. By default, no attribute is specified and the list returned from the speech configuration database is in the order that attributes were discovered.

Return Value

An ISpeechObjectTokens collection containing the selected outputs.

Remarks

The format of selection criteria may either be *Value* or "*Attribute = Value*". Values may be excluded by "*Attribute != Value*".

Example

The following Visual Basic form code demonstrates the use of the GetAudioOutputs method and the AudioOutput property. To run this code, create a form with the following commands:

- A command button called Command1
- A list box called List1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates an SpVoice object and calls a subroutine called ShowAudioOutputs. This subroutine uses the GetAudioOutputs method to select all available output devices, and adds each device to the list box. The string (Current) is appended to the current audio device name.

Select a device in the list box, and then click Command1. The Command1_Click procedure resets the voice's AudioOutput property to the device selected in the list box, and calls the ShowAudioOutputs subroutine, which will display the device selected as the current audio output.

```
Option Explicit
```

```
Private V As SpeechLib.SpVoice
```

```

Private T As SpeechLib.SpObjectToken

Private Sub Command1_Click()
    If List1.ListIndex > -1 Then
        Set V.AudioOutput = V.GetAudioOutputs().Item(List1.L
        Call ShowAudioOutputs
    End If
End Sub

Private Sub Form_Load()
    Set V = New SpVoice
    Call ShowAudioOutputs
End Sub

Private Sub ShowAudioOutputs()
    Dim strAudio As String
    Dim strCurrentAudio As String

    List1.Clear
    Set T = V.AudioOutput           'Token for current a
    strCurrentAudio = T.GetDescription 'Get description fro

    For Each T In V.GetAudioOutputs
        strAudio = T.GetDescription 'Get description fro
        If strAudio = strCurrentAudio Then
            strAudio = strAudio & " (Current)" 'Show curren
        End If
        List1.AddItem strAudio           'Add description to .
    Next
End Sub

```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

GetVoices Method

The **GetVoices** method returns a selection of voices available to the voice.

Selection criteria may be applied optionally. In the absence of selection criteria, all voices are returned in the selection, ordered alphabetically by the voice name. If no voices match the criteria, GetVoices returns an empty selection, that is, an ISpeechObjectTokens collection with a Count of zero.

See [Object Tokens and Registry Settings White Paper](#) for a list of SAPI 5-defined attributes.

```
SpVoice.GetVoices(  
    [RequiredAttributes As String = ""],  
    [OptionalAttributes As String = ""]  
) As ISpeechObjectTokens
```

Parameters

RequiredAttributes

[Optional] Specifies the RequiredAttributes. All voices selected will match these specifications. If no voices match the selection, the selection returned will contain no voices. By default, no attributes are required and so the list returns all the tokens discovered.

OptionalAttributes

[Optional] Specifies the OptionalAttributes. Voices which match these specifications will be returned at the front of the selection. By default, no attribute is specified and the list returned from the speech configuration database is in the order that attributes were discovered.

Return Value

An ISpeechObjectTokens variable containing the collection of voice tokens selected.

Remarks

The format of selection criteria is "*Attribute = Value*" and "*Attribute != Value*." Voice attributes include "Gender," "Age," "Name," "Language," and "Vendor."

Example

The following Visual Basic form code demonstrates the use of the GetVoices method and the Voice property. To run this code, create a form with the following controls:

- A command button called Command1
- A list box called List1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object, and displays the names of all available voices in the list box. Select a voice name in the list box, and then click Command1. The Command1 procedure sets the voice object's Voice property to the selected name, and causes the voice to speak its new name.

```
Option Explicit
```

```
Private V As SpeechLib.SpVoice  
Private T As SpeechLib.ISpeechObjectToken
```

```
Private Sub Command1_Click()
```

```
    If List1.ListIndex > -1 Then
```

```
'Set voice object to voice name selected in list box  
'The new voice speaks its own name
```

```
Set V.Voice = V.GetVoices().Item(List1.ListIndex)  
V.Speak V.Voice.GetDescription
```

```
Else
```

```
    MsgBox "Please select a voice from the listbox"
```

```
End If
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Dim strVoice As String
```

```
    Set V = New SpVoice
```

```
    'Get each token in the collection returned by GetVoices
```

```
    For Each T In V.GetVoices
```

```
        strVoice = T.GetDescription           'The token's name
```

```
        List1.AddItem strVoice              'Add to listbox
```

```
    Next
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

IsUISupported Method

The **IsUISupported** method determines whether the specified UI is supported.

The speech recognition (SR) and text-to-speech (TTS) engines are capable of displaying and running various user interfaces (UI). These displays assist with different aspects of the speech environment such as user training, microphone wizards, adding and removing words, or setting controls for the engine. Many of these UIs are available using Speech properties in Control Panel. In addition, the engines are capable of requesting that the user run specific UIs to improve recognition. For example, the SR could request more user training if recognition is consistently poor.

Engines are not required to support UI and not all engines will have the same UI. Consult the manufacturer's engine documentation for specific details. An application may call `IsUISupported` before attempting to invoke a particular UI to see if the engine supports it. Invoking unsupported UIs will cause a run-time error. If the UI is available, use `DisplayUI` to invoke the display.

```
SpVoice.IsUISupported(  
    TypeOfUI As String,  
    [ExtraData As Variant = Nothing]  
) As Boolean
```

Parameters

TypeOfUI

A String specifying the name of the UI to display. For a list of available SAPI 5 UI, see [Engine User Interfaces](#).

ExtraData

[Optional] Specifies the ExtraData. This information is unique to the application and may be used to provide additional or more specific information to the UI. By default, the Nothing value is used and indicates the UI does not use any additional information provided by this method.

Return Value

A Boolean variable indicating whether the specified UI is supported. It returns True if supported, or False if not supported.

Remarks

See [SPVoice.DisplayUI](#) for additional information.

Example

The following Visual Basic form code demonstrates the use of the DisplayUI and IsUISupported methods. The application runs the audio sound panel, the same sound panel that is available using Speech properties in Control Panel.

To run this code, create a single form without any items on it. Paste this code into the Declarations section of the form.

```
Option Explicit
Public WithEvents vox As SpeechLib.SpVoice

Private Sub Form_Load()
    Set vox = New SpVoice
    RunUI SpeechAudioVolume
End Sub

Private Function RunUI(theUI As String)
    If vox.IsUISupported(theUI) = True Then
        vox.DisplayUI Form1.hwnd, "My App's Sound Levels", tl
```

```
Else
    MsgBox theUI & " UI not supported"
End If
End Function
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

Pause Method

The **Pause** method pauses the voice at the nearest alert boundary and closes the output device, allowing it to be used by other voices.

```
SpVoice.Pause()
```

Parameters

None.

Return Value

None.

Example

The following Visual Basic form code demonstrates the use of the Pause and Resume methods. To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object. The first call of the Command1_Click procedure causes the voice to begin speaking a text stream. Subsequent Command1 clicks alternately pause and resume the voice. When the voice has finished speaking the stream, the EndStream causes the voice to speak it again.

The voice's AlertBoundary setting of SVEPhoneme means that the Pause method can interrupt the voice within word

boundaries. The text stream spoken contains a number of long words in order to show this interruption of words more clearly.

```
Option Explicit
```

```
Private WithEvents V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Select Case Command1.Caption
```

```
    Case "Start"
```

```
        Call SpeakAgain
```

```
        Command1.Caption = "Pause"
```

```
    Case "Pause"
```

```
        V.Pause
```

```
        Command1.Caption = "Resume"
```

```
    Case "Resume"
```

```
        V.Resume
```

```
        Command1.Caption = "Pause"
```

```
    End Select
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpVoice
```

```
    V.AlertBoundary = SVEPhoneme      'Let words be interrupted
```

```
    Command1.Caption = "Start"
```

```
End Sub
```

```
Private Sub V_EndStream(ByVal StreamNumber As Long, ByVal StreamNumber As Long, ByVal StreamNumber As Long)
```

```
    Call SpeakAgain
```

```
End Sub
```

```
Private Sub SpeakAgain()
```

```
    V.Speak "this phenomenal asynchronous stream contains mu."
```

End Sub & "pronunciations and circumlocutions.", SVSFlag:

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

Priority Property

The **Priority** property gets and sets the priority level of the voice.

The priority level defines the order in which the text-to-speech (TTS) engine processes a voice object's speech requests relative to requests from other voice objects. Higher priority levels are assigned to error-handling voices and the lowest priority level is assigned to normal voices. Because of their priority level, voice requests from error-handling voices are spoken ahead of normal priority voice requests; as a result, error-handling voices can appear to interrupt normal voices.

The voice priority levels are contained in the `SpeechVoicePriority` enumeration. An `SpVoice` object is created with normal priority. To create an alert or over voice, create a voice and set its `Priority` property appropriately.

Syntax

```
Set: SpVoice.Priority = SpeechVoicePriority
```

```
Get: SpeechVoicePriority = SpVoice.Priority
```

Parts

SpVoice

The owning object.

SpeechVoicePriority

Set: A `SpeechVoicePriority` constant that sets the priority level.

Get: A `SpeechVoicePriority` constant that returns the current priority level.

Remarks

A voice with a `Priority` setting of `SVPAlert` is referred to as an alert voice. Alert voices are designed to be the primary vehicle for TTS error-handling. Other `SpVoice` elements, such as the `AlertBoundary` property, support error-handling functionality in alert voices that is not available to other voices.

Example

The following Visual Basic code demonstrates the use of the `Priority` property. To run this code, create a form with the following controls:

- Two command buttons called `Command1` and `Command2`

Paste this code into the `Declarations` section of the form.

The `Form_Load` procedure creates three voice objects, and assigns different `Voice` and `Priority` settings to each. Each voice is named after its `Priority` setting.

In the `Command1_Click` procedure, the normal voice begins speaking asynchronously. The alert voice waits a few seconds, and then begins speaking. The normal voice is interrupted by the alert voice, and resumes speaking when the alert voice has finished.

In the `Command2_Click` procedure, the normal voice and the over voice begin speaking asynchronously. The alert voice waits a few seconds, and then begins speaking. The normal voice is interrupted by the alert voice, and resumes speaking when the alert voice has finished. The over voice speaks over, or mixes with, the other two voices.

Please see the [AlertBoundary](#) property for another code

example using the Priority property.

Option Explicit

```
Dim Normal As SpeechLib.SpVoice
Dim Alert As SpeechLib.SpVoice
Dim Over As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    'Enqueue streams from normal voice
    Normal.Speak "a normal voice. a normal priority voice.",
    Normal.Speak "a normal voice. a normal priority voice.",
```

```
    Call Wait(3)      'Alert voice interrupts normal
    Alert.Speak "excuse me, alert voice!", SVSFlagsAsync
```

```
End Sub
```

```
Private Sub Command2_Click()
```

```
    'Enqueue streams from Normal voice and Over voice
    Normal.Speak "a normal voice. a normal priority voice.",
    Normal.Speak "a normal voice. a normal priority voice.",
```

```
    Over.Speak "over voice speaking over the other voices.",
    Over.Speak "over voice speaking over the other voices.",
    Over.Speak "over voice speaking over the other voices.",
```

```
    Call Wait(3)      'Alert voice interrupts normal
    Alert.Speak "excuse me, alert voice!", SVSFlagsAsync
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set Normal = New SpVoice
    Set Alert = New SpVoice
    Set Over = New SpVoice
```

```
    Alert.Priority = SVPAAlert      'From SVPNormal to SVPAAlert
    Over.Priority = SVPOver        'From SVPNormal to SVPOver
```

```
'Presumes two male voices and one female voice
Set Normal.Voice = Normal.GetVoices("gender = male").Item(0)
Set Alert.Voice = Alert.GetVoices("gender = male").Item(0)
Set Over.Voice = Normal.GetVoices("gender = female").Item(0)
```

```
End Sub
```

```
Private Sub Wait(ByVal Seconds As Integer)
```

```
    Dim sglWait As Single
```

```
    sglWait = Timer() + Seconds
```

```
    Do
```

```
        DoEvents
```

```
    Loop Until Timer > sglWait
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

Rate Property

The **Rate** property gets and sets the speaking rate of the voice.

Values for the Rate property range from -10 to 10, which represent the slowest and the fastest speaking rates, respectively.

At the beginning of each Speak or SpeakStream method, the voice sets its speaking rate according to the value of its Rate property, and speaks the entire stream at that rate. The Rate property can be changed at any time, but the actual speaking rate will not reflect the changed property value until it begins a new stream.

Syntax

```
Set: SpVoice.Rate = Long
```

```
Get: Long = SpVoice.Rate
```

Parts

SpVoice

The owning object.

Long

Set: A Long variable that sets the property value.

Get: A Long variable that gets the property value.

Example

The following Visual Basic form code demonstrates the use of

the Rate and the Volume properties. To run this code, create a form with the following controls:

- A command button called Command1
- A text box called Text1
- A VScrollbar called VScroll1
- An HScrollbar HScroll1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object and associates the VScrollbar with the voice's Volume property and the HScrollbar with the voice's Rate property. Adjusting the scroll bars changes the settings of the Volume and Rate properties. The Command1_Click procedure speaks a phrase in order to demonstrate the effects of the changes.

Option Explicit

```
Private V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    V.Speak "The quick brown fox jumped over the lazy dog.",  
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpeechLib.SpVoice
```

```
    VScroll1.Min = 0
```

```
    VScroll1.Max = 100
```

```
    VScroll1.Value = V.Volume
```

```
    HScroll1.Min = -10
```

```
    HScroll1.Max = 10
```

```
    HScroll1.Value = V.Rate
```

```
    Text1.Text = "Vol: " & VScroll1.Value & "; Rate: " & HS
```

```
End Sub
```

```
Private Sub HScroll1_Change()
```

```
    V.Rate = HScroll1.Value
```

```
    Text1.Text = "Vol: " & VScroll1.Value & "; Rate: " & HS  
End Sub
```

```
Private Sub VScroll1_Change()
```

```
    V.Volume = VScroll1.Value
```

```
    Text1.Text = "Vol: " & VScroll1.Value & "; Rate: " & HS  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

Resume Method

The **Resume** method causes the voice to resume speaking when paused.

```
SpVoice.Resume()
```

Parameters

None.

Return Value

None.

Example

The following Visual Basic form code demonstrates the use of the Pause and Resume methods. To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object. The first call of the Command1_Click procedure causes the voice to begin speaking a text stream. Subsequent Command1 clicks alternately pause and resume the voice. When the voice has finished speaking the stream, the EndStream causes the voice to speak it again.

The voice's AlertBoundary setting of SVEPhoneme means that the Pause method can interrupt the voice within word boundaries. The text stream spoken contains a number of long

words in order to show this interruption of words more clearly.

```
Option Explicit
```

```
Private WithEvents V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Select Case Command1.Caption
```

```
        Case "Start"  
            Call SpeakAgain  
            Command1.Caption = "Pause"
```

```
        Case "Pause"  
            V.Pause  
            Command1.Caption = "Resume"
```

```
        Case "Resume"  
            V.Resume  
            Command1.Caption = "Pause"
```

```
    End Select
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpVoice  
    V.AlertBoundary = SVEPhoneme      'Let words be interrupted  
    Command1.Caption = "Start"
```

```
End Sub
```

```
Private Sub V_EndStream(ByVal StreamNumber As Long, ByVal StreamNumber As Long, ByVal StreamNumber As Long)  
    Call SpeakAgain
```

```
End Sub
```

```
Private Sub SpeakAgain()
```

```
    V.Speak "this phenomenal asynchronous stream contains multiple  
            & "pronunciations and circumlocutions.", SVSFlag: SVSFlag: SVSFlag:
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

Skip Method

The **Skip** method skips the voice forward or backward by the specified number of items within the current input text stream.

```
SpVoice.Skip(  
    Type As String,  
    NumItems As Long  
) As Long
```

Parameters

Type

The type of items to be skipped. Currently, Sentence is the only type supported.

NumItems

The number of items to be skipped forward in the voice input stream. A negative value specifies skipping backward.

Return Value

A Long variable containing the number of items skipped.

Example

The following Visual Basic form code demonstrates the use of the Skip method. To run this code, create a form with the following controls:

- A text box called Text1

- Two command buttons called Command1 and Command2
- An HScrollbar control called HScroll1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice, sets up the HScrollbar with a value of -2 in a range from -5 to 5, and puts a string of numbers in the textbox. Because the numbers are followed by periods and separated by spaces, SAPI considers each number to be a sentence. The scrollbar specifies the number of sentences that the voice object will skip.

Click Command1 to start the voice speaking the sentences in the textbox. Click Command2 to skip the voice forward or backward, depending on the value of the scrollbar control. Click the left side or the right side of the scrollbar to increase or decrease the number of sentences to be skipped by the Command2 button.

Option Explicit

```
Private WithEvents V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()  
    V.Speak Text1.Text, SVSFlagsAsync  
End Sub
```

```
Private Sub Command2_Click()  
    V.Skip "Sentence", HScroll1.Value  
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpVoice
```

```
    HScroll1.Min = -5  
    HScroll1.Max = 5  
    HScroll1.Value = -2
```

```
    Text1.Text = "1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13."
```

```
End Sub
```

```
Private Sub HScroll1_Change()
```

```
    If HScroll1.Value > 0 Then
```

```
        Command2.Caption = " Skip forward " & HScroll1.Value
```

```
    Else
```

```
        Command2.Caption = " Skip backward " & Abs(HScroll1.'
```

```
    End If
```

```
End Sub
```

```
Private Sub V_Word(ByVal StreamNumber As Long, ByVal StreamP
```

```
                ByVal CharacterPosition As Long, ByVal Le
```

```
    Text1.SetFocus
```

```
    Text1.SelStart = CharacterPosition
```

```
    Text1.SelLength = Length
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

Speak Method

The **Speak** method initiates the speaking of a text string, a text file, an XML file, or a wave file by the voice.

The Speak method can be called synchronously or asynchronously. When called synchronously, the method does not return until the text has been spoken; when called asynchronously, it returns immediately, and the voice speaks as a background process.

When synchronous speech is used in an application, the application's execution is blocked while the voice speaks, and the user is effectively locked out. This may be acceptable for simple applications, or those with no graphical user interface (GUI), but when sophisticated user interaction is intended, asynchronous speaking will generally be more appropriate.

The [WaitUntilDone](#) and [SpeakCompleteEvent](#) methods can be used to block an application's forward progress while allowing user interaction with the mouse or keyboard.

```
SpVoice.Speak(  
    Text As String,  
    [Flags As SpeechVoiceSpeakFlags = SVSFDefault]  
) As Long
```

Parameters

Text

The text to be spoken, or if the SVSFIsFilename flag is included in the Flags parameter, the path of the file to be spoken.

Flags

[Optional] Flags. Default value is SVSFDefault.

Return Value

A Long variable containing the stream number. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate events with the appropriate stream.

Remarks

The Speak method inserts a stream into the text-to-speech (TTS) engine's queue, and returns a stream number, assigned by the engine. This distinguishes the stream from other streams in the queue. This number is a temporary identifier which functions like an index into the TTS queue. The first stream spoken into an empty queue will always have a stream number of 1.

A voice object can enqueue numerous streams, and each of these streams can generate events. SpVoice events always return the stream number as a parameter. If an application saves the stream numbers of the streams it enqueues, events can be associated with the proper stream.

Example

The following code snippet demonstrates the Speak method with several commonly used flag settings.

```
Const cstrTextName = "c:\Speech Voice Speak.txt"
```

```
Dim V As SpeechLib.SpVoice  
Set V = New SpVoice
```

```
'Build a simple text file for demonstration purposes  
Open cstrTextName For Output As #1
```

```
Print #1, "The name of this file is " & cstrTextName  
Close #1
```

```
'Speak literal text  
V.Speak "This is some text", SVSFDefault
```

```
'Speak the text of the test file  
V.Speak cstrTextName, SVSFIsFilename + SVSFlagsAsync
```

```
'Speak with/without punctuation  
V.Speak "one, two, three!", SVSFlagsAsync  
V.Speak "one, two, three!", SVSFNLPSpeakPunc + SVSFlagsAsync
```

```
'Speak text with/without XML tags  
V.Speak "text with XML", SVSFIsXML + SVSFlagsAsync  
V.Speak "text with XML", SVSFIsNotXML + SVSFlagsAsync
```

```
V.WaitUntilDone 10000
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

Type: Hidden

SpeakCompleteEvent Method

The **SpeakCompleteEvent** method gets an event handle from the voice that will be signaled when the voice finishes speaking.

The SpeakCompleteEvent method is similar to the [WaitUntilDone](#) method, but supports more sophisticated ways of waiting for the voice to finish speaking.

The WaitUntilDone method explicitly blocks program execution until the voice finishes. The SpeakCompleteEvent method does not block execution, but returns an event handle that can be used with API wait functions such as WaitForSingleObject.

Because these functions can wait for short periods of time, applications may be able to perform useful tasks while polling the event handle.

```
SpVoice.SpeakCompleteEvent() As Long
```

Parameters

None.

Return Value

A Long variable containing the event handle.

Example

The following code snippet demonstrates the use of the SpeakCompleteEvent method. The asynchronous Speak call returns immediately, and causes the voice to begin speaking as

a background process. The `SpeakCompleteEvent` method returns the event handle of the speaking process. This handle is passed to `WaitForSingleObject`, which waits for a completion signal from the process. When the background speaking process signals its completion, the call to `WaitForSingleObject` returns, and the program continues.

```
Dim objVoice As SpeechLib.SpVoice
Dim lngHandle As Long
Dim lngRtn As Long
Const INFINITE = -1&

Set objVoice = New SpVoice
objVoice.Speak "please wait until this text has been spoken"

lngHandle = objVoice.SpeakCompleteEvent           'Get a h
lngRtn = WaitForSingleObject(lngHandle, INFINITE) 'Wait fo
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

SpeakStream Method

The **SpeakStream** method initiates speaking of a sound file by the voice.

```
SpVoice.SpeakStream(  
    Stream As ISpeechBaseStream,  
    [Flags As SpeechVoiceSpeakFlags = SVSFDefault]  
) As Long
```

Parameters

Stream

Specifies an [ISpeechBaseStream](#) object containing the stream.

Flags

[Optional] Specifies the Flags. Default value is [SVSFDefault](#).

Return Value

A Long variable containing the stream number. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate events with the appropriate stream.

Example

The following code snippet demonstrates the use of the `SpeakStream` method. To run this code, create a form with the following control:

- A command button called Command1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a male and a female voice. The Command1_Click procedure causes the female voice to speak text into a file, and the male voice to play that file using the SpeakStream method.

Option Explicit

```
Private M As SpeechLib.SpVoice      'M is a male voice
Private F As SpeechLib.SpVoice      'F is a female voice
Private S As SpeechLib.SpFileStream
```

```
Private Sub Command1_Click()
```

```
    'Build a local file path and open it as a stream
    Set S = New SpFileStream
    Call S.Open("C:\SpeakStream.wav", SSFMCreateForWrite, Fa.
```

```
    'Female voice speaks into the file stream and creates a \
    Set F.AudioOutputStream = S
    F.Speak "cee : \ speak stream dot wave", SVSFNLPSpeakPun
    S.Close
```

```
    'Male voice speaks female voice's stream
    Call S.Open("C:\SpeakStream.wav", , False)
    M.Speak "i will now demonstrate the speak stream method.
    M.SpeakStream S
    M.Speak "that sounded like " & F.Voice.GetDescription &
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    'Create voices
    Set F = New SpVoice
    Set F.Voice = F.GetVoices("gender=female").Item(0)
    Set M = New SpVoice
    Set M.Voice = M.GetVoices("gender=male").Item(0)
```

End Sub

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

Status Property

The **Status** property returns the current speaking and event status of the voice in an ISpeechVoiceStatus object.

Syntax

Set: (This property is read-only)

Get: ***ISpeechVoiceStatus*** = *SpVoice*.**Status**

Parts

SpVoice

The owning object.

ISpeechVoiceStatus

Set: (This property is read-only)

Get: An ISpeechVoiceStatus object containing Status information.

Remarks

Properties of the ISpeechVoiceStatus object may also be accessed through an implicit status object by means of the syntax "*propertyvalue* = *SpVoice*.*Status*.*propertyname*." Please see the example below.

The Status method is designed for use with voices speaking to audio devices. Because the Status method is closely associated with audio device status, it will not return an active status for a voice speaking to an audio output stream.

Example

The following code snippet demonstrates two ways of using the Status method. The first uses an implicit status object; the second creates the status object explicitly.

Use of the voice Status method and the ISpeechVoiceStatus interface is demonstrated with more detail in the [ISpeechVoiceStatus code example](#).

```
Dim objVOICE As SpeechLib.SpVoice
Dim objSTATUS As SpeechLib.ISpeechVoiceStatus

' Assume that objVOICE has been created, and
' has spoken some text asynchronously.

' ISpeechVoiceStatus object is implicit here
'
If objVOICE.Status.CurrentStreamNumber = 2 Then
    'Do something
End If

' ISpeechVoiceStatus object is explicit here
'
Set objSTATUS = objVOICE.Status
If objSTATUS.CurrentStreamNumber = 2 Then
    'Do something
End If
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

SynchronousSpeakTimeout Property

The **SynchronousSpeakTimeout** property gets and sets the interval, in milliseconds, after which the voice's synchronous Speak and SpeakStream calls will time out when its output device is unavailable.

When a voice enqueues a text stream, the audio output device represented by its AudioOutput property may be in use. When the text stream is enqueued synchronously, the voice will wait for the amount of time specified in its SynchronousSpeakTimeout property. If the output device does not become available to the voice before the time has elapsed, the voice will time out, the synchronous speech request is cancelled, and the application receives an SPERR_DEVICE_BUSY error. This and other SAPI errors are detailed in [Error Codes](#)

There is no equivalent timeout for asynchronous speech. Because synchronous speech prevents applications from receiving events from mouse movements and keyboard input, unexpected voice streams from other applications could freeze an application attempting synchronous speech. The SynchronousSpeakTimeout is designed so that applications can recover from such situations.

Syntax

Set: *SpVoice*.**SynchronousSpeakTimeout** = *Long*

Get: *Long* = *SpVoice*.**SynchronousSpeakTimeout**

Parts

SpVoice

The owning object.

Long

Set: A Long variable that sets the property value.

Get: A Long variable that gets the property value.

Example

The following Visual Basic form code demonstrates the use of the SynchronousSpeakTimeout property. To run this code, create a form with the following controls:

- A list box control called List1
- A command button called Command1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates three voices. The Command1_Click procedure starts the first voice speaking, which makes the audio output device unavailable to the other voices. The SynchronousSpeakTimeout property of the second voice is set to one millisecond, ensuring that it will time out before the first voice finishes speaking. The third voice simply waits for the first voice to finish, and then speaks.

The voices use a subroutine to speak; this subroutine adds each speech request to the list box and tests for the error that occurs when a voice times out.

```
Option Explicit
```

```
Const SPERR_DEVICE_BUSY = &H80045006;
```

```
Dim v1 As SpeechLib.SpVoice
```

```
Dim v2 As SpeechLib.SpVoice
```

```
Dim v3 As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```

List1.Clear

'Voice 1 takes control of the audio
Call SafeSpeak(v1, "This is voice number 1", SVSFlagsAsy
Call SafeSpeak(v1, "Voice 2 and voice 3 will wait for me
v1.WaitUntilDone 100    'ensure that the voice 1 starts

'Voice 2 starts waiting until voice 1 is done,
'but its timeout is very short -- 1 millisecond.
'So it times out before voice 1 is done.
v2.SynchronousSpeakTimeout = 1
Call SafeSpeak(v2, "This is voice 2", SVSFDefault)
Call SafeSpeak(v2, "This is voice 2 again", SVSFDefault)

'Voice 3 simply waits until voice 1 is done.
Call SafeSpeak(v3, "This is voice 3 now", SVSFDefault)

End Sub

Private Sub Form_Load()
    Set v1 = New SpVoice
    Set v2 = New SpVoice
    Set v3 = New SpVoice
End Sub

Private Sub SafeSpeak(who As SpVoice, ByVal txt, ByVal flags
    On Error GoTo SafeSpeakExit
    DoEvents
    who.Speak txt, flags

SafeSpeakExit:
    Select Case Err.Number
    Case 0:                List1.AddItem "queued: " &
    Case SPERR_DEVICE_BUSY: List1.AddItem "timeout: " &
    End Select
    Err.Clear
End Sub

```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

Voice Property

The **Voice** property gets and sets the currently active member of the Voices collection.

The Voice property can be thought of as the person of a voice object; examples of Voices are "Microsoft Mary" and "Microsoft Mike." Use the [GetVoices](#) method to determine what voices are available.

Syntax

```
Set: SpVoice.Voice = SpObjectToken
```

```
Get: SpObjectToken = SpVoice.Voice
```

Parts

SpVoice

The owning object.

SpObjectToken

Set: An SpObjectToken object that sets the voice property.

Get: An SpObjectToken object that gets the current voice.

Remarks

If there is not a voice currently in use, this property will return the token for the default voice.

Example

The following Visual Basic form code demonstrates the use of

the GetVoices method and the Voice property. To run this code, create a form with the following controls:

- A command button called Command1
- A list box called List1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object, and displays the names of all available voices in the list box. Select a voice name in the list box and then click Command1; the Command1 procedure sets the voice object's Voice property to the selected name, and causes the voice to speak its new name.

Option Explicit

```
Private V As SpeechLib.SpVoice
Private T As SpeechLib.ISpeechObjectToken

Private Sub Command1_Click()

    If List1.ListIndex > -1 Then

        'Set voice object to voice name selected in list box
        'The new voice speaks its own name

        Set V.Voice = V.GetVoices().Item(List1.ListIndex)
        V.Speak V.Voice.GetDescription

    Else
        MsgBox "Please select a voice from the listbox"
    End If

End Sub

Private Sub Form_Load()
    Dim strVoice As String

    Set V = New SpVoice

    'Get each token in the collection returned by GetVoices
```

```
For Each T In V.GetVoices
    strVoice = T.GetDescription    'The token's name
    List1.AddItem strVoice        'Add to listbox
Next
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

Volume Property

The **Volume** property gets and sets the base volume (loudness) level of the voice.

Values for the Volume property range from 0 to 100, representing the minimum and maximum volume levels, respectively.

At the beginning of each Speak or SpeakStream method, the voice sets the volume level according to the value of the Volume property, and speaks the entire stream at that level. The voice's Volume property can be changed at any time, but the actual volume level will not reflect the changed property value until it begins a new stream.

Syntax

```
Set: SpVoice.Volume = Long
```

```
Get: Long = SpVoice.Volume
```

Parts

SpVoice

The owning object.

Long

Set: A Long variable that sets the property value.

Get: A Long variable that gets the property value.

Example

The following Visual Basic form code demonstrates the use of the Rate and the Volume properties. To run this code, create a

form with the following controls:

- A command button called Command1
- A text box called Text1
- A VScrollbar called VScroll1
- An HScrollbar called HScroll1

Paste this code into the Declarations section of the form.

The Form_Load procedure creates a voice object and associates the VScrollbar with the voice's Volume property and the HScrollbar with the voice's Rate property. Adjusting the scroll bars changes the settings of the Volume and Rate properties. The Command1_Click procedure speaks a phrase in order to demonstrate the effects of the changes.

```
Option Explicit
```

```
Private V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    V.Speak "The quick brown fox jumped over the lazy dog.",  
End Sub
```

```
Private Sub Form_Load()
```

```
    Set V = New SpeechLib.SpVoice
```

```
    VScroll1.Min = 0
```

```
    VScroll1.Max = 100
```

```
    VScroll1.Value = V.Volume
```

```
    HScroll1.Min = -10
```

```
    HScroll1.Max = 10
```

```
    HScroll1.Value = V.Rate
```

```
    Text1.Text = "Vol: " & VScroll1.Value & "; Rate: " & HS
```

```
End Sub
```

```
Private Sub HScroll1_Change()  
    V.Rate = HScroll1.Value  
    Text1.Text = "Vol: " & VScroll1.Value & "; Rate: " & HS  
End Sub
```

```
Private Sub VScroll1_Change()  
    V.Volume = VScroll1.Value  
    Text1.Text = "Vol: " & VScroll1.Value & "; Rate: " & HS  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice](#)

WaitUntilDone Method

The **WaitUntilDone** method blocks the caller until either the voice has finished speaking or the specified time interval has elapsed.

The purpose of this method is to block application execution while a voice is speaking asynchronously. The effect of performing a single `WaitUntilDone` call following a `Speak` or `SpeakStream` call is similar to performing those calls synchronously. But the `WaitUntilDone` method can be used in conjunction with the `DoEvents` statement to block an application's forward progress while allowing it to receive events. This is demonstrated in the example below.

```
SpVoice.WaitUntilDone(  
    msTimeout As Long  
) As Boolean
```

Parameters

msTimeout

Specifies the timeout in milliseconds. If -1, the time interval is ignored and the method simply waits for the voice to finish speaking.

Return Value

A Boolean variable indicating which case terminated the call. If True, the voice finished speaking; if False, the time interval elapsed.

Example

The following Visual Basic form code demonstrates the use of the `WaitUntilDone` method. To run this code, create a form with the following controls:

- Three picture controls called `Picture1`, `Picture2`, and `Picture3`
- Three command buttons called `Command`, `Command2`, and `Command3`

Paste this code into the Declarations section of the form.

The three picture controls have `MouseMove` event procedures which change their background colors when the mouse is moved over them. The `Form_MouseMove` event procedure resets the picture controls to a white background. Moving the mouse across the pictures will cause each to change color for as long as the mouse is over it.

The three command button procedures are similar. Each procedure disables all the command buttons, causes a text-to-speech (TTS) voice to speak a short phrase, and enables all command buttons when the voice finishes speaking. There is no significance to disabling and enabling the buttons except to show the duration of the speech.

Click the `Command1` button to cause the voice to speak synchronously. Moving the mouse over the three picture controls will not cause them to change colors; the `MouseMove` events are blocked by the synchronous `Speak` call.

Click the `Command2` button to cause the voice to speak asynchronously. The `WaitUntilDone` call prevents the `Command2` procedure from ending before the voice is finished. Moving the mouse over the three picture controls will not cause them to change colors; the `MouseMove` events are blocked by the `WaitUntilDone` call.

Click the `Command3` button to cause the voice to speak asynchronously. The `WaitUntilDone` method is called inside a loop which also contains a `DoEvents` statement. This loop prevents the `Command3` procedure from ending before the

voice is finished, and also allows MouseMove events to be received by the form. As a result, moving the mouse over the three picture controls causes them to change colors while the voice is speaking.

Option Explicit

```
Const INFINITE = -1& 'Tells WaitUntilDone to wait forever
```

```
Dim V As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Call EnableButtons(False) 'Speak synchronously  
                                'Disable buttons while voice is speaking
```

```
    V.Speak "Please move the mouse over the picture controls  
            & " while i speak synchronously."
```

```
    Call EnableButtons(True) 'Enable buttons when voice is finished
```

```
End Sub
```

```
Private Sub Command2_Click()
```

```
    Call EnableButtons(False) 'Speak asynchronously  
                                'Disable buttons while voice is speaking
```

```
    V.Speak "Please move the mouse over the picture controls  
            & " while i speak with a single Wait Until Done"  
            SVSFlagsAsync
```

```
    V.WaitUntilDone (INFINITE)  
    Call EnableButtons(True) 'Enable buttons when voice is finished
```

```
End Sub
```

```
Private Sub Command3_Click()
```

```
    Call EnableButtons(False) 'Speak asynchronously  
                                'Disable buttons while voice is speaking
```

```
    V.Speak "Please move the mouse over the picture controls
```

```

        & " while i speak with a smart loop.", _
        SVSFlagsAsync
    Do                               'Smart loop
        DoEvents                    'DoEvents lets events happen
    Loop Until V.WaitUntilDone(10)  'Loop until voice finished

    Call EnableButtons(True)        'Enable buttons when voice finished

End Sub

Private Sub Form_Load()
    Set V = New SpVoice
    V.Speak "Please move the mouse over the picture controls"
        & " before clicking the buttons", _
        SVSFlagsAsync
End Sub

Private Sub Form_MouseMove(Button As Integer, Shift As Integer,
                            X As Single,
                            Y As Single)
    Picture1.BackColor = vbWhite
    Picture2.BackColor = vbWhite
    Picture3.BackColor = vbWhite
End Sub

Private Sub Picture1_MouseMove(Button As Integer, Shift As Integer,
                                X As Single,
                                Y As Single)
    Picture1.BackColor = vbRed
End Sub

Private Sub Picture2_MouseMove(Button As Integer, Shift As Integer,
                                X As Single,
                                Y As Single)
    Picture2.BackColor = vbGreen
End Sub

Private Sub Picture3_MouseMove(Button As Integer, Shift As Integer,
                                X As Single,
                                Y As Single)
    Picture3.BackColor = vbBlue
End Sub

Private Sub EnableButtons(TrueFalse As Boolean)
    Command1.Enabled = TrueFalse
    Command2.Enabled = TrueFalse

```

```
Command3.Enabled = TrueFalse  
End Sub
```



SpVoice (Events)

The **SpVoice (Events)** automation object defines the types of events that can be received by an [SpVoice](#) object from a text-to-speech (TTS) engine.

In order to understand voice events, it is necessary to distinguish between the TTS engine, which synthesizes speech from text, and the SpVoice object, which applications employ to communicate with the engine. The TTS engine is somewhat like a server, and the SpVoice object like a client. The voice object sends the engine a request to speak a string of text. The engine processes the request as soon as it can. The interval between a speech request and the production of the speech is unpredictable. SpVoice events overcome this difficulty by providing applications with real-time feedback from the engine as it speaks, making it possible to synchronize application functions with speech. For example, an application can use a voice object's Viseme event to drive animations that display mouth movements as the engine speaks.

The voice object initiates requests with the [Speak](#) and [SpeakStream](#) methods, which send text strings and audio files to the TTS engine. These methods can be called synchronously or asynchronously. Because a synchronous speech request suspends execution of the calling application while the engine speaks the stream, events from the speaking of the stream are received after the stream has been spoken. Applications which need to receive events as real-time feedback should use asynchronous [Speak](#) and [SpeakStream](#) calls.

Examples of voice events are the beginning and the end of a text stream, and the boundaries of visemes, phonemes, words, and sentences. The [SpeechVoiceEvents](#) enumeration defines a constant for each type of voice event. Use one or more [SpeechVoiceEvents](#) constants to set the [EventInterests](#) property of a voice object. Only the types of events specified by [EventInterests](#) property will be sent by the TTS engine. The

default setting of this property specifies all voice event types except AudioLevel.

When using Visual Basic, you must use the "WithEvents" keyword to define an SpVoice object which receives events.

Events in file streams

When a voice object speaks into a filestream object, the TTS engine will embed event data in the file stream if all the following conditions are true:

- The voice object is defined using the "WithEvents" keyword
- The voice object's EventInterests property specifies at least one event type
- The audio output contains event conditions of a type specified in the voice's EventInterests
- The filestream object is opened for writing with its "DoEvents" parameter True

When TTS engine speaks a filestream object which contains embedded events, it will send events to the voice if all the following conditions are true:

- The voice object is defined using the "WithEvents" keyword
- The voice object's EventInterests property specifies at least one event type
- The file stream contains an embedded event of a type specified in the voice's EventInterests
- The filestream object is opened for reading with its "DoEvents" parameter True

When the TTS engine speaks a filestream object for a voice, if the voice's EventInterests specify StartStream and EndStream

events, the engine will send it a StartStream and an EndStream event, even if these events are not embedded in the stream. If StartStream and EndStream events are embedded in that file stream, the engine will send the voice two StartStream events and two EndStream events.

Automation Interfaces

The SpVoice (Events) automation object has the following elements:

Events	Description
AudioLevel Event	Occurs when the TTS engine detects an audio level change while speaking a stream for the SpVoice object.
Bookmark Event	Occurs when the TTS engine detects a bookmark while speaking a stream for the SpVoice object.
EndStream Event	Occurs when the TTS engine reaches the end of a stream it is speaking for the SpVoice object.
EnginePrivate Event	Occurs when a private TTS engine detects a custom event condition boundary while speaking a stream for the SpVoice object.
Phoneme Event	Occurs when the TTS engine detects a phoneme boundary while speaking a stream for the SpVoice object.
Sentence Event	Occurs when the TTS engine detects a sentence boundary while speaking a stream for the SpVoice object.
StartStream Event	Occurs when the TTS engine begins

	speaking a stream for the SpVoice object.
<u>Viseme</u> Event	Occurs when the TTS engine detects a viseme boundary while speaking a stream for the SpVoice object.
<u>VoiceChange</u> Event	Occurs when the TTS engine detects a change of voice while speaking a stream for the SpVoice object.
<u>Word</u> Event	Occurs when the TTS engine detects a word boundary while speaking a stream for the SpVoice object.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice \(Events\)](#)

AudioLevel Event

The **AudioLevel** event occurs when the text-to-speech (TTS) engine detects an audio level change while speaking a stream for the SpVoice object.

```
SpVoice.AudioLevel(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    AudioLevel As Long  
)
```

Parameters

StreamNumber

The stream number which generated the event. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate an event with the appropriate stream.

StreamPosition

The character position in the output stream at which the audio level change occurs.

AudioLevel

The new audio level.

Example

The following Visual Basic form code demonstrates the use of the AudioLevel event. To run this code, create a form with the

following controls:

- A command button called Command1
- A text box called Text1
- A list box called List1

Paste this code into the Declarations section of the form.

The Form_Load code creates an SpVoice object, adds AudioLevel to its event interests, and places sample text in Text1. The Command1_Click procedure speaks the text in Text1. The Word event code displays each word spoken. The AudioLevel event code converts each new audio level to a string of asterisks, effectively displaying the audio levels in a graph format.

Option Explicit

```
Public WithEvents vox As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    List1.Clear
```

```
    vox.Speak Text1.Text, SVSFlagsAsync + SVSFIsXML
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    ' SVEAudioLevel not in default EventInterests -- must be
```

```
    Set vox = New SpVoice
```

```
    vox.EventInterests = vox.EventInterests Or SVEAudioLevel
```

```
    Text1.Text = "audio levels change often"
```

```
End Sub
```

```
Private Sub vox_AudioLevel(ByVal StreamNumber As Long, ByVal  
    ByVal AudioLevel As Long)
```

```
    List1.AddItem String(AudioLevel, "*") 'AudioLevel value
```

```
End Sub
```

```
Private Sub vox_Word(ByVal StreamNumber As Long, ByVal StreamName As String,
                    ByVal CharacterPosition As Long, ByVal Length As Integer)
    List1.AddItem Mid(Text1.Text, CharacterPosition + 1, Length)
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice \(Events\)](#)

Bookmark Event

The **Bookmark** event occurs when the text-to-speech (TTS) engine detects a bookmark while speaking a stream for the SpVoice object.

It should be noted that Bookmark events may not be synchronized with the actual speaking of the words in text streams containing bookmarks. In some circumstances, TTS buffering considerations may cause a Bookmark event to be received sooner than the voice speaks the word preceding the bookmark in the text stream.

```
SpVoice.Bookmark(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    Bookmark As String,  
    BookmarkId As Long  
)
```

Parameters

StreamNumber

The stream number which generated the event. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate an event with the appropriate stream.

StreamPosition

The character position in the output stream at which the bookmark occurs.

Bookmark

The string value of the Mark attribute within the bookmark.

BookmarkId

The string value of the leading (left-most) numeric characters in the Mark attribute within the bookmark.

Example

The following Visual Basic form code demonstrates the use of the Bookmark event. To run this code, create a form with the following controls:

- A command button called Command1
- A text box called Text1

Paste this code into the Declarations section of the form.

The Form_Load code creates an SpVoice object. The Command1_Click procedure enqueues a short sentence containing a bookmark. The Bookmark event code displays the values of the BookmarkId and Bookmark parameters in Text1.

```
Option Explicit
```

```
Public WithEvents vox As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    Dim strTemp As String
```

```
    strTemp = "this is text <BOOKMARK mark='123456.789 abcde  
    vox.Speak strTemp, SVSFlagsAsync + SVSFlagsXML
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set vox = New SpVoice
```

```
End Sub
```

```
Private Sub vox_Bookmark(ByVal StreamNumber As Long, ByVal S
                        ByVal Bookmark As String, ByVal Boo

        Text1.Text = "BookmarkId: "" & BookmarkId & """, Bookma

End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice \(Events\)](#)

EndStream Event

The **EndStream** event occurs when the text-to-speech (TTS) engine reaches the end of a stream while speaking for the SpVoice object.

The [StartStream](#) and EndStream events can be used together to determine the duration of a stream being spoken.

```
SpVoice.EndStream(  
    StreamNumber As Long,  
    StreamPosition As Variant  
)
```

Parameters

StreamNumber

The stream number which generated the event. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate an event with the appropriate stream.

StreamPosition

The ending character position in the output stream.

Example

The following Visual Basic form code demonstrates the use of the StartStream and EndStream events. To run this code, create a form with the following controls:

- A command button called Command1

- A text box called Text1

Paste this code into the Declarations section of the form.

The Form_Load procedure puts a text string in Text1 and creates a voice object. The command1_Click procedure calls the Speak method. This will cause the TTS engine to send the EStream event and EndStream events to the voice. The StartStream event code changes the color of the text in the text box to red; the EndStream event changes the color back to black.

The text color change in this example has no significance other than showing the duration of the text stream.

```
Option Explicit
```

```
Public WithEvents vox As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()  
    vox.Speak Text1.Text, SVSFlagsAsync  
End Sub
```

```
Private Sub Form_Load()  
    Set vox = New SpVoice  
    Text1.Text = "This text turns red while being spoken."  
End Sub
```

```
Private Sub vox_EndStream(ByVal StreamNumber As Long, ByVal :  
    Text1.ForeColor = vbBlack  
End Sub
```

```
Private Sub vox_StartStream(ByVal StreamNumber As Long, ByVa.  
    Text1.ForeColor = vbRed  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice \(Events\)](#)

EnginePrivate Event

The **EnginePrivate** event occurs when a private text-to-speech (TTS) engine detects a custom event condition boundary while speaking a stream for the SpVoice object.

A private event is a custom event defined by the TTS engine. This event allows engines to define a specialized event beyond the standard suite of events and send custom events to SpVoice objects as though they were standard events. There is no requirement that engines support this event. The SAPI 5 Microsoft engines do not use the EnginePrivate event. If using another manufacturer's engine, check their documentation for possible implementation of this event.

```
SpVoice.EnginePrivate(  
    StreamNumber As Long,  
    StreamPosition As Long,  
    EngineData As Variant  
)
```

Parameters

StreamNumber

The stream number which generated the event. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate an event with the appropriate stream.

StreamPosition

The character position in the output stream at which the private event occurs.

EngineData

Data returned by the engine with the event. When using another manufacturer's TTS engine, consult its documentation for details.

Example

No sample code is available. The event is unique to manufacturer's engines and will vary among engines. The SAPI 5 Microsoft engines do not use the EnginePrivate event.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice \(Events\)](#)

Phoneme Event

The **Phoneme** event occurs when the text-to-speech (TTS) engine detects a phoneme boundary while speaking a stream for the SpVoice object.

```
SpVoice.Phoneme(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    Duration As Long,  
    NextPhoneId As Integer,  
    Feature As SpeechVisemeFeature,  
    CurrentPhoneId As Integer  
)
```

Parameters

StreamNumber

The stream number which generated the event. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate an event with the appropriate stream.

StreamPosition

The character position in the output stream at which the phoneme begins.

Duration

The duration of the phoneme, in milliseconds.

NextPhoneId

The next phone ID.

Feature

The SpeechVisemeFeature, which may indicate emphasis or stress on the viseme.

CurrentPhoneld

The current phone ID.

Remarks

When the engine synthesizes a phoneme comprised of more than one phoneme element, it raises an event for each element. For example, when a Japanese TTS engine speaks the phoneme "KYA," which is comprised of the phoneme elements "KI" and "XYA," it raises an SPEI_PHONEME event for each element. Because the element "KI" in this case modifies the sound of the element following it, rather than initiating a sound, the duration of its SPEI_PHONEME event is zero.

Example

The following Visual Basic form code demonstrates the Phoneme event. To run this code, create a form with the following controls:

- A command button called Command1
- Two text boxes called Text1 and Text2

Paste this code into the Declarations section of the form.

The Form_Load procedure puts a text string in Text1 and creates a voice object, leaving all its properties with their default settings. The command1_Click procedure calls the Speak method. This will cause the TTS engine to send the Phoneme event to the voice; the Phoneme event code will display the

phoneme values in Text2.

```
Option Explicit
```

```
Public WithEvents vox As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    vox.Speak Text1.Text, SVSFlagsAsync
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set vox = New SpVoice
```

```
    Text1.Text = "This is text in a text box."
```

```
End Sub
```

```
Private Sub vox_Phoneme(ByVal StreamNumber As Long, ByVal St
```

```
    Text2.Text = Text2.Text & CurrentPhoneId & " "
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice \(Events\)](#)

Sentence Event

The **Sentence** event when the text-to-speech (TTS) engine detects a sentence boundary while speaking a stream for the SpVoice object.

```
SpVoice.Sentence(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    CharacterPosition As Long,  
    Length As Long  
)
```

Parameters

StreamNumber

The stream number which generated the event. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate an event with the appropriate stream.

StreamPosition

The character position in the output stream at which the sentence begins.

CharacterPosition

The character position in the input stream one character before the start of the sentence. In the case of the first sentence in a stream, this parameter is zero.

Length

The length of the sentence.


```
' In order to show this selection,  
' the Text1.HideSelection property must be False
```

```
Text1.SelStart = CharacterPosition  
Text1.SelLength = Length
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice \(Events\)](#)

StartStream Event

The **StartStream** event occurs when the text-to-speech (TTS) engine begins speaking a stream for the SpVoice object.

The StartStream and [EndStream](#) events can be used together to determine the duration of a stream being spoken.

```
SpVoice.StartStream(  
    StreamNumber As Long,  
    StreamPosition As Variant  
)
```

Parameters

StreamNumber

The stream number which generated the event. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate an event with the appropriate stream.

StreamPosition

The character position in the output stream at which the stream begins.

Example

The following Visual Basic form code demonstrates the use of the StartStream and EndStream events. To run this code, create a form with the following controls:

- A command button called Command1

- A text box called Text1

Paste this code into the Declarations section of the form.

The Form_Load procedure puts a text string in Text1 and creates a voice object. The command1_Click procedure calls the Speak method. This will cause the TTS engine to send the StartStream and EndStream events to the voice. The StartStream event code changes the color of the text in the text box to red; the EndStream event changes the color back to black.

The text color change in this example has no significance other than showing the duration of the text stream.

```
Option Explicit
```

```
Public WithEvents vox As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()  
    vox.Speak Text1.Text, SVSFlagsAsync  
End Sub
```

```
Private Sub Form_Load()  
    Set vox = New SpVoice  
    Text1.Text = "This text turns red while being spoken."  
End Sub
```

```
Private Sub vox_EndStream(ByVal StreamNumber As Long, ByVal :  
    Text1.ForeColor = vbBlack  
End Sub
```

```
Private Sub vox_StartStream(ByVal StreamNumber As Long, ByVa.  
    Text1.ForeColor = vbRed  
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice \(Events\)](#)

Viseme Event

The **Viseme** event occurs when the text-to-speech (TTS) engine detects a viseme boundary while speaking a stream for the SpVoice object.

```
SpVoice.Viseme(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    Duration As Long,  
    NextVisemeId As SpeechVisemeType,  
    Feature As SpeechVisemeFeature,  
    CurrentVisemeId As SpeechVisemeType  
)
```

Parameters

StreamNumber

The stream number which generated the event. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate an event with the appropriate stream.

StreamPosition

The character position in the output stream at which the viseme begins.

Duration

The duration of the viseme state.

NextVisemeld

The next viseme ID.

Feature

The SpeechVisemeFeature, which may indicate emphasis or stress on the viseme.

CurrentVisemeld

The current viseme ID.

Example

The following Visual Basic form code demonstrates the Viseme event. To run this code, create a form with the following controls:

- A command button called Command1
- Two text boxes called Text1 and Text2

Paste this code into the Declarations section of the form.

The Form_Load procedure puts a text string in Text1 and creates a voice object. The command1_Click procedure calls the Speak method. This will cause the TTS engine to send the Viseme event to the voice. The Viseme event code will display the viseme values in Text2.

```
Option Explicit
```

```
Public WithEvents vox As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    vox.Speak Text1.Text, SVSFlagsAsync
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set vox = New SpVoice
```

```
Text1.Text = "This is text in a text box."
```

```
End Sub
```

```
Private Sub vox_Viseme(ByVal StreamNumber As Long, ByVal Str
```

```
Text2.Text = Text2.Text & CurrentVisemeId & " "
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice \(Events\)](#)

VoiceChange Event

The **VoiceChange** event occurs when the text-to-speech (TTS) engine detects a change of voice while speaking a stream for the SpVoice object.

```
SpVoice.VoiceChange(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    VoiceObjectToken As SpObjectToken  
)
```

Parameters

StreamNumber

The stream number which generated the event. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate an event with the appropriate stream.

StreamPosition

The character position in the output stream at which the change of voice occurs.

VoiceObjectToken

The ObjectToken of the new voice.

Example

The following Visual Basic form code demonstrates the use of the VoiceChange event. To run this code, create a form with the

following controls:

- A command button called Command1
- A text box called Text1

Paste this code into the Declarations section of the form.

The Form_Load code creates an SpVoice object. The Command1_Click procedure sets the object's Voice property to three different voices, and enqueues a short sentence in each voice. Each time the TTS engine speaks with a new Voice property, a VoiceChange event is raised. The VoiceChange event code displays the name of the new voice in Text1.

Option Explicit

```
Public WithEvents vox As SpeechLib.SpVoice
Const cstrText = "my voice just changed."
```

```
Private Sub Command1_Click()
```

```
    Set vox.voice = vox.GetVoices("name = microsoft mary").It
    vox.Speak cstrText, SVSFlagsAsync
```

```
    Set vox.voice = vox.GetVoices("name = microsoft mike").It
    vox.Speak cstrText, SVSFlagsAsync
```

```
    Set vox.voice = vox.GetVoices("name = microsoft sam").It
    vox.Speak cstrText, SVSFlagsAsync
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set vox = New SpVoice
```

```
End Sub
```

```
Private Sub vox_VoiceChange(ByVal StreamNumber As Long, ByVal
    ByVal VoiceObjectToken As Speech
```

```
Text1.Text = VoiceObjectToken.GetDescription
```

```
End Sub
```

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpVoice \(Events\)](#)

Word Event

The **Word** event occurs when the text-to-speech (TTS) engine detects a word boundary while speaking a stream for the SpVoice object.

```
SpVoice.Word(  
    StreamNumber As Long,  
    StreamPosition As Variant,  
    CharacterPosition As Long,  
    Length As Long  
)
```

Parameters

StreamNumber

The stream number which generated the event. When a voice enqueues more than one stream by speaking asynchronously, the stream number is necessary to associate an event with the appropriate stream.

StreamPosition

The character position in the output stream at which the word begins.

CharacterPosition

The character position in the input stream one character before the start of the word. In the case of the first word in a stream, this parameter is zero.

Length

The length of the word in the input stream.

Example

The following Visual Basic form code demonstrates the use of the Word event. To run this code, create a form with the following controls:

- A command button called Command1
- A text box called Text1
- Set the HideSelection property of Text1 to False

Paste this code into the Declarations section of the form.

The Form_Load procedure puts a text string in Text1 and creates a voice object. The command1_Click procedure calls the Speak method. This will cause the TTS engine to send the Word event to the voice; the Word event code will use the event parameters to highlight the word associated with the event.

```
Option Explicit
```

```
Public WithEvents vox As SpeechLib.SpVoice
```

```
Private Sub Command1_Click()
```

```
    vox.Speak Text1.Text, SVSFlagsAsync
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Set vox = New SpVoice
```

```
    Text1.Text = "This is some text in a textbox."
```

```
End Sub
```

```
Private Sub vox_Word(ByVal StreamNumber As Long, ByVal Stream  
                    ByVal CharacterPosition As Long, ByVal |
```

```
' In order to show this selection,  
' the Text1.HideSelection property must be False!
```

```
Text1.SelStart = CharacterPosition  
Text1.SelLength = Length
```

```
End Sub
```

Microsoft Speech SDK

Speech Automation 5.1



SpWaveFormatEx

The **SpWaveFormatEx** automation object represents the format of waveform-audio data.

The SpWaveFormatEx object gets and sets the audio format property of the [SpAudioFormat](#) object. Please see a code example in the SpAudioFormat [GetWaveFormatEx](#) section.

Automation Interface Elements

The SpWaveFormatEx automation object has the following elements:

Properties	Description
AvgBytesPerSec Property	Gets and sets the required average data-transfer rate for the format tag in bytes per second.
BitsPerSample Property	Gets and sets the bits per sample for the FormatTag format type.
BlockAlign Property	Gets and sets the block alignment in bytes.
Channels Property	Gets and sets the number of channels in the waveform-audio data.
ExtraData Property	Gets and sets extra format information.
FormatTag Property	Gets and sets the waveform-audio format type.
SamplesPerSec Property	Gets and sets the sample rate at which each channel should be played or recorded.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpWaveFormatEx](#)

AvgBytesPerSec Property

The **AvgBytesPerSec** property gets and sets the required average data-transfer rate for the format tag in bytes per second.

If the FormatTag property is WAVE_FORMAT_PCM, AvgBytesPerSec should be equal to the product of SamplesPerSec and BlockAlign. For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

Playback and record software can estimate buffer sizes using the AvgBytesPerSec member.

Syntax

```
Set: SpWaveFormatEx.AvgBytesPerSec = Long
```

```
Get: Long = SpWaveFormatEx.AvgBytesPerSec
```

Parts

SpWaveFormatEx

The owning object.

Long

Set: A Long variable that sets the property.

Get: A Long variable that gets the property.

Example

For an example of the use of the AvgBytesPerSec property, see the code example in the SpAudioFormat [GetWaveFormatEx](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpWaveFormatEx](#)

BitsPerSample Property

The **BitsPerSample** property gets and sets the bits per sample for the FormatTag format type.

If FormatTag is WAVE_FORMAT_PCM, BitsPerSample should be equal to 8 or 16. For non-PCM formats, this member must be set according to the manufacturer's specification of the format tag.

Note that some compression schemes cannot define a value for BitsPerSample, so this member can be zero.

Syntax

Set: *SpWaveFormatEx*.**BitsPerSample** = *Integer*

Get: *Integer* = *SpWaveFormatEx*.**BitsPerSample**

Parts

SpWaveFormatEx

The owning object.

Integer

Set: An Integer variable that sets the property.

Get: An Integer variable that gets the property.

Example

For an example of the use of the BitsPerSample property, see the code example in the SpAudioFormat [GetWaveFormatEx](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpWaveFormatEx](#)

BlockAlign Property

The **BlockAlign** property gets and sets the block alignment in bytes.

Syntax

Set: *SpWaveFormatEx*.**BlockAlign** = *Integer*

Get: *Integer* = *SpWaveFormatEx*.**BlockAlign**

Parts

SpWaveFormatEx

The owning object.

Integer

Set: An Integer variable that sets the property.

Get: An Integer variable that gets the property.

Remarks

The block alignment is the minimum atomic unit of data for the FormatTag format type. If the FormatTag is WAVE_FORMAT_PCM, BlockAlign should be equal to the product of Channels and BitsPerSample divided by 8 (bits per byte). For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

Playback and record software handles audio data in blocks. The sizes of these blocks are multiples of the value of the BlockAlign property. Data written and read from a device must always start at the beginning of a block. For example, it is illegal to start playback of PCM data in the middle of a sample (that is, on a

non-block-aligned boundary).

Example

For an example of the use of the BlockAlign property, see the code example in the SpAudioFormat [GetWaveFormatEx](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpWaveFormatEx](#)

Channels Property

The **Channels** property gets and sets the number of channels in the waveform-audio data.

Monaural data uses one channel and stereo data uses two channels.

Syntax

```
Set: SpWaveFormatEx.Channels = Integer
```

```
Get: Integer = SpWaveFormatEx.Channels
```

Parts

SpWaveFormatEx

The owning object.

Integer

Set: An Integer variable that sets the property.

Get: An Integer variable that gets the property.

Example

For an example of the use of the Channels property, see the code example in the SpAudioFormat [GetWaveFormatEx](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpWaveFormatEx](#)

ExtraData Property

The **ExtraData** property gets and sets extra format information. This information can be used by non-PCM formats to store extra attributes for the FormatTag property. For WAVE_FORMAT_PCM formats, this parameter is ignored.

Syntax

Set: *SpWaveFormatEx*.**ExtraData** = *Variant*

Get: *Variant* = *SpWaveFormatEx*.**ExtraData**

Parts

SpWaveFormatEx
The owning object.

Variant

Set: A Variant variable that sets the property.

Get: A Variant variable that gets the property.

Example

For an example of the use of the ExtraData property, see the code example in the SpAudioFormat [GetWaveFormatEx](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpWaveFormatEx](#)

FormatTag Property

The **FormatTag** property gets and sets the waveform-audio format type.

Format tags are registered with Microsoft Corporation for many compression algorithms. A complete list of format tags is located in the Mmsystem.h header file.

Syntax

```
Set: SpWaveFormatEx.FormatTag = Integer
```

```
Get: Integer = SpWaveFormatEx.FormatTag
```

Parts

SpWaveFormatEx

The owning object.

Integer

Set: An Integer variable that sets the property.

Get: An Integer variable that gets the property.

Example

For an example of the use of the FormatTag property, see the code example in the SpAudioFormat [GetWaveFormatEx](#) section.

Microsoft Speech SDK



Speech Automation 5.1

Object: [SpWaveFormatEx](#)

SamplesPerSec Property

The **SamplesPerSec** property gets and sets the sample rate at which each channel should be played or recorded.

If FormatTag is WAVE_FORMAT_PCM, common values for SamplesPerSec are 8.0 kHz, 11.025 kHz, 22.05 kHz, and 44.1 kHz. For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

Syntax

Set: *SpWaveFormatEx*.**SamplesPerSec** = Long

Get: Long = *SpWaveFormatEx*.**SamplesPerSec**

Parts

SpWaveFormatEx
The owning object.

Long
Set: A Long variable that sets the property.
Get: A Long variable that gets the property.

Example

For an example of the use of the SamplesPerSec property, see the code example in the SpAudioFormat [GetWaveFormatEx](#) section.



Microsoft Speech SDK Setup 5.1

Introduction

The Microsoft Speech SDK setup is built on the Windows Installer technology. The SAPI 5.1 core components are only available for distribution through Windows Installer merge modules and are called .msm files. These .msm files should be included in an applications setup and packaged in a Microsoft Installer .msi file. The .msi files consume the merge modules and handle the actual installation process. All other application-specific installation files should be included in this module. The .msi is run by the setup.exe file. The setup.exe file determines if the Windows Installer is present on your system and installs it if necessary. SAPI 5.1 is redistributable by independent software vendors (ISVs) or individuals by including the Speech SDK MSMs in their setup process and using the Windows Installer merge module technologies.

The following topics are covered in this section:

- [SAPI 4.0 and SAPI 5.0 coexistence](#)
- [Speech SDK merge modules \(MSM\)](#)
- [Speech SDK core modules](#)
- [Speech recognition \(SR\) modules](#)
- [Text-to-speech \(TTS\) modules](#)
- [Speech SDK modules](#)
- [Speech SDK file locations](#)
- [Speech SDK files](#)
- [Quiet Install](#)
- [Registry settings](#)

- [Building your setup package](#)
- [Glossary of terms and abbreviations](#)

SAPI 4.0 and SAPI 5.1 coexistence

In order to ensure that SAPI 4.0 and SAPI 5.1 can coexist on a single machine, the following four steps have been taken:

1. SAPI 5.1 dlls have different names from SAPI 4.0 dlls
2. SAPI 5.1 registry keys are registered in different locations
3. SAPI 5.1 GUIDs are different from SAPI 4.0 GUIDs
4. SAPI 5.1 registry keys are different from SAPI 4.0 registry keys

There is one known issue: SAPI 4.0 and SAPI 5.1 cannot share a common microphone. If one version of SAPI has control of the microphone, the other cannot use it. However, the functionality of TTS is not affected.

The engines that will be shipped with the Microsoft Speech SDK are shown below:

Engine	Language
SR	English
	Japanese
	Chinese
TTS	English
	Chinese

[Back to top](#)

Speech SDK merge modules (MSM)

The Speech SDK setup will produce merge modules for use with MSI to support the following configurations:

- [Speech SDK core modules](#)
- [Speech Recognition \(SR\) modules](#)
- [Text to Speech \(TTS\) modules](#)
- [Speech SDK modules](#)

Speech SDK core modules

sp5.msm

SAPI 5.1 includes the following: SAPI.DLL ,SAPISVR.exe, sapi.cpl, sapi.cpl help files: This is made available as a redistributable component for ISVs, both application vendors and engine vendors. The ISV will be expected to install their engines in the proper location and make the proper registry settings for the engines. For more information, please see the [Speech SDK file locations](#) section below.

Sp5intl.msm

These modules contain localized resource .dlls needed for the Control Panel, sapi.cpl. (Sapi.cpl contains English resources so that if none of the sp5intl.msm's are installed it would default to English). The Control Panel is the GUI used to select the various speech engines and for setting a voice enabled application. The Control Panel has been localized for other languages. Currently, there are three separate language dependent modules:

- English
- Japanese
- Chinese (simplified)

If the number of localized languages increases, separate msms for each language will be made available. It is recommended to include all localized msm's in 3rd party Setups so that if the user wants to install on Japanese or Chinese systems will be correctly localized.

This module has a dependency on sp5.msm.

[Back to top](#)

Speech Recognition (SR) modules

Sp5sr.msm

The Microsoft SAPI 5.1 SR engine files are contained within this module (spsreng.dll and spsrx.dll). There are no speech data files located within this module, as the Microsoft SR engine is language independent. This implies that one engine can perform SR processing for multiple languages by loading different speech data files.

This module has a dependency on sp5.msm and sp5intl.msm

Sp5itn.msm

The language specific SAPI 5.1 ITN components are located within this file. The ITN modules enable developers to include Inverse Text Normalization in the SR applications. There are three separate language dependent modules:

- English
- Japanese
- Chinese (simplified)

This module has a dependency on sp5sr.msm, sp5intl.msm and sp5.msm.

Sp5ccint.msm

All the acoustic and language modules of the Microsoft SAPI 5.1 SR engine are contained in this merge module. This module also contains localized resource .dlls for the Microsoft SR engine (spsrx.dll). These resource .dlls contain User Interfaces for the Training wizard and Microphone wizard. (Spsrx.dll contains English resources. If no sp5ccint.msm's are installed, it would default to English). There are three separate language dependent modules:

- English
- Japanese
- Chinese (simplified)

This module has a dependency on sp5sr.msm, sp5intl.msm and sp5.msm.

[Back to top](#)

Text to Speech (TTS) modules

Sp5ttint.msm

The Microsoft SAPI 5.1 TTS English engine is a language dependent TTS engine. This implies that the TTS engine module will contain the engine as well as the data files for the engine. Currently, Microsoft is shipping the following language TTS engines:

- English
- Chinese (simplified)

This module has a dependency on sp5.msm and sp5intl.msm.

spcommon.msm

Contains files that are common to both the Microsoft SAPI 5.1 TTS and SR engine. Currently, this is shipped for the following languages:

- English

[Back to top](#)

Microsoft Speech SDK modules

Sp5sdk.msi

The full installation of the Microsoft Speech SDK includes the following modules:

- sp5.msm
- sp5intl.msm
- sp5sr.msm
- sp5ccint.msm
- Sp5ttint.msm
- spcommon.msm

The Speech SDK samples and help documentation for SAPI 5.1 API/DDI will be included in the installation.

[Back to top](#)

Speech SDK file locations

Setup will verify the versions of the various installed components. Setup will detect if the operating system one of the following:

Supported Operating Systems

- Microsoft Windows(r) NT Workstation 4.0, service pack 6a, English, Japanese or Simplified Chinese edition.
- Microsoft Windows 2000 Professional Workstation, English edition or English edition with Japanese or Simplified Chinese Language support.
- Microsoft Windows 98. However, Windows 95 is not supported.
- Microsoft Windows Millennium edition.

When attempting to install the Microsoft Speech SDK on a non-supported operating system, a dialog box will appear with the string "SAPI5 is currently not supported on this Operating System. You must upgrade to Windows 98 or higher." After the Speech SDK has been installed on your computer, the footprint will not be deleted from the drive. If you accidentally delete the sapi.dll and then tries to run one of the applications, then the footprint file will be able to get the sapi.dll file and install it. All engine files should follow the 8.3 naming convention.

NOTE: To obtain the Chinese (Simplified) and Japanese Microsoft SR Engines and the Chinese (Simplified) TTS engine, please install the Microsoft Speech SDK 5.1 Language Pack.

SAPI.DLL

The sapi.dll file is the main dll for SAPI 5.1. This file should

be independent of engine vendor and language. As a result, this file should be located in the system directory.

Control Panel

The Control Panel file is the file for the Control Panel. This file should be independent of engine vendor and language. As a result, this file should be located in the system directory.

Lexicons

The SAPI 5.1 user lexicons should be placed in the user's profile directory under the speech directory.

For example, for Windows 2000 installations, this would be Documents and Settings*<user name>*\Speech.

SR Engine

The SR engines may be installed on any drive on a user's computer. The SR engine is language independent (i.e., the same engine is loaded with different data to create a different language engine). As a result, the SR engine files, by default, should be located in a language independent path, as follows:

Microsoft SR Engine

The SR engines may be installed on any drive on a user's computer. The SR engine is language independent (i.e., the same engine is loaded with different data to create a different language engine). As a result, the SR engine files by default should be located in a language independent path, as follows: program files\common files\speechengines\Microsoft\sr

The SR data files contain the language specific information. The SR data files (including the files needed for command and control (C and C) and ITN) should be located in the following path: program

files\common files\speechengines\Microsoft\sr\>
where the <LCID> is 1033 (English), 2052 (Chinese
(simplified)) and 1041 (Japanese)

TTS Engine

Microsoft TTS Engine

The TTS engines may be installed on any drive on a user's computer. The Microsoft TTS engine is not language independent (i.e., currently, each language is based on a different TTS code base). As a result, the TTS engine by default should be placed under the LCID it represents. The TTS engine files should be located in the following path: program files\common files\speechengines\Microsoft\TTS\

[Back to top](#)

Speech SDK files

The Microsoft Speech SDK contains a number of samples and tools. These samples should be located in the following directories:

Executable Files

All compiled executable files of the samples and the grammar compiler are located in the following directory:
\\Microsoft Speech SDK 5.1\\bin

Help Documentation

The reference file SAPI5SDK.chm is located in: \\Microsoft Speech SDK 5.1\\docs\\help

SAPI 5.1 IDL

The sapi.idl contains all of the API function declarations in SAPI 5.1. This is the main file used by application developers when developing speech enabled applications.
\\Microsoft Speech SDK 5.1\\idl

SAPI5ddk idl

The sapi5ddk.idl contains the DDI function declarations for SAPI. This is the main file used by engine developers when developing speech engines. \\Microsoft Speech SDK 5.1\\idl

Header files

The header files for SAPI 5.1 should be located in:
\\Microsoft Speech SDK 5.1\\include

Miscellaneous

The following folder contains the sapi.lib \\Microsoft Speech

SDK 5.1\lib\i386

Samples

The following table outlines the location of the source code for the various samples application and tools.

Name	Path
Dictation Pad	\Microsoft Speech SDK 5.1\samples\cpp\dictpad
Simple Dictation	\Microsoft Speech SDK 5.1\samples\cpp\simpledict
TTSApp	\Microsoft Speech SDK 5.1\samples\cpp\TTSApp
Tutorial - Coffee S0	\Microsoft Speech SDK 5.1\samples\cpp\tutorials\CoffeeS0
Tutorial - Coffee S1	\Microsoft Speech SDK 5.1\samples\cpp\tutorials\CoffeeS1
Tutorial - Coffee S2	\Microsoft Speech SDK 5.1\samples\cpp\tutorials\CoffeeS2
Tutorial - Coffee S3	\Microsoft Speech SDK 5.1\samples\cpp\tutorials\CoffeeS3
Tutorial - Coffee S4	\Microsoft Speech SDK 5.1\samples\cpp\tutorials\CoffeeS4
Tutorial - Coffee S5	\Microsoft Speech SDK 5.1\samples\cpp\tutorials\CoffeeS5
Tutorial - Coffee S6	\Microsoft Speech SDK 5.1\samples\cpp\tutorials\CoffeeS6
Talkback	\Microsoft Speech SDK 5.1\samples\cpp\talkback
Telephony Application	\Microsoft Speech SDK 5.1\samples\cpp\telephony
SR Engine (Null engine)	\Microsoft Speech SDK 5.1\samples\cpp\engines\SR
TTS engine (Null	\Microsoft Speech SDK

engine)	5.1\samples\cpp\engines\TTS
SPComp	\Microsoft Speech SDK 5.1\bin
SRComp	\Microsoft Speech SDK 5.1\tools\comp\SR
TTSComp	\Microsoft Speech SDK 5.1\tools\comp\TTS
Grammar Editor	\Microsoft Speech SDK 5.1\bin
SimpleAudioDll	\Microsoft Speech SDK 5.1\Samples\CPP\SimpleAudioDll
TapiCustomStream	\Microsoft Speech SDK 5.1\Samples\CPP\TapiCustomStream
ListBoxCSharp	\Microsoft Speech SDK 5.1\Samples\CSharp\Listbox
SimpleTTSCSharp	\Microsoft Speech SDK 5.1\Samples\CSharp\SimpleTTS
SimpleTTSJScript	\Microsoft Speech SDK 5.1\Samples\Scripts\SimpleTTS
AudioApp	\Microsoft Speech SDK 5.1\Samples\VB\AudioApp
ListboxVB	\Microsoft Speech SDK 5.1\Samples\VB>ListboxVB
RecoVB	\Microsoft Speech SDK 5.1\Samples\VB\RecoVB
SimpleDictVB	\Microsoft Speech SDK 5.1\Samples\VB\SimpleDict
SimpleTTSVB	\Microsoft Speech SDK 5.1\Samples\VB\SimpleTTS
TTSAppVB	\Microsoft Speech SDK 5.1\Samples\VB\TTSAppVB
VBTapiSample	\Microsoft Speech SDK 5.1\Samples\VB\VBTAPIOSamples
Mkvoice	\Microsoft Speech SDK 5.1\samples\cpp\engines\TTS\mkVoice

[Back to top](#)

Quiet Install

The command used for quiet install is "msiexec /i "Microsoft Speech SDK 5.1.msi" /qn". "setup.exe /S /v/qn" is also another usable option.

[Back to top](#)

Registry Settings"

For engine specific registry settings, please see the [Object Tokens and Registry Settings](#) white paper. All registry keys will be manually created and deleted upon installation and uninstallation respectively. This means that nothing in the setup procedure that will use self registration. Microsoft will not handle the lazy initialization for the user profiles.

Setup is not able to know about the individuals who will be using speech features after installation. The lazy registration information will continue to be built up by sapi.dll at run time.

[Back to top](#)

Building your Setup Package

The easiest way to build a Setup package that incorporates the SAPI 5.1 redistributable merge modules is to use a Setup tool that is designed specifically for the Windows Installer technology. Currently a number of these exist on the market including, but not limited to: Install Shield for Windows Installer, Visual Studio Installer, Wise for Windows Installer, and Seagate WinINSTALL. These tools can consume the SAPI 5.1 merge modules (.msm's) and seamlessly install the components along with the rest your setup. Simply consult the documentation on these products or follow the built in Wizards to build your Setup package and include the SAPI 5.1 .msm's.

If these tools are not available, consider downloading the Windows Installer SDK and building your Setup package manually. The following steps provide a walk through on how this is done.

1. Download the Windows Installer SDK from [Windows Installer 1.5](#).
2. Plan the Sample Installation. When the installation of an existing application is moved to the Windows Installer from another setup technology, the setup developer may start authoring a Windows Installer package using the source and target file images of the existing installation. A detailed plan of how the files and other resources are organized at the source and target is also a good starting point for developing a package for a new application.

For example, if you have a TTS application (YourTTSApp.exe) that you want to install along with the SAPI 5.1 merge modules, simply determine the source and destination locations of your application.

File	Path To Source	Path To
YourTTSApp.exe	C:\YourApp\YourTTSApp.exe	D:\Program Files\Com

3. Obtain the blank installation database Schema.msi from the Windows Installer SDK and rename it to yourProduct.msi.
4. Use the database editor Orca, which is provided with the SDK, or another editor, to open the installation database yourProduct.msi.
5. Use the editor to modify the following tables in the .msi:

Directory Table

Component Table

File Table

Media Table

Feature Table

Feature Components Table

Registry Table

ShortCut Table

Icon Table

Property Table

InstallExecuteSequence Table

InstallUISequence Table

AdminExecuteSequence Table

AdminUISequence Table

AdvExecuteSequence Table

For details on specific table values and entries, see the "Windows Installer Examples / An installation example" section of the Msi.chm help file that is included in the Windows Installer SDK.

6. Use the MsiInfo.exe tool provided with the Windows Installer SDK to add Summary Information to yourProduct.msi. The following properties must be set for your product to pass Package Validation. It is recommended that authors run validation on every new, or newly modified, installation package before attempting to install the package (see the Package Validation section of the Windows Installer SDK documentation for more about Package Validation).

Summary Information Property	Data	Notes
Template (Platform and Language)	;1033	Platform and language used by the database. Leaving the platform field empty indicates the package is platform independent. The ProductLanguage property from the database is typically used for this summary property. The sample's Language ID indicates that the package uses U.S. English.
Revision	{49D185A1-	This is the package

Number (Package Code)	D7FD-11D2-9159-00C04FD70856}	code GUID that uniquely identifies the sample package. If you reproduce this sample, use a utility such as GUIDGEN to generate a different GUID for your package. The results of GUIDGEN contain lowercase characters, note that you must change all lowercase characters to uppercase for a valid package code. See Package and Product Codes.
Page Count (Minimum Installer Version)	100	For Windows Installer version 1.0, this property should be set to the integer 100.
Word Count (Type of Source)	;1033	Platform and language used by the database. Leaving the platform field empty indicates the package is platform independent. The ProductLanguage property from the database is typically used for this summary property. The sample's

		Language ID indicates that the package uses U.S. English.
--	--	---

The remaining summary information stream properties are not required, but should be set for yourProduct.msi.

Summary Information Property	Data	Notes
Title	Installation Database	Informs users that this database is for an installation rather than a transform or a patch.
Subject	yourProduct	File browsers can display this as the product to be installed with this database.
Keywords	Installer, MSI, Database	File browsers that are capable of keyword searching can search for these words.
Author	Your Company Name	Name of the product's manufacturer.
Comments	This installer database contains the logic and data required to install	Informs the user about the purpose of this database.

	YourProduct.	
Creating Application	Orca	Application used to create the installation database.
Security	0	The sample database is unrestricted read-write.

To use MsiInfo to add the summary information to the sample, change to the directory containing the database yourProduct.msi and use the following command line:

```
MsiInfo.exe yourProduct.msi -T "Installation Database" -J Subject -A "Your Company Name" -K "Installer, MSI, Database" -O "This installer database contains the logic and data required to install YourProduct." -P ;1033 -V {49D185A1-D7FD-11D2-9159-00C04FD70856} -G 100 -W 0 -N Orca -U 0
```

7. Add the following User Interface information to the Property Table:

Property	Value
DefaultUIFont	DlgFont8
INSTALLLEVEL	3
LIMITUI	1
Manufacturer	Your Company Name
ProductCode	{19BED231-30AB-11D3-91D3-00C04FD70856}
ProductLanguage	1033
ProductName	yourProduct
ProductVersion	01.20.0000t
UpgradeCode	{ACFBE060-33B8-11D3-91D6-00C04FD70856}

8. Validate your installation. See the Package Validation section of the Windows Installer SDK documentation for more about Package Validation.

[□ Back to top](#)

Glossary of Terms and Abbreviations">

API

Application programming interface, the "top" side of the SAPI 5.1 middleware.

C and C

Command and control

CSR

Continuous speech recognition, also called dictation

DDI

Device Driver Interface. In SAPI 5.1, this is the interface speech engine providers code to (the underside of the middleware)

MSI

Microsoft Installer file containing the instructions and data required to install an application.

MSM

Microsoft Merge Module

SAPI

Microsoft Speech Application Programming Interface. SAPI 5.1 is middleware that provides an API for applications and a DDI for speech providers.

SR

Speech Recognition (includes both CSR and C and C)

TTS

Text-to-Speech, also called speech synthesis

[□ Back to top](#)



Global Variables

The following variables and constants are of special importance to SAPI developers.

- [Other Global Constants](#)
- [User Interface](#)

See also [ISpTokenUI](#) for a description of how to query if an object supports this UI type (see [ISpTokenUI::IsUISupported](#)) or how to display the UI (see [ISpTokenUI::DisplayUI](#)).



Using Sample Audio Object (SpAudioPlug)

Overview

This paper presents an overview to assist writing a custom audio object. It is intended to be used with the SAPI 5.1 SDK sample Visual Basic Audio Application. That application allows a user to enter the text in an edit box and perform speech recognition dictation on that text. A custom audio object is used to replace the traditional speaker. Rather than having the text talk over the speakers, the voice is redirected through the custom audio object to the speech recognition (SR) engine. At that point the voice is attempted to be recognized.

Implementation

The Visual Basic example VB Audio Application uses the automation interface provided by the sample audio object to do the audio data management. The application creates two instances of sample audio objects. One audio object is for text-to-speech (TTS) output, the other one is for SR input. The application would route the audio data from TTS output to SR input.

Additional custom audio processing is available in the SAPI 5.1 SDK samples [VB SAPI with Internet](#) and [VB Outgoing Call](#). The white paper [Speech Telephony Application Guide](#). VB SAPI with Internet provides implementation details.

Set up the TTS output

For TTS, we create an instance of the sample audio object and set to write mode.

```
Set AudioPlugOut = New SpAudioPlug  
AudioPlugOut.Init True, AUDIOFORMAT
```

Then the Voice's output is set to point to this audio object.

```
Set Voice = New SpVoice  
Set Voice.AudioOutputStream = AudioPlugOut
```

Set up the SR input

For SR, we create an instance of the sample audio object and set to read mode.

```
Set AudioPlugIn = New SpAudioPlug  
AudioPlugIn.Init False, AUDIOFORMAT
```

Then the Recognizer's input is set to point to this audio object.

```
Set Recognizer.AudioInputStream = AudioPlugIn
```

Start processing

The following code starts the TTS and SR processes, and routes audio data from TTS output to SR input

```
output = AudioPlugOut.GetData  
'Output the audio data to the input audio object  
  If (Len(output) * 2 <> 0) Then  
    AudioPlugIn.SetData (output)  
  End If
```



VendorPreferred Attribute

VendorPreferred is an attribute used by a token. The token may be for either a speech recognition (SR) engine or a text-to-speech (TTS) voice. It indicates that the vendor has a preference for a particular token if the vendor has more than one token installed, and that all tokens satisfy other requirements. However, this preference is considered only if the system does not already have a default selected.

For example, a TTS system may have three voices installed by the same vendor: Mike, Mary, and Sam. At various points in the application or in SAPI, the programmer would want to know what the default engine is for the current user. If the user has already chosen one, or the system administrator has set one up, that default is used and the VendorPreferred key is never evaluated. If no default is available, SAPI selects one based on, among other values, VendorPreferred.

For SAPI to select a default, the language of the user (which can be retrieved by [SpGetUserDefaultUILanguage](#)) is considered first. If a specific provider has multiple tokens installed, which all support the specified language, the one marked VendorPreferred becomes the default. Only one token should be marked as the VendorPreferred; however, if more than one is marked, SAPI automatically selects one. In any case, the selected token becomes the default for that category.

This process only applies if the user has not already chosen a default. Once the user chooses a default, VendorPreferred is not looked at further. However, a caller can also use the VendorPreferred attribute to look for a match for a specific vendor. For example, a user could call `SpEnumTokens(SPCAT_VOICES, L"Vendor=Microsoft;VendorPreferred", ...)` to ensure that they get Microsoft's pick for their voice, or similarly for the SR engine.



Speech Telephony Application Guide

Purpose

The purpose of this paper is to document the ability to add speech capabilities to telephony applications.

Overview

Application developers can use SAPI 5.x to speech-enable Microsoft TAPI applications. This includes processing speech with either telecommunication devices (such as a modem), or across a network. This paper provides two examples: one uses devices such as a standard voice modem, and the other uses a Internet connection. These samples assume that you are familiar with TAPI programming and possibly already have a telephony application that you wish to speech enable.

This paper primarily covers the following topics:

- [Adding SAPI automation to a telephony application](#)
- [Custom real time audio stream](#)
- [Pitfalls: Common problems encountered](#)

The section, Adding SAPI Automation to a Telephony Application, discusses how to set up SAPI audio input and output to telecommunication devices. The next section demonstrates a method to process the recognition results. This section is intended to help developers of TAPI add SAPI automation to telephony applications using a standard voice modem or other telephony hardware communication devices. However, the samples in the Recognition Results Storage and Retrieval section can be used with the Internet as well.

The next section, Custom Real Time Audio Stream, describes how to build a custom real time audio stream and connect SAPI audio input and output to the streams. The stream object, enables an application to have voice communication using SAPI on either a telephony device or the network.

Adding SAPI automation to a telephony application

The following sections demonstrate adding SAPI to a telephony application. The procedures and code samples are suggested methods only, but other methods may be used to fit your specific needs.

To incorporate the SAPI run time, the SAPI text-to-speech (TTS) and Speech Recognition (SR), the following methods or procedures may be used. SimpleTelephony, a simple speech-enabled telephony application using device connections in C++ is available with the SAPI SDK. See the SAPI SDK for additional details. Examples in this section will demonstrate this with Visual Basic 6.0 or later. In order to use the sample code in Visual Basic, SAPI 5.1 or later must also be installed on your system.

Set up SAPI audio input/output

To transmit voice data over telecommunications devices using SAPI, it is very important to set up the audio input and output to the specific audio device correctly. The following are examples of how to set the audio output and input in C++ and Visual Basic, respectively.

To set the audio output object for TTS in C/C++, use the following steps:

1. Create an ISpeechMMSysAudio audio object.
2. Retrieve the wav/out device identifier and set it to the audio object calling ISpMMSysAudio::SetDeviceId().
3. Find the wav format that your audio device supports and assign it to the audio object using ISpMMSysAudio::SetFormat().
4. Call ISpVoice::SetOutput () to inform the TTS engine of the audio object.

Note that the second parameter of ISpVoice::SetOutput() is set to False. This means that the ISpVoice object will use the SAPI format converter to translate between the data being rendered by TTS engines and the format of the output audio data. This prevents the audio format from being changed on the output device.

To set the audio input object for SR in C/C++, use the following steps:

1. Create an ISpeechMMSysAudio audio object.
2. Get the wav/in device identifier and set it to the audio object by calling ISpMMSysAudio:: SetDeviceId().
3. Find the wav format that your audio device supports and assign it to the audio object using

ISpMMSysAudio::SetFormat().

4. Call ISpRecognizer::SetInput() to inform the SR engine of the audio object. Set the second parameter of SetInput() to False to prevent the audio format on the input device from changing.

For additional information regarding setting up audio input and output in C++, please consult Simple Telephony Application in SAPI SDK.

Similarly, you can follow the above procedures to set the audio input and output in Visual Basic. You may discover that it is difficult to obtain the device identifier using TAPI ITLegacyCallMediaControl.GetID in Visual Basic. Hence, you may write supporting code in C++ to help the Visual Basic application to obtain the device identifier. In the following example, a TAPI helper interface, ITAPIHelper, is created in C/C++ to retrieve the device identifier and the wav format supported by the audio device for Visual Basic applications.

Snippet 1: ITAPIHelper Interface

Assume that the following APIs are exposed by the ITAPIHelper interface:

```
interface ITAPIHelper : IDispatch
{
    [id(1), helpstring("method GetDeviceIDWaveOut")]
    HRESULT GetDeviceIDWaveOut ([in]IUnknown * pBasicCallCtl,
    [out, retval]long *pDeviceId);
    [id(2), helpstring("method GetDeviceIDWaveIn")]
    HRESULT GetDeviceIDWaveIn ([in]IUnknown * pBasicCallCtl,
    [out, retval]long *pDeviceId);
    [id(3), helpstring("method FindSupportedWaveOutFormat")]
    HRESULT FindSupportedWaveOutFormat ([in]long DeviceId,
    [out, retval]SpeechAudioFormatType *pFormat);
    [id(4), helpstring("method FindSupportedWaveInFormat")]
    HRESULT FindSupportedWaveInFormat ([in]long DeviceId,
    [out, retval]SpeechAudioFormatType *pFormat);
};
```

GetDeviceIDWaveOut() and GetDeviceIDWaveIn() takes pBasicCallControl, which points to an ITBasicCallControl object created by the Visual Basic application. From the ITBasicCallControl object, you can query the ITLegacyCallMediaControl interface and then call ITLegacyCallMediaControl::GetID() to obtain the device identifier. The following sample code demonstrates the implementation of GetDeviceIDWaveOut() method.

```

STDMETHODIMP CTAPIHelper:: GetDeviceIDWaveOut (IUnknown *pBa
    long *pDeviceId)
{
    HRESULT hr = S_OK;
    CComPtr<ITBasicCallControl> cpBasicCallCtl;

    hr = pBasicCallCtl->QueryInterface( IID_ITBasicCallC
        (void**)&cpB

    // Get the LegacyCallMediaControl interface so that \
    // get a device ID to reroute the audio
    ITLegacyCallMediaControl *pLegacyCallMediaControl;
    if ( SUCCEEDED(hr) )
    {
hr = cpBasicCallCtl->QueryInterface( IID_ITLegacyCallMediaCo

        }

    // Get the device ID through ITLegacyCallMediaControl inter
    UINT *puDeviceID;
    BSTR bstrWavOut = ::SysAllocString( L"wave/out" );
    if ( !bstrWavOut )
    {
        return E_OUTOFMEMORY;
    }

    DWORD dwSize = sizeof( puDeviceID );
    if ( SUCCEEDED(hr) )
    {
hr = pLegacyCallMediaControl->GetID( bstrWavOut, &dwSize;,
    (BYTE**) &puDeviceID; );
    }

    *pDeviceId = *puDeviceID;
}

```

```

//clean up
::SysFreeString( bstrWavOut );
::CoTaskMemFree( puDeviceID );
pLegacyCallMediaControl->Release();
cpBasicCallCtl.Release ();

return hr;
}

```

To implement of GetDeviceIDWaveIn (), you only need to change wav/out to wav/in in SysAllocString().

The following sample demonstrates implementation of the FindSupportedWaveOutFormat() method. The sample loops through all of the SAPI audio formats and queries the wav/out device as to whether it supports the given format. The method returns as soon as it finds one. Similarly, you can change waveOutOpen to waveInOpen for the implementation of the FindSupportedWaveInFormat() method.

```

STDMETHODIMP CStHelper::FindSupportedWaveOutFormat(long DeviceID)
{
    HRESULT hr = S_OK;

    //Initialization
    GUID guidWave = SPDFID_WaveFormatEx;
    WAVEFORMATEX *pWaveFormatEx = NULL;
    SPSTREAMFORMAT enumFmtId= SPSF_NoAssignedFormat;

    // Find out what formats are supported
    if ( SUCCEEDED(hr) )
    {
        // Loop through all of the SAPI audio formats and query the device
        // about whether it supports each one. We will take the first one
        // that supports the given format.

        MMRESULT mmr = MMSYSERR_ALLOCATED;
        for ( DWORD dw = 0;
              (MMSYSERR_NOERROR != mmr) &&
              (dw < SPSF_NUM_FORMATS); dw++ )
        {

```

```

if ( pWaveFormatEx && ( MMSYSERR_NOERROR != mmr ) )
{
    // The audio device does not support this format
    // Free up the WAVEFORMATEX pointer
    ::CoTaskMemFree( pWaveFormatEx );
    pWaveFormatEx = NULL;
}

// Get the next format from SAPI and convert it into
enumFmtId = (SPSTREAMFORMAT) (SPSF_8kHz8BitM
HRESULT hrConvert = SpConvertStreamFormatEnum
enumFmtId, &guidWave;, &pWaveFormatEx);

if ( SUCCEEDED( hrConvert ) )
{
    // This call to waveOutOpen() does not actually open the device.
    // it just queries the device whether it supports the given format.
    mmr = ::waveOutOpen( NULL, DeviceId, pWaveFormatEx, 0, 0,
        }
}

// If we made it all the way through the loop without breaking
// means we found no supported formats
if ( enumFmtId == SPSF_NUM_FORMATS )
{
    return SPERR_DEVICE_NOT_SUPPORTED;
}

}

if ( SUCCEEDED( hr ) )
{
    *pFormat = (SpeechAudioFormatType)enumFmtId;

if ( pWaveFormatEx )
{
    ::CoTaskMemFree( pWaveFormatEx );
}
}

return hr;
}

```

Snippet 2: Use of ITAPIHelper Object in Visual Basic

The following code snippet illustrates setting up audio input and output to the devices in Visual Basic using the ITAPIHelper object.

Set up the audio output for TTS:

```
Dim MMSysAudioOut As ISpeechMMSysAudio
Dim TapiHelper As TAPIHelper

    'Create SpMMAudioOut object
Set MMSysAudioOut = New SpMMAudioOut

'Create helper object
Set TapiHelper = New TAPIHelper

'Get the device identifier and set it to audio out
MMSysAudioOut.DeviceId = TapiHelper.GetDeviceIDWaveOut(...)

'Find the supported wav format and set it to audio out object
MMSysAudioOut.Format.Type = TapiHelper.FindSupportedWaveOutFormat(...)

'Prevent format changes
VoiceObj.AllowAudioOutputFormatChangesOnNextSet = False

'Set the object as the audio output
Set VoiceObj.AudioOutputStream = MMSysAudioOut
```

Set up the audio input for SR:

```
Dim MMSysAudioIn As ISpeechMMSysAudio

'Create an SpMMAudioIn object
Set MMSysAudioIn = New SpMMAudioIn

'Get the device identifier and assign it to audio in object
MMSysAudioIn.DeviceId = TapiHelper.GetDeviceIDWaveIn(...)

'Find the supported wave in format and set it to audio in object
MMSysAudioIn.Format.Type = TapiHelper.FindSupportedWaveInFormat(...)
```

'Prevents format changes
RecognizerObj.AllowAudioInputFormatChangesOnNextSet = False

'Set the object as the audio input
Set RecognizerObj.AudioInputStream = MMSysAudioIn

'Release the helper object
Set TapiHelper = Nothing

Recognition result storage and retrieval

Once the audio input and output are set up, you can use SAPI TTS and SR functions to play or transcribe audio through the audio devices. This section details the processing of recognition results after a call has been connected. The sample code in this section has general purpose so it may be used for other connections such as the Internet.

To demonstrate using SAPI playback and transcribing for a telephony application, the following code examples use the case of a simple speech-enabled voice mail system. Using the sample application, the caller chooses from a menu of two options: leave a message and check messages. In order to use these functions, the application needs to store the recognition results for the left messages, and retrieve the results for checked messages. The following code snippet illustrates the initialization of SAPI, the use of ISpeechMemoryStream and the recognition results storage and retrieval in Visual Basic.

Declaration of variables

The following are declared as global variables and used in the speech-related APIs in the example.

```
Dim WithEvents VoiceObj As SpVoice
Dim RecognizerObj As SpInprocRecognizer 'SR
Dim WithEvents RecoContextObj As SpInProcRecoContext 'Reco
Dim DictationGrammarObj As ISpeechRecoGrammar 'Dic
Dim gMemStream As SpMemoryStream 'Mem

Dim StreamLength(100) 'Array of the lengths of each
Dim NumOfResults As Long 'Number of stored results
Enum GRAMMARIDS
    GID_DICTATION = 1 'ID for the dictation grammar
    GID_CC = 2 'ID for the C and C grammar
End Enum
```

SAPI initialization

The TTS voice object, in the following sample, is obtained from the RecoContextObj instead of from a separate voice object. This allows the application to play back the retained audio later using ISpeechRecoResult.SpeakAudio. Code snippets 3 through 6 demonstrate initializing SAPI objects. For simplicity, dictation is used, although realistically you may use command and control (C and C) grammar for better recognition of the menu of choices and dictation grammar for transcribing messages.

Snippet 3: Initialization

```
'Create a recognizer object
Set RecognizerObj = New SpInprocRecognizer

'Create a RecoContext object
Set RecoContextObj = RecognizerObj.CreateRecoContext

'Get the voice object from the RecoContext object
Set VoiceObj = RecoContextObj.Voice

'Although by default, all of SR events, except the audio level
RecoContextObj.EventInterests = SRERecognition + SRESoundEnd
                                SREStreamStart + SRESoundEnd

'Retain the audio data in recognition result
RecoContextObj.RetainedAudio = SRA0RetainAudio

'Create the dictation grammar
Set DictationGrammarObj = RecoContextObj.CreateGrammar(GID_D:

'Load dictation grammar
DictationGrammarObj.DictationLoad vbNullString, SLOStatic
```

Answer the call

After the application receives a call notification, it will perform the following to handle the call:

1. Set the audio input and output to the right audio device (refer to the first section, Set up SAPI Audio Input/Output section) or streams.
2. Answer the call using `ITBasicCallControl.Answer`.
3. Prompt using `ISpeechVoice::Speak`. For example, "Welcome! Please select from the following two options: Leave a message or Check your messages."
4. Activate the recognition.
5. Process recognition results.
6. Disconnect the call.

Implementation for handling the answer call is straight forward now except for processing the recognition results. Snippet 3 will discuss in detail about how to use `ISpeechMemoryStream` to store the recognition results and extract them later on.

Snippet 4: Leave a message

After the caller selects the Leave a message option, the application creates an `ISpeechMemoryStream` object. The stream will be used to save the recognition results in recognition event handler.

```
'Reset the number of recognition results
NumOfResults = 0

'Cleanup the stream and create a new one
Set gMemStream = New SpMemoryStream

'Activate the recognition
RecoDictationGrammar.DictationSetState SGDSActive

'Wait for maximum 30 seconds to allow the caller to leave a
Dim Start
Start = Timer      ' Get start time.
Do While (Timer < Start + 30)
    DoEvents      ' Yield to other processes.
```

Loop

```
'Deactivate the recognition  
RecoDictationGrammar.DictationSetState SGDSInactive
```

Snippet 5: Handle the recognition event

ISpeechRecoResult.SaveToMemory and *ISpeechMemoryStream.Write* are used in the following example to save the entire current recognition result to the memory stream. In the meantime, the application, increases the number of recognition results which have been saved in the memory stream and records the length of each recognition result in bytes. These two variables will be used while retrieving the recognition information from the memory stream.

```
Private Sub RecoContextObj _Recognition (... , ByVal Result As ISpeechRecoResult)

    Select Case Result.PhraseInfo.GrammarId

        Case GID_DICTATION
            Dim SerializeResult as Variant

            'Save the entire recognition result to memory
            SerializeResult = Result.SaveToMemoryStream()

            'Write the result to the memory and store the length of the result
            MemoryStream.StreamLength (NumOfResults) = gMemStream.StreamLength + SerializeResult.Length

            'Record the number of recognition results having been saved
            NumOfResults = NumOfResults + 1

            'Additional speech processing code here

    End Select

End Sub
```

Snippet 6: Check the message

The following code snippet may be used when the caller

chooses the Check the message option. NumOfResults stores the number of recognition results in the file stream. If this variable is zero, then there is no message. Otherwise, the sample uses ISpeechMemoryStream.Read and ISpeechRecoResult.CreateResultFromMemory to restore each recognition result from the file stream and call ISpeechRecoResult.SpeakAudio to play the message in the original voice.

```
If (NumOfResults <> 0) Then
```

```
Dim resultGet As Variant, length As Long  
Dim RecoResultGet As ISpeechRecoResult
```

```
'Set the pointer to the beginning of the stream  
gMemStream.Seek 0, SSSPTRelativeToStart
```

```
'Speak using TTS voice  
VoiceObj.Speak "Your message is paused."
```

```
Dim i as Integer  
For i = 0 To NumOfResults - 1
```

```
'Extract data in bytes from the stream  
length = gMemStream.Read(resultGet, StreamLength (i))
```

```
'Restore the recognition results  
Set RecoResultGet = RecoContextObj.CreateResultFromMemor
```

```
'Speak the audio  
RecoResultGet.SpeakAudio
```

```
'Release the result object  
Set RecoResultGet = Nothing
```

```
Next i
```

```
'Speak using TTS voice  
VoiceObj.Speak "End of your messages"
```

```
Else  
'Speak using TTS voice
```

```
VoiceObj.Speak "You have no messages, good b
```

```
End If
```

Custom real time audio stream

Under some circumstances, you might want a custom real-time audio stream to read the audio data from one entity and write it to another. You can use this stream object in two ways: call using the phone system or call over the Internet (VoIP) from or to another computer. In addition, you can play the SAPI TTS voice and transcribe the message using SAPI SR functionality over the network. Before using the code snippets in this example, you need Windows 2000 Operating System or later installed on your system. The example needs TAPI 3.x which is installed with Windows 2000 OS.

The following is an example that builds a custom real time audio stream to send and receive audio data between SAPI and TAPI objects using media streaming terminals (MST) and other media controls provided by TAPI Media Service Providers (MSPs). Assume that STCustomStream is the name of the component of the custom audio stream containing two interfaces: ITTSStream and IASRStream. ITTSStream handles data exchanges between the TTS object and the media stream while IASRStream deals with data transition between the SR object and the media stream. The media stream interfaces used in ITTSStream and IASRStream are queried from the TAPI media streaming terminals. The terminals are created by the TAPI application. Using these two media streaming terminals with the aid of other media streaming interfaces, a TAPI applications should be able capture the audio data from the SAPI TTS engine and send it out to the remote caller side or render the audio data arriving from the remote end to the SAPI SR engine over the network.

Code samples are provided for the following topics: " TTS Custom stream " SR Custom stream

TTS custom stream

The TTS custom stream, (the aforementioned ITTSStream), is

used to capture the audio data from a TTS engine and inject the live audio data into a TAPI media stream. In this example, ITTSSStream inherits from ISpStreamFormat. This allows SAPI to eventually call ITTSSStream::Write() to feed the live audio data to the ITTSSStream object. The object then simply uses the media stream terminal to send out the audio data to the remote ends. The following are the sample code snippets illustrate ITTSSStream and its uses.

Snippet 7: ITTSSStream idl

```
interface ITTSSStream : IDispatch
{
[id(1), helpstring("method InitTTSSStream")]
HRESULT InitTTSSStream(IUnknown *pCaptureTerminal);
};
```

ITTSSStream::InitTTSSStream () initializes the IMediaStream object by querying from a capture terminal, pCaptureTerminal, pointing to an ITTerminal object. The method can also obtains the audio wav format using ITAMMediaFormat::get_MediaFormat() and store the format for later use.

Snippet 8: Use of the ITTSSStream in Visual Basic

When the call is connected, the TAPI application creates an MST for capture. The word "capture" is used in the DirectShow sense, and indicates the fact that MST captures an application's data to be introduced into the TAPI data stream.

```
Dim objTTSTerminal As ITTerminal
Dim MediaStreamTerminalClsid As String

MediaStreamTerminalClsid = "{E2F7AEF7-4971-11D1-A671-006097C

'Create a capture terminal
Set objTTSTerminal = objTerminalSupport.CreateTerminal( _
MediaStreamTerminalClsid, TAPIMEDIATYPE_AUDIO, TD_CAPTURE)
```

```
'Process here for selecting terminals, answering calls, etc.
```

```
'Set the output for SAPI TTS  
Dim CustomStream As New SpCustomStream  
Dim MyTTSSStream As New TTSSStream
```

```
'Initialize the TTSSStream object  
MyTTSSStream.InitTTSSStream objTTSTerminal
```

```
'Set MyTTSSStream as a BaseStream for the SAPI ISpeechCustomS  
Set CustomStream.BaseStream = MyTTSSStream
```

```
'Prevent the format change  
gObjVoice.AllowAudioOutputFormatChangesOnNextSet = False
```

```
'Set the CustomStream as an audio output  
Set gObjVoice.AudioOutputStream = CustomStream
```

```
'Release  
Set MyTTSSStream = Nothing  
Set CustomStream = Nothing
```

After your application receives the media event, CME_STREAM_

```
gObjVoice.Speak "Welcome!"
```

Snippet 9: Implementation for ITTSSStream methods

Listed below are the methods that must be implemented in the TTS custom stream. Currently in SAPI 5.1, other methods, such as CopyTo(), Commit(), etc., may return as E_NOTIMPL if those methods are not defined by the application.

```
// IStream interface
```

```
STDMETHODIMP Write(const void * pv, ULONG cb, ULONG  
STDMETHODIMP Seek(LARGE_INTEGER dlibMove, DWORD dwOr.
```

```
// ISpStreamFormat interface
```

```
STDMETHODIMP GetFormat(GUID * pFormatId, WAVEFORMATEX
```

SAPI calls the `ITTSStream::Write()` using the TTS engine whenever the audio data is ready. This method copies the data from the input buffer, `void *pv` to a `IStreamSample` buffer and then submits them to MST. SAPI calls `Seek()` to move the Seek pointer to a new location in the stream. SAPI calls `GetFormat` to locate the current stream format.

```
STDMETHODIMP CTTSSStream::Write(const void * pv, ULONG cb, ULONG
{
HRESULT hr = S_OK;

m_hCritSec.Lock();

ULONG lWritten = 0;
ULONG ulPos =0;
BYTE *pbData = (BYTE *)pv;

// Keep reading samples from void *pv and sending them on.
while ( SUCCEEDED ( hr ) )
{
    //Allocate a sample on the terminal's media stream. m_

    IStreamSample *pStreamSample = NULL;

    hr = m_cpTTSMediaStream->AllocateSample(0, &pStreamSampl

    // Check hr

    // Get IMemoryData interface from the sample
    IMemoryData *pSampleMemoryData = NULL;

    hr = pStreamSample->QueryInterface(IID_IMemoryData, (v

    // Check hr

    //Get the sample buffer information

    ULONG nBufferSize = 0;
```

```

    BYTE *pBuffer = NULL;

    hr = pSampleMemoryData->GetInfo(&nBufferSize;, &pBuffer

    // Check hr

    // Copy the audio data to the buffer provided by the s

    nBufferSize = min ( nBufferSize, cb - ulPos);
    memcpy ( pBuffer, (BYTE *) (pbData+ ulPos), nBufferSize

    ulPos += nBufferSize;

    // Tell the sample how many useful bytes are available
    hr = pSampleMemoryData->SetActual(nBufferSize);

    // Check hr

    pSampleMemoryData->Release();
    pSampleMemoryData = NULL;

    //Tell the MST that the sample is ready for processing
    hr = pStreamSample->Update(NULL, NULL, NULL, 0);

    //Break the while loop when the current data process c
    if ( FAILED(hr) || ulPos == cb )
    {
        pStreamSample->Release();
        pStreamSample = NULL;
        break;
    }
}

m_hCritSec.Unlock();
return hr;
};

STDMETHODIMP CTTStream::Seek(LARGE_INTEGER dlibMove,
DWORD dwOrigin,
ULARGE_INTEGER *plibNewPosition)
{
    // We only accept queries for the current stream position

```

```

        if (STREAM_SEEK_CUR != dwOrigin || dlibMove.QuadPart
        {
            return E_INVALIDARG;
        }

        // Validate the OUT parameter
        if (SPlsBadWritePtr(plibNewPosition, sizeof(ULARGE_I
        {
            return E_POINTER;
        }

        m_hCritSec.Lock();

        plibNewPosition->QuadPart = (LONG)dlibMove.LowPart;

m_hCritSec.Unlock();

        return S_OK;
    }

STDMETHODIMP CTTSSStream::GetFormat(GUID * pFmtId,
WAVEFORMATEX ** ppCoMemWaveFormatEx)
{
    m_hCritSec.Lock();

    HRESULT hr = S_OK;

    hr = m_StreamFormat.ParamValidateCopyTo( pFmtId, ppCo

    m_hCritSec.Unlock();

    return hr;
}

```

Notes

- SPlsBadWritePtr() and SPlsBadReadPtr() used in the above example are parameter checking help functions. They are defined in Spddkhlp.h in the SAPI SDK.
- Variable m_StreamFormat is declared as

CSpStreamFormat. It is defined in Sphelper.h.

- The m_hCritSec variable is defined as CComAutoCriticalSection.
- IStreamSample::Update() in the above Write() method performs a synchronous update of a sample. If you would like to update the samples asynchronously, you need to define a mechanism to keep track of all of the samples that having been submitted in order to ensure that these submitted samples are completely processed by the MST. For further information, please refer to tapisend, a TAPI 3.0 sample application, in the Microsoft Platform SDK.

SR custom stream

The SR custom stream, (the above-mentioned IASRStream), is used for rendering the audio data from a TAPI media stream to the SAPI SR object using MST. In this example, IASRStream inherits from ISpStreamFormat. This allows SAPI to eventually call IASRStream::Read() to retrieve the live audio data from the media stream. The following are sample code snippets about IASRStream and its uses.

Snippet 10: IASRStream idl

```
interface IASRStream : IDispatch
{
    [id(1), helpstring("method InitSRStream ")]
    HRESULT InitSRStream(IUnknown *pRenderTerminal);
    [id(2), helpstring("method StopRenderStream ")] HRESULT
};
```

IASRStream::InitSRStream () initializes the IMediaStream object by querying from pRenderTerminal, which points to an ITTerminal object. The method also obtains the audio wav format using ITAMMediaFormat::get_MediaFormat() and stores the format for later use. IASRStream::StopRenderStream () is used by TAPI applications to notify the IASRStream object to stop providing the audio data to SAPI during the read operation.

Snippet 11: Use of the IASRStream in Visual Basic

```
Dim objSRTerminal As ITTerminal
Dim MediaStreamTerminalClsid As String
MediaStreamTerminalClsid = "{E2F7AEF7-4971-11D1-A671-006097C

'Create a render terminal
Set objSRTerminal = objTerminalSupport.CreateTerminal( _
    MediaStreamTerminalClsid, TAPIMEDIATYPE_AUDIO, TI
```

```

'Process here for selecting terminals, answering calls, etc.

'Set input for SAPI SR
Dim CustomStream As New SpCustomStream
Dim MySRStream As New ASRStream

'Initialize the ASRStream object
MySRStream.InitSRStream objSRTerminal

'Set MySRStream as the BaseStream for the SAPI ISpeechCustom:
Set CustomStream.BaseStream = MySRStream

'Prevent the format change
gObjRecognizer.AllowAudioInputFormatChangesOnNextSet = False

'Set the CustomStream as an audio input
Set gObjRecognizer.AudioInputStream = CustomStream

'Release
Set CustomStream = Nothing

'Assume the RecoDictationGrammar, ISpeechRecoGrammar, is val.
RecoDictationGrammar.DictationSetState SGDSActive

'Wait here for a few seconds for recognition events

'Deactivate the dictation
RecoDictationGrammar.DictationSetState SGDSInactive

'Tell the ASRStream object to stop providing any audio data
MySRStream.StopRenderStream

'Release ASRStream object
Set MySRStream = Nothing

```

Snippet 12: Implementation for IASRStream methods

The following snippet lists the methods that must be implemented in the SR custom stream. Currently in SAPI 5., other methods like Commit(), CopyTo(), etc., may simply return

E_NOTIMPL.

```
// IStream interface
```

```
"          STDMETHODCALLTYPE Read(void * pv, ULONG cb, ULONG * pcbRea  
"          STDMETHODCALLTYPE Seek(LARGE_INTEGER dlibMove, DWORD dwOr
```

```
// ISpStreamFormat interface
```

```
"          STDMETHODCALLTYPE GetFormat(GUID * pFormatId, WAVEFORMATE
```

SR engines call `IASRStream::Read()` using SAPI. This method retrieves the audio data from a TAPI media stream and copies it to the buffer, pointed to by *void *pv*.

The following are sample code snippets for the `Read()` method. For the implementation of `Seek()` and `GetFormat()`, please refer to the TTS Custom Stream section.

```
STDMETHODIMP CASRStream::Read(void * pv, ULONG cb, ULONG *pcbRea  
{  
    m_hCritSec.Lock();  
    HRESULT hr = S_OK;  
  
    BYTE *pbData = (BYTE *)pv;  
  
    if (m_bPurgeFlag)  
    {  
// Add code here for Cleanup such as, release events, sample  
return hr;  
    }  
  
    //allocate the buffer  
    if ( m_pnDataBuffer == NULL )  
    {  
m_ulBufferSize = cb;  
m_pnDataBuffer = new BYTE [m_ulBufferSize ];  
    }  
    else if ( m_ulBufferSize != cb)  
    {  
//cb might be different from that in the previous Read()
```

```

delete []m_pnDataBuffer;
m_ulBufferSize = cb;
m_pnDataBuffer = new BYTE [ m_ulBufferSize ];
    }

    //Retrieve cb bytes audio data from the TAPI media s
    If ( SUCCEEDED ( hr ) )
    {
hr = RenderAudioStream();
    }

    if ( SUCCEEDED ( hr ) )
    {
        *pcbRead = m_ulActualRead;
        memcpy ( & pbData, (BYTE *)m_pnDataBuffer, m_ulAc
    }

    m_hCritSec.Unlock();

    return hr;
}

```

The RenderAudioStream() function extracts the audio data from the media streaming terminal to the buffer m_pnDataBuffer. The function first reads the terminal's allocator properties to perform the following:

- Obtains the number of samples.
- Calls IMediaStream::AllocateSample() on the terminal's IMediaStream interface to allocate an array for each stream sample.
- Creates an array of events and associates each sample with an event. An event is signaled when the corresponding sample is filled with data by the Media Streaming Terminal and is ready for use.
- Calls IStreamSample::CompletionStatus to ensure that the sample contains valid data and then copies data to buffer m_pnDataBuffer.

- Calls `IStreamSample::Update()` to return the sample to the terminal in order to be notified again when the sample refills with a new port of data.

For detailed information, please refer to "TAPI 3.0 TAPIRecv Media Streaming Terminal Sample Application" in the Microsoft Platform SDK.

Notes

- `m_pnDataBuffer` is a data member, pointing to `BYTE`. It stores the audio data received from the TAPI media stream.
- `m_ulActualRead` is a data member, containing the number of audio data in bytes stored in `m_pnDataBuffer`.
- `m_bPurgeFlag` is a data member. It gets set when the application calls `StopRenderStream()` of the `IASRStream` object.

Pitfalls: Common problems Encountered

The following are possible issues that developers might encounter during development:

Audio input/output devices

If your audio input or output source is not a standard windows Multimedia device, you need to create an audio object first and then call SAPI SetInput and SetOutput to the device (see the Set Audio Input and Output to an Audio section of this paper). Your application will not work if you simply select your wave In/Out device as the default audio input or output device using Speech properties in Control Panel.

Custom stream object

In your SR custom stream object, when the Read() method returns an error, SAPI will deactivate the recognizer state. In the case of telephony, you must explicitly set the recognizer state to active in every connection even through the recognizer state, by default, is set to active. If connections are not set to active, the Read() method might return E_ABORT or other error message after the caller disconnects the phone and the recognizer will be tuned off. This might cause troubles during the next calls.

After your application sets either the dictation or command and control grammar state to inactive, you may purge the stream by simply returning zero bytes in Read() to inform the SAPI that SR engine the end of stream has been reached. Otherwise, some SR engines might keep calling the Read() method, so this might cause your application to hang.



Audio Object

Overview

This document is intended to help developers write custom audio objects. Application developers can use this tool to direct speech data from memory into SAPI for speech recognition (SR) and for text-to-speech (TTS). The object does not generate or consume any audio data. Instead, it works as an audio buffer manager. For SR, audio data is passed to this object using a custom method `ISpAudioPlug::SetData`. SAPI retrieves the audio data from this object using `IStream::Read`. For TTS, audio data is passed from SAPI to this object using `IStream::Write` and the audio data can be retrieved calling a custom method `ISpAudioPlug::GetData`.

In order to use audio object for TTS output, the application uses asynchronous speak because the audio object does not consume the audio data. The application must consume the audio data by calling [*ISpAudioPlug.GetData*](#). If the application uses synchronous speak, SAPI blocks the client thread. SAPI's write call on the audio object will block if the internal queue does not allocate more space. If the application retrieves the audio data on a different thread, the problem is averted.

Interface description

SAPI ISpAudio

The SAPI audio object needs to implement the ISpAudio interface.

Custom interface ISpAudioPlug

ISpAudioPlug inherits from ISpeechAudio. It provides methods to send and retrieve audio data. The interface is automation compliant and can be used easily in languages that support automation.

```
ISpAudioPlug::Init(VARIANT_BOOL fWrite, SpeechAudioFormatType
```

This method is used to initialize the audio object's basic mode, including the read/write mode, as well as initialize the audio data format. If *fWrite* is TRUE, then the object is in write mode; if *fWrite* is FALSE the object is in read mode. FormatType specifies the audio format. By default, the object is in write mode and the format is set to SPSF_22kHz16BitMono. If the method is called while the object is processing audio data, SPERR_DEVICE_BUSY is returned.

```
ISpAudioPlug::SetData(VARIANT vData, long * pWritten)
```

SR uses this method when the object is set to read mode. The caller uses this method to send audio data so that SAPI can retrieve the audio data by Istream::Read.

```
ISpAudioPlug::GetData(VARIANT* vData)
```

TTS uses this method when the object is set to be write mode. The caller uses this method to retrieve audio data.

SAPI automation ISpeechAudio

The sample audio object provides an empty implementation of ISpeechAudio. In order to make the audio object usable in languages that support automation, SAPI requires that the object implement ISpeechAudio, which inherits from IDispatch. Internally, SAPI would not use ISpeechAudio directly. It uses QueryInterface on ISpAudio and calls the methods on ISpAudio. This way, the audio object only needs to provide an empty implementation of ISpeechAudio.

Buffer management

Internally, the audio object uses a queue object CBasicQueueByArray to manage the incoming and outgoing audio data. The queue internally uses an array to store data. When the data reaches the end of the array, it would move the head of the array to fill the data. The methods on the queue objects are thread safe.

State management

When SAPI starts an audio stream, the audio state changes to SPAS_RUN. When SAPI closes an audio stream, the audio state changes to SPAS_CLOSE. The audio object must perform the appropriate action according to the audio state. For example, when the audio state changes to SPAS_CLOSE, the audio object needs to free the audio buffer and signal other threads waiting for audio data.

Threading

Because the thread calling IStream::Read on the audio object is the same one that the SR engine uses to call SAPI, the client thread calling SetData/GetData is different from SAPI's IStream::Read thread. The audio object needs to be thread safe.

Event rerouting

The audio object implements ISpEventSink and ISpEventSource. SAPI forwards the SR/TTS events to the audio object. The audio object forwards SAPI the events with the audio position later than the current device position.



Using MFC to Automate SAPI

Introduction

The Microsoft Foundation Classes (MFC) provides an easy and convenient way to automate calls to SAPI using its Class Wizard to generate wrappers for the SAPI layer from the SAPI Type Library.

In order to accomplish this, perform the following steps:

1. Create a new MFCAppWizard(exe) project in Visual C++.
2. Based on the type of application you are creating, follow the wizard prompts. In Step 3 of the wizard prompts, (or Step 2 if you are creating a Dialog Based application) make sure that the Automation check box is selected under the heading, What other support would you like to include?

Once the new project is ready, access Class Wizard.

1. Click the Automation tab, and then click Add Class and select From a type library in the drop-down list.
2. Browse for the sapi.dll file and open it.
3. Select the classes you would like Class Wizard to generate a wrapper for. The resulting default header and implementation files are sapi.h and sapi.cpp respectively. The rest of this document assumes that you have chosen to use these default file names. Click OK.
4. You should now be back in the Class Wizard window. Click OK.

After you are done with the above steps, Visual C++ will automatically add the Class Wizard generated files sapi.cpp and sapi.h to your project.

Upon viewing the sapi.h file, you should notice that it is nothing more than an automation wrapper that has been generated for all the classes you selected. Notice that all the classes inherit from COleDispatchDriver, hence the dispatch interface needs to be set up. This only requires a few lines of simple code, after which the wrapper class can be used just like any other C++ class.

Example

This example assumes that you chose to generate a wrapper for the ISpeechVoice class from among any other classes you may have selected. Using the project created above, include the sapi.h file within a source file in the project that will make automation calls to SAPI using the wrapper. In that source file, type the following code.

```
CLSID CLSID_SpVoice;    // class ID for the SAPI SpVoice object
LPDISPATCH pDisp;    // dispatch interface for the class
ISpeechVoice voice;    // use the MFC Class Wizard generated wrapper

CLSIDFromProgID(L"SAPI.SpVoice", &CLSID_SpVoice);
voice.CreateDispatch(CLSID_SpVoice);
pDisp = voice.m_lpDispatch;

HRESULT hr = pDisp->QueryInterface(CLSID_SpVoice, (void**)&voice);

if (hr == S_OK) {
    pDisp->Release();
}
else {
    voice.AttachDispatch(pDisp, TRUE);
}

voice.Speak("hello world", 1);    // asynchronous call to Speak
```

If you have been following the steps outlined above properly, you should hear your computer say "hello world!" That's all there is to using MFC to make automation calls to SAPI. Currently however, not all the wrapper classes generated by MFC's Class Wizard work properly. For instance, the ISpeechLexicon interface does not work. The work around for this is to implement your own automation wrapper classes using C++. The steps to do that are beyond the scope of this document. Of course, you can always use the COM interfaces in C++ and Automation in Visual Basic to ensure that every

interface in SAPI works easily and flawlessly.

Further Reading

The following links are recommended to learn more about the IDispatch interface and automation.

- [MSDN Automation: Overview](#). This is a starting point for those new to automation programming and will outline fundamental concepts and procedures.
- [MSDN's Automation: IDispatch Interface](#). This is the interface used to expose objects for automation.