SMAPI Developer's Guide

IBM ViaVoice(tm) Software Developer's Kit
Version 1.7

Printed in the USA

Note: Before using this information and the product it supports, be sure to read the general information under Appendix A [Notices], page 151.

First Edition (December 1999)

Table of Contents

About This Document

This document provides detailed information on developing speech-aware applications using the IBM ViaVoice(tm) Software Developer's Kit (SDK) and the IBM Speech Manager API (SMAPI) interface set.

Who Should Read This Document

Read this document if you are a programmer interested in writing speech-aware applications that use ViaVoice for speech and are built using the IBM SMAPI interface set.

How This Document Is Organized

This document is divided into the following parts:

1. Introduction, describes speech-aware application programming using the ViaVoice SDK, discusses how to design applications that use ViaVoice.
2. Developing Command and Control Applications, explains how to develop command and control applications using the ViaVoice SDK. It covers identifying grammars, creating grammar files, compiling grammars, building a dictionary, and testing grammars. It also includes information on using dynamic command vocabularies as a method for implementing command and control in an application.
3. Developing Dictation Applications, explains how to develop dictation applications using the ViaVoice SDK. It presents basic functions that dictation applications should support. It also covers the steps involved in developing dictation macros and templates.
4. Speech Manager API (SMAPI), provides an overview of the Speech Manager APIs, describes function call processing, and explains the tasks associated with each phase of a speech-aware application session.
5. SMAPI Grammar Compiler API, provides an overview of the SMAPI Grammar Compiler APIs and explains the tasks involved in compiling grammars from within an application.
6. Custom Audio DLLs, provides an overview of the Custom Audio DLLs and explains the tasks involved with using multiple audio sources within an application.
7. Appendix A, Notices, describes copyright and trademark information for this publication.
8. Appendix B, Glossary, defines terms and abbreviations found in this publication.

Related Publications

Refer to the following publications included with this version for additional programming, reference, and design information:

- SMAPI Reference
- SAPI Reference (Windows only)

- ActiveX Developer's Guide (Windows only)

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

IBM ViaVoice Developer's Corner website at
ww.ibm.com/viavoice/dev_home.html

IBM ViaVoice SDK Web Channel at:
ww.software.ibm.com/viavoice/subscribe.html

OLE Automation Reference from Microsoft
(Windows only)

1 Introduction to SMAPI Developer's Guide

The IBM ViaVoice SDK for Windows 95/98, Windows NT and Linux provides programmers with the necessary tools to develop applications that incorporate speech. It includes a robust set of application programming interfaces (APIs) that allows an application to access speech resources. It contains several utilities that enable developers to define and manage what the user can say within an application. There are also several sample programs that can help programmers as they develop their applications for speech. Finally, there are distributable runtime elements that are included with an application that uses IBM ViaVoice.

1.1 IBM Native Architecture Overview

IBM SMAPI supports only speech recognition functions. The SMAPI interface set is the native interface for the ViaVoice engine. This section contains a description of the overall architecture of ViaVoice.

The heart of a speech recognition system is known as the speech recognition engine. The speech recognition engine recognizes speech input and translates it into text that an application understands. The application decides what to do with the recognized text. It can transcribe it literally for dictation, or it can act on it for commands.

Applications can access the speech recognition engine through a speech recognition API. For ViaVoice, this API is known as the Speech Manager API, or SMAPI, for short. SMAPI is a conventional API. This means that the API is defined as part of the resource; in this case, SMAPI is defined as part of the speech engine. With an API, speech becomes a resource to all applications, just like any system resource (mouse, video, and so on). Any of the ViaVoice SDK interfaces can be used in this manner, but SMAPI is the focus of the material in this guide.

Windows developers note: The SMAPI interface set cannot be used in conjunction with other speech interfaces, such as SAPI, from within the same source code.

1.1.1 Speech Resources

Before we can discuss the architecture of the speech recognition engine, we should first tell you about the various resources with which the speech recognition engine works. The speech recognition engine uses the following resources to process spoken words:

- User's language of origin
- Domains

1.1.1.1 User's Language of Origin

The language of origin is the language used by the speaker. ViaVoice on Windows supports U.S. English, six European, and three Asian languages. The supported languages and their associated language keys are:

Language	Language Key
U.S. English	En_US
U.K. English	En_UK
French	Fr_FR
German	Gr_GR
Italian	It_IT
Spanish	Es_ES
Arabic	Ar_AR
Japanese	Ja_JP
Chinese (Simplified)	Zh_CN
Chinese (Traditional)	Zh_TW

Please note:

Currently only U.S. English is supported on Unix systems.

The speech recognition engine is language-neutral and data-driven. Applications can easily be enabled to support multiple languages. In fact, multiple languages can be installed on one system, and ViaVoice allows the user to switch between them as needed.

1.1.1.2 Domains

Each language can include several different domains. A domain is a set of vocabularies, pronunciations, and word-usage models designed to support a specific application. The vocabularies, pronunciations, and word-use models are used together by the speech engine to decode speech for your application.

ViaVoice SDK, by default, runs with the general office domain in the language you selected when you installed the tools. This general office domain contains 64,000 to 230,000 words representative of the office environment. The number of words in the general office domain depends on the language. The general office domains that are available for the ViaVoice SDK are:

Language	Domain Name
U.S. English	cstartus
U.K. English	cssv#uk *
French	cssv#ft *
German	cssv#gr *
Italian	cssv#it *
Spanish	cssv#es *
Arabic	cssv1ar
Japanese	cstartjp
Chinese (Simplified)	ccsv1cn
Chinese (Traditional)	cssv1tw

* The number symbol (#) in the domain name represents a version number which will increase with each update.

For command and control applications, the ViaVoice SDK provides a set of tools for you to create the vocabularies specific to your application. Your application-specific vocabulary,

word-usage models, and pronunciations will be distributed along with your application. These application-specific objects, along with the predefined navigation domain supplied with ViaVoice, comprise the domain for your command and control application.

For dictation applications, the ViaVoice SDK provides a predefined general office domain, either isolated or continuous, for you to use when writing your application. For isolated dictation, other domains (such as Radiology, Journalism, Legal, and Emergency Medical) are also available. For continuous dictation, these other domains are not available currently. However, they will become available in the near future for use with the next product release.

1.1.2 Speech Engine Architecture

The speech engine has a rather complex task to handle, that of taking the raw audio input and translating it to recognized text that an application understands.

Please note:

The audio input source module encapsulates the methods used by the engine to retrieve the audio input stream. By default, the engine retrieves its audio input from the standard microphone input device in the system. A developer can write a custom audio library so that the input of the engine would be a custom piece of hardware. For more information on how to do this, see Chapter 16 [Overview of the Custom Audio Library], page 127.

The acoustic processor takes raw audio data and converts it to the appropriate format for use. The acoustic processor consists of two components: the signal processor and the labeler.

In the ViaVoice engine, audio input picked up by the microphone is analyzed by the signal processor. This raw audio data is captured at 22 kHz by default, but 11 kHz and 8 kHz sampling is also supported. It contains both speech data and background noise.

1.2 Application Programming Interfaces

ViaVoice SDK provides several programming interfaces that developers can use to incorporate speech into their applications. This guide describes the IBM SMAPI and Grammar Compiler API.

Speech Manager APIs (SMAPI)
: IBM speech recognition engine APIs.

SMAPI Grammar Compiler APIs
: APIs used to compile grammars used by the speech recognition engine.

1.2.1 SMAPI

There is significantly more function in the ViaVoice engine beyond raw recognition of spoken words, including dynamic vocabulary handling, database functions to query and select installed users, languages, and domains, and the ability to add new words to the user's vocabulary. SMAPI supports:

- Verifying the API version

- Establishing a database session to query system parameters (language, domain, user, etc.)
- Establishing a recognition session
- Setting up vocabularies
- Setting speech engine parameters
- Processing speech input
- Adding new words to the user's vocabulary
- Handling errors
- Disconnecting from the speech engine
- Closing a speech session

The SMAPI is provided as a library, which is linked into an application. The engine is a separate executable. The ViaVoice architecture supports many speech applications through a single engine, connected to one microphone. SMAPI was derived from earlier IBM speech products so that all functions used by IBM applications are available to the developer.

For more information about how to use SMAPI, refer to the following sections: See Chapter 9 [Overview of the C Language SMAPI], page 55 for an overview of SMAPI. See Chapter 10 [Function Call Processing], page 57 for information on coding SMAPI function calls. See Chapter 11 [Session Sharing], page 67 for information on coexisting with other speech enabled applications. See Chapter 12 [Parallel Session API Calls], page 87 for information on implementing multiple connections with the speech engine from one application. See Chapter 13 [Programming Tasks], page 89 for a description of various programming tasks. The individual API calls are documented in the IBM SMAPI Reference.

1.2.2 DMAPI

Please note: The DMAPI is only available on Windows

The ViaVoice SDK also provides a Dictation Macro API (DMAPI) to access dictation macros and templates. DMAPI supports:

- Initializing DMAPI
- Getting macro and template definitions
- Querying the current set of macros and templates
- Extracting information from macro actions and expansion text
- Updating the application's copy of the macro database
- Handling errors
- Closing DMAPI

DMAPI is provided as a DLL, which is linked into an application. For more information about how to use the DMAPI, refer to the DMAPI manual.

1.2.3 SMAPI Grammar Compiler API

The ViaVoice SDK provides the ability for developers to compile grammars programmatically. This function is known as the SMAPI Grammar Compiler API and supports:

- Specifying parameters to the grammar compiler
- Compiling a grammar
- Obtaining error messages from the grammar compiler

The SMAPI Grammar Compiler APIs are provided as a library, which is linked into an application. For more information on how to use the SMAPI Grammar Compiler APIs, see Chapter 14 [Overview of the SMAPI Grammar Compiler API], page 123. The individual SMAPI Grammar Compiler API calls are documented in the IBM SMAPI Reference.

2 Introduction to SMAPI Programming

This chapter introduces SMAPI application programming using the ViaVoice SDK. The process of developing an application that incorporates speech is different, depending on whether you're developing a command and control application, a dictation application, or an application that uses both. This chapter describes the processes for all three of these options.

2.1 Developing a Command and Control Application

1. Install the ViaVoice SDK.
2. Identify what the user can say to your application (the vocabulary).
3. Create a grammar file and/or define a dynamic command vocabulary to represent this vocabulary.
4. Compile the grammar.
5. Build a dictionary of pronunciations for your vocabulary.
6. Test your vocabulary.
7. Write the application interface.
8. Build a distributable runtime for your application.

"Developing Command and Control Applications" section covers this process in detail.

2.1.1 Identifying What the User Can Say

The first step to incorporating command and control in your application is deciding what the users will want to say. Are there different parts of the application where different things will be said? Are there things they will always want to say, regardless of where they are in the application? Do you want the users to be able to say the names of buttons and menu items? Do you want to support synonyms - more than one way of saying a command? Do you want to support a more natural way of saying something versus providing specific command sequences (for example, "Can I see my mail" versus "Mail-open")? These are just some of the considerations you should make when identifying what users will be able to say to your application.

Each collection of words and phrases that a user can say is called a vocabulary. ViaVoice allows you to have multiple vocabularies active at the same time. This helps you group and combine things the user will say to your application. Your vocabulary can be as restrictive or as flexible as your application needs to be. Of course, there is a trade-off of recognition speed and accuracy versus the size of the vocabulary. You may want to experiment with different vocabularies to validate a design that best matches the requirements and expectations of your users.

2.1.2 Creating a Vocabulary

You can specify the vocabulary in one of two ways: either as a structured grammar or as a dynamic command vocabulary. A grammar defines the syntax, or set of rules, for the words and phrases that a user can say. You use a plain text editor to create the grammar file. The grammar is specified using a specialized speech recognition control language, or SRCL. SRCL was developed as a joint effort between the SRAPI (Speech Recognition API) Committee and ECTF (Enterprise Computer Telephony Forum). SRCL is a high-level logic language that permits the use of repeated patterns and substitutable parameters to define a syntax and set of valid phrases. For example, a symbol <digit> could be defined as representing the words "zero" through "nine." Now, if a multiple-digit number is used elsewhere in the grammar, it can be defined using the previously defined symbol <digit>. That is, a three-digit number can be defined as <digit><digit><digit> rather than defining all 1000 possible combinations. For more information about SRCL and defining grammars, please see Chapter 4 [SMAPI Grammars], page 17.

Dynamic command vocabularies are lists of words and/or phrases that are defined at run-time. Dynamic vocabularies are quite useful when you don't know all of the words in the vocabulary at the time you're developing your grammar. For example, think of a telephone dialer application, where you have a list of names that are not known to you when you define your grammar. The names can be read from an address book at runtime, and be added dynamically to your vocabulary using the appropriate SMAPI calls.

The key difference between grammar vocabularies and dynamic command vocabularies is substitution and repetition operators. Grammar vocabularies support substitution, while dynamic vocabularies do not. This makes grammar vocabularies suitable for more complex vocabularies, and dynamic vocabularies suitable for simple voice commands.

2.1.3 Compiling the Grammar

The ViaVoice SDK SMAPI grammar compiler converts a grammar file defined in SRCL syntax into a binary file that can be used by the speech engine (FSG). At application run time, the speech engine uses the FSG file to determine the words and phrases that are currently available for the user to say. You can compile multiple grammars for the same application or share grammars across multiple applications.

For more information about using the SMAPI grammar compiler, see Chapter 14 [Overview of the SMAPI Grammar Compiler API], page 123.

2.1.4 Refining the Grammar

Please note: The grammar test tools are only available on Windows.

Three new command line tools are provided to assist you as you design, test, and debug grammars. These tools can be used in conjunction with the SMAPI Grammar Compiler, Dictionary Builder, and SMAPI Grammar Test Tool to verify the behavior of your grammar.

For more information about using the grammar test tools, refer to "SMAPI Grammar Development and Test Tools" in the Speech Developer's Tools Guide.

2.1.5 Building a Dictionary

Please note: The Dictionary Builder is only available on Windows.

Once you've created your grammar and/or dynamic vocabulary, you must ensure that ViaVoice's recognition engine will find pronunciations for every word used by your application. ViaVoice already comes with a variety of pronunciations; however, your vocabulary may contain words not yet stored in any of its pronunciation pools. For these words you must create a new baseform pool. This pool, together with all other pools present, is used by the recognition engine to tell it how the words it needs to recognize are pronounced.

Dictionary Builder provides for an integrated way to create this pronunciation pool by capitalizing on the reuse of existing baseforms and the ability to create new ones where needed. It reads and extracts the words from the vocabulary and lets you attach pronunciations. Dictionary Builder integrates various ways of creating such pronunciations, like retrieval from existing pools, on-line acoustic creation of new baseforms or the interface to off-line transcription programs. Through reuse and automatic baseform creation the amount of work required to come up with a set of pronunciations is kept to a minimum. When all pronunciations are in place, Dictionary Builder generates a new baseform pool for your application. The new pronunciation pool is distributed with your application. For information about creating and maintaining dictionary files, refer to "Dictionary Builder" in the Speech Developer's Tools Guide.

2.1.6 Testing the Vocabulary

Please note: The Grammar Test Tool is only available on Windows.

The ViaVoice SDK provides a SMAPI grammar test tool that allows you to test your compiled grammars and your dynamic command vocabularies. Using the SMAPI grammar test tool, you speak the words from your vocabulary into the microphone. If there are any words that aren't recognized, you will need to modify the grammar (BNF) file or the command vocabulary, the system values, or re-train the word using the train word function of the Dictionary Builder to resolve the problem. If you have more than one vocabulary, you may want to test the vocabularies individually first, and then combined. You should also test "out-of-vocabulary" phrases.

For more information about using the SMAPI grammar test tool to test your vocabularies, refer to "SMAPI Grammar Test Tool" in the Speech Developer's Tools Guide.

2.1.7 Writing the Application Interface

Using the SMAPI, your application can interact with the speech engine; for example, establishing a session with the speech engine or controlling when a grammar is activated. The SMAPI is designed for use with the C language, but any language that supports C Language calls can access the ViaVoice SDK library. While the calls are procedural C, the headers are compatible with C++.

The SMAPI calls needed for command and control applications include establishing a connection with the speech engine, directing the engine to start processing speech, setting up grammars and dynamic command vocabularies and activating them, processing recognized

speech, and disconnecting from the engine when you are done. Refer to "Speech Manager API (SMAPI)" section for more details on speech-aware programming using the SMAPI.

Individual SMAPI calls are documented in the ViaVoice SDK SMAPI Reference. SMAPI provides a robust set of APIs for interacting with the speech engine. However, you do not need to learn or use all of these APIs to develop a command and control application. There is a starter set of less than 20 APIs that you can use to take advantage of command and control in your application. Starter Set APIs in the ViaVoice SDK API Reference describes these APIs in more detail.

2.1.8 Building a Distributable Runtime for your Application

To distribute an application that uses ViaVoice for command and control, you will need to include your compiled grammar (FSG) files and, if you are developing on Windows, your dictionary file (PPL) along with your application. You must also include a redistributable version of ViaVoice with your application, unless you require the user have a version of ViaVoice 98 already on the client system.

The ViaVoice Run Time Kit includes tools for helping you prepare a distributable runtime version of ViaVoice for your application if you require it. You must comply with the Terms and Conditions of the IBM ViaVoice Run Time Kit Runtime License Agreement in order to redistribute runtimes. The IBM ViaVoice Run Time Kit Runtime License Agreement is available from your IBM sales representative.

2.2 Developing a Dictation Application

You will need to perform the following steps to develop a dictation application for ViaVoice:

1. Install the ViaVoice SDK.
2. Write the application interface.
3. Build a distributable runtime for your application.

For more details about this process see Chapter 7 [Developing Dictation Applications], page 45.

2.2.1 Writing the Application Interface

Your application will need to use the SMAPI to interact with the speech engine to establish a connection with the speech engine, to set up a dictation vocabulary, and to process recognized text. Since the goal is to produce completely correct text, you will need to handle error correction. This includes playing back what was spoken, displaying alternate words, and updating the vocabulary. The SMAPI is designed for use with the C language, but any language that supports C Language calls can access the ViaVoice SDK library.

The SMAPI calls needed for dictation applications include establishing a connection with the speech engine, directing the engine to start processing speech, setting up dictation vocabularies, processing speech input, playing back what was recognized, correcting misrecognized words, and disconnecting from the engine when you are done. See Chapter 9 [Overview of

the C Language SMAPI], page 55 for more details on speech-aware programming using the SMAPI.

Individual SMAPI calls are documented in the ViaVoice SDK API Reference. SMAPI provides a robust set of APIs for interacting with the speech engine. However, you do not need to learn or use all of these APIs to develop a dictation application. There is a starter set of around 25 APIs that you can use to take advantage of dictation in your application. "Starter Set APIs" in the ViaVoice SDK API Reference describes these APIs in more detail.

If you are a Windows developer and your application supports dictation macros and templates, you will need to use DMAPI. With these calls, you can initialize DMAPI, get the definitions of macros and templates, query the current set of macros and templates, and refresh your application's copy of the macro and template database. Refer to "Dictation Macro API (DMAPI)" in the Speech Developer's Tools Guide for more details on programming using DMAPI.

If you are a Windows developer and you need additional dictation macros or templates for your application, you create and export these macros and templates using the Dictation Macro Editor. This produces a macro (DCT) file, which your users then import using the Dictation Macro Editor. The Dictation Macro Editor is provided as part of the ViaVoice Distribution Package.

2.2.2 Building a Distributable Runtime for your Application

Dictation applications use the selected domain that is supplied with ViaVoice Run Time Kit. The ViaVoice Run Time Kit includes tools for helping you prepare a distributable runtime version of ViaVoice for your application if you require it. You do not need to include any special files with your dictation application that define the vocabulary or pronunciations for your application. On Windows if your application provides additional dictation macros or templates, you will need to include the DCT file for these macros and templates along with your application. Alternatively, you can require the user to have ViaVoice as a pre-requisite.

You must comply with the Terms and Conditions of the IBM ViaVoice Run Time Kit License Agreement in order to redistribute runtimes. The IBM ViaVoice Run Time Kit License Agreement is available from your IBM sales representative.

2.3 Developing an Application for Both Command and Control and Dictation

The process for developing an application that incorporates both command and control and dictation is not that different from the processes described previously. The primary consideration is the order in which you enable the command and control and dictation vocabularies for your application. For the command and control portions of your application, you will still need to define a grammar, compile it, build a dictionary of pronunciations for your grammar, and test it. You will also need to write the appropriate application interfaces to the engine to support the command and control features. For the dictation portions of your application, you will need to write the application interfaces to support dictation.

For more information about handling both command and control vocabularies and dictation vocabularies within the same application, see Section 13.2.1 [Setting Up Vocabularies], page 96.

If your application supports both command and control and dictation, you will need to distribute your compiled grammar (FSG) files and dictionary (PPL) file along with your application. If you are providing additional dictation macros and templates with your application, you will need to include your macro (DCT) file as well. You must include a redistributable version of ViaVoice with your application, unless a version of ViaVoice already exists on the client system.

The ViaVoice Run Time Kit includes tools for helping you prepare a distributable runtime version of ViaVoice for your application if you require it.

You must comply with the Terms and Conditions of the IBM ViaVoice Run Time Kit License Agreement in order to redistribute runtimes. The IBM ViaVoice Run Time Kit License Agreement is available from your IBM sales representative.

2.4 Speech Engine Runtime Limitations

For information about speech engine runtime maximum values and limitations, consult the SMLIMITS.H file in the \Include directory where the ViaVoice SDK is installed.

3 Dynamic Command Vocabularies

This chapter discusses the following topics:

- What is a dynamic command vocabulary?
- Why might you want to use one?
- Building pronunciations for a dynamic command vocabulary.
- Testing a dynamic command vocabulary.
- An example of a dynamic command vocabulary.

3.1 What is a Dynamic Command Vocabulary?

A dynamic command vocabulary is a vocabulary that is defined by your application at run time, rather than pre-compiled. A dynamic command vocabulary contains simple, enumerated words ("file") or phrases ("open the file"). It is defined through an API, not through predefined grammar files. Substitutions of words within a phrase are not allowed, although you can add words and phrases to the vocabulary and remove words and phrases from the vocabulary as needed.

In a dynamic command vocabulary, words and phrases are decoded entirely based on their pronunciations. That is, each word and phrase in a dynamic command vocabulary is equally likely to occur.

3.2 When to Use a Dynamic Vocabulary

Dynamic command vocabularies serve two major purposes: to provide simple command and control capability within applications, and to provide auxiliary vocabulary capability in conjunction with grammar and dictation vocabularies.

Dynamic command vocabularies afford a quick and easy way to incorporate basic speech recognition into your application. For example, you might want to allow your users to be able to speak the words and phrases on menus and buttons. To provide this capability, you can define these words and phrases within your application at run time. This is known as defining a dynamic command vocabulary.

Dynamic command vocabularies can be quite useful in this way, since you reap many of the benefits of speech recognition without the development overhead of defining and compiling grammars. Keep in mind, though, that dynamic command vocabularies support a "what-you-see-is-what-you-say" model (for example, if you defined "Open the mail" as a valid phrase, the user must say "Open the mail," not "Open my mail," "Show me my mail," or "Get my mail please").

Dynamic command vocabularies can be used in conjunction with grammar vocabularies. This comes in quite handy when you don't know all of the words in your vocabulary at the time you're defining your grammar. Consider a telephone dialer application, where you have a list of names that aren't known to you when you're developing your grammar. These names could be read from an address book at run time, and added dynamically to your vocabulary with the appropriate API calls.

Another particularly useful example for dynamic command vocabularies is in conjunction with dictation. Dynamic command vocabularies allow you to provide support for commands such as "stop dictation" during dictation. Both the dictation and the dynamic command vocabulary can be active simultaneously.

3.3 Building Pronunciations for a Dynamic Command Vocabulary

Please note: The Dictionary Builder tool is only available on Windows.

Even though the words and phrases in a dynamic vocabulary are defined within your application, you must have pronunciations available for these words and phrases. Use the Dictionary Builder to verify that pronunciations for the words and phrases in your vocabulary are present in one of the system dictionary (pronunciation pool) files. For more information about using the Dictionary Builder with dynamic command vocabularies, refer to "Dictionary Builder" in the Speech Developer's Tools Guide.

3.4 Testing a Dynamic Command Vocabulary

Please note: The Grammar Test Tool is only available on Windows.

You should also test your dynamic command vocabulary and its pronunciations. Use the SMAPI Grammar Test utility to verify that the words and phrases in your vocabulary, along with their associated pronunciations, can be recognized by ViaVoice. You can enter the words and phrases dynamically as you run the SMAPI Grammar Test utility, or you can provide a word list (WDL) file as input. For more information on using the SMAPI Grammar Test utility to test a dynamic command vocabulary, refer to "SMAPI Grammar Development and Test Tools" in the Speech Developer's Tools Guide.

4 SMAPI Grammars

4.1 What is a Grammar?

Before defining the term "grammar" it is necessary to define the term "utterance." For our purposes, we will define the term utterance to mean any stream of speech that represents a complete command. With this in mind, we will now define grammar as a structured collection of words and phrases bound together by rules that define the set of all utterances that can be recognized by the speech recognition engine at a given point in time.

4.2 Why is a Grammar Necessary?

Grammars are an extension of the single words or simple phrases supported by dynamic command vocabularies. Grammars support substitution and repetition, while dynamic vocabularies do not. As a result, grammars support vocabularies that are more flexible and more complex than do dynamic command vocabularies. Grammars formally define the set of allowable phrases that can be recognized by the speech engine, such as:

"show me my mail"

"do I have any mail from <name>"

"send a note to <name>"

These phrases can be spoken continuously, without pausing between words. The grammar provides a language model for the speech engine, constraining the valid set of words to be considered, increasing recognition accuracy while minimizing computational requirements.

The trade-off for speed and accuracy, however, is that the user of the system is also constrained to producing commands that stay within the grammar definition. Larger grammars can be more tolerant of user variation (for example, adding alternative commands such as "is there mail from <name>"), but at the potential expense of command latency and reduced accuracy.

There is also a trade-off with user expectations. Larger grammars might also set high user expectations, since it may be perceived that they can say "just about anything." Without real natural-language processing, the speech recognition system cannot anticipate all the different ways a user will structure a command. Also, you don't want to support so many commands that the user can't remember what to say.

Consideration of these inherent design trade-offs is fundamental to designing robust and efficient command grammars.

4.2.1 Acceptance or Rejection of Utterances

Since the user may say commands that are not in the vocabulary, speak to others with the microphone live, or be in an environment with background noise, the engine implements

accept/reject logic for grammars. Providing the logic in the engine simplifies application development, and makes performance consistent across applications.

Not all utterances are accepted by the recognition engine. Acceptance or rejection is flagged on each returned phrase. The engine will reject an utterance for these reasons:

- An incomplete path through the grammar is encountered, and the time-out is exceeded. This happens if the user stops talking before completing a phrase, or if the engine incorrectly goes down the wrong path. An application has control over this by setting the incomplete time-out, which effectively determines the tolerance for pausing within a grammar.
- The scores of the words are sufficiently low relative to the threshold setting. Again, the application can control this through an API to set the rejection threshold. In ViaVoice on Windows, this setting is available through Properties in Control Panel.

4.2.2 Handling Embedded Silence and Mumbles

Users may not always speak commands using exact continuous speech. For example, users might pause briefly while speaking commands, or they might interject extraneous speech into utterances (such as saying "open the...uh...file"). These occurrences are known as embedded silences and mumbles, respectively. They can be handled transparently by the engine, with optional notification to your application of where they occur in the word sequence.

You can request that the engine handle embedded silences and mumbles for your grammar. This information is passed back to your application. You can specify either or both of these options at run time when you define your grammar using the SmDefineGrammar call. You can also specify them when you compile your grammar. See Chapter 15 [SMAPI Grammar Compiler Programming Tasks], page 125 for more details. Note that any flags set at run time override those compiled into the FSG.

4.3 Introduction to SRCL Grammars

One way to represent a speech grammar is to use Backus-Naur Form, or BNF. This form of grammar representation is used to describe the syntax of a given language and its notation. These are generally understood and used throughout the world of formal language theory and processing.

This section describes the syntax of a particular type of BNF grammar that has been adapted to the task of speech recognition. It is called Speech Recognition Control Language (abbreviated SRCL and pronounced "circle"). SRCL was developed jointly between the SRAPI (Speech Recognition API) Committee and ECTF (Enterprise Computer Telephony Forum). IBM is a member of both organizations. Information on SRAPI can be found at http://www.srapi.com; likewise, information on ECTF can be found at http://www.ectf.org.

Grammars constructed using SRCL offer an organized view of the words and phrases that are part of the speech grammar, since they define a notation for identifying common phrases,

optional phrases, and repeated phrases. Throughout this section, the terms BNF and SRCL refer to the SRCL command language.

A speech grammar is defined by enumerating the valid words and phrases. The general form of a SRCL grammar is as follows:

```
<rule> = sentences and phrases.
```

This general form is called a production rule. Every SRCL grammar must start with and contain at least one production rule. The production rule has four parts:

- The left side (<rule>): This part of the production rule is required, although it is not necessary to specify the string exactly as we have done. In general, any string may be used as long as it is placed between angle brackets <>, and the string formed by the angle brackets and the enclosed word is unique (that is, it is not used elsewhere in the BNF). Hence, <rule> must not appear elsewhere in the grammar.
- The assignment operator (=): This part of the production rule is required and must be written as shown.
- The right side (sentences and phrases): This part of the production rule defines all of the sentences and phrases that are valid in the speech grammar. The exact structure of these definitions is covered in the remaining parts of this section.
- The end-of-production delimiter (period): This part of the production rule is required and must be specified as shown. Although it is not necessary to separate the period from the sentences-and-phrases section of the rule, doing so often improves readability.

From looking at this definition, you might have noticed that the structure of the production rule is much more complex than it need be for a language that is capable of having only one such rule. In fact, a SRCL grammar can have an arbitrary number of such rules. Later in this chapter, you will see how to define these grammars and how to relate their rules such that the entire speech grammar fits together into a single sophisticated speech grammar. For now, however, we will confine ourselves to those that contain only one rule.

The following is an example SRCL BNF grammar that defines menu selections for a windowed application. This simple example is shown for educational purposes. If this was all you wanted to do, it would be much easier to define a dynamic command vocabulary and skip the compilation altogether.

```
<root> =  FILE
        | EDIT
        | OPTIONS
        | HELP.
```

Here is a slightly more complex example, involving two phrases:

```
<root> =  hello world
        | hello there.
```

The only symbol that we have not yet covered in these two examples is the vertical bar (|). This operator is used to indicate mutually exclusive choices. Hence, in the first example we can say any of the words listed. In the second example, we can say either "hello world" or "hello there" just as in the original examples. Notice that we cannot say just the single words "hello" or "world" or "there." In this case, the choices are for phrases, not just for individual words.

4.3.1 Defining Common Words and Phrases with Nonterminal Symbols

The sentences that you use in everyday life are composed of many hundreds of common phrases and words. A SRCL grammar gives you a notation to define these common elements and to use them throughout your speech grammars. In so doing, it allows you to create sophisticated grammars out of a few sentence forms with many common words and phrases.

To see how the SRCL grammar syntax supports this, let's return to our two-phrase grammar above:

```
<root> =  hello world
        | hello there.
```

In the preceding discussions, the grammars have contained only terminal symbols. In our two-phrase example, the words "hello," "world," and "there" are examples of terminal symbols. Although there are numerous definitions of the term "terminal symbol" in the linguistic and programming community, when we use the term in describing a SRCL grammar, we mean words to be spoken. All of the grammars, so far, have contained only the words to be spoken arranged into collections of individual words, phrases, or both, separated by a vertical bar or "OR" symbol. However, everywhere that a terminal symbol can be used, it is possible to use a nonterminal symbol instead. To define a nonterminal symbol, you create a production rule of the form just described. In the preceding example, we could define a nonterminal symbol as:

```
<root> =    hello world
          | hello there .
<object> =  world
          | there .
```

Now, we have defined a grammar with two nonterminal symbols, <root> and <object>. The nonterminal symbol <root> is the special starting nonterminal and must be unique as previously outlined. The nonterminal symbol <object> is a simple nonterminal. Like the starting nonterminal symbol, it is a string of alphanumeric characters enclosed in angle brackets. In order to use a nonterminal symbol, place it somewhere in the right side of the production rule, either as a stand-alone word or in a phrase. Anywhere a terminal symbol can be used, a nonterminal symbol can also be used.

One way to use the nonterminal symbol is:

```
<root> =    hello <object>
          | hello <object> .
<object> =  world
          | there .
```

Notice that by using the nonterminal symbol <object> we have created a grammar with two identical clauses. Hence, we could rewrite the grammar without the redundant clause.

```
<root> =    hello <object> .
<object> =  world
          | there .
```

Although this example is intentionally simple to allow each point to be explained easily, the use of nonterminal symbols simplifies the writing and maintenance of production grammars that might be both large and complex.

4.3.2 Defining Optional Words and Phrases

It is often desirable to define sentences that contain optional phrases. Many of these are in the form of an imperative preceded by a noun or pronoun of address. For example the command "May I see your passport?" might well be optionally preceded by "Sir," "Madam," or "Miss." One way to define this construct is:

```
<root>   =   <command>
           | <title> <command> .
<title>   = sir | madam | miss .
<command> = may I see your passport .
```

Although this example is a perfectly reasonable way to define such constructs, they are so common that SRCL provides the ? operator to define them. The above example is rewritten below using the ? operator.

```
<root>  =   <title>? <command> .
<title>   = sir | madam | miss .
<command> = may I see your passport .
```

When the ? operator appears it causes the symbol to its immediate left to be defined as optional. If the symbol is a terminal symbol the operator causes it to be defined as an optional word. If the symbol is a nonterminal symbol the operator causes all of the clauses defined by that nonterminal symbol to be treated as optional. The ? operator cannot be used on the left side of a production rule.

4.3.3 Defining Repeated Word and Phrases

Another commonly seen construct in speech grammars is that of the repeated word or phrase. These are quite common in grammars that define arbitrarily long sequences of natural numbers. Phone numbers, credit card numbers, account numbers, and serial numbers fall into this category. As in the preceding example, SRCL supports this special case with a special purpose operator. We'll use a digit string grammar as an example. This grammar will cause the speech engine to recognize and accept as valid any number of digits zero through nine in any order:

```
<numbers>   = <digit>+ .
  <digit>   =  zero
            | one
            | two
            | three
            | four
            | five
            | six
            | seven
            | eight
            | nine .
```

When the + operator appears, it causes the symbol to its immediate left to be defined as one or more of. Hence, in our example, the <numbers> production rule is read: "Numbers is defined as one or more digit." The * operator is analogous to the + operator, except that it defines zero or more of the symbols to its left. Grammars should be written with the

supported ?, +, and * operators, rather than with recursive rules. The use of recursion can often result in larger and less efficient grammars than those written using the operators, and these grammars can often produce unintended, yet legal, word sequences. The grammar compiler, therefore, does not support recursive grammars.

4.3.4 Grammar Annotations–A Post Parsing Aid

The SRCL command language contains, in addition to the features that support defining a command grammar, a feature that reduces the complexity of parsing command grammar sentences. This feature, called grammar annotations or simply annotations, accomplishes its goal by allowing key words and phrases to be marked or annotated when the grammar is defined. Then, when the grammar is used in an application, and whenever an annotated word is recognized, both the word and the annotation are returned to the application. By choosing the annotations properly, you can simplify most parsing of command grammar sentences. This also simplifies development of grammars for multiple languages. Careful use of annotations can make the parsing independent of word order in a command, which will vary from language to language (for example, "start <program>" in U.S. English versus "<program> start" in German).

4.3.4.1 Defining Annotations

Here are some general rules for defining annotations and some general suggestions to help you get started. In the following BNF fragment we see the two major ways to define an annotation:

```
... tuesday:"Day_of_week"
... february:2
```

The first method allows a string to be used as an annotation. To use this form of annotation, place a colon after the symbol to be annotated and follow it with a quoted string that contains the data. The second method allows any decimal value in unsigned long to be used as tags. To use this form, place a colon after the symbol to be annotated and follow it with the number. Aside from their syntax, keep the following rules and suggestions in mind when using annotations:

- Annotations can be attached to any terminal symbol.
- Annotations cannot be attached to a nonterminal symbol.
- Only one annotation per symbol is permitted.

4.4 The Kiosk Example

To illustrate the use of annotations, we will use a kiosk speech grammar. This is the type of grammar that you might find in an information kiosk at either the Olympics or an international pavilion in a major airport. The purpose of such a kiosk would be to provide automated assistance to visitors on a variety of topics. Its SRCL grammar is as follows:

```
<kiosk> =  <greeting1>? <greeting2>? <sentence1>
                  | <greeting1>? <sentence2> .
<greeting1> = hello | excuse me | excuse me but .
<greeting2> =  can you tell me
                  | I need to know
                  | please tell me .
<sentence1> =  where <destination1> is located
                  | where is <destination1>
                  | where am I
                  | when will <transportation> <destination2>? arrive
                  | when <transportation> <destination2>? will arrive
                  | what time it is
                  | the local time
                  | the phone number of <destination1>
                  | the cost of <transportation> <destination2>? .
<sentence2> =  I am lost
                  | I need help
                  | please help me
                  | help
                  | help me
                  | help me please .
    <destination1> =  a restaurant
                   | the <RestaurantType> restaurant
                   | <BusinessType>? <BusinessName> .
    <RestaurantType> = best | nearest | cheapest | fastest .
    <BusinessType> = a | the nearest .
    <BusinessName> =   filling station
                   | public rest room
                   | police station .
    <transportation> = the <TransportType>? <TransportName> .
    <TransportType>  = next | first |last .
    <TransportName>  = bus | train .
    <destination2> =   to metro central
                   | to union station
                   | to downtown
                   | to national airport .
```

Take a minute to examine this grammar and try to imagine how you might design an application to process its requests. In case you are wondering, there are a possible 2664 sentences. Let's list a few of them:

hello I need to know where the nearest police station is located

where is the nearest public rest room

please tell me the phone number of the nearest filling station

where am I

when will the next bus to union station arrive

excuse me but when will the next train to national airport arrive

help

I am lost

excuse me what time is it

This small grammar has the beginnings of a reasonable amount of flexibility. The sentences generated by this grammar could also present something of a challenge to an application parser. From a language processing perspective, the key to designing an efficient program is to recognize that of the 2664 possible sentences, only six different categories of information are requested:

1. A nonspecific request for assistance
2. A request for the local time of day
3. A request to locate a given landmark
4. A request for the phone number for a given business or facility
5. A request for information for public transportation
6. A request for fare information for public transportation

Providing such information, in and of itself, is not a difficult task, provided that the requests for it are straightforward. The difficulty lies in the fact that the information is being requested through a near-natural language interface. That interface, in order to be desirable and useful to speakers, must allow them to request information in a "natural" way. Hence, the grammar must anticipate many of the common ways that users might pose their questions and requests.

Fortunately, there is a way to almost totally eliminate the job of parsing. By attaching annotations to key phrases it is possible, through a relatively simple set of post-processing steps that involve only simple string manipulation, to quickly reduce the 2664 sentences to the six forms in the list and to place those requests in an almost template-like form that will be easy to process.

To illustrate how this is done, we will first rewrite the grammar to include the annotations.

```
<kiosk> =  <greeting1>? <greeting2>? <sentence1>
             | <greeting1>? <sentence2> .
<greeting1> = hello | excuse me | excuse me but .
<greeting2> =  can you tell me
             | I need to know
             | please tell me .
<sentence1> =  where:"op_locate" <destination1> is located
             | where:"op_locate" is <destination1>
             | where:"op_locate" am I
             | when:"op_sched" will <transportation> <destination2>? arrive
             | when:"op_sched" <transportation> <destination2>? will arrive
             | what:"op_time_of_day" time it is
             | the local:"op_time_of_day" time
             | the phone:"op_phone" number of <destination1>
             | the cost:"op_fare" of <transportation> <destination2>?
<sentence2> =  I:"op_help" am lost
             | I:"op_help" need help
             | please:"op_help" help me
             | help:"op_help"
             | help:"op_help" me
             | help:"op_help" me please .
<destination1> =  a restaurant:"landmark"
                | the <RestaurantType> restaurant:"landmark"
                | <BusinessType>? <BusinessName> .
<RestaurantType> = best:"type"
                | nearest:"type"
                | cheapest:"type"
                | fastest:"type" .
<BusinessType> = a:"type" | the nearest:"type" .
<BusinessName> =   filling:"landmark" station:"landmark"
                 | public:"landmark" rest:"landmark" room:"landmark"
                 | police:"landmark" station:"landmark" .
<transportation> = the <TransportType>? <TransportName> .
<TransportType>  = next:"type" | first:"type" |last:"type" .
<TransportName>  = bus:"transport" | train:"transport" .
<destination2> =   to metro:"dest" central:"dest"
                 | to union:"dest" station:"dest"
                 | to downtown:"dest"
                 | to national:"dest" airport:"dest" .
```

After adding the annotations, the sample sentences generated by our grammar look as follows:

Please note: These sentence forms are conceptual. It will take programming logic to convert the real output to this form.

1. hello I need to know where."op_locate" the nearest."type" police."landmark" station."landmark" is located
2. hello excuse me where."op_locate" is the nearest."type" landmark" rest."landmark" room."landmark"

3. please tell me the phone."op_phone" number of the nearest."type" filling."landmark" station."landmark"
4. where."op_locate" am I
5. when."op_sched" will the next."type" bus."transport" to union."dest" station."dest" arrive
6. excuse me but when."op_sched" will the next."type" train."transport" to national."dest" airport."dest" arrive
7. help."op_help"
8. I am lost."op_help"
9. excuse me what time is it."op_time_of_day"

Please note: Words and annotations are returned in parallel structures through SMAPI. As such, they do not require additional tokenization or parsing, as in the previous examples.

With this set of sentences as input, the task of writing a parser becomes straightforward. No longer is it a job of parsing, it is one of separating the annotated parts of the sentence, which contain the information required by the application, from the non-annotated parts, and placing them into a standard form for processing.

4.5 Dynamic Command Vocabularies

A dynamic command vocabulary is a set of words, phrases, or both, defined at run time. The list might be created based on the user's response or on the current state of the application. Take a phone dialer application, for example. When you're defining the grammar, you do not know what names the user might want to call. So, at run time, your application reads these names in from an address book as a dynamic command vocabulary. Now the names will be available for the user to speak, along with the rest of the words and phrases in your grammar.

Dynamic command vocabularies are specified as external lists in the grammar in the following manner:

```
EXTERN <list-name>
```

The dynamic command vocabulary must have the same name as the external non-terminal symbol used in the grammar. For more information about how to define an external vocabulary, refer to "Creating a Dynamic Vocabulary or External List" in the Speech Developer's Tools Guide.

Please note: In the current definition of SRCL, external lists have the following limitations:

- Annotations cannot be attached to an external list (since it is a nonterminal)
- Annotations cannot be attached to individual items in an external list, even though these are terminals

4.6 Guidelines for Designing SMAPI Grammars

The following are suggestions for designing SMAPI Grammars:

- Use grammars with few phrases and a simple syntax to permit a faster and more accurate speech recognition process.
- Limit the size of the grammar so that fewer words must be matched. Note that a grammar cannot contain more than 65,000 words.
- Develop long and narrow grammars, rather than short and wide grammars, to limit the number of legal possibilities at any given point in the grammar.
- Avoid using words that are very similar in pronunciation in the same portion of your grammar. For example, if you are writing a child's game, you would not want to define a grammar "pick up the hat" and "pick up the cat." Instead, you could specify the grammars as "pick up the blue hat" and "pick up the yellow cat."
- Allowing the user multiple ways of saying a particular command can enhance usability. But, large grammars might also set user expectations unrealistically high; that is, users might perceive that they can say almost anything, which, of course, is not true. Recognition accuracy and speed can deteriorate as users speak more randomly based on their expectations.
- Don't support so many ways of saying commands that your users won't remember what to say. You should probably test your grammars with a number of potential users, and support the phrases that are most intuitive and meaningful to them. You might also want to provide a visual mechanism for reminding users what they can say at any given point.

4.7 SRCL Syntax

This section sets forth the syntax for the Speech Recognition Command Language (SRCL).

SRCL is a language that is easy to implement and learn, with a minimum set of constructs to implement a reasonable capability. We have avoided providing multiple ways to accomplish the same thing. This has resulted in a small, consistent language definition that is easy to learn.

The language elements addressed in the syntax are:

- Comment formats
- Terminals
- Nonterminals
- Grammar rules (productions)
- External lists
- Include declarations

These aspects are discussed in the following sections.

4.7.1 Language Definition

The following are general language considerations:

- Syntactically significant characters such as ", =, <, >, and . can be used to represent words and annotations if they are preceded by a \ escape character.
- Blanks can be used within a quoted string or a quoted annotation. The characters " and \ may be included within quoted strings and nonterminals if they are preceded by a \ escape character (that is, \" and \\, respectively).
- The \ escape character may be used in a quoted terminal that is continued on the next line, with white space on the next line being significant.
- The parser provides support for source level include files using the syntax:

  ```
  INCLUDE "filespec"
  ```

- White space is not significant except within quoted terminal and variable definitions and to separate unquoted words.
- SRCL is a case-sensitive language and its string-handling syntax is modeled after the C language.

4.7.2 Language Elements

4.7.2.1 Comment Formats

SCRL accepts both the double-slash "//" and the semicolon ";" as comment markers, which occur at the beginning of a line or within the line. All characters up to the next new line are then taken as comment characters.

4.7.2.2 Terminals

For terminology we use "variable" and nonterminal as well as "word" and terminal for the basic elements of the language in the following discussions. Terminals can be single spoken units (such as "San Francisco") that are optionally enclosed in quotation marks.

This syntax implies that a production such as:

```
<var>=Name that Tune.
or
<var>="Name" "that" "Tune".
```

uses three separate words for name, that, and tune, and they are pronounced as "n ey m", "th ae td", and "t uw n".

A production such as:

```
<var>="Name that Tune".
```

uses a single phrase for the three words that are pronounced "n ey m th ae td uw n" and processed as a single word.

Please note:

Productions like "Name that Tune" should be used with caution. The primary intent of "quoted phrases" in grammars is to override the space-delimited tokenization of spoken words that include blanks, such as "West Palm Beach" or "Van Dyke." You can use

"quoted phrases," but understand that the user won't be able to pause between the words in the phrase, nor will the user be able to mumble between the words in the phrase. Also, "name" "that" "tune" will be returned as a recognized phrase message and word structures, potentially with three different annotations. "Name that Tune" will be returned as one SM_WORD structure, with only one annotation.

4.7.2.3 Nonterminals

The definition of a nonterminal consists of a mandatory set of angle brackets enclosing an unquoted word identifier, with case being significant. Nonterminals should not cross line boundaries. Syntactically significant characters may be used in nonterminals without the \ escape character.

Examples of distinct nonterminals include:

```
<nonterminal>
<Nonterminal>
<this_is_a_nonterminal>
```

4.7.2.4 Grammar Rules (Productions)

The grammar rule format is given as follows:

1. A simple and consistent format for variables (nonterminals) of the form <id>, where the "<" and ">" are mandatory, and id is in the format of an unquoted word identifier.
2. Tokens, or words, may be represented as identifier strings without the need for quotation marks. It is sometimes necessary to delimit the words with quotation marks to include imbedded blanks.
3. The "=" symbol to separate the left-hand side (lhs), and right-hand side (rhs) clauses of grammar rules, rather than the classic BNF notation of ::=.
4. A repetition operator set to include:
   - + for one or more occurrences of the preceding word or variable.
   - * for zero or more occurrences of the preceding word or variable
   - ? for zero or one occurrences of the preceding word or variable

   Any repetition operator can follow either a word identifier or a variable identifier used on the right-hand side of a rule. Repetition operators are not supported on the left-hand side of the rules.
5. Annotations that can be represented as character strings in the form of word identifiers, which would include decimal numbers or quoted strings like "this is an annotation" or "this is another annotation." The form of the annotation is given as:

   ```
   word identifier:annotation identifier
   ```
6. Variables can be defined only once. Multiple rules written with the same left-hand side (lhs) cause an error.
7. The root production carries a special designator in that the left-hand side uses a double bracket construct as follows:

```
<<root_name>>= a | <b> | c | ... .
```

The root production does not need to be the first production in a grammar file. Also, you do not need to have a <<...>> root production. In this case, the first nonterminal becomes the root production.

8. Each grammar rule must be terminated with a period ".".

An example grammar rule might be as follows:

```
<transaction>=Buy <amount> "of" <companies> "at market".
```

Please note: There must be a dictionary entry that provides a pronunciation for "at market" or, for example, "Amalgamated Rhubarb," if that were a word defined in <companies>.

4.7.2.5 External Lists

External lists are indicated with the following syntax:

```
EXTERN <Variable> [, <variable 1> , ... <variable n>]
```

An external list is a list of words or phrases defined at run time as a dynamic vocabulary, with the same name as the external nonterminal. Currently, they must be defined with SmDefineVocab before the grammar is defined with SmDefineGrammar, but you can add and remove items from the word list at any point using SmAddToVocab and SmRemoveFromVocab.

Please note: External file names are case sensitive.

4.7.2.6 Include Declarations

External source files are included into grammar files with a directive having the following syntax:

```
INCLUDE "filespec"
```

Please note: SRCL supports letters and digits and all national language characters in code page 1252. SRCL also supports the following special characters:

~	-
‘	+
!	=
@	{
#	}
$	[
%	]
&	|
*	\
(	,
)	.

4.8 Using the Grammar Translation Facility

The translation facility enables the application developer to associate a translation with any expression in the Extended BNF grammar. The grammar writer associates a translation with a BNF expression by writing the symbol "->" followed by the translation after the expression.

Consider for example the following rule:

```
<<start>> = my dog has fleas -> scratch.
```

This rule defines a grammar that accepts exactly one phrase, "my dog has fleas", and produces the string "scratch" as a translation. A translation may be given any place where a sequence of expressions is specified, including each alternative in a list of alternatives. Consider for example the following rule:

```
<<start>> = my dog has fleas -> scratch
          | my dog is happy -> wag.
```

This rule defines a grammar that accepts two phrases, each with a different translation ("scratch" and "wag" respectively).

In addition to literal strings, as in the preceding examples, the translation of a sequence of BNF expressions may include the translations of the constituent expressions. This is done by marking with braces "" the expressions in the sequence of expressions whose translations are to contribute to the translation of the whole sequence, and referring in the translation to the nth marked expression by using the string "n". So for example the following grammar

```
<pet> = dog | cat | fish.
<<start>> = my <pet> has fleas -> 1 scratches
          | my <pet> is happy -> 1 wags.
```

defines a grammar that accepts six phrases; the input "my dog has fleas" produces the translation "dog scratches", while the input "my fish is happy" produces the translation "fish wags", and so on. The same grammar can be expressed more succinctly as:

```
<pet> = dog | cat | fish.
<state> = has fleas -> scratches | is happy -> wags.
<<start>> = my <pet> <state> -> 1 2.
```

4.8.1 Working with the Details

Every expression in the Extended BNF has a translation. Consider the following example:

```
<digit> = one -> 1 | two -> 2.
<<number>> = <digit> (hundred -> 00).
```

This grammar accepts the phrases "one hundred" and "two hundred" and produces the translations "1 00" and "2 00", respectively, as per the following rules:

- The translation of a terminal is the terminal itself, although this may be overridden by treating any given occurrence of the terminal as a sequence of length one and specifying a translation using the "->" notation, as in "one -> 1" and "two -> 2" above. If necessary, the expression can be enclosed in parentheses as in "(hundred -> 00)" above.

- The translation of a list of alternatives is the particular alternative actually used in parsing the input. Thus the translation of the expression "one -> 1 | two -> 2" in the example above is "1" or "2" according to whether the first or second alternative is used in parsing the input.

- The translation of a non-terminal is the translation of the expression on the right-hand side of the BNF statement defining the non-terminal. Thus in the example above the translation of <digit> is the translation of "one -> 1 | two -> 2".

- Finally, the default translation of a sequence of expressions is the concatenation of the translations of the expressions in the sequence, separated by spaces. Thus the translation of <<number>> is either "1 00" or "2 00". This of course may be overridden by explicitly specifying a translation as described previously. For example, if the definition of <<number>> is changed as follows:

```
<digit> = one -> 1 | two -> 2.
<<number>> = <digit> hundred -> 00 -> 12.
```

  then the translation of <<number>> becomes "100" or "200", because an explicit translation "12" is given which does not include a space.

4.8.2 Example

The following example is taken from a simple travel expense reporting application. The first part of the example is a grammar for English numbers up to six digits. Each of the rules <n-digit> specifies all way of saying numbers up to n digits, and produces as a translation exactly n digits, using leading zeros if necessary. It is assumed that the application will strip the leading zeros if desired.

```
<1-digit> = oh -> 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .
<2-digit> = oh? <1-digit> -> 01 |
 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |
 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 |
 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 |
 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 |
 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 .

<3-digit>
= <1-digit> (hundred and?)? <2-digit> -> 12
| <1-digit> hundred                    -> 100
| <2-digit>                            -> 01
.

<6-digit>
= <3-digit> thousand <3-digit>        -> 12
| <3-digit> thousand and <2-digit>    -> 102
| <3-digit> thousand                    -> 1000
| <2-digit> (hundred and?)? <2-digit> -> 0012
| <2-digit> hundred                     -> 00100
| <2-digit>                            -> 00001
.

<number> = <6-digit>.
```

The following rule uses the number rules given above to specify amounts of U.S. currency.

```
<money>
= <number>
| <number> dollars and? <2-digit> cents? | "" -> 00 -> (1.2)
| <number> point? <2-digit>                          -> (1.2)
.
```

The following rules accept English dates and produce a translation in the form yy/mm/dd. For purposes of illustration, the grammar accepts only a limited number of years, and supplies 1996 as a default year.

```
<day> =
  1st -> 01 |  2nd -> 02 |  3rd -> 03 |  4th -> 04 |  5th -> 05 |
  6th -> 06 |  7th -> 07 |  8th -> 08 |  9th -> 09 | 10th -> 10 |
 11th -> 11 | 12th -> 12 | 13th -> 13 | 14th -> 14 | 15th -> 15 |
 16th -> 16 | 17th -> 17 | 18th -> 18 | 19th -> 19 | 20th -> 20 |
 21st -> 21 | 22nd -> 22 | 23rd -> 23 | 24th -> 24 | 25th -> 25 |
 26th -> 26 | 27th -> 27 | 28th -> 28 | 29th -> 29 | 30th -> 30 |
 31st -> 31.

<month> =
 January   -> 01 | February -> 02 | March    -> 03 | April    -> 04 |
 May       -> 05 | June     -> 06 | July     -> 07 | August   -> 08 |
 September -> 09 | October  -> 10 | November -> 11 | December -> 12.

<year> = 1994 -> 94 | 1995 -> 95 | 1996 -> 96 | 1997 -> 97.

<date>
= <month> <day>                  -> 96/1/2
| the <day> of <month>           -> 96/2/1
| <month> <day> <year>         -> 3/1/2
| the <day> of <month> <year> -> 3/2/1
.
```

The following section of the grammar uses the rules defined above to define a grammar that accepts sentences concerning amounts and dates of expense items, and produces a translation that is a JavaScript statement that will set the appropriated row and column of the travel expense report to the stated dollar amount. The translations assume that the application developer has defined two JavaScript functions: <tt>ondate(date)</tt> accepts a date and moves a cursor to the appropriate row in the table for that date; and <tt>setnum(col, amount)</tt> that puts the specified amount in the specified column of the table.

```
<item>
= hotel      -> HotelCol
| lodging    -> HotelCol
| air        -> AirCol
| airplane   -> AirCol
| food       -> FoodCol
| meal       -> FoodCol
| car        -> CarCol
| automobile -> CarCol
.

<datespec> = on <date> -> (ondate('1')).

<itemspec> = <item> (is|was) <money> -> (setnum(1, CurrentRow, '2')).

<<command>>
= <itemspec>             -> 1
| <datespec>             -> 1
| <itemspec> <datespec> -> (2; 1)
| <datespec> <itemspec> -> (1; 2)
.
```

The following table lists some sample inputs accepted by the grammar and the resulting translation specified by the grammar.

Input	Translation
Air was 2 86 oh 5 on February 19th	ondate('96/02/19'); setnum(AirCol, CurrentRow, '000286.05')
Lodging was 52 dollars 87 cents	setnum(HotelCol, CurrentRow, '000052.87')
On June 8th 1995	ondate('95/06/08')
Air was 5 thousand 6 dollars and 32 cents	setnum(AirCol, CurrentRow, '005006.32')

For information about translation parameters for the grammar compiler, see Section 5.2 [SMAPI Grammar Compiler Options], page 38.

5 SMAPI Grammar Compiler

Grammars specify the words and phrases that a user can say to your application. Grammars are defined using SRCL syntax in plain-text BNF files. These BNF files must be compiled into a format appropriate for use by the speech engine. The SMAPI Grammar Compiler takes a plain-text grammar file (BNF) and compiles it into a finite state grammar file (FSG). An FSG file is the binary representation of the BNF file. The FSG file is what the speech engine uses at run time to determine which words and phrases are currently available for a user to say to your application. Grammars can also be compiled programmatically at run time.

This chapter describes how to use the SMAPI Grammar Compiler to create the FSG file(s) for your application. It also describes the options that are available for use in the compilation process.

Windows developers note: Do not use the SAPI Grammar Compiler for your SMAPI-based applications. Use the SMAPI compiler only. Each compiler generates different target behaviors.

5.1 Using the SMAPI Grammar Compiler

The syntax of the SMAPI Grammar Compiler is as follows:

```
VTBNFC [-tr] [-en] [-n] [-m | -m- | -m+] [-s | -s- | -s+] [-o outfile | -
d outdir] grammarfile
```

The parameters are:

-tr Generates an fsg file usable only by the translation API. This fsg file will not be accepted by the engine. For more information about translation rules, see Section 4.8 [Using the Grammar Translation Facility], page 31.

-en Generates an fsg file without translations for the engine.

If the grammar contains translations, you must specify either -tr or -en when you compile the grammar. This means that you compile the grammar twice: once with -tr, producing an fsg to be provided to the translation API (VtLoadFSG), and once with -en, producing an fsg to be provided to the engine (SmDefineGrammar). For more information about translation rules, see Section 4.8 [Using the Grammar Translation Facility], page 31.

-n Causes the uniform probability computation to be turned off.

-m Instructs the compiler to enable mumble words. Use -m to tell the speech engine to enable mumble words, but not to return the mumble text. Use -m+ to tell the speech engine to enable mumble words and to return the mumble text. Use -m- to disable mumble words. (This is the default).

-s Instructs the compiler to enable silence words. Use -s to tell the speech engine to enable silences, but not to return silence words. (This is the default). Use

-s+ to tell the speech engine to enable silences and to return silence words. Use -s- to disable silence words.

-o outfile The name of the output FSG file. By default, the grammar compiler uses the file name of the grammar file and adds an extension of .fsg.

-d outdir The name of the directory in which the compiler places multiple FSG files, one for each root.

grammarfile

The name of the BNF grammar file to be compiled. The grammar file name must be fully qualified if it does not reside in the current directory.

Windows developers note: The file extension does not have to be specified (it defaults to .bnf). Also, if you write non-English grammars (for example, those with accented characters or "umlaute"), the grammar file must be in code page 1252 so that the special characters are handled properly.

5.2 SMAPI Grammar Compiler Options

There are four options associated with the grammar compiler:

Non-uniform probability computation

By default, grammars are compiled with what is known as the uniform probability computation, which means that all words out of a given state are equally likely. The non-uniform probability option enables the developer to turn the uniform probability computation off.

The recognition performance differences between the two options is strongly grammar dependent. An extreme example is:

```
<Number> = "point" <digit> | <digit>
<digit>  = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The first word that can be spoken is "point" or a digit. With the uniform probability computation, all initial words have probability 1/11. Without the uniform probability computation, "point" has a probability of 0.5, while each digit has a probability of only 0.05.

Without the uniform probability computation, language model probabilities are assigned to be locally uniform for each production rule in the BNF file. Take the following example:

```
<A> = <a1> | <a2>
<a1> = "West Palm Beach"
<a2> = "Bejing" | "Yorktown"
```

Without a uniform probability computation, <a1> and <a2> each get probability 0.5. <a1> expands to a single terminal, "West Palm Beach," which has probability 0.5 x 1 or 0.5. Since <a2> expands to two terminals, which are considered equally likely, "Bejing" and "Yorktown" each get probability 0.5 x 0.5 or 0.25. Consequently, "West Palm Beach" has twice the probability of the

other alternative phrases. This allows an application developer the opportunity to bias word probabilities, where they are known or where they make sense to use.

If <A> were rewritten as:

```
<A> = "West Palm Beach" | "Bejing" | "Yorktown"
```

then all possibilities are equally likely. This is what the uniform probability computation will generate, independent of how the rules are expressed in the BNF file.

To be independent of rule construction in BNF files, most application developers would likely want to keep the default; that is, they should use the uniform probability computation.

Mumble words

Use this option to generate an FSG file that handles cases where the end user injects mumbling or other non-speech noise while speaking. For example, in a grammar with the following construct:

```
<rule> = open <program>
```

the user might say "uhm...open Lotus Notes" instead of "open Lotus Notes." Specifying the mumble words option causes the speech engine to ignore any speech that is not defined as valid by the BNF grammar.

You can also tell the speech engine to return the non-speech noise text back to the application, along with the word or phrase that was recognized. A mumble is flagged as an SM_WORD with an empty spelling ("") in an SM_RECOGNIZED_PHRASE message. Flag settings for mumble on a run-time SmDefineGrammar call override the flags compiled into the FSG.

Silence words

Use this option to generate an FSG file that handles cases where the end user pauses briefly while speaking a command. For example, in a grammar with the following construct:

```
<rule> = open <program>
```

the user might say "open <pause> Lotus Notes" instead of "open Lotus Notes." Specifying this option causes the speech engine to ignore any silence within an utterance.

SMAPI treats silence similarly to mumbles. Silence is allowed within a phrase, and is returned as an SM_WORD with a spelling of "<silence>." As with the mumble words option, flag settings for silence on a run-time SmDefineGrammar call override the flags compiled into the FSG.

Multiple Roots

By default, if a BNF grammar file contains multiple roots marked by <<>> notation, the grammar compiler will return an error. Use this option to enable multiple roots to be specified in a single BNF file. For example, if the BNF file contains:

```
<1> = ...
<2> = ...
<3> = ...
<<a>> = <1> | <2>
<<b>> = <1> | <3>
<<c>> = <2> | <3>
```

Using the multiple roots option, the compiler generates a.fsg, b.fsg, and c.fsg, corresponding to the root rules <<a>>, <<b>>, and <<c>>.

Please note: There are no restrictions on the ordering of multiple roots within the BNF file.

5.3 Compiling a Grammar

To compile a grammar:

1. From a command prompt switch to the directory that contains the SMAPI Grammar Compiler. If that directory is in your path, it is not necessary to switch to that directory location.
2. Type VTBNFC followed by any optional parameters and the name of the grammar file to be compiled. Press the Enter key.
3. If the syntax of the command has errors, command-line syntax help is displayed; otherwise, the Grammar Compiler runs. If the grammar has errors, error messages are displayed.
4. To redirect the compilation to an output log, add " > filename " to the end of the command. For example,

```
vtbnfc active.bnf > active.log
```

The following example compiles the grammar file, ACTIVE.BNF, and enables mumble words. The output file name is ACTIVE.FSG.

```
vtbnfc -m active.bnf
```

Please note: You can include compiler statements in your make file.

6 Writing the Application Interface

All command and control applications need to perform certain functions. Writing the application interface to perform these functions is relatively straightforward; in fact, with less than 20 SMAPIs, you can implement command and control in your application. Plus, the ViaVoice SDK includes many other SMAPIs that enable you to provide even more capabilities than what are prescribed as the basics.

This chapter describes the basic functions that a command and control application should support. It also suggests some optional features that developers of command and control applications might want to consider.

6.1 Basic Command and Control Tasks

At a minimum, all command and control applications should handle the following tasks:

- Establishing a recognition session
- Defining and enabling vocabularies
- Directing the engine to process speech
- Processing recognized commands
- Interacting (coexisting) with other speech-aware applications
- Disconnecting from the engine

To communicate with the engine and to received decoded text, a command and control application must first establish a recognition session. Next, the application must tell the engine what words the user can say. This is done by defining and enabling vocabularies. Once the vocabularies are known to the engine, the application must then tell the engine to process speech. This is done by acquiring control of the microphone and requesting that the engine begin recognizing speech on behalf of your application. As speech is recognized by the engine and decoded into text, the application must process this information and decide what to do with it. Finally, when your application is closing down, it must disconnect from the speech recognition engine.

For a more detailed description of these tasks, see Chapter 13 [Programming Tasks], page 89. The ViaVoice SDK SMAPI Reference documents all of the SMAPIs that are available to develop command and control applications.

6.1.1 Enabling and Disabling Vocabularies

You can have multiple vocabularies active for your command and control application at any time. However, to improve performance (both recognition and accuracy), your application should narrow the possibilities by enabling and disabling vocabularies as they are needed.

The speech recognition engine stops decoding after it recognizes a word or phrase. This is an opportune time for your application to change the active vocabularies for recognizing the next command. (Of course, while the engine is halted, it is still capturing and processing audio, so no words are lost.)

6.1.2 Handling Speech Focus

ViaVoice allows multiple applications to connect to the speech recognition engine at the same time. There is only one microphone, and the application with control of the microphone is referred to as the application with speech focus. When an application has speech focus, it receives decoded text from the speech recognition engine. For more information on speech focus, see Section 11.2 [Speech Focus], page 68.

Be aware that other speech-aware applications can take speech focus away. When an application requests speech focus, it gets it. Since there can be multiple speech-aware applications connected to the engine at any time, speech focus can be taken away from your application.

The engine does not impose a speech focus model tied to window focus. It is up to the application to implement, if desired.

6.1.3 Notification

Notification refers to the asynchronous status messages sent from the speech recognition engine to interested speech-aware applications. An application can use notification messages to display the following types of information sent from the engine:

- Microphone state (on/off)
- Focus state (release, pending, denied, granted)
- Speech engine state (normal/reduced CPU mode)
- Recognized command words
- Audio level

While notification messages primarily support the speech monitoring application, all speech-aware applications can subscribe to notification groups. See Section 11.3 [Notification], page 70 for more information.

6.2 Other Command and Control Tasks

This section describes some optional tasks that a command and control application might choose to handle.

6.2.1 Querying System Parameters

An application can query the engine for several system parameters, including task (domain) ID, user ID, and enroll ID. There are several reasons an application would want to do this, including:

- A task-specific application should determine if the current task is the correct one for the application. For example, a command and control application using German vocabularies would need to query the engine for the active task. If German command and control is not the active task, the application would need to change the task accordingly.
- A command and control application might want to indicate which user and enrollment is being used (so that the user can change either, if necessary). The application queries

the engine for this information, and can display it appropriately to the user. For example, when the application starts, it might want to indicate that it is starting for "User Eric with enrollment Office."

On Windows, system parameters such as these can be specified by the end user through the Options program that is included with the ViaVoice Distribution Package.

6.2.2 Enrolling

Command and control application developers should ensure that their users understand the benefit of enrollment, which is increased accuracy. ViaVoice is a speaker-independent system, and most users will enjoy acceptable recognition accuracy without enrolling. You might want to suggest to users that they enroll if their early attempts yield less than desirable accuracy. Since untrained accuracy varies from user to user depending on voice characteristics such as pitch or accent, enrollment can improve accuracy considerably for certain users.

On Windows an enrollment application is included in the ViaVoice Distribution Package. Note that no other speech-aware application can be running at the same time as enrollment.

6.2.3 Supporting Annotations

Using annotations in command and control grammars can make it easier to parse recognized text and decide the appropriate course of action. With annotations, an application can base its actions on the annotation rather than on the recognized text. As an example, consider the following simple grammar defined using numeric annotations:

```
<color> = orange:1
        | aqua:2
        | blue:3
        | gold:4
        .
```

The developer could implement programming logic (using a switch statement, for example) around the numeric annotation instead of the actual recognized text. In other words, instead of taking some action based on the text "orange," the application would take some action based on the integer 1. Using annotations in this way is especially helpful when developing applications that support multiple languages. The application does not have to change any logic based on the translation of the recognized text. For more information on using annotations in grammars, refer to "Grammar Annotations–A Post Parsing Aid" in Section 4.3 [Introduction to SRCL Grammars], page 18.

6.2.4 Supporting Dictation as well as Command and Control

Applications can support both command and control and dictation. Command and control is implemented using grammars and/or dynamic command vocabularies; dictation is implemented using the general office vocabulary provided with ViaVoice. Grammars and/or dynamic command vocabularies can be used during dictation.

6.3 User Interface Considerations

Applications need to provide user interface elements to support command and control. These elements can be visual (graphical), audio, or some combination of both. Some techniques include:

- All speech-aware applications should consider providing visual feedback to the user for such information as microphone status, audio level, and connect status.
- If your application supports both command and control and dictation, you might want to provide feedback to users so that they know which "mode" they are in.

7 Developing Dictation Applications

All dictation applications need to perform certain functions. Writing a dictation application is not a difficult or complex undertaking; in fact, it is fairly straightforward. Using less than 25 APIs, developers can implement robust dictation applications for their users. Furthermore, ViaVoice SDK includes many other SMAPIs that enable developers to provide even more capabilities than what are prescribed as the basics.

This chapter describes the basic functions that a dictation application should support. It also suggests some optional features that developers of dictation applications might want to consider.

7.1 Basic Dictation Tasks

At a minimum, all dictation applications should handle the following tasks:

- Establishing a recognition session
- Defining and enabling vocabularies
- Directing the engine to process speech
- Processing recognized text
- Correcting errors
- Interacting (coexisting) with other speech-aware applications
- Disconnecting from the engine

To communicate with the engine and to receive decoded text, a dictation application must first establish a recognition session. Next, the application must tell the engine what words the user can say. This is done by defining and enabling vocabularies. (For the general office vocabulary for dictation, the application enables the "text" vocabulary.) Once the vocabularies are known to the engine, the application must then tell the engine to process speech. This is done by acquiring control of the microphone and requesting that the engine begin recognizing speech on behalf of your application. As speech is recognized by the engine and decoded into text, the application must process this information and decide what to do with it. The application must also handle error correction; that is, the user needs to be able to correct misrecognitions by the engine, which will result in improved recognition accuracy. Finally, when your application is closing down, it must disconnect from the speech recognition engine.

For a more detailed description of these tasks, see Chapter 13 [Programming Tasks], page 89. The ViaVoice SDK SMAPI Reference documents all of the SMAPIs that are available to develop dictation applications.

7.1.1 Correcting Errors

Error correction is an important task for a dictation application to handle because it enables the user to improve recognition accuracy. Error correction improves accuracy in two ways:

By adding pronunciations that the engine does not know about, and by helping the engine better understand the user's style of speaking through the context of the words that were spoken. If the user corrects a word, the next time the user says that word in the same context, the engine has a better chance of getting it right. However, if the user corrects all of his/her dictation errors by typing over the dictated text, recognition accuracy will not improve - the engine has no way of knowing about its errors and "learning from its mistakes."

Users can also improve recognition accuracy by training individual words. On Windows this can be done using the Vocabulary Manager or Vocabulary Expander, which is included in the ViaVoice Run Time Kit. This is not as effective as error correction. Since error correction provides the context of adjacent words and the Vocabulary Manager or Vocabulary Expander does not, the user's word-usage model can be further refined.

A dictation application must provide the user interface through which the user corrects dictation errors. Developers can provide an interface that resembles the error correction facility of IBM SpeakPad, or they can provide one that is consistent with their application interface. Some of the features of an error correction user interface should include:

- Playing back what was said to the user
- Displaying the list of alternative words provided by the engine
- Allowing the user to type in the word if it is not available from the list of alternatives
- Allowing the user to add or change the pronunciation (If you are migrating from discrete to continuous, there is a new return code to handle the re-recording of pronunciations for correction.)

An application must save audio information as part of the correction process. Saved audio is used to play back dictated words to the user. It also saves information required to provide alternative word lists and to build new word pronunciation models during correction. For more information, see Section 13.2.6 [Processing Speech Engine Audio], page 116.

Audio (along with other speech and application state data) must also be saved if the dictated text is going to be corrected during another session or by another user. For details on what your application must do to save and restore speech sessions, see Section 13.2.7 [Writing ViaVoice Applications to Save and Restore Speech Sessions], page 117.

7.1.2 Processing Firm and Infirm Words

Firm words are those words which the engine has completed decoding. Once the engine has declared a word "firm," it is finished processing that word. The application should display all firm words to the user as they are recognized by the engine.

Infirm words, on the other had, are those that the engine is still decoding. One or more infirm words might eventually become a single firm word, and a single infirm word might result in one or more firm words. Earlier versions of the ViaVoice engine returned infirm words along with firm words. Currently the engine only returns firm words, and does not return any infirm words.

SmGetFirmWords is the API function that an application uses to acquire firm and words. This call is an access function on the SM_RECOGNIZED_TEXT message. The SM_WORD structure contains the word's spelling, size, tag, and the vocabulary in which the word was

found. The tag is a unique ID that the developer uses to refer to the word. Tags are used by the engine to map dictated text to saved audio and to a list of alternative words.

7.1.3 Handling Speech Focus

ViaVoice allows multiple applications to connect to the speech recognition engine at the same time. There is only one microphone, the application with control of the microphone is referred to as the application with speech focus. When an application has speech focus, it receives decoded text from the speech recognition engine. For more information on speech focus, see Section 11.2 [Speech Focus], page 68.

Dictation applications need to have speech focus for the following tasks:

- To turn the microphone on and off
- To play back audio for corrections
- To correct errors

Be aware that other speech-aware applications can take speech focus away. When an application requests speech focus, it gets it. Since there can be multiple speech-aware applications connected to the engine at any time, speech focus can be taken away from your application.

The engine does not impose a speech focus model tied to window focus. It is up to the application to implement, if desired. On Windows such a model is associated with SAPI implementations by default.

7.1.4 Notification

Notification refers to the asynchronous status messages sent from the speech recognition engine to interested speech-aware applications. An application can use notification messages to display the following types of information sent from the engine:

- Microphone state (on/off)
- Focus state (release, pending, denied, granted)
- Speech engine state (normal/reduced CPU mode)
- Recognized command words
- Audio level

While notification messages primarily support the speech monitoring application, all speech-aware applications can subscribe to notification groups. See Section 11.3 [Notification], page 70 for more information.

7.2 Other Dictation Tasks

This section describes some optional tasks that a dictation application might choose to handle:

7.2.1 Querying System Parameters

An application can query the engine for several system parameters, including task (domain) ID, user ID, and enroll ID. There are several reasons an application would want to do this, including:

- A task-specific application should determine if the current task is the correct one for the application. For example, a radiology application should query the engine for the active task. If radiology is not the active task, the application would need to change the task.
- Dictation applications should ensure that the recognition session that is about to be established is capable of doing what the application needs it to do. For example, it is possible that only command and control is installed on the user's system. A dictation application could query the engine to determine if the general office vocabulary (or any other dictation task) is available. If not, the application should instruct the user to install dictation.
- A dictation application might want to indicate which user and enrollment is being used (so that the user can change either, if necessary.) The application queries the engine for this information, and can display it appropriately to the user. For example, when the application starts, it might want to indicate that it is starting for "User Eric with Enrollment Office." Or, while the user is dictating, the application might display "Dictating for user Eric..."

On Windows system parameters such as these can be specified by the end user through the Options program that is included with the ViaVoice Distribution Package.

7.2.2 Providing Commands During Dictation

Command and control is implemented using grammars and/or dynamic command vocabularies. Both vocabulary types can be used during dictation.

7.2.3 Supporting Dictation Macros and Templates

Dictation macros and templates can enable rather powerful productivity and usability gains for users. You may want to support macros and templates in your application. If you do, keep the following considerations in mind:

- Macros are added to the dictation text vocabulary, while templates are added as a dynamic command vocabulary. This is because templates must be expanded immediately, and dynamic command vocabularies are the only type of vocabulary to be immediately firmed by the speech recognition engine. Normal vocabulary processing rules apply. For more information, see Section 13.2.1 [Setting Up Vocabularies], page 96.
- The Dictation Macro Editor (available only on Windows) maintains a macro and template database across all speech-aware applications running on the system. Macros can be updated by the user at any time. You should update your application's local copy of the database each time you switch to dictation so that your application has the latest definition of macros and templates available.

- Macros and templates, by their very nature, can be memory-intensive. You should be careful to free these resources when they are no longer needed using the appropriate DMAPI calls.

7.2.4 Enrolling

Dictation application developers should ensure that their users understand the benefit of enrollment, which is increased accuracy. ViaVoice is a speaker-independent system, and most users will enjoy acceptable recognition accuracy without enrolling. You might want to suggest to users that they enroll if their early attempts at dictation yield less than desirable accuracy. Since untrained accuracy varies from user to user depending on voice characteristics such as pitch or accent, enrollment can improve accuracy considerably for certain users.

On Windows an enrollment application is included in the ViaVoice Run Time Kit. Note that no other speech-aware application can be running at the same time as enrollment.

7.3 User Interface Considerations

Applications need to provide user interface elements to support dictation. These elements can be visual (graphical), audio, or some combination of both. Some techniques were described earlier (correcting errors, displaying the active user and enrollment, providing "stop dictation" as a command vocabulary during dictation, and enrollment.) Other considerations include:

- All speech-aware applications should consider providing visual feedback to the user for such information as microphone status, audio level, and connect status.
- You might want to provide a reminder to users to use isolated speech while dictating.
- In ViaVoice, when users begin to dictate, an audio waveform is played ("Dictation started") and a visual cue ("Start dictating..." in reverse video) is displayed. When users stop dictation, a corresponding waveform is played ("Dictation stopped".) You might want to provide similar feedback.

8 Developing Enrollment Applications

One of the features of the ViaVoice speech recognition system is the ability to train the system for a particular speaker. The system comes with an acoustic model which can result in accuracy of over 90% for the average speaker. A speaker with a heavy accent will not see these results. Adapting the system for a specific speaker can greatly improve the accuracy for that speaker. Even the average speaker can see accuracy improvements after adapting the system.

In order to train the system a sample of acoustic data must be processed by the training program which modifies the speaker independent acoustic model. The process of recording this acoustic data is called enrollment. When performing what's known as supervised enrollment, an enrollment application interacts with the speech recognition engine to display sentences from a pre-defined script to the user. As the user reads the sentences the speech recognition engine records the user's voice, and saves this acoustic data in files while performing speech recognition. The recognition results are returned to the enrollment application which can reflect feedback to the user on the progress of the enrollment process. Once the user has read a minimum number of sentences the enrollment application tells the speech recognition engine to start the training program which will use the recorded audio data to create a modified acoustic model. Other types of enrollment processes exist such as batch enrollment and unsupervised enrollment. These are beyond the scope of this document.

This chapter describes how to develop a basic supervised enrollment application. It also suggests some optional features that developers of enrollment applications might want to consider.

8.1 Basic Enrollment Tasks

At a minimum, all enrollment applications should handle the following tasks:

- Establishing an enrollment session
- Defining and enabling grammar vocabularies
- Processing the user's speech
- Starting and monitoring the training program

8.1.1 Establishing An Enrollment Session

As in any other speech application, the enrollment application must first open a session with the speech recognition engine using the SMAPI function SmOpen. Any callbacks for asynchronous requests should be added using the SMAPI function SmAddCallback. Callbacks should also be added for the recognized phrase and utterance complete messages which will be sent to the application by the speech recognition engine during the enrollment process. The arguments SmNrecognizedPhraseCallback and SmNutteranceCompletedCallback are defined in smargs.h for this purpose.

The enrollment application will then connect to the speech recognition engine as a database session using the SMAPI function SmConnect. At this point a new userid and or enrollid can be added to the system using the SMAPI functions SmAddUser and SmAddEnrollId. A new enrollid can be created for an existing user or an existing enrollid can be trained. Alternatively, at this point the database connection can change the default userid and or enrollid. Also the list of available scripts can be queried using the SMAPI function SmQueryScripts. Currently supervised enrollment can only be done using existing scripts installed with the system. SmQueryScripts will also return the minimum number of correctly decoded sentences which should be read before the training program is started. The access function SmGetIncrements can be used to retrieve this minimum number of sentences from the SmQueryScripts reply.

Once the userid, enrollid, and script have been selected the database connection can be terminated and the enrollment application can then connect to the speech recognition engine as an enrollment session. The application will build an argument list using the SMAPI function SmSetArgs. This argument list will then be passed to the SMAPI function SmConnect, similarly to when connecting to the speech recognition engine as a recognition session. The following attributes must be specified when connecting for an enrollment session:

SmNenrollment	Set to TRUE to specify enrollment session
SmNuserId	Selected userid or default
SmNenrollId	Selected enrollid or default
SmNenrollIdDescription	Enrollid description (eg "First 22khz enrollment")
SmNscript	Selected script
SmNaudioHost	Audio source specification or default
SmNconnectionId	Windows only: Identifies connection, returned as WPARAM in async replies

Once the application has established an enrollment session with the engine it must tell the engine to save the audio data which will be gathered during the enrollment process in the correct form required by the training program. The SMAPI function SmSet should be used for this purpose. The item argument should be set to SM_SAVE_AUDIO, with the value argument set to SM_SAVE_AUDIO_ADAPTATION. Example:

```
SM_MSG reply;
SmSet ( SM_SAVE_AUDIO, SM_SAVE_AUDIO_ADAPTATION, &reply );
```

If the enrollment application is to support playback of the recorded audio for providing feedback to the user the value argument should be set to the logical OR of SM_SAVE_AUDIO_ADAPTATION and SM_SAVE_AUDIO_PLAYBACK. Example:

```
SM_MSG reply;
SmSet ( SM_SAVE_AUDIO,
        SM_SAVE_AUDIO_ADAPTATION | SM_SAVE_AUDIO_PLAYBACK, &reply );
```

8.1.2 Defining and Enabling Grammar Vocabularies

Once the enrollment session has been established the application will request a sentence of text from the script using the SMAPI function SmRequestScriptText. This text will be used to define a grammar vocabulary using SmDefineVocabEx. The options flags

SM_VOCAB_WORDS and SM_VOCAB_PHRASE must be set. The data parameter will point to a string which is the concatenation of each spelling field from the array of SM_WORDs returned by SmRequestScriptText. The length parameter will be set to the length of this string. Defining the grammar will limit the speech recognition engine's choices to the words in the grammar in the order in which they appear in the grammar. The text returned by SmRequestScriptText will also be displayed to the user to read.

An alternative to the grammar vocabulary which has resulted in improved accuracy after adaptation is the select vocabulary. The select vocabulary also limits the speech recognition engine's choices to the words in the vocabulary in the order in which they appear. But it does not require the engine successfully recognize to the end of the sentence. With a grammar vocabulary if any word in the phrase is misrecognized the entire utterance is rejected. With a select vocabulary if a word is misrecognized, all previous words in the utterance can be accepted. A new select vocabulary can be defined starting with the mis-recognized word. The select vocabulary is defined using the SMAPI function SmDefineVocabEx. The options flags SM_VOCAB_VOCWORDS and SM_VOCAB_SELECT must be set. The data parameter will point to an array of SM_VOCWORD structures. The spelling fields in this array will be filled from the spelling fields in the array of SM_WORDs returned by SmRequestScriptText. The length parameter will be set to the number of elements in the array of SM_VOCWORD structures. The select vocabulary also allows the ability to mark optional words which may or may not be spoken. Punctuation words, for example, can be marked as optional, which allows the user to optionally skip these words when reading the enrollment script. To mark a word as optional set the SM_WORD_RESERVED_USER1 flag in the flags field of the SM_VOCWORD structure.

The application must enable the grammar vocabulary using the SMAPI function SmEnable-Vocab. The utterance number must also be set at this point using the SMAPI function SmSetUtteranceNumber. Generally the utterance number is set to 1 for the first sentence and incremented for each sentence.

8.1.3 Processing the User's Speech

The application must provide some feedback to the user to indicate that the system is now ready for the user to turn on the mic and begin reading. Now the microphone can be turned on using the SMAPI function SmMicOn and the speech recognition engine can be told to begin recognizing using the SMAPI function SmRecognizeNextWord. This is similar to the processing done for a dictation session. As the speech recognition engine performs recognition of the user's speech it will return the decoded words in the form of an asychronous SM_RECOGNIZED_PHRASE message. The application can use this information to provide feedback to the user as sentences are decoded correctly or incorrectly.

When the speech recognition engine has completed decoding the sentence it will send an asynchronous SM_UTTERANCE_COMPLETE message. When this message is received the microphone can be turned off using the SMAPI function SmMicOff.

The access function SmGetPhraseState can be used to retreive the flags field from the SM_RECOGNIZED_PHRASE message. If the sentence was decoded correctly the flag SM_PHRASE_ACCEPTED will be set. In this case the next sentence of text can be displayed, read by the user and decoded using the process outlined above. If the sentence was

not decoded correctly the flag SM_PHRASE_REJECTED will be set. In this case the application may want to have the user try reading the sentence again. The audio data stored for this utterance must first be deleted by calling the SMAPI function SmDiscardUtterance.

8.1.4 Starting and Monitoring the Training Program

Once the speech recognition engine has successfully decoded the required minimum number of sentences the enrollment application can request the training program be started. This is done by calling the SMAPI function SmDisconnect with the SmNcompleteEnrollment attribute set to TRUE. The speech recognition engine will start the training program which will modify the acoustic model using the data accumulated during the enrollment process.

It may be useful to add to the application the ability to suspend the enrollment. This allows a user to stop reading before the minimum number of sentence has been successfully decoded. In this case the application can call SmDisconnect with the SmNsuspendEnrollment attribute set to TRUE.

After the training program is started, he application must then connect to the speech recognition engine as a database session is order to determine when the training program is complete. By calling the SMAPI function SmQueryEnrollIds the application can monitor the progress of the training program and provide feedback to the user. The access function SmGetEnrollIds will return an array of enrollids. The access function SmGetStates will return an array of states. Use the index of the enrollid used for enrollment in the enrollids array to find the status for that enrollid in the states array. When the status is SM_STAT_ENROLLMENT_COMPLETE the training program is complete. If the status is SM_STAT_ENROLLMENT_RUNNING the training program is still processing.

The access function SmGetPercentages will return an array of percentages. Use the index of the enrollid used for enrollment in the enrollids array to find the percentage complete for that enrollid in the percentages array. This percentage can be used by the application to display a progress bar.

If the status is SM_STAT_ENROLLMENT_FAILED the training program has failed for some reason. Often the cause is insufficient disk space. The application should provide a mechanism for the user, if possible, to correct the condition creating the problem and restart the training program. The training program can be restarted by the application connecting to the speech recognition engine as an enrollment session and then disconnecting with the attribute SmNcompleteEnrollment set. The training program will start where the failure occurred. It will not need to perform all of the processing again up to the point of failure.

9 Overview of the C Language SMAPI

All speech engine functions can be accessed from the Speech Manager API, SMAPI. API calls from the application to the engine can either by synchronous or asynchronous. For synchronous calls, SMAPI blocks locally, waits for the engine to reply, and returns a message through a function parameter. For asynchronous calls, SMAPI returns control to the application, the engine puts the reply message into a shared memory buffer, and notifies the application. On Windows, for instance, this notification is through a posted Windows message. The application processes messages from the speech engine in a Windows procedure tied to a window handle owned by the application. The window handle need not be for any particular window in the application, and indeed may be for a dummy window that is established for no other purpose than communication with the speech engine.

Calls from the speech engine to the application are only asynchronous, so the application needs to establish a message-handling procedure for these. Some unsolicited events from the engine are delivered and handled through the same mechanisms (case/callbacks) as asynchronous replies to SMAPI function calls. One example of an unsolicited event from the engine would be recognized words. These are inherently asynchronous, because they depend on when the user speaks. The SmGet* data access functions are used to retrieve data from reply messages or unsolicited messages.

Some operations take several seconds to complete (such as the first application to load and initialize the speech engine). Synchronous calls can lock up an application. You can solve this problem by using asynchronous calls.

SMAPI calls fall into three phases: initialization, recognition, and termination. In the initialization phase, the application sets items that determine how recognition will be performed, such as the user ID, the enrollment ID, and the speech domain. During the recognition phase, the application interacts with the speech engine to set vocabularies and grammars from which recognition takes place, processes recognized speech, controls the microphone, and performs other tasks related to speech recognition itself. During the termination phase, the application makes sure that its session with the speech engine ends properly.

The speech engine allows multiple concurrent connections to it, even from within the same application. Two concurrent sessions from separate applications are referred to as shared sessions, and two concurrent sessions within the same application are referred to as parallel sessions. The API is not guaranteed to be thread-safe, so any individual session should be restricted to a single thread. All SMAPI function calls for a session should be made from the same thread.

Although SMAPI is a C language interface, it is written in an object-oriented style. For example, the reply structure SmReply should be thought of as an object with SmGet* access methods. Keeping this in mind should help you understand the API.

10 Function Call Processing

Function calls to the speech engine can be processed synchronously or asynchronously. For synchronous calls, the application waits for the call to be completed and receives the result of the call in a reply structure.

The engine is a separate process from the application. SMAPI either blocks waiting for a return message (synchronous calls) or returns control to the application (asynchronous calls).

10.1 Message Passing

The engine handles the synchronous and asynchronous calls issued by the application as messages. Here is an example of the synchronous and asynchronous versions of a particular message:

```
SmMicOn ( &Reply );           /* synchronous */

SmMicOn ( SmAsynchronous );   /* asynchronous */
```

The only difference between the two versions is that the synchronous version blocks locally in the speech API waiting for the reply message from the engine, and the asynchronous version returns control to the application immediately and the reply message is later dispatched explicitly by the application or through a registered callback.

The last parameter of each synchronous call is a reply pointer that is set to the address of the reply message. In the asynchronous call, the reply pointer is provided as the first parameter to the callback function, or it is accessible through SmReceiveMsg. The last parameter of an asynchronous call is the constant SmAsynchronous.

These reply message structures are transparent to the application; only the logical contents of the message, such as the lists of user names or word spellings, are available through the SmGet* data access functions. The application cannot directly access these fields in a message through the reply pointer, because the detailed structures are not visible.

10.2 Synchronous Function Calls

Synchronous calls do not return control to the application until the call has been processed or until a time-out waiting for the reply from the engine has been exceeded, which generates an error condition.

A special parameter identifies whether a function call is synchronous. This parameter is always the last parameter in the corresponding call and it points to a reply structure.

The following example shows the synchronous call format:

```
SM_MSG reply;          /* reply message structure */
SmMicOn ( &reply );    /* synchronous             */
```

For more information about reply structures, refer to Section 10.5 [Accessing Data Returned By Function Calls], page 64 and "Reply Message Structures Received from the Speech Engine" in the API Reference.

The following code fragment contains a synchronous function call.

```
unsigned long   nuserids;
char            **userids;
SM_MSG          reply;
int             rc;

/*-------------------------------------------------------------------*/
/* Synchronously query the users known to the speech engine          */
/*-------------------------------------------------------------------*/

rc = SmQueryUsers ( &reply ); // synchronous form

if ( rc == SM_RC_OK ) {             // SmQueryUsers successful
   /*----------------------------------------------------------------*/
   /* Get the user IDs.                                               */
   /*----------------------------------------------------------------*/
   rc = SmGetUserIds ( reply, &nuserids, &userids );

   if ( rc == SM_RC_OK ) {          // SmGetUserIds successful
      for ( ; nuserids-- > 0; userids++ ) {
         /*-------------------------------------------------------*/
         /* Process the list of user IDs.                         */
         /*-------------------------------------------------------*/
      }
   }
}
```

10.3 Asynchronous Function Calls

Asynchronous calls return control to the application immediately after the function call has been initiated successfully. Control returns to the application after a message is sent to the engine. The application does not have to wait for the engine to finish processing before continuing.

A special parameter identifies whether a function call is asynchronous. This parameter is always the last parameter in the corresponding call and it is set to SmAsynchronous. The following example shows the asynchronous call format:

```
SmMicOn ( SmAsynchronous );         /* asynchronous */
```

When an asynchronous call has been processed, the speech engine notifies the calling application with a reply message. Message types are defined in SMCOMM.H with reply messages identified by the suffix, _REPLY).

For more information about reply structures, refer to Section 10.5 [Accessing Data Returned By Function Calls], page 64 and "Reply Message Structures Received from the Speech Engine" in the API Reference.

There are two choices for handling asynchronous messages: through an explicit message receive switch, or through preregistered callbacks. The decision to choose one method over the other is a matter of programming style.

The following code fragment contains an asynchronous function call. This call is exactly the same whether the reply is handled through callbacks or through explicit receive. Any errors detected at this point would only be that the call exceeds parameter limits (for example, sizes on strings) or there was a protocol error communicating with the engine.

```
char         szErrMsg [ 50 ];
int          rc;

/*-----------------------------------------------------------------*/
/* Asynchronously query the users                                  */
/*-----------------------------------------------------------------*/
rc = SmQueryUsers ( SmAsynchronous );  /* asynchronous form */

if ( rc != SM_RC_OK ) {
  /*---------------------------------------------------------------*/
  /* Process error condition                                       */
  /*---------------------------------------------------------------*/
}
```

10.3.1 Asynchronous Function Calls without Callbacks

This section describes how a Windows application processes a reply message from the speech engine without using callbacks. Non-windows applications can similarly dispatch their own asynchronous replies from the speech engine.

To process a reply message from the engine without using callbacks, place the code in the window procedure for the window handle passed to the engine in SmOpen. The application can use a switch statement to explicitly process Windows messages from the engine after a speech session has been established.

The engine sends a standard Windows WM_COMMAND message to alert the application to the presence of a speech message. The application issues SmReceiveMsg to receive the message and SmGetMsgType to determine the message type. The application typically uses a switch statement on the message type (see the following code segment) to process the message.

Please note:

The message field of the Windows message returned by the engine is WM_COMMAND. The wParam field is set to an ID established with SmOpen (reference the connection ID specified on SmConnect) and the lParam field determines the actual message from the engine.

The following code fragment contains processing of the associated message sent from the speech engine:

```
SM_MSG          sm_msg;
int             sm_msg_type;
int             rc;
unsigned long   nuserids;
char            **userids;
 switch ( msg ) {
       ...
  case WM_COMMAND:
    switch ( wParam ) {
      /**
       ** CONNECT_ID is an application-defined value set
       ** when the current session was established using the
       ** following commands:
       **
       ** SmSetArg ( smArgs[smc],
       **            SmNconnectionId,
       **            CONNECT_ID );
       ** smc++;
       **/

    case CONNECT_ID:
        SmReceiveMsg ( lParam, &sm_msg );
        SmGetMsgType ( sm_msg, &sm_msg_type );
         switch ( sm_msg_type ) {
          ...
          case SM_QUERY_USERS_REPLY:
                  /*-------------------------------------------------*/
                  /* Check if SmQueryUsers was successful.           */
                  /*-------------------------------------------------*/
            SmGetRc ( sm_msg, &rc );
             if ( rc == SM_RC_OK )           // SmQueryUsers successful
               {
                /*-------------------------------------------------*/
                /* Get the user IDs.                               */
                /*-------------------------------------------------*/
                rc = SmGetUserIds ( sm_msg, &nuserids, &userids );
                 if ( rc == SM_RC_OK )       // SmGetUserIds successful
                  {
                    for ( ; nuserids-- > 0; userids++ )
                      {
                        MessageBox((HWND) NULL,
                                   *userids,
                                   "User ID",
                                   MB_OK |
                                   MB_ICONEXCLAMATION);
                      }
                  }
               }
```

```
            return 0;


          ...
          default:
            break;
        }
        break;
      default:
        break;
    }
    break;
  default:
    return ( DefWindowProc ( hwnd, msg, wParam, lParam ) );
}
```

10.3.2 Asynchronous Function Calls with Callbacks

The application can use callbacks to process messages sent by the speech engine after a speech session has been established. To use this method, the application must register callbacks prior to issuing the function call. The previously registered callbacks are automatically dispatched when a message is sent by the engine. Multiple callbacks are permitted for a given message. These callbacks are processed in the order that they were registered.

There are special speech API functions to add and remove functions from callback lists. However, such lists must be defined carefully. Because the list of callbacks is processed sequentially, the procedure sequence must guarantee proper processing so that changes of state by early routines do not unintentionally disrupt processing by later routines in the list. The functions include error checking for some conditions that could cause problems in callback list processing; for example, if an SmClose call is detected in any procedure, the remaining procedures in the callback list are not called, and callback-list processing is stopped.

Callback procedures are defined as returning a particular data type and having the following parameters:

```
SmHandler Callback_Handler ( SM_MSG reply, void *client_data,
                             void *call_data );
```

reply The message to be received

client_data
Data supplied by the speech-aware application when registering the callback

call_data Dynamic data to be supplied

Callbacks are registered by using the SmAddCallback function, which specifies the type of callback, the name of the callback routine, and any user data. The type of callback is sometimes referred to as the attribute name for the callback. User data is application-specific information that is passed along to the callback procedure when it is called. One procedure can be registered per SmAddCallback function.

The final link is how reply messages are routed to the different callback functions. This is accomplished by issuing an SmDispatch call. The following code fragments contain an asynchronous function call using a callback. For example, the prototype of the callback routine is:

```
/*-------------------------------------------------------------*/
/* Callback prototype                                          */
/*-------------------------------------------------------------*/
SmHandler  QueryUsersCB ( SM_MSG reply, void *client_data,
                          void *call_data );
```

The callback to the SmQueryUsers call is added as follows:

```
/*-------------------------------------------------------------*/
/* Register the callback for the SmQueryUser asynchronous event */
/*-------------------------------------------------------------*/
SmAddCallback ( SmNqueryUsersCallback,  QueryUsersCB,  NULL );
```

The callback can take the following form:

```
/*-------------------------------------------------------------*/
/* QueryUsersCB                                                */
/*-------------------------------------------------------------*/
SmHandler QueryUsersCB ( SM_MSG reply, void *client_data,
                         void *call_data )
{
  /*-----------------------------------------------------------*/
  /* Internal variables                                        */
  /*-----------------------------------------------------------*/
  int             rc;
  unsigned long   nuserids;
  char            **userids;
   /*----------------------------------------------------------*/
  /* Check if SmQueryUsers was successful                      */
  /*-----------------------------------------------------------*/
  SmGetRc ( reply, &rc );

  if ( rc == SM_RC_OK )      // SmQueryUsers successful
    {
      /*-------------------------------------------------------*/
      /* Get the user IDs.                                     */
      /*-------------------------------------------------------*/
      rc = SmGetUserIds ( reply, &nuserids, &userids );

      if ( rc == SM_RC_OK )    // SmGetUserIds successful
        {
          for ( ; nuserids-- > 0; userids++ )
            {
            /*-----------------------------------------------*/
            /* Process each userid.                          */
            /*-----------------------------------------------*/
```

```
            }
          }
      }

  return ( SM_RC_OK );
}
```

The following code fragment contains processing of the associated message sent from the speech engine on Windows:

```
switch ( msg ) {
       ...
  case WM_COMMAND:
    switch ( wParam ) {
      /**
       ** CONNECT_ID is an application-defined value set
       ** when the current session was established using the
       ** following commands:
       ** SmSetArg ( smArgs[smc],
       **            SmNconnectionId,
       **            CONNECT_ID );
       ** smc++;
       **/
      case CONNECT_ID:
        //
        // Get a reply message structure from
        // speech engine and call previously registered callback,
        // QueryUsersCB, to process it.
        //
               SmDispatch ( lParam );
        return 0;
 default:
        break;
    }
    break;
  default:
     return ( DefWindowProc ( hwnd, msg, wParam, lParam ) );
}
```

10.4 Function Call Error Reporting

The following information explains error reporting by function call type:

Synchronous function calls
: The return code for synchronous function calls specifies whether the request was successful.

Asynchronous function calls

The return code for an asynchronous function call specifies only whether the call is valid. When processing of the call is completed, the engine sends a message data structure to the caller. To check for engine errors, use SmGetRc to extract the return code from the returned reply message structure.

A SM_RC_OK (0) return code indicates success. For more information, see "SMAPI Return Codes and Messages" in the API Reference.

10.5 Accessing Data Returned By Function Calls

The speech engine uses messages to interact with the application.

10.5.1 Access Functions

The application must use access functions to retrieve data from the reply messages. For example, when the application makes an SmMicOn call, the utterance number is retrieved from the reply structure sent by the engine, as follows:

```
SmGetUtteranceNumber ( Reply, &UttNo );
```

This object-oriented approach has several advantages:

- Any changes to message structures in future releases of the API will not impact the application because the SmGet functions access the data.
- The messages are typed internally; for example, compare an SM_MIC_ON_REPLY message with an SM_RECOGNIZED_WORD message. This allows strong type checking in the access function. Applying a particular access function, such as SmGetFirmWords, to a microphone-on reply message, will return an rc = SM_RC_SM_EINVAL_MSG_TYPE.

Unsolicited speech events, such as recognized words, are also sent as messages from the engine to the application and are handled through the same access functions.

For clarity, all of the data access functions have the SmGet or SmReturn prefix. These functions do not interact with the engine; they simply provide local access to the logical contents of a message that has already been received. Since these functions are working on local data, they are all inherently synchronous calls.

10.5.2 Function Calls

The SmGet* functions return logical speech-related data, such as firm words, rather than typed data, such as char arrays. For example, after the call:

```
SmQueryUsers ( &Reply );
```

The reply message contains a list of user names, user IDs, and optional user descriptions. It also contains the return code that indicates the success or failure of this call. Examples of valid access calls follow:

SmGetRc (Reply, &Rc);

SmGetUsers (Reply, &NumUsers, &Users);

SmGetUserIds (Reply, &NumUserIds, &UserIds);

SmGetDescriptions (Reply, &NumDescripts, &Descripts);

Refer to "Reply Message Structures and Callbacks" in the API Reference for the logical content of the message and the valid access functions of each message.

10.5.3 Unsolicited Events

Some events from the engine are unsolicited (for example, recognized words). They are delivered and handled through the same mechanisms as asynchronous replies to function calls. These events are handled by unsolicited callbacks that are defined in "Reply Message Structures and Callbacks" in the API Reference. For example, when the user turns the microphone on and begins to dictate, asynchronous events such as SmNrecognizedTextCallback or SmNrecognizedWordCallback are generated as soon as words are spoken into the microphone and decoded.

Applications need to handle unsolicited events, even if all function calls are made synchronously.

10.5.4 Reply Access Functions

Because an SmGet* function provides access to logical data types, it can be valid for more than one reply message type; the application can use the same SmGet* function for all messages that contain the same logical data. For example, SmGetRc can be applied to any reply pointer because all reply messages contain the return code field. SmGetDescriptions can be used for all reply messages that return a list of descriptive strings, such as the following:

SmQueryEnrollIds (UserId, EnrollId, Language, &Reply)

SmQueryTasks (Language, &Reply);

SmQueryUsers (&Reply);

Refer to "Data Access Functions" in the API Reference for a list of the SmGet* and SmReturn* access functions.

10.5.5 Memory Handling

Pointers returned by SmGet* functions point to private speech API storage, such as pointers to a list of users or a list of enrollments. These pointers should not be freed by the application. In addition, only the reply structure of the last received message is available to process, regardless of whether the message was received through a direct synchronous call, through a callback to an asynchronous call, or through a callback for an unsolicited message.

For efficiency and simplicity of memory handling by the application, there is a single receive buffer for each session (shared or parallel). If the application issues two synchronous SmQuery calls, even with different reply pointers, the receipt of the second message invalidates the contents of the first.

10.5.6 Use of Reply Structure

The API maintains only one static reply buffer per session. This implies that any SMAPI call that returns a reply structure "invalidates" the data pointed to by the previous reply pointer. To preserve data in a reply structure, code your application to remove the data from the reply structure and store it in application memory. The following code fragment illustrates the problem:

```
SM_MSG  Reply1, Reply2;

// Get the list of defined vocabularies
SmQueryVocabs        ( &Reply1 );

// Get the list of enabled vocabularies
SmQueryEnabledVocabs ( &Reply2 );
```

In this example, receipt of the second reply (Reply2) invalidates the data pointed to by the first reply (Reply1).

When the application owns a version of the message contents, it must handle the freeing of those private copies.

11 Session Sharing

The ViaVoice speech recognition engine supports shared sessions, which means that several speech-aware applications can connect to a single speech recognition engine at the same time. You might even want to have more than one connection open to the engine from your application. This ability to have more than one connection to the engine from a single application is known as a parallel session. This chapter describes the following session-sharing concepts:

11.1 Examples of Session-Sharing Components

Session sharing supports a desktop speech environment that can consist of the following logical components:

Speech Monitoring Application
: Provides a single visual focal point for speech state, such as whether the microphone is on or off, audio level, focus control, engine status, and command word history. It also provides common microphone control and recognition parameter control for all session-sharing applications. **Please note:** This typically runs without acquiring the speech focus, but it requires notification of all recognized command words across all speech sessions. See Section 11.3 [Notification], page 70.

Navigator Application
: Provides command support for speech-aware applications. To allow command processing on behalf of a speech-aware application, even while that application has the focus, the concept of a privileged navigation application has been added (see Section 11.4 [Navigator Session], page 76). This application is permitted to keep active vocabularies without having the speech focus.

Dictation Application
: Provides a text entry and correction field.

Enrollment Application
: Provides an interface to record user's speech and to initiate and monitor training status.

Speech-Aware Application
: Provides other services offered by applications created with the ViaVoice SDK. **Window developers please note:** Options (controls user settings) and Enrollment are included in the ViaVoice Run Time Kit, as well as a Microphone Setup utility and a Vocabulary Manager.

11.2 Speech Focus

Session-sharing allows the user to switch easily from one speech-aware application to another. Multiplexed speech-aware applications cooperate for control of the microphone, or speech focus, but only one application has speech focus at a time. To cooperatively share the engine, speech-aware applications use the request to release focus (see the SMAPI Reference for more information). or the request from the speech monitoring application to change the microphone state.

Speech focus works similarly to the way mouse/keyboard focus works. Just as there's one mouse that defines keyboard focus, there is one microphone that defines speech focus. However, the engine doesn't enforce any policy or model on tying keyboard and speech focus–that's left open to the application developer.

In a session sharing environment, after an application is asked to release the speech focus, that application is permitted further interactions with the speech recognition engine to put itself into a consistent state. This simplifies the amount of state information an application needs to carry across focus loss or gain. For example, when the speech recognition engine recognizes words from command vocabularies, the application can make multiple vocabulary manipulation calls so that it is in a known and consistent state the next time speech focus is restored.

11.2.1 Requesting and Releasing Focus

Notification of the loss of focus is delivered through an asynchronous, unsolicited SM_FOCUS_LOST message. There is no need to acknowledge receipt of this message.

An application requests focus through the SmRequestFocus call, which follows the "event notify" model; an immediate reply is generated indicating whether the request has been accepted. If the request is accepted, an asynchronous SM_FOCUS_GRANTED notification message is sent when focus has been granted (see "Granting Focus" below and Section 11.3 [Notification], page 70).

Applications can unilaterally release the speech focus without an explicit request through the SmReleaseFocus call; for example, the request to release focus might be tied to the loss of windowing system focus.

11.2.2 Granting Focus

Focus is automatically granted to the first speech-aware application that connects to the speech recognition engine for recognition. Focus switches are granted based on user initiation. Applications are notified with an asynchronous, unsolicited SM_FOCUS_GRANTED message. Or, if a navigator application is present, it gets focus if another application releases focus without a request pending. Otherwise, if no navigator application is present, the microphone is turned off.

When the user initiates a focus switch, the requesting application signals the speech recognition engine and the speech recognition engine sends a request to the application that

currently has the focus. While an application has the speech focus, all API interactions with the engine continue as if the engine were completely dedicated to that application.

Other speech-aware applications are blocked from some interactions with the engine when they do not have the speech focus (typically involving use of the audio input/output channel). However, application state is preserved across focus switches by the engine; when speech focus is re-acquired, the engine automatically restores the speech state (defined and enabled command and dictation vocabularies). The current microphone state is also preserved across focus switches; for example, if the microphone is on when focus is released, then the microphone will be on when the next application receives focus. The intent is to maintain a consistent state for the user.

The audio saving state is not preserved across focus switch. Audio saving is turned off when focus changes. In order to restore it, the microphone must be turned off, turn on the audio saving, and then turn the microphone back on.

When an explicit focus request is made through SmRequestFocus, the engine limits audio processing for the current focused application. This limitation allows the application to process all the currently queued audio. The process is identical to the processing of an SM_UTTERANCE_COMPLETED message. The message arrives as an unsolicited indication that the audio stream has been processed.

11.2.3 Restrictions

Although each multiplexed speech-aware application has equal access to speech recognition services, the following restrictions apply:

- Only a single speech-aware program is supported during an enrollment session. A status indicator program, such as the speech monitoring application, can run with the enrollment application, but the navigator application cannot. On Windows enrollment is provided by IBM as part of the ViaVoice Run Time Kit.
- During a recognition session, only concurrent sessions for the same user ID, enroll ID, task, and audio source are accepted.

11.2.4 Requesting Next Word

To guarantee ongoing processing of audio during command recognition, the application with speech focus is responsible for always having an SmRecognizeNextWord request pending while the microphone is on.

11.2.5 Guidelines for Handling Focus

The following guidelines show how the session-sharing components typically handle focus:

Speech Monitoring Application
: The speech monitoring application never needs to request focus.

Navigator Application
: The navigator application normally runs without acquiring focus; however; specific actions, such as removing pronunciations and going into reduced CPU

mode, require that the navigator application acquire focus (see Section 11.6 [Allowable API Calls], page 81 for other instances).

Dictation Application
: The dictation application requests speech focus only when necessary, rather than whenever it has keyboard focus; for example, the dictation application requests focus when dictation starts.

Enrollment Application
: The enrollment application, which always requests focus, coexists with the speech monitoring application without conflict.

Speech-Aware Applications
: A speech-aware application requests speech focus only when necessary, rather than whenever it has keyboard focus.

11.3 Notification

Notification refers to the asynchronous status messages sent from the speech recognition engine to interested speech-aware applications. An application can use notification messages to display the following types of information sent from the engine:

- Microphone state (on/off)
- Focus state (release, pending, denied, granted)
- Speech engine state (normal/reduced CPU mode)
- Recognized command words
- Audio level

While notification messages primarily support the speech monitoring application, all speech-aware applications can subscribe to notification groups (refer to "Requesting Notification" below for more information).

11.3.1 Requesting Notification

An application does not automatically receive notification messages after connecting to the speech recognition engine. Notification messages are enabled when an application joins a notification group. A notification group describes the set of applications that subscribe to one of the notification types.

To join a notification group, the application issues an SmSet call that specifies one of the following flags and an associated value of TRUE:

SmSet/SmQuery Flags	Notification Event
SM_NOTIFY_AUDIO_LEVEL	Requests audio level.
SM_NOTIFY_COMMAND_WORD	Requests recognized command words.
SM_NOTIFY_MIC_STATE	Requests changes to microphone state.
SM_NOTIFY_FOCUS_STATE	Requests changes to focus state.
SM_NOTIFY_ENGINE_STATE	Requests changes to engine state.

For example, an application interested in receiving microphone status messages issues one of the following calls:

```
SmSet (SM_NOTIFY_MIC_STATE, TRUE, &reply);          /* synchronous */
```

or

```
SmSet (SM_NOTIFY_MIC_STATE, TRUE, SmAsynchronous);  /* asynchronous */
```

To determine current membership status within a notification group, an application can issue an SmQuery function call, such as one of the following:

```
SmQuery (SM_NOTIFY_MIC_STATE, &reply);           /* synchronous */
```

or

```
SmQuery (SM_NOTIFY_MIC_STATE, SmAsynchronous);   /* asynchronous */
```

A return value of TRUE from SmGetItemValue (...) indicates that the application is a member of the group. Flags for SmSet() and SmQuery() reside in SMCOMM.H.

11.3.2 Receiving Notification

Notification messages are sent as asynchronous, unsolicited events. Applications use explicit message dispatching or registered callbacks to process these messages.

In general, the notification message is determined by the corresponding engine event; for example, the engine sends a notification message when the microphone is turned on or when a command word is recognized. However, a speech-aware application, can be started at any time and then seek to display the current engine state. If notification messages were sent only at discrete event times, the state would not be displayed properly until the next change. To trigger an immediate notification message that reflects the current state, enable a notification group that contains persistent state. Of the available notification groups, microphone state, focus state, and engine state notification groups contain persistent state and support immediate notification. Command word notification does not result in a message until the next word is recognized.

The following information shows the notification messages, their descriptions, the access functions they use to extract information, and the callbacks required to receive the notification.

Message Type

SM_AUDIO_LEVEL - Contains audio level.

Access Functions

SmGetAudioLevel

Retrieves one volume level. The value will be in the range SM_MIN_AUDIO_LEVEL - SM_MAX_AUDIO_LEVEL

SmGetTimes

Retrieves a timestamp which can be used to determine the amount of time in milliseconds since the microphone was turned on.

Callback

SmNaudioLevelCallback

Message Type

SM_COMMAND_WORD - Contains the name of the application that currently has focus and the last recognized word or phrase from a dynamic command vocabulary or grammar.

Access Functions

SmGetFirmWords
: Retrieves the SM_WORD structure associated with the recognized word or phrase.

SmGetApplication
: Retrieves the application name.

SmGetWordTimes
: Retrieves timestamps associated with the start time of the first word and end time of the last word. These timestamps can be used to determine the amount of time in milliseconds since the microphone was turned on.

Callback

SmNcommandWordCallback

Message Type

SM_MIC_STATE - Indicates whether the microphone is on or off.

Access Functions

SmGetMicState

Retrieves the microphone state. Valid values are:

SM_NOTIFY_MIC_ON

SM_NOTIFY_MIC_OFF

SmGetTimes

Retrieves a timestamp associated with when the microphone is turned on or off. This timestamp can be used to determine the amount of time in milliseconds between various speech events. For instance, it can be used to determine the amount of time between when the microphone was turned on and when a word was spoken.

Callback

SmNmicStateCallback

Message Type

SM_FOCUS_STATE - Shows a change in focus status and includes the name of the associated application.

Access Functions

SmGetFocusState
: Retrieves the focus state. Valid values:
  SM_NOTIFY_FOCUS_REQUESTED
  SM_NOTIFY_FOCUS_GRANTED
  SM_NOTIFY_FOCUS_DENIED
  SM_NOTIFY_FOCUS_RELEASED

SmGetApplication
: Retrieves the application name.

SmGetFocusChangeReason
: Retrieves one of the following reason codes on focus changes:
  SM_FOCUS_CHANGE_ON_RELEASE - An application released the focus
  SM_FOCUS_CHANGE_ON_REQUEST - An application requested the focus

Callback

SmNfocusStateCallback

Message Type

SM_ENGINE_STATE - Indicates current engine status.

Access Functions

SmGetEngineState

Retrieves the engine state. The engine state will be in the following set:

SM_NOTIFY_NAVIGATOR_EXCLUSIVE
SM_NOTIFY_FOCUS_APP_EXCLUSIVE
SM_NOTIFY_NONE_EXCLUSIVE
SM_NOTIFY_NORMAL_CPU
SM_NOTIFY_REDUCED_CPU
SM_NOTIFY_PRONUNCIATIONS_ADDED
SM_NOTIFY_PRONUNCIATIONS_DELETED
SM_NOTIFY_ENGINE_SETTINGS_CHANGED
SM_NOTIFY_APPLICATION_CONNECTED
SM_NOTIFY_APPLICATION_DISCONNECTED
SM_NOTIFY_SPEECH_START
SM_NOTIFY_SPEECH_STOP
SM_NOTIFY_SPEECH_TOO_HIGH
SM_NOTIFY_SPEECH_TOO_LOW
SM_NOTIFY_SPEECH_TOO_NOISY
SM_NOTIFY_RECOGNIZED_SPEECH

SmGetTimes

Retrieves a timestamp which can be used to determine the amount of time in milliseconds since the microphone was turned on when the engine state is either:

SM_NOTIFY_SPEECH_START
SM_NOTIFY_SPEECH_STOP.

Callback

SmNengineStateCallback

11.4 Navigator Session

The Navigator session is unique in that it is allowed to do command recognition without requiring speech focus. This permits the navigator application to process commands while speech-aware applications have the speech focus.

Command recognition API calls include those to define, enable, disable, add to, remove from, and undefine dynamic vocabularies, to halt the recognizer, and to request recognition of the next word.

Other calls that extend beyond support of command recognition, such as add-pronunciation, require the navigator application to request and receive speech focus. This is consistent with the model for all other speech-aware applications.

The navigator application receives the speech focus by default when no other speech-aware application has the focus. This notification comes from the unsolicited SM_FOCUS_GRANTED message.

The Navigator session is defined by an attribute on the SmOpen or SmConnect call, as follows:

```
SmSetArg ( smArgs[n], SmNnavigator, TRUE );  n++;
```

Only one active Navigator session can be defined for an engine; if a second application asserts the SmNnavigator field, the SmConnect call fails with a SM_RC_NAV_ALREADY_DEFINED return code.

11.4.1 Exclusive Vocabularies

Command vocabularies defined by the navigator session are global in that they are active, by default, in parallel with any additional vocabularies defined by the application with speech focus. This allows the user to escape back to the navigator application by voice.

There are conditions when an application will want to exclude these global vocabularies, and force the engine to perform recognition using only the vocabularies enabled by the application. For example, a transcription application that does not want any possibility of interruption of text transcription could disable all global command vocabularies and only enable the text vocabulary.

The application with speech focus can enable exclusive access to vocabularies as follows:

```
SmSet(SM_ENABLE_EXCLUSIVE_VOCABS, TRUE, &Reply);
```

and disable it as follows:

```
SmSet(SM_ENABLE_EXCLUSIVE_VOCABS, FALSE, &Reply);
```

This SmSet always returns SM_RC_OK and generates SM_ENGINE_STATE notifications with the flags SM_NOTIFY_NAVIGATOR_EXCLUSIVE, SM_NOTIFY_FOCUS_APP_EXCLUSIVE, and SM_NOTIFY_NONE_EXCLUSIVE.

11.4.2 Vocabulary Scope

The following information describes the relationship between the application and various types of vocabularies:

Predefined vocabularies are owned by the application that currently has focus. Dynamic vocabularies are owned by the application that defines them; they cannot be manipulated by other applications because they are not visible to other applications.

Applications without focus can manipulate their vocabularies, as well as their image of the predefined vocabularies; these changes become visible the next time the application

receives focus. Recognized words are returned to the application that owns the vocabulary; the vocabulary type (dynamic as compared to grammars or predefined dictation) determines whether the recognized words are returned with the SM_RECOGNIZED_WORD, SM_RECOGNIZED_PHRASE, or SM_RECOGNIZED_TEXT message.

11.4.3 Reduced CPU Mode

In a navigation or command and control environment, the user will probably want to ask the speech recognition engine to "stop listening" until it is "wakened" again–probably by voice. The engine needs only to listen for the wake-up word (or a limited set of wake-up words); therefore, the CPU resources consumed by the speech recognition engine could be minimized; this is called reduced CPU or reduced power mode.

The application can make a call that explicitly reduces the engine's CPU requirements; however, this reduces accuracy. The call is independent of dictation and command recognition processing, and the size or number of active words. In the speech recognition engine, reduced CPU mode is not explicitly tied to a limited number of active words. This provides more flexibility for the developer.

Reduced CPU mode is initiated as follows:

```
SmSet ( SM_REDUCED_CPU_MODE, TRUE, &Reply );
```

and terminated as follows:

```
SmSet ( SM_REDUCED_CPU_MODE, FALSE, &Reply );
```

The current state can be queried as follows:

```
SmQuery ( SM_REDUCED_CPU_MODE, &Reply );
```

The SM_ENGINE_STATE notification group reflects changes to this mode to interested applications.

Although setting reduced CPU mode is not restricted to the navigator application, the application that sets reduced CPU mode must have the focus. Reduced CPU mode is terminated either through an explicit request from the application that set it, or when focus is requested by another application. To receive notification when the mode changes, applications that set reduced CPU mode should register for the SM_ENGINE_STATE notification messages.

In the reduced CPU mode, the engine implicitly disables the vocabularies of any other active session. These vocabularies are re-enabled when the engine returns to normal CPU mode. This disabling and reactivation of vocabularies is transparent to the application. The application without focus never receives a recognized word while the engine is in reduced CPU mode.

11.5 Related Functions

The following information explains how the API supports various aspects of the session-sharing environment.

11.5.1 Request Microphone On/Off

To support a speech monitoring application that provides a single microphone control shared by all speech-aware applications, there are functions that allow the speech monitoring application to send microphone on/off requests to the application that currently has speech focus:

```
SmRequestMicOn ( &Reply );
SmRequestMicOff ( &Reply );
```

If no application has focus, the request returns SM_RC_NO_FOCUS_APP. If the request is sent to the application with focus, these calls return SM_RC_OK. There is no further handshaking between applications. If the application with focus accepts the request and acts on it, the corresponding change in microphone state can be tracked through the notification group messages.

11.5.2 Default Values for Initialization

In a session-sharing environment, once the engine has been initialized for a particular user, additional applications can connect without having to explicitly specify the user ID, enroll ID, and task on the SmConnect call. This simplifies the bookkeeping required of speech-aware applications, because this can be handled through the speech monitoring application.

To indicate this "don't care" condition on the SmConnect, there is a SM_USE_CURRENT argument flag, which is valid only during a recognition session. For example:

```
SmSetArg ( smArgs [n], SmNuserId, SM_USE_CURRENT ); n++
SmSetArg ( smArgs [n], SmNenrollId, SM_USE_CURRENT ); n++
SmSetArg ( smArgs [n], SmNtask, SM_USE_CURRENT ); n++
```

SM_USE_CURRENT is type-defined as an empty string, so that it can be used in place of an actual user ID, enroll ID, or task string in an application's resource file.

This flag is interpreted by the engine as follows:

- If the engine is not currently initialized, use the persistent default values that would be returned by SmQueryDefaults.
- If the engine is already initialized, then use the current values.

The actual values used by the engine are returned on the SmConnect reply message and can be retrieved through the following access functions:

```
SmGetUserId ( Reply, &UserId );
SmGetEnrollId ( Reply, &EnrollId );
SmGetTask  ( Reply, &Task );
```

This allows an application to specify initialization with default parameters, but display the actual parameters for the user's information.

11.5.3 Querying and Setting Defaults

The initial default values can be determined through the SmQueryDefault call:

```
SmQueryUserDefault ( SM_DEFAULT_USERID, &Reply );
SmQueryUserDefault ( SM_DEFAULT_ENROLLID, &Reply );
```

```
SmQueryUserDefault ( SM_DEFAULT_TASK, &Reply );
```

The appropriate return values can be retrieved through the SmGetUserId, SmGetEnrollId, and SmGetTask access functions.

The defaults are not changed by explicit initialization, that is, changing from user "sdg" to user "eddie" on SmConnect does not change the default user ID. However, enrolling "eddie" after "sdg" does change the default. The userid default values are maintained by SMAPI and the enrollid, taskid, and topics are maintained by the engine on a per-user basis.

The default values can also be changed by an application through the SmSetUserDefault call:

```
SmSetUserDefault ( SM_DEFAULT_USERID, "sdg", &Reply );
SmSetUserDefault ( SM_DEFAULT_ENROLLID, "ibmsdg1", &Reply );
SmSetUserDefault ( SM_DEFAULT_TASK, "radio", &Reply );
```

11.5.4 Query Sessions

To assist with the management of multiple speech-aware applications, the following query function has been provided to return the list of connected sessions:

```
SmQuerySessions ( &Reply );
```

The associated access functions are SmGetUserIds, SmGetEnrollIds, SmGetTasks, and SmGetApplication to get the name of the registered application. There are parallel lists of user IDs, enroll IDs, tasks, and application names. The number of items in each list is the same.

11.5.5 Detach Sessions

The engine does not support different user IDs, enroll IDs, or task IDs in multiple concurrent sessions. Multiple sessions connected to the engine concurrently must use the same user ID, enroll ID, and task ID. If a session needs to change the user ID, enroll ID, or task ID, the session should call SmDetachSessions to request that all other sessions disconnect from the speech recognition engine. Coupled with the SmQuerySessions call and engine state notifications, an application can monitor all sessions that remain connected to the engine, identify these applications to the user, and prompt the user to take the appropriate action. The engine will not accept changes to the user ID, enroll ID, or task ID until only the session requesting the changes remains connected to the engine.

SmDetachSessions sends a SM_REQUEST_DETACH message to all sessions. This message is informational only. The engine does not force a session to disconnect, nor does it wait for a reply to this request from each session. If a session wishes to comply with the request, it should call SmDisconnect. Applications can close after disconnecting; however, if the application remains up, it should provide feedback to the user to indicate that the application does not have an active speech session. If an application has the ability to re-connect to the engine, it will need to re-define and re-enable all vocabularies and grammars once it re-connects to the engine. Re-connecting to the engine should occur based upon a user request (either from a menu item or another defined event).

11.5.6 Automatically Start and Stop the Speech Engine

If an engine is not loaded, SmConnect will start the engine executable. When all applications disconnect from the engine, the engine may start a countdown timer. The auto_kill engine tag accepts an integer value which is the number of seconds until the engine terminates. A value of True defaults to a 10 second timeout. A value of False means never timeout. Set these values in the defaults stanza of the ENGINE.CFG file in \ViaVoice\Bin. This timeout length can also be set using the SMAPI function SmSet with the value of the item argument set to SM_DELAY_EXIT.

11.6 Allowable API Calls

The tables in the remaining portion of this chapter define the allowable API calls in a session-sharing environment. The tables use the following terminology:

Focus Application
: Application that currently has speech focus.

Navigation Application
: The navigator application when it does not have explicit speech focus. When the navigator application has the speech focus, it has the same functions permitted for the focus application.

No Focus
: The speech-aware application that does not currently have speech focus.

11.6.1 Attribute Functions

Attribute functions are implemented locally in the application's address space by the SMAPI layer, and they do not require any interaction with the speech recognition engine. Consequently, they can be made at any appropriate time, independent of the speech focus.

API Call	Focus Application	Navigation Application	No Focus
SmSetArg	X	X	X

11.6.2 Callback and Dispatching Functions

Callback and dispatching functions are implemented locally in the application's address space by the SMAPI layer.

API Call	Focus Application	Navigation Application	No Focus
SmAddCallback	X	X	X
SmRemoveCallback	X	X	X

11.6.3 Access Functions

Because access functions manipulate only received, application-side data, they are independent of the speech focus.

API Call	Focus Application	Navigation Application	No Focus
SmGet* calls	X	X	X
SmReturn* calls	X	X	X

11.6.4 Connection Functions

API functions that allow an application to connect or disconnect from the speech recognition engine are permitted at all times.

API Call	Focus Application	Navigation Application	No Focus
SmApiVersionCheck	X	X	X
SmClose	X	X	X
SmConnect	X	X	X
SmDisconnect	X	X	X
SmOpen	X	X	X

11.6.5 Session Functions

Some calls related to session sharing are appropriate only when an application is in a particular state (for example, releasing focus only makes sense if the application has the focus). These calls will not be blocked, thereby simplifying application development. These calls are denoted with a plus (+). Allowing other calls, such as requesting a change in microphone state, even when the requesting application has the focus, might also simplify application development.

API Call	Focus Application	Navigation Application	No Focus
SmDetachSessions	X	X	X
SmReleaseFocus	X	X(+)	X(+)
SmConnect	X(+)	X	X
SmRequestFocus	X	X	X
SmRequestMicOff	X	X	X
SmRequestMicOn	X	X	X

Please note: (+) These calls are not applicable.

11.6.6 Database Functions

Database functions are primarily for database access and, other than diverting some engine resource, do not interfere with the focus application's view of the engine. Consequently, they can be permitted without regard to the speech focus. Some of the returned information, however, is kept on a per-session basis, or globally across all connected sessions. The following table identifies database functions.

API Call	Focus Application	Navigation Application	No Focus	State
SmQueryAddedWords	X	X	X	both(*)
SmQueryAddedWordsEx	X	X	X	both(*)
SmQueryAlternates	X	X	X	session
SmQueryDefaults	X	X	X	global
SmQueryEnabledVocabs	X	X	X	session
SmQueryEnrollIds	X	X	X	global
SmQueryLanguages	X	X	X	global
SmQueryPronunciation	X	X	X	global
SmQueryPronunciations	X	X	X	global
SmQueryPronunciationsEx	X	X	X	global
SmQuerySessions	X	X	X	global
SmQueryTasks	X	X	X	global
SmQueryTopics	X	X	X	global
SmQueryUserDefault	X	X	X	global
SmQueryUserInfo	X	X	X	global
SmQueryUsers	X	X	X	global
SmQueryVocabs	X	X	X	session
SmQueryWord	X	X	X	session

Please note: (*) This depends on the scope of the associated vocabulary.

The following functions are primarily administrative and do not change the state of the engine for the application with focus. Some state changes are not done right away (synchronously); for example, changing the default user ID, enroll ID, and task requires three interactions with the engine.

API Call	Focus Application	Navigation Application	No Focus	State
SmSetDefault	X	X	X	global
SmSetUserInfo	X	X	X	global

The parameters of the SmSet call need to be addressed individually. Changing of the audio input mode can take effect only when the microphone is off; this changes the global state of the engine. Other changes are kept on a per-session basis. Settings changes that are not allowed are indicated by NA.

SmSet Flag	Focus App.	Navigation Application	No Focus	State
SM_COMPLETE_COMMAND_TIMEOUT	X	X	X	global
SM_ENABLE_EXCLUSIVE_VOCABS	X	X+	X	global
SM_NOTIFY_*	X	X	X	session
SM_PARTIAL_COMMAND_TIMEOUT	X	X	X	global
SM_REDUCED_CPU_MODE	X	NA	NA	global
SM_REJECTION_THRESHOLD	X	X	X	global
SM_SAVE_AUDIO	X	X	X	session

Note: (+) These calls are not applicable.

The parameters to the SmQuery call are also handled globally or per session:

SmQuery Flag	Focus App.	Navigation Application	No Focus	State
SM_AUDIO_INPUT_MODE	X	X	X	global
SM_AUDIO_OUTPUT_MODE	X	X+	X	global
SM_AUDIO_DEVICE	X	X	X	global
SM_AUDIO_CONFIGURATION	X	X	X	global
SM_COMPLETE_COMMAND_TIMEOUT	X	X	X	global
SM_ENABLE_EXCLUSIVE_VOCABS	X	X	X	global
SM_NOTIFY_*	X	X	X	session
SM_PARTIAL_COMMAND_TIMEOUT	X	X	X	global
SM_RECOGNIZE_MODE	X	X	X	session
SM_REDUCED_CPU_MODE	X	X	X	global
SM_REJECTION_THRESHOLD	X	X	X	global
SM_SAVE_AUDIO	X	X	X	session

11.6.7 Vocabulary Functions

These functions change the state of the active vocabularies, but all dynamically defined vocabularies are handled independently per session by the engine; this is transparent to the application.

API Call	Focus Application	Navigation Application	No Focus	State
SmAddPronunciation	X	NA	NA	global
SmAddToVocab	X	X+	X	session
SmCorrectText	X	X	X	global
SmCorrectTextCancel	X	X	X	global
SmDefineGrammar	X	X	X	session
SmDefineVocab	X	X	X	session
SmDisableVocab	X	X	X	session
SmDiscardData	X	X	X	session
SmEnableVocab	X	X	X	session
SmEventNotify	X	X	NA	session
SmHaltRecognizer	X	X	X	session
SmNewContext	X	NA	NA	session
SmRecognizeNextWord	X	X	X	global
SmRemoveFromVocab	X	X	X	global
SmRemovePronunciation	X	NA	NA	global
SmUndefineVocab	X	X	X	session
SmWordCorrection	X	NA	NA	global

11.6.8 Audio Functions

These functions change the state of the audio system. Given a single audio source, this would change the state seen by the application with focus; therefore, these functions are allowed only by the application with focus.

API Call	Focus Application	Navigation Application	No Focus
SmCancelPlayback	X	NA	NA
SmMicOn	X	NA	NA
SmMicOff	X	NA	NA
SmPlayMessage	X	NA	NA
SmPlayUtterance	X	NA	NA
SmPlayWords	X	NA	NA

12 Parallel Session API Calls

The shared-session API calls are all documented as SmXxxxx (SmDisconnect, for example). These calls can be used with exactly these names if there is only one connection to the engine from the application. If you want more than one connection from your application, you must use the parallel session calls. (Remember the terminology. Shared sessions refers to all concurrent sessions with the engine. Two sessions are said to be parallel if they are both from the same application.)

The name of a parallel session call is the same as the regular call except for two things. First, the name of the call has the characters "Ses" inserted after the "Sm." For example, the parallel session disconnect call is SmSesDisconnect. Second, each parallel session call takes one additional parameter, which is the session ID. This is always the first parameter. So, for our disconnect example, the call would be SmSesDisconnect (hSession), where hSession is the session ID. The session ID is returned in the first parameter of the SmSesOpen call.

Parallel sessions enable independent connections from within the same application. The requirement for parallel sessions arises, for example, when a speech-aware library routine is embedded in a speech-aware application. Each session has a full and complete connection to the engine, including separate shared memory segments for communication, and is therefore relatively expensive with regard to resource use. As a result, parallel sessions are not intended to be used casually within an application. In addition, parallel sessions are completely independent, with each session requiring a unique window handle for engine messages.

Improper use of parallel sessions (for example, to handle vocabulary management within an application, where one session is used for grammars and another for dictation, or where a different session is used for each field in a form) will result in larger memory overhead and will not improve recognition performance. Also, the engine is a single executable, sequentially processing a single audio stream. Parallel sessions cannot reduce recognition latency for a given connection.

13 Programming Tasks

There are three phases of an application session: initialization, recognition, and termination.

13.1 Initialization Phase

During the initialization phase, the speech-aware application connects to the speech recognition engine. The engine is a separate process. The first user of the engine causes the SMAPI to start the engine, if it is not already running.

Many of the sample code fragments used in this chapter include asynchronous function calls. For more information, see Chapter 10 [Function Call Processing], page 57.

13.1.1 Verifying the SMAPI Version

Begin by verifying that the ViaVoice SDK SMAPI version used to compile the speech-aware application matches the version of the speech recognition engine currently installed on the computer of the user of your application. ViaVoice 5.x is SMAPI-compatible, however, ViaVoice 4.1 applications are not backward compatible to previous versions of VoiceType runtimes.

You can use the following statements to verify the SMAPI version:

```
char *sm_version;
char szErrMsg [50];
int rc;

rc = SmApiVersionCheck(SM_API_VERSION_STRING, &sm_version);

if (rc == SM_RC_WRONG_SM_VERSION) {

   // Error processing goes here

}
```

13.1.2 Establishing a Speech Session

A speech session is established when the speech function calls connect an application to the speech recognition engine. SmOpen initializes a local structure that is updated when the application calls SmConnect to connect to the speech recognition engine. The ViaVoice SDK SMAPI supports the following session types:

- Database
- Enrollment
- Recognition

The speech attribute values used with SmOpen and SmConnect determine the session type. Some attributes are common for all sessions and some are valid only for specific sessions:

13.1.2.1 All Sessions:

SmNapplicationName
: The speech-aware application name.

SmNwindowHandle
: The window handle to receive messages from the engine (both asynchronous notifications and replies from asynchronous function calls).

SmNconnectionId
: A number that uniquely identifies this connection within an application. All messages sent to this application by the engine will be WM_COMMAND messages with this number in the WPARAM value.

13.1.2.2 Database Sessions:

SmNdatabase
: The type of session (set to TRUE for a database session).

13.1.2.3 Enrollment Sessions:

SmNenrollment
: The type of session (set to TRUE for an enrollment session).

13.1.2.4 Recognition Sessions:

SmNrecognize
: The type of session (set to TRUE for a recognition session).

SmNnavigator
: Indicates that this recognition session is a special Navigator session. (Refer to Section 11.4 [Navigator Session], page 76.)

SmNenrollId
: The enrollment ID.

SmNtask The recognition domain.

SmNuserId
: The user ID.

Note: SmNenrollID, SmNtask, and SmNuserID don't need to be explicitly provided. If they are not specified, the engine will use the current values.

SmOpen creates a shared memory segment and protocols for communicating with the engine. Typically, SmOpen sets the attributes required for all sessions (SmNwindowHandle, SmNconnectionId, and SmNapplicationName), and SmConnect sets all the remaining session-specific attributes. The speech attributes are stored locally in the application

address space and are not communicated to the engine until the SmConnect call. Consequently, after an attribute has been set, the change does not take effect until the application issues the next SmConnect call.

The application can set speech attributes any time after an SmOpen function call. The attributes can be specified in any order. Attribute names are case sensitive, but attribute values are not.

Pass the session attributes to the SmOpen and SmConnect calls using an array of SmArg structures. The last specified value for a speech attribute takes precedence over previously set values. The arguments in an SmArg array are acted on sequentially, and the last specified attribute takes precedence here as well. For example, the application can request only one type of session at SmConnect, so the last session type specified is used.

The SmSetArg macro has been provided to assist with setting values into the array. The following Windows code fragment illustrates the correct method of initializing your array using SmSetArg, and passing the array to the SmOpen function call:

```
/*----------------------------------------------------------------*/
/* Establish a recognition session                                */
/*----------------------------------------------------------------*/
#define CONNECT_ID   100

int smc = 0;
SmArg smArgs[20];
HWND hWnd;

SmSetArg(smArgs[smc], SmNapplicationName, "SmSampleApp"); smc++;
SmSetArg(smArgs[smc], SmNwindowHandle, hWnd); smc++;
SmSetArg(smArgs[smc], SmNconnectionId, CONNECT_ID); smc++;

SmOpen(smc, SmArgs);
```

Please note: Do not place the statement to increment the counter ("smc++" in the previous figure) within the SmSetArg macro. The macro actually makes two references to the first argument, which results in the counter being incremented twice.

13.1.2.5 Initializing

The speech recognition engine runs as a separate process in support of a speech-aware application. It is started when the SmConnect call is issued. Starting this process and performing the initialization for the requested speaker is a time-consuming process. It is suggested that this call be made asynchronously for applications that must be responsive to user interaction. Connection time is significant only for the first application that starts and initializes the engine; it is insignificant for others. However, there is no way to tell when an application will be first.

13.1.2.6 Database Sessions

During a database session, the user can request status-related information and set various engine and session attributes. The following functions are valid:

- Query and set information about the speech session and speech recognition engine
    - SmQuery
    - SmQueryDefault
    - SmQuerySessions
    - SmSet
    - SmSetDefault
- Query and set information about users
    - SmQueryUserInfo
    - SmQueryUsers
    - SmSetUserInfo
- Query information about the available languages and domains
    - SmQueryLanguages
    - SmQueryTasks
- Query information about the enrollments and their statuses
    - SmQueryEnrollIds
- Request to change the microphone state
    - SmRequestMicOff
    - SmRequestMicOn

The following Windows code sample is an example of how to initialize a database session with the speech engine:

```
#define CONNECT_ID    100

SmArg smArgs[10];
int smc;
int rc;
/*-------------------------------------------------------------------*/
/* Set speech session attributes for SmOpen call.                    */
/*-------------------------------------------------------------------*/
smc = 0;
SmSetArg(smArgs[smc], SmNapplicationName, "SmSampleApp"); smc++;
SmSetArg(smArgs[smc], SmNwindowHandle, hwnd); smc++;
SmSetArg(smArgs[smc], SmNconnectionId, CONNECT_ID); smc++;
rc = SmOpen(smc,      // Number of attributes in array
            smArgs);  // Array of attributes
if (rc == SM_RC_OK)   // SmOpen successful
{
  /*-----------------------------------------------------------------*/
  /* Set speech session attributes for SmConnect call.               */
  /*-----------------------------------------------------------------*/
  smc = 0;
  SmSetArg(smArgs[smc], SmNdatabase, TRUE); smc++;
  // SmConnect is called asynchronously; therefore, the SM_CONNECT_REPLY
  // message from the speech recognition engine must be processed to determine
  // the success of this call.
  //
  rc = SmConnect(smc,               // Number of attributes in array
                 smArgs,            // Array of attributes
                 SmAsynchronous);   // Call is asynchronous
  if ( rc != SM_RC_OK )
  {
    // Error processing goes here...
  }
}
```

13.1.2.7 Recognition Sessions

In addition to all database functions, the following tasks characterize a recognition session:

- Defining and working with dynamic vocabularies
- Defining and working with grammars
- Speaking and working with the recognized words
- Playing back words or utterances
- Correcting misrecognized words to improve future recognition

Some of the SMAPI functions calls which are available from a recognition session have certain restrictions based on the current state of the engine (for example, the engine must be halted or the microphone must be off). For more detailed information, refer to the SMAPI Reference.

For the engine to successfully initialize a session, the language associated with an enrollment ID must match the domain language. For example, you can't run a U.S. English enrollment against a German domain.

The following Windows code sample is an example of how to initialize a recognition session with the speech engine:

```
#define CONNECT_ID   100
SmArg smArgs[10];
int smc;
int rc;
/*-------------------------------------------------------------------*/
/* Set speech session attributes for SmOpen call.                    */
/*-------------------------------------------------------------------*/
smc = 0;
SmSetArg(smArgs[smc], SmNapplicationName, "SmSampleApp"); smc++;
SmSetArg(smArgs[smc], SmNwindowHandle, hwnd);             smc++;
SmSetArg(smArgs[smc], SmNconnectionId, CONNECT_ID);       smc++;
rc = SmOpen(smc,      // Number of attributes in array
            smArgs);  // Array of attributes
if (rc == SM_RC_OK)   // SmOpen successful
{
  /*-----------------------------------------------------------------*/
  /* Set speech session attributes for SmConnect call.               */
  /*-----------------------------------------------------------------*/
  smc = 0;
  SmSetArg(smArgs[smc], SmNrecognize, TRUE);          smc++;
  SmSetArg(smArgs[smc], SmNuserId, SM_USE_CURRENT);   smc++;
  SmSetArg(smArgs[smc], SmNtask, SM_USE_CURRENT);     smc++;
  SmSetArg(smArgs[smc], SmNenrollId, SM_USE_CURRENT); smc++;
  // SmConnect is called asynchronously; therefore, the SM_CONNECT_REPLY
  // message from the speech recognition engine must be processed to determine
  // the success of this call.
  rc = SmConnect(smc,             // Number of attributes in array
                 smArgs,          // Array of attributes
                 SmAsynchronous); // Call is asynchronous
  if ( rc != SM_RC_OK )
  {
      // Error processing goes here...
  }
}
```

13.1.3 Changing Speech Sessions

In order to change a speech session, disconnect the current session, modify the session attributes, then connect the new session. In this case, the new attribute values replace the current values, and the values of attributes that are not passed to SmConnect remain

unchanged. The following code sample illustrates the steps required to change from a database session to a recognition session.

```
SmArg smArgs[10];
SM_MSG reply;
int smc;
int rc;

/*-------------------------------------------------------------*/
/* Change from database session to recognition session.     */
/*-------------------------------------------------------------*/

smc = 0;

rc = SmDisconnect(smc,          // Number of attributes in array
                  smArgs,       // Array of attributes
                  &reply);      // Pointer to reply structure

if (rc == SM_RC_OK)             // SmDisconnect successful
{

  /*-------------------------------------------------------------------*/
  /* Set speech session attributes for SmConnect call.               */
  /*-------------------------------------------------------------------*/
  smc = 0;
  SmSetArg(smArgs[smc], SmNrecognize, TRUE); smc++;
  SmSetArg(smArgs[smc], SmNdatabase, FALSE); smc++;

  // SmConnect is called asynchronously; therefore, the SM_CONNECT_REPLY
  // message from the speech recognition engine must be processed to determine
  // the success of this call.

  rc = SmConnect(smc,              // Number of attributes in array
                 smArgs,           // Array of attributes
                 SmAsynchronous);  // Call is asynchronous

  if ( rc != SM_RC_OK )
  {
      // Error processing goes here...
  }
}
```

In order to completely refresh a session, use both SmDisconnect and SmClose to purge the session data, then initialize a new session using SmOpen and SmConnect. Then, all of the attributes from the previous session will be discarded and replaced by the new attributes or defaults.

13.2 Recognition Phase

During the recognition phase, the speech-aware application controls and interacts with the speech recognition engine, which converts speech input to text. This chapter explains how to do the following related tasks:

Setting Up Vocabularies

Processing Speech Input

Changing The Engine Decoding State

Setting/Querying Speech Engine Parameters

Improving Recognition By Updating Personal Data

Processing Speech Engine Audio

Saving and Restoring Speech Sessions

Handling Speech Recognition Engine Errors

Playing Audio Through The Speakers

Many of the sample code fragments used in this chapter include asynchronous speech function calls. See Chapter 10 [Function Call Processing], page 57 for more information.

13.2.1 Setting Up Vocabularies

A vocabulary is a list of words the engine uses to match speech input and translate it into text. An application specifies the set of active words by defining and enabling one or more vocabularies.

The application must enable at least one vocabulary before the speech recognition engine can process speech. The speech recognition engine uses the enabled vocabularies to find a match to the incoming words, and the vocabulary that provides the match determines the engine state (refer to Section 13.2.3 [Changing the Engine Decoding State], page 114 for more information).

There are three types of vocabularies:

- Command vocabularies are used to recognize words from a list created dynamically by the application in response to user input or the current state of the application. They differ from dictation vocabularies in that the words are decoded based entirely on how they sound. When processing a command vocabulary, the engine waits for an explicit request before decoding and sending a recognized word to the application.
- Grammar vocabularies are used to recognize words and phrases contained in a compiled BNF grammar (this compiled SMAPI grammar is known as a Finite State Grammar (FSG) file). Grammar vocabularies are decoded based on the word-order syntax rules specified in the BNF. When processing a grammar vocabulary, the engine waits for an explicit request before decoding and sending a recognized phrase to the application.
- Dictation vocabularies are used for free-form speech-to-text translation. When decoding words from a dictation vocabulary, the engine uses the sound of the words as well as their context–the words preceding and following the current word– to determine what the speaker said. When processing a dictation vocabulary, the engine decodes

and sends words to the application without waiting for an explicit request for the next word.

To decode incoming words, the engine searches all the currently enabled vocabularies. Multiple vocabularies can be enabled at one time. To improve performance (speed and accuracy), the application should narrow the possibilities by enabling and disabling vocabularies as needed based on its current state.

13.2.1.1 Setting Up a Command Vocabulary

Command vocabularies are used to recognize words from a list created dynamically by the application in response to user input or the current state of the application. An application defines a command vocabulary using the SmDefineVocabEx function call, passing a list of the words which make up the vocabulary. This definition remains in effect for the duration of a recognition session, or until the application explicitly removes it by calling SmUndefineVocab. The words in a command vocabulary are assumed to already have pronunciations in the domain or among the user's personal vocabulary. If the engine does not have the pronunciations required to recognize a word or words, then a list of those words is returned to the application. In this case, the vocabulary is still defined, since the words in a command vocabulary are independent of each other. The words for which the engine does not have a pronunciation do not become part of this defined vocabulary.

The command vocabulary must be created and then enabled before the speech recognition engine can use it to decode speech. Rather than defining and undefining a vocabulary multiple times, it is more efficient to define a vocabulary once with SmDefineVocabEx and then enable and disable the vocabulary as needed using SmEnableVocab and SmDisableVocab. After a vocabulary has been defined, use SmAddToVocab, and SmRemoveFromVocab to modify it. SmDefineVocabEx requires that a name be specified for the vocabulary. This name is an alphanumeric character string defined by the application, and it is used to identify the vocabulary in all subsequent vocabulary-related function calls and in SM_RECOGNIZED_WORD messages sent by the engine.

Use the SM_VOCWORD structure for all information pertinent to a vocabulary word when performing such actions as defining the vocabulary, or adding, removing, or retrieving a word from a vocabulary (refer to "Data Types" in the API Reference for a description of the SM_VOCWORD data structure).

The following code sample shows how to set up a dynamic command vocabulary:

```
char *apszCmds[] =  "open","close","copy","move",
                    "delete","find","select";
int iNumWords = (sizeof(apszCmds)/sizeof(char *));
SM_VOCWORD **pVocPtrs;           // Dynamic array of pointers to
                                 // vocabulary structures.
SM_MSG        reply;
int           i, rc;

// Allocate array of pointer to the vocab word structs
pVocPtrs = malloc(sizeof(SM_VOCWORD *) * iNumWords);

// Allocate each vocabulary word structure, then fill it in.
for (i = 0; i < iNumWords; i++) {
    pVocPtrs[i] = malloc(sizeof(SM_VOCWORD));
    pVocPtrs[i]-> flags = 0; // Reserved value
    pVocPtrs[i]->spelling_size = strlen(apszCmds[i])+1;
    pVocPtrs[i]->spelling = apszCmds[i];
}

/*-------------------------------------------------------------------*/
/* Define the "SampleCmds" command vocabulary.                       */
/*-------------------------------------------------------------------*/
rc = SmDefineVocab("SampleCmds", // Name of vocabulary
                   iNumWords,    // Number of words in vocabulary
                   pVocPtrs,     // Array of word structures
                   &reply);      // Synchronous call
if (rc == SM_RC_OK) {
  rc = SmEnableVocab( "SampleCmds", // Name of vocabulary
                      &reply);      // Synchronous call
  if (rc != SM_RC_OK) {
     // Error processing goes here...
  }
}

// Free the memory - once a vocabulary is defined to the engine,
// local storage can be freed, since the engine keeps its
// own copy
for( i = 0; i < iNumWords; i++ ) {
  free(pVocPtrs[i]);
}

free (pVocPtrs);
```

13.2.1.2 Setting Up a Grammar Vocabulary (FSG)

Grammar vocabularies are used to recognize words and phrases contained in an FSG. An application defines a grammar vocabulary using the SmDefineGrammar function call, pass-

ing the path name of the FSG file. This definition remains in effect for the duration of a recognition session, or until the application explicitly removes it by calling SmUndefineGrammar.

The words in a grammar vocabulary are assumed to already have pronunciations in the domain or among the user's personal vocabulary. If the engine does not have the pronunciations required to recognize a word or words, then the grammar is not defined, and the engine returns a list of those words to the application. A grammar that contains words with missing pronunciations would be unpredictable (for example, if the missing word was the root to all possible sequences, nothing could be recognized). This is different from dynamic command vocabularies, where all words and phrases are inherently independent. With dynamic command vocabularies, if pronunciations are missing, there is no coupling to other words and phrases. The grammar effectively replaces the language model for dictation, defining the order in which words are recognized.

The grammar vocabulary must be defined and then enabled before the speech recognition engine can use it to decode speech. Rather than defining and undefining a vocabulary multiple times, it is more efficient to define a vocabulary once with SmDefineGrammar and then enable and disable the vocabulary as needed using SmEnableVocab and SmDisableVocab. SmDefineGrammar requires that a name be specified for the vocabulary. This name is an alphanumeric character string defined by the application, and it is used to identify the vocabulary in all subsequent vocabulary-related function calls and in SM_RECOGNIZED_PHRASE messages sent by the engine.

The following code sample shows how to set up a grammar vocabulary. Note that the full path for "sample.fsg" file would normally be specified. It needs to be accessible to the engine to extract the words and grammar network, and by the SMAPI library to extract the annotation data. SMAPI makes a local copy of the annotation data; this storage is freed when the function SmUndefineVocab is called.

```
SM_MSG        reply;
int           rc;

/*-------------------------------------------------------------------*/
/* Define the "SampleGrammar" grammar vocabulary.                    */
/*-------------------------------------------------------------------*/

rc = SmDefineGrammar("SampleGrammar", // Name of vocabulary
                     "sample.fsg",    // FSG file name
                     0,               // Vocabulary flags
                     &reply);         // Synchronous call

if (rc == SM_RC_OK) {
   rc = SmEnableVocab( "SampleGrammar", // Name of vocabulary
                       &reply);         // Synchronous call

   if (rc != SM_RC_OK) {
      // Error processing goes here...
   }
}
```

13.2.1.3 Setting Up a Grammar Vocabulary with External Lists

Grammar vocabularies have a capability that allows some of the words to be defined at runtime instead of being defined directly in the grammar itself. These additional words make up an external list. One use for an external list could be in an application that needs to read information from a database and still be able to recognize these words when the user speaks them. For complete details about writing grammars that contain external lists, refer to Chapter 4 [Grammars], page 17.

Defining and enabling a grammar vocabulary that contains an external list is essentially a combination of the processes required to define and enable command and grammar vocabularies. The first step is to define the list using SmDefineVocabEx. The vocabulary name specified on the SmDefineVocabEx call must match the name used for the list in the grammar itself. Once the list has been defined successfully, define and enable the grammar vocabulary as usual by calling SmDefineGrammar and SmEnableVocab.

Please note: The external list vocabulary should NOT be explicitly enabled or disabled. This is inherent in the enabling and disabling of the grammar containing the list. The SmAddToVocab and SmRemoveFromVocab calls can be used to dynamically change the content of the list.

The following is an example of defining an external list in a BNF:

```
<query> = Where is the nearest <food_type> restaurant
extern <food_type>
```

The code sample below illustrates the coding steps required to define and enable a grammar vocabulary that uses an external list.

```
// Words to be used in food_type list
char *apszCmds[] = {"italian", "seafood","indian","american","barbeque","thai",
                                    "fast food"};
int iNumWords = (sizeof(apszCmds)/sizeof(char *));
SM_VOCWORD **pVocPtrs;           // Dynamic array of pointers to
                                 // vocabulary structures.
SM_MSG        reply;
int           i, rc;
// Allocate array of pointer to the vocab word structs
pVocPtrs = malloc(sizeof(SM_VOCWORD *) * iNumWords);
// Allocate each vocabulary word structure, then fill it in.
for (i = 0; i < iNumWords; i++) {
   pVocPtrs[i] = malloc(sizeof(SM_VOCWORD));
   pVocPtrs[i]->flags = 0; // Reserved value
   pVocPtrs[i]->spelling_size = strlen(apszCmds[i])+1;
   pVocPtrs[i]->spelling = apszCmds[i];
}

/*-------------------------------------------------------------------*/
/* Define the "food_type" external list.                             */
/*-------------------------------------------------------------------*/
rc = SmDefineVocab("food_type",  // Name of external list
                   iNumWords,    // Number of words in vocabulary
                   pVocPtrs,     // Array of word structures
                   &reply);      // Synchronous call
if (rc == SM_RC_OK) {
   // Define the grammar that uses the external list
   rc = SmDefineGrammar("Restaurants",   // Name of vocabulary
                        "resttype.fsg",  // FSG file name
                        0,               // Vocabulary flags
                        &reply);         // Synchronous call
   if (rc == SM_RC_OK) {
      // Enable the grammar that uses the external list
      rc = SmEnableVocab( "Restaurants",   // Name of vocabulary
                          &reply);         // Synchronous call
      if (rc != SM_RC_OK) {
         // Error processing goes here...
      }
   }
}

// Free the memory for the food_type word list
for( i = 0; i < iNumWords; i++ ) {
   free(pVocPtrs[i]);
}

free (pVocPtrs);
```

13.2.1.4 Setting Up a Dictation Vocabulary

The application must enable a predefined dictation vocabulary before the engine can use it to decode dictated text. The SMAPI functions SmEnableVocab and SmDisableVocab control which vocabularies can be used by the speech recognition engine.

The application can use SmWordCorrection or SmAddToVocab to include personal words in a dictation vocabulary (refer to Section 13.2.5 [Improving Recognition by Updating Personal Data Files], page 115). The personal extensions added with SmWordCorrection persist across recognition sessions and can be removed with SmRemoveFromVocab.

The code sample below shows how to use SmEnableVocab to enable the ViaVoice predefined dictation vocabulary, called "text". Since dictation vocabularies are all predefined, there is no need for an explicit function call to define the vocabulary.

```
SM_MSG reply;
int rc;

/*-------------------------------------------------------------*/
/* Enable the predefined "text" vocabulary.                    */
/*-------------------------------------------------------------*/

rc = SmEnableVocab("text",       // Name of vocabulary
                   &reply);      // synchronous form

if (rc != SM_RC_OK) {
   // Error processing goes here...
}
```

13.2.2 Processing Speech Input

The bulk of the work that a speech-aware application performs involves getting the recognized text from the engine and determining what to do with it. The responsibilities of an application in this area are:

- Ensuring that the application has speech focus when it needs it. If the application needs to have speech focus whenever it has input focus, then it needs to call SmRequestFocus when it receives notification from Windows that it has input focus. For a complete discussion of issues related to speech focus, refer to "Speech Focus" on page 148.
- Managing and displaying current microphone state. Having a visual display that indicates the current microphone state is a good idea, and it makes the application easier to use. Also, a well-designed speech application responds to asynchronous requests to turn the microphone on or off.
- Keeping audio processing moving by handling recognized text messages and ensuring that an SmRecognizeNextWord call is pending while the engine decoder is running. For command or grammar vocabulary processing, this means calling SmRecognizeNextWord each time it receives an SM_RECOGNIZED_WORD or SM_RECOGNIZED_PHRASE message from the engine.

13.2.2.1 Vocabulary Processing

The speech recognition engine processes words from command and grammar vocabularies differently from the way it processes dictation vocabularies, and it sends the data to the application in different messages. In all cases, however, the engine uses the SM_WORD structure to communicate relevant information about each word to the application (refer to "Data Types" in the API Reference for a description of the SM_WORD data type).

When the engine finds the best match for a spoken word in a command vocabulary, the decoded word and the alternative choices made by the engine are stored in an array of SM_WORD structures and sent to the application in an SM_RECOGNIZED_WORD message. After sending the SM_RECOGNIZED_WORD message, the engine stops decoding until the application calls SmRecognizeNextWord. Use SmGetFirmWords to retrieve the recognized text from the message.

When the engine finds the best match for a spoken word in a grammar vocabulary, it creates an array of SM_WORD structures containing the decoded word and alternative choices and passes it to the application in an SM_RECOGNIZED_PHRASE message. After sending the SM_RECOGNIZED_PHRASE message, the engine stops decoding until the application calls SmRecognizeNextWord. Use SmGetFirmWords to retrieve the recognized text from the message.

When the engine finds the best match for a spoken word in a dictation vocabulary, it creates an array of SM_WORD structures for the firm words, and returns this array in an SM_RECOGNIZED_TEXT message. To improve accuracy during dictation, the engine uses the neighboring words to help in its selection of the best decoded word. After sending the SM_RECOGNIZED_TEXT message, the engine continues decoding, and sends additional, asynchronous SM_RECOGNIZED_TEXT messages as subsequent words are recognized. Use SmGetFirmWords to retrieve the recognized text from the message.

If a recognized word occurs in two or more vocabularies enabled at the same time, the engine selects the word from the more recently enabled command or grammar vocabulary. Command vocabularies always override dictation vocabularies. When a recognized word occurs in a command vocabulary and a dictation vocabulary that are enabled at the same time, the engine selects the command vocabulary word. As a result, if a command vocabulary with a single word (such as "Stop Dictation") is enabled during a dictation session, the user can indicate the end of the session with a spoken command.

If a word from a command vocabulary follows words from a dictation vocabulary, the engine sends an SM_RECOGNIZED_TEXT message before it sends the SM_RECOGNIZED_WORD message.

13.2.2.2 Handling Rejections

Occasionally, the engine cannot adequately match an input sound against any word in the defined vocabularies. This might be caused by an out-of-vocabulary command, or by extraneous noise or speech. The engine returns an indication of these events to the application, so that it can inform the user that some sound was processed, and it was rejected.

The rejection is conveyed through an SM_RECOGNIZED_WORD message with an empty word spelling string and an empty vocabulary name string. As with all the other SM_RECOGNIZED_WORD messages, the engine then halts, waits for any vocabulary state changes from the application, then continues on an SmRecognizeNextWord call.

Consequently, all applications need to define a handler for the SM_RECOGNIZED_WORD message (or SmRecognizedWordCallback), even if no dynamic command vocabularies are defined. This handler should, at a minimum, issue the SmRecognizeNextWord call when processing a rejection.

13.2.2.3 Command and Grammar Vocabulary Processing

The engine decodes command and grammar vocabulary utterances one at a time without using the context of surrounding words. The application explicitly controls speech recognition. It requests the next recognized word from the engine. The engine returns the word, then halts and waits for further instructions from the application.

The following is a summary of the command and grammar vocabulary processing:

1. To receive recognition results, the application must have speech focus, so it calls SmRequestFocus and waits for notification through an SM_GRANTED_FOCUS message, which also indicates the current microphone state (on or off).
2. The application calls SmMicOn (if it is not already on), and the engine begins processing audio data.
3. The application calls SmRecognizeNextWord.
4. The engine starts running, processing the audio stream until a word is recognized.
5. When the engine recognizes a word, it sends a message to the application. If the word is in a command vocabulary, the message is SM_RECOGNIZED_WORD. If the word is in a grammar vocabulary, the message is SM_RECOGNIZED_PHRASE.
6. The engine then stops decoding and waits for further instructions from the application. While waiting for further instructions from the application, the engine continues capturing audio.
7. The application extracts the recognized text from the message using SmGetFirmWords and determines what to do next. For dynamic command vocabularies, alternatives are provided along with the recognized word, and those alternatives can be extracted with the SmGetAlternates function. For grammar vocabularies, SmGetAnnotations can be used to extract annotation data for phrases.

   At this point, the application can change the set of potentially recognizable words to fit the current context by changing the currently enabled vocabularies. This is especially useful when a user traverses different menus, changing the set of potentially recognizable words with each menu command. The application can modify, enable, disable, define, and undefine several different vocabularies at this time.
8. Repeat steps 3 through 7 as often as necessary based on the user's actions.
9. When the application needs to stop recognition, it calls SmMicOff. The engine stops reading new data from the audio source, but there could still be audio data remaining to be processed. To ensure that it has handled all of the user's spoken input, the

application should continue calling SmRecognizeNextWord and processing the returned words.

10. When all the buffered audio data has been processed by the engine, it sends an SM_UTTERANCE_COMPLETED message to the application. This message does not change the engine state. Any pending SmRecognizeNextWord call is still valid after an SM_UTTERANCE_COMPLETED has been received. For more information about how the recognition engine processes audio input, refer to Section 13.2.6 [Processing Speech Engine Audio], page 116.

13.2.2.4 Command Recognition Events

The following table presents an overview of the sequence of command recognition events.

Event	User Says	Speech Application	Speech Recognition Engine
0		Calls SmDefineVocabEx for command vocabulary called "People". Calls SmDefineGrammar for grammar vocabulary called "Places".	Creates definition for the vocabularies, and marks them "disabled".
1		Calls SmEnableVocab for "People".	Marks "People" vocabulary "enabled"
2		Calls SmRequestFocus (synchronous).	
3			Assigns speech focus to application, and sends SM_FOCUS_GRANTED message.
4		Calls SmMicOn (synchronous).	Starts capturing and buffering audio input stream
5	"Smith"		
6	"Miami"		
7		Calls SmRecognizeNextWord (synchronous).	Starts speech-to-text decoding of audio input stream.
8			Recognizes "Smith". Sends SM_RECOGNIZED_WORD reply message loaded with "Smith" to application. Stops speech-to-text decoding of audio input stream.

9	Calls SmGetFirmWords to extract "Smith" from SM_RECOGNIZED_WORD. Determines that it needs to switch active vocabularies. Calls SmDisableVocab (synchronous) for "People" vocabulary.	Marks "People" vocabulary "disabled".
10	Calls SmEnableVocab (synchronous) for "Places" vocabulary.	Marks "Places" vocabulary "enabled".
11	Calls SmRecognizeNextWord (synchronous).	Starts speech-to-text decoding of audio input stream from last recognized word.
12		Recognizes "Miami". Sends SM_RECOGNIZED_PHRASE message loaded with "Miami" to the application. Stops speech-to-text decoding of audio input stream.
13	Calls SmGetFirmWords to extract "Miami" from SM_RECOGNIZED_PHRASE.	
N	Calls SmMicOff.	Terminates capture of audio input stream.
Last		When the entire audio input stream has been processed for recognized words, sends an SM_UTTERANCE_COMPLETED message to the application. Pending SmRecognizeNextWord is still valid.

Please note:

- The function calls are assumed to be successful.
- When function calls are made synchronously, the API waits until the associated reply message has been received from the engine, then copies it to the reply message pointed to by the last parameter of this call.

- N is a variable denoting the event number and it depends on whether the event is synchronous or asynchronous.

The following code sample illustrates the code required to initiate command or grammar recognition after vocabularies have been enabled.

```
SM_MSG reply;
int rc;

// Turn on microphone to capture audio input stream of user
rc = SmMicOn(&reply);                    // synchronous form

if (rc == SM_RC_OK) {                   // SmMicOn successful
   /*-------------------------------------------------------*/
   /* Decode audio input stream for first recognized word   */
   /* Note: SmRecognizeNextWord must be called for each word*/
   /*       in the audio input stream to be recognized      */
   /*-------------------------------------------------------*/
   rc = SmRecognizeNextWord(&reply);    // synchronous form

   if (rc != SM_RC_OK) {
      // Error processing goes here...
   }
}
```

The following code sample illustrates code on Windows to process messages from the speech recognition engine.

```
SM_MSG          sm_msg;
int             sm_msg_type, rc, i;
ULONG           num_firm_words;
SM_WORD         *firm_p;
switch ( msg ) {
  case WM_COMMAND:
    switch (wParam) {
      /** CONNECT_ID is an application-defined value set
       ** when the current session was established. **/
      case CONNECT_ID:
        SmReceiveMsg(lParam, &sm_msg);
        SmGetMsgType (sm_msg, &sm_msg_type);
        switch (sm_msg_type) {
          case SM_RECOGNIZED_WORD:
            SmGetRc(sm_msg, &rc);   // verify no error
            if (rc == SM_RC_OK) {
              // retrieve the firm words
              SmGetFirmWords(sm_msg, &num_firm_words, &firm_p);
            }
            return 0;
           case SM_RECOGNIZED_PHRASE:
            SmGetRc(sm_msg, &rc);        // verify no error
            if (rc == SM_RC_OK) {
              // retrieve the firm words
              SmGetFirmWords(sm_msg, &num_firm_words, &firm_p);
              // for all the firm words in the list
              for ( ; num_firm_words-- > 0; firm_p++) {
                // Process firm words here...
              }
            }
            return 0;
           case SM_UTTERANCE_COMPLETED:
            SmGetRc ( sm_msg, &rc );     // check for error
            if (rc != SM_RC_OK) {
              // Error processing goes here...
            }
            return 0;
           default:
            break;
        }
        break;
      default:
        break;
    }
    break;
    default:
    return (DefWindowProc(hwnd, msg, wParam, lParam));
}
```

13.2.2.5 Dictation Vocabulary Processing

When processing words from a dictation vocabulary, the engine uses a word-usage model, merging a score based on the context of the surrounding words with an acoustic score.

The dictation recognition process can be summarized as follows:

1. Call SmEnableVocab to enable a dictation vocabulary.
2. Call SmDefineVocabEx and SmEnableVocab to set up a command vocabulary containing a word that will allow the user to stop dictation using a spoken command.
3. Call SmRequestFocus to secure speech focus.
4. Call SmMicOn to start audio processing.
5. Call SmRecognizeNextWord to start the engine running, looking for words to decode.
6. If the engine recognizes a word from a dictation vocabulary, the available decoded words are sent to the application in a SM_RECOGNIZED_TEXT reply structure.
7. The SM_RECOGNIZED_TEXT reply structure provides a list of firm words.
8. Use the SmGetFirmWords reply structure access function to retrieve the list of words from the SM_RECOGNIZED_TEXT reply structure.
9. The engine continues decoding words as they are spoken. Refer to "Dictation Recognition Events" below for a "map" of the dictation decoding speech process.

   If the engine recognizes a word from a command vocabulary, the word is sent to the application in a SM_RECOGNIZED_WORD reply structure. The engine halts and waits for further instructions from the application.
10. Sometime after an SmMicOff call, the engine sends an unsolicited SM_UTTERANCE_COMPLETED message to indicate the audio stream has been processed. This message does not change the engine's state (running or halted). Any pending SmRecognizeNextWord call is still valid after an SM_UTTERANCE_COMPLETED has been received.

For more details about audio, refer to Section 13.2.6 [Processing Speech Engine Audio], page 116.

13.2.2.6 Dictation Recognition Events

The following table shows an overview of dictation recognition events:

Event	User Says	Speech Application	Speech Recognition Engine
1		Calls SmEnableVocab to enable the dictation vocabulary called "text." Calls SmEnableVocab to enable the command vocabulary containing "Stop Dictation." Calls SmDefineVocab to define the command vocabulary containing "Stop Dictation."	Halted
2		Calls SmRequestFocus to get the speech focus.	
3			Sends SM_FOCUS_GRANTED to application.
4		Calls SmMicOn to begin audio processing (synchronous).	Starts capturing audio input stream.
5		Calls SmRecognizeNextWord (synchronous).	Starts speech-to-text decoding of audio input stream.
6	"sales"		
7	"are"		
8	"up"		
9			Sends SM_RECOGNIZED_TEXT reply message loaded with the firm word "sales" to application.
10		Calls SmGetFirmWords to extract firm word "sales" from the SM_RECOGNIZED_TEXT message.	

11	"period"		
12	"Stop dictation"		
13			Recognizes "StopDictation" from command vocabulary. Sends SM_RECOGNIZED_TEXT reply message loaded with the firm words "are", "up", and "." to application. Sends SM_RECOGNIZED_WORD reply message loaded with the "StopDictation" to application. Speech-to-text decoding halts and engine waits for further instructions.
14		Calls SmGetFirmWords to extract firm words "are", "up", and "." from the SM_RECOGNIZED_TEXT message. Calls SmGetFirmWords to extract the command word "StopDictation" from the SM_RECOGNIZED_WORD message.	
15		Calls SmMicOff (synchronous).	Stops capturing audio input.
16		Calls SmRecognizeNextWord (synchronous) repeatedly to get all recognizable words from remaining audio input stream. Calls SmGetFirmWords to extract recognized words from each SM_RECOGNIZED_WORD message received.	Starts speech-to-text decoding with last recognized word. Sends SM_RECOGNIZED_WORD for each word found.

Last	The entire audio input stream has been processed for recognized words. Sends an unsolicited SM_UTTERANCE_COMPLETED reply message to the application. Any pending SmRecognizeNextWord is still valid.

Please notes:

- The function calls are assumed to be successful.
- When function calls are made synchronously, the API waits until the associated reply message has been received from the engine, then copies it to the reply message pointed to by the last parameter of this call.
- The precise sequence of events on this chart may not be reproducible.

The following code sample illustrates the statements required to initiate dictation recognition.

```
SM_MSG reply;
int rc;

/*-------------------------------------------------------------*/
/* Turn on microphone to capture audio input stream of user.   */
/*-------------------------------------------------------------*/
rc = SmMicOn (&reply);                // Synchronous form

if (rc == SM_RC_OK || rc == SM_RC_MIC_ALREADY_ON)
{
    // Start decoding audio to get recognized words
    rc = SmRecognizeNextWord(&reply); // Synchronous form

    if (rc != SM_RC_OK)
    {
        // Error processing goes here...
    }
}
```

The following code sample illustrates the code on Windows to handle messages from the speech recognition engine.

```
SM_MSG          sm_msg;
int             sm_msg_type, rc, i;
ULONG           num_firm_words;
SM_WORD         *firm_p;
switch (msg) {
  case WM_COMMAND:
    switch (wParam) {
      /** CONNECT_ID is an application-defined value set
       ** when the current session was established. **/
      case CONNECT_ID:
        SmReceiveMsg(lParam, &sm_msg);
        SmGetMsgType(sm_msg, &sm_msg_type);
        switch (sm_msg_type) {
          case SM_RECOGNIZED_TEXT:
            SmGetRc (sm_msg, &rc);   // verify no error
            if (rc == SM_RC_OK) {
              // retrieve the firm words
              SmGetFirmWords ( sm_msg, &num_firm_words, &firm_p );
              // for all the firm words in the list
              for (; num_firm_words-- > 0; firm_p++) {
                // Process firm word here...
              }
            }
            return 0;
          case SM_UTTERANCE_COMPLETED:
            SmGetRc ( sm_msg, &rc );     // check for error
            if ( rc != SM_RC_OK ) {
              // Error processing here...
            }
            return 0;
          default:
            break;
        }
        break;
       default:
        break;
    }
    break;
  default:
    return ( DefWindowProc ( hwnd, msg, wParam, lParam ) );
}
```

13.2.3 Changing the Engine Decoding State

The engine is always in one of two decoding states:

1. When the engine is looking for words to decode, the decoding state is running.
2. When it is waiting for further instructions from the application it is halted.

The engine state changes from halted to running when the application calls SmRecognizeNextWord. If the engine is already running when the application calls SmRecognizeNextWord, it continues running.

The engine state changes from running to halted when any of the following events occurs:

- The application calls SmHaltRecognizer.
- The engine recognizes a word in a command or grammar vocabulary, and sends an SM_RECOGNIZED_WORD or SM_RECOGNIZED_PHRASE message.
- The engine reaches an event synchronization point in the data stream and sends an SM_EVENT_SYNCH message with the halt option to the application. The application sets a halting event synchronization point by calling SmEventNotify with the SM_EVENT_HALT_RECOGNITION option.

Turning the microphone off does not change the engine state. The unsolicited SM_UTTERANCE_COMPLE sent to the application after an audio stream has been processed simply notifies the application that the boundary has been crossed. If the engine was running when the microphone was turned off, it continues decoding when the microphone is turned on again.

13.2.4 Setting/Querying Speech Engine Parameters

Use SmSet to set and fine-tune speech recognition engine and audio parameters (refer to Section 13.2.6 [Processing Speech Engine Audio], page 116). For example, set the SM_REJECTION_THRESHOLD parameter to achieve optimum recognition performance. Use SmQuery to retrieve the current values for engine and audio parameters.

Refer to "SmSet" and "SmQuery" in the SMAPI Reference for a complete list of engine parameters. On Windows the Properties program allows users to set various system parameters from Control Panel.

13.2.5 Improving Recognition by Updating Personal Data Files

Future recognition can be improved if the following personal data files are updated during recognition:

Initial Personal Voice Model
: A speaker-independent voice model is provided with ViaVoice. It can be updated if the user completes the enrollment process.

Text Vocabulary User Extension
: This file contains words that have been added to the dictation vocabulary of a domain. It is updated during word correction and macro creation. Call SmWordCorrection to update the vocabulary extension when the user corrects an incorrectly recognized word.

Word-Usage Model User Extension
: This file contains word-usage patterns derived from the user's correctly dictated text for the domain. It is updated when the user corrects a word. Call

SmWordCorrection to update the word-usage extension during a recognition session. This is valid only for dictation vocabularies.

Pronunciation Pool User Extension
: This file contains phonetic representations of the user's personal vocabulary for the appropriate language. It is updated during word correction and macro creation. Call SmWordCorrection to update the pronunciation extension when the user corrects an incorrectly recognized word. Call SmAddPronunciation to update the pronunciation extension when the user wishes to create a new pronunciation for a word.

Use these guidelines for updating a user's personal voice model:

- Call SmQueryAlternates to provide alternative choices when incorrectly recognized words are detected by the user during a recognition session.

Some notes about SmAddPronounciation:

- The SmAddPronunciation call requires an utterance. Create an utterance by turning the microphone on, speaking the word, and then turning the microphone off. The application must receive an SM_UTTERANCE_COMPLETED reply message before making the call.
- This call takes an optional argument that holds a phonetic spelling for the word. This argument indicates how the word is pronounced. Usually the phonetic (or "sounds-like") spelling for a word can be determined from its actual spelling. For example, "IBM" is pronounced "I B M" or "eye bee em". However, some words require that a phonetic spelling be provided; for example, "NCAA" can be pronounced "N-C-double-A". The engine might also need help with names such as "Gillian," which is pronounced "jill-e-an", not "gill-e-an".

13.2.6 Processing Speech Engine Audio

This section summarizes the audio processing that occurs between calls to SmMicOn and SmMicOff.

- Audio is sampled at 22 kHz and processed in real time.
- Up to 30 seconds of audio can be stored in an internal audio memory buffer.
- The audio saved to disk is used by the speech engine for playback and word correction. It is not recommended that applications directly manipulate these audio files. If an application has a requirement to use audio data captured during speech recognition it can make use of the SMAPI function SmPlayWords which allows playback to an audio file instead of an audio device. Refer to the SMAPI Reference for more information on the SmPlayWords function.
- Audio is saved to disk at 11KB per second.
- The SmSet SM_SAVE_AUDIO parameter controls whether or not the audio is saved to disk.
- Audio saved to disk can be freed with SmDiscardData.

While the SM_SAVE_AUDIO parameter specifies whether pulse code modulation (PCM) audio is saved to disk, it also includes all of the data required to do correction. This data includes tags and alternative words, as well as the processed audio data.

IMPORTANT:

SM_SAVE_AUDIO is false by default. The engine does not save session data by default because it is processor and disk intensive. To save all session data, an application must first call SmSet(SM_SAVE_AUDIO, TRUE, &reply). An application need not save all session data. If an application requires playback but not word correction only the files required for playback need to be saved. In this case the application calls SmSet (SM_SAVE_AUDIO, SM_SAVE_AUDIO_PLAYBACK, &reply). Refer to the SmSet function in the SMAPI Reference for more information on various audio saving settings.

SM_SAVE_AUDIO takes effect only at microphone on/off boundaries, so the application needs to toggle the microphone state appropriately. Note that all audio saving is turned off when focus is lost, so an application will need to restore audio saving settings after regaining focus.

These SmSet calls do not take effect until the next time the microphone is turned on, which means that you must do the following to change the PCM saving state:

1. Set the desired PCM state.
2. If the microphone is on, turn it off.
3. Process SM_RECOGNIZED_WORD and SM_RECOGNIZED_TEXT messages until SM_UTTERANCE_COMPLETED is received.
4. Turn the microphone on.

If the engine is running (meaning an SmRecognizeNextWord has been issued) and the microphone is turned off, the engine will automatically continue decoding when the microphone is turned on again (see also "Granting Focus" in Section 11.2 [Speech Focus], page 68).

Audio settings can be queried and set in all sessions.

13.2.7 Writing ViaVoice Applications to Save and Restore Speech Sessions

Saving and restoring speech sessions is necessary to support certain functions. You must save speech session data to support in-session correction (for example, tags must be saved if you want to play back words for correction). Deferred and delegated correction also require that speech session data be saved. In addition, the speech data must also be restored, since these features allow for recognized text to be corrected at a later time or, perhaps, on a different computer by a different user.

IBM SpeakPad is an example of an application that saves and restores speech data. It creates a VPS file, which includes both engine and application state data. The Microsoft Word dictation feature also saves application and engine state data in the DOC file normally created by Word.

The ViaVoice SDK SMAPI includes SmSaveSpeechData and SmRestoreSpeechData. These two SMAPI functions are used to save the audio (PCM) data required by the engine, along

with other ViaVoice engine state data required to restore the engine session. This other data includes tags that provide an index into the PCM audio data, processed audio data, and list(s) of alternative words. The processed audio data is used for adding words and is stored to prevent the need to reprocess the PCM audio data.

SmSaveSpeechData and SmRestoreSpeechData should be called asynchronously, as they can take a fairly long time to complete. The engine spawns a thread to handle the save/restore, and will continue to service requests from other speech-aware applications. To avoid changing the engine state underneath the save/restore, though, any additional API calls from the application session requesting the save/restore are blocked. They get a return code of SM_RC_NOT_VALID_REQUEST.

If you are writing a ViaVoice application that implements any kind of correction on a saved session, you must design and implement saving and restoring the application state yourself, along with using the SmSaveSpeechData and SmRestoreSpeechData SMAPI functions for the engine state. The items you must store yourself are:

- Recognized text
- Rich text data (such as font)
- User, enrollment, and task identification from SmConnect reply
- After an SmRecognizeNextWord call, word tags from SmGetFirmWords and most recent utterance number from SmGetUtteranceNumber
- Anything else it takes to restore the application state

13.2.8 Handling Speech Engine Errors

The speech recognition engine uses the SM_REPORT_ENGINE_ERROR reply message to notify the application of errors. This reply structure includes error severity, which can be extracted with SmGetSeverity, and a specific error code, which can be extracted with SmGetRc. (For more information, refer to Chapter 11 [Session Sharing], page 67.) The severity can be one of the following values:

SM_ENGINE_INFO
: This is purely an informational message and can be safely ignored. Applications that are logging information might want to add this to their log.

SM_ENGINE_WARNING
: The engine has detected an error that did not cause it to shut down (for example, input audio buffer overflow). Some of these errors cause the engine to turn off the microphone, so the application might need to turn it back on.

SM_ENGINE_ERROR
: The engine has detected a serious error and might not be able to resume normal operation.

SM_ENGINE_TERMINAL_ERROR
: The engine has suffered an unrecoverable failure and has been shut down. The application no longer has access to speech recognition, and must re-initialize its connection to the engine.

The following code sample shows how to process engine errors on Windows.

```
SM_MSG          sm_msg;
int             sm_msg_type, rc;
unsigned long   severity, micstate;
switch (msg) {
  case WM_COMMAND:
    switch ( wParam ) {
      /** CONNECT_ID is an application-defined value set
       ** when the current session was established. **/
      case CONNECT_ID:
        SmReceiveMsg ( lParam, &sm_msg );
        SmGetMsgType ( sm_msg, &sm_msg_type );
        switch (sm_msg_type) {
          // speech recognition engine error occurred
          case SM_REPORT_ENGINE_ERROR:
            // determine error severity
            SmGetSeverity (sm_msg, &severity);
            switch (severity) {
              // The engine has detected a non-terminal error
              // such as SM_RC_BAD_DECO or SM_RC_AUDIO_OVERRUN,
              // and has returned the state of the microphone
              case SM_ENGINE_WARNING:
                // Check if microphone was forced off by engine
                SmGetMicState ( sm_msg, &micstate );
                if ( micstate == SM_ENGINE_MIC_OFF ) {
                  // Turn the microphone back on...
                }
                return 0;
              // The engine has terminated because of an unrecoverable
              // error such as SM_RC_SERVER_TERMINATED
              case SM_ENGINE_TERMINAL_ERROR:
                // Error processing here...notify user.
                return 0;
              default: // Ignore other cases...
                return 0;
                break;
            } // switch severity
            break;
          default:
            break;
        } // switch sm_msg_type
        break;
      default:
        break;
    } // switch wparam
    break;
  default:
    return ( DefWindowProc ( hwnd, msg, wParam, lParam ) );
} // switch msg
```

13.2.9 Playing Audio through the Speakers

You can use the following SMAPI function calls to program your application to play audio through the speakers:

SmPlayMessage
: Plays a prerecorded audio file. This should be used sparingly. With multimedia machines, new applications should use the facilities provided by the operating system.

SmPlayUtterance
: Plays back audio captured between SmMicOn and SmMicOff calls. Requires SmSet(SM_SAVE_AUDIO, TRUE, &reply).

SmPlayWords
: Plays back one or more spoken words. This is typically used for word correction during dictation or enrollment. It is done on a per-tag basis. Requires SmSet(SM_SAVE_AUDIO, TRUE, &reply).

Please note:

The following restrictions apply to the SmPlay function calls:

- These calls can play only the audio files recorded by the recognition engine. They cannot be used to play other audio files.
- The microphone must be off, since the engine assumes a half-duplex audio system. For further information, refer to "SmMicOff" in the API Reference.
- The speech recognition engine must not be decoding speech to text. In other words, the application must have received SM_UTTERANCE_COMPLETED from the engine.

All of the SmPlay functions work in a similar manner:

1. The application calls the SmPlay function.
2. When the file begins to play, the engine sends an SM_PLAY_..._STATUS message containing a status value indicating that playback has begun successfully. Use SmGetStatus to retrieve the status value from the message.
3. When the file finishes playing, the engine sends another SM_PLAY_..._STATUS message indicating successful completion.
4. If there is an error at any time during this process, the engine sends an SM_PLAY_..._STATUS message containing an error code.
5. There is a new function that allows you to play audio to a file in a standard audio file format. Please refer to SmPlayWords in the SMAPI Reference Manual, for more information.

13.3 Termination Phase

During the termination phase, the speech-aware application disconnects from the speech recognition engine and closes the speech session.

13.3.1 Disconnecting from the Speech Engine

SmDisconnect breaks the connection between the application and the speech recognition engine. If SmClose has not been called, you can reconnect to the engine by calling SmConnect. The speech attribute values passed to SmOpen are still valid, but the ones passed to SmConnect are not. For information about how to set the speech attributes, see Section 13.1 [Initialization Phase], page 89.

When disconnecting from a recognition session, you can use speech attributes to specify whether to do one of the following:

- Discard or save data accumulated during dictation. You can discard the data to free disk space, or you can save the data to use for error reporting.
- Reset the user's personal voice model to its previous state.

An application can disconnect from the speech recognition engine with the following code fragment:

```
int rc;
SM_MSG reply;

/*-----------------------------------------------------------------*/
/* Disconnect from the speech recognition engine.                  */
/*-----------------------------------------------------------------*/

rc = SmDisconnect (0, NULL, &reply);
if (rc != SM_RC_OK) {
   // Error processing goes here...
}
```

13.3.2 Closing the Speech Session

Use SmClose to end the speech session. An application can close the speech session with the following code fragment. The internal connection structure, allocated and initialized with SmOpen, is freed with SmClose, and all the speech session attributes are purged.

```
int    rc;

/*-----------------------------------------------------------------*/
/* Close the speech session                                        */
/*-----------------------------------------------------------------*/
rc = SmClose ();
if ( rc != SM_RC_OK ) {
   // Error processing goes here...
}
```

14 Overview of the SMAPI Grammar Compiler API

The SMAPI Grammar Compiler API is a set of C-language functions that allows developers to compile grammars from within their applications. The Compiler API is intended for use in command and control applications written for ViaVoice. A good understanding of the ViaVoice Speech Manager API (SMAPI) is assumed.

The Compiler API is provided as a separate library from SMAPI.

To compile applications that use the Compiler API, include the header file, VTBNFC.H, which contains all of the necessary definitions. On Windows, be sure to include VTBNFC.H after the standard Windows include files. Also, depending on your build environment, you might need to set the INCLUDE environment variable to point to the path where VTBNFC.H is located, or otherwise specify this path (such as on the compiler command line.) Likewise, you will need to add VTBNFC31 to the link step of your product build.

If your application uses the Compiler APIs, you need to include the library with your application. The ViaVoice Run Time Kit includes VTBNFC31.

15 SMAPI Grammar Compiler Programming Tasks

The Compiler API offers functions to perform the following tasks:

- Setting up Compiler argument structures
- Compiling SMAPI grammars
- Handling compilation errors

An application may need to compile a grammar programmatically if the grammar cannot be determined completely when the application is developed (in other words, the grammar is determined at run time.) Dynamic vocabularies provide a similar capability, but are much less flexible than grammars compiled at run time.

For more information on the syntax and parameters of the individual Grammar Compiler API calls, refer to "Grammar Compiler API Function Calls" in the IBM SMAPI Reference.

15.1 Setting up SMAPI Grammar Compiler Argument Structures

The command-line version of the SMAPI Grammar Compiler supports several parameters. These parameters are passed to the VtCompileGrammar call in an argument structure (of type VtArg). Use VtAddArg and VtSetArg to set up the argument structure used by VtCompileGrammar.

The argument structure specifies the input BNF file and, optionally:

- The output FSG file
- Whether the non-uniform probability computation is to be used for the grammar
- The output FSG directory, if multiple roots are supported in the grammar
- Whether embedded silences and mumbles are allowed within the words of phrases in the grammar

For more information on Compiler parameters, refer to Chapter 5 [SMAPI Grammar Compiler], page 37.

15.2 Compiling Grammars

Use VtCompileGrammar to compile a grammar from within your application. VtCompileGrammar converts a BNF file to an FSG using the options specified in the argument structure.

15.3 Handling Compilation Errors

Use VtGetMessage whenever you receive a return code other than zero from VtCompileGrammar. VtGetMessage provides error and warning information that can be used to determine the cause of the error and also the appropriate action for your application to take.

The information returned by VtGetMessage is the same information returned by the command-line SMAPI Grammar Compiler.

16 Overview of the Custom Audio Library

The audio library functions must be packaged as a library, On Windows this is a Dynamically Linked Library (DLL), on Unix based systems it is a shared library. The implementation of the audio library functions are determined by the requirements of the application developer. The audio library will be loaded by the engine upon the first client connection. The engine will then resolve audio library exported function addresses. The audio library will be unloaded by the engine when the last client disconnects and a new client connects specifying a different userid, enrollid, taskid, or audio source specification from that with which the engine is currently initialized. The engine will also unload the audio library when the engine is terminating.

A client application sets the SmNaudioHost argument (using SmSetArg) before calling SmConnect to specify the audio source. Note that the argument values for SmConnect are preserved. This means after setting the Audio Host for a connection, all future connections from the same client will use the previously set Audio Host. Therefore, if the client needs to connect to the default audio library after connection to a custom Audio Host, the client must set the SmNaudioHost argument to an empty string (`""`) to reset the Audio Host argument to its default value.

The format for the audio source string is defined as:

```
type;dllname;key;init_str
```

This is a zero delimited string with semicolons separating the individual fields. The fields are defined:

type The type of audio data. Current valid values are pcm, cep. If not specified, default is pcm.

dllname The name of audio library to load. If not fully qualified, on Windows, the engine will attempt to load the library from the directory specified by the value of registry key

```
HKEY_LOCAL_MACHINE/SOFTWARE/IBM/VoiceType/Engine/Directories/Bin
```

(e.g., C:\ViaVoice\Bin). On Unix based systems the engine will load the audio library from the directory specified by the SPCH_BIN environment variable. If no library name is specified, on Windows the default is audmme.dll, on Unix based systems the default is audany.dll, except on Linux where the default is audlinux.so.

key It is now possible for multiple speech recognition engines to run on the same machine. This allows, for instance, one client application to connect to an engine performing live audio speech recognition, while another client application is connected to a different engine which gets audio data from a telephony card. The key is used to specify whether to connect to an existing engine or autostart another. The key must consist of alphanumeric characters and must not exceed 256 characters in length. If a key is not specified, the connect request will be sent to a running engine which was started without a key (or one will be autostarted if one is not running). If a key is specified, the connect request will be sent to a running engine started with this key (or one will be autostarted).

init_str Application data sent to audio library in first argument of AudioConnect function.

If the SmNaudioHost argument is not set before connecting, the default behavior is for the engine to process live pcm audio data.

Each buffer of audio data returned to the engine via the AudioGetPCM function must be divided into blocks. Each block must have a block header at the beginning of the block of length PCM_HEADER_LENGTH. Values for samples per block, block volume, total bytes per block and data bytes per block must be in this header. See SMAUDIO.H for header definitions.

The block headers should be created using these defines:

```
#define PCM_HEADER_LENGTH             26 /* Bytes in header,13 shorts */
#define PCM_HEADER_OFFSET_SAMPLES     2  /* Complete samples/block    */
#define PCM_HEADER_OFFSET_VOLUME      3  /* Block volume              */
#define PCM_HEADER_OFFSET_PCM_LENGTH  11 /* PCM bytes per block       */
#define PCM_HEADER_OFFSET_DATA_LENGTH 12 /* Data bytes per block      */
```

The header is PCM_HEADER_LENGTH bytes long, an array of 13 shorts. The other defines are indexes into that array. The PCM_HEADER_OFFSET_PCM_LENGTH and PCM_HEADER_OFFSET_DATA_LENGTH should hold the same values. This determines the size of the block and should be one of the following depending on the sampling rate:

```
#define PCM_BLOCK_LENGTH8  2020          /* 8 khz block size          */
#define PCM_BLOCK_LENGTH11 3030          /* 11 khz block size         */
#define PCM_BLOCK_LENGTH22 6102          /* 22 khz block size         */
```

The PCM_HEADER_OFFSET_SAMPLES should then be the number of samples in a block. With 16 bit pcm this would be PCM_HEADER_OFFSET_PCM_LENGTH / 2. The PCM_HEADER_OFFSET_VOLUME is the maximum absolute value of all pcm samples in the block.

17 Audio Library Functions

The following twenty functions, which are declared in SMAUDIO.H, define the interface between the engine and audio library. The first ten are required. If any of them are not resolvable after loading the audio library the connect request will fail with a return code of SM_RC_BAD_AUDIO. The other ten are optional. If any of them are not resolvable, a default handler will be used in its place. If the default handler is called, it will log a warning and return with a return code of SM_RC_NOT_VALID_REQUEST.

When developing an audio library in conjunction with an application it is strongly recommended that the audio library implement all optional audio functions which will be called as a result of SMAPI requests made by that application.

All return values must be defined in the SMRC.H file.

17.1 Required Functions

The following 10 functions must be implemented in an audio library.

AudioConnect Function
AudioCreate Function
AudioDestroy Function
AudioDisconnect Function
AudioGetPCM Function
AudioPutPCM Function
AudioStartPlayback Function
AudioStartRecording Function
AudioStopPlayback Function
AudioStopRecording Function

17.1.1 AudioConnect

Called as a result of an SmConnect SMAPI request. This function is used to establish a connection between the engine and the audio source.

```
int AudioConnect ( const char  *source,
                   const char  *client_name,
                   int         *handle,
                   int          sample_rate,
                   int          file_format,
                   int         *data_size,
                   int         *byte_order,
                   const char **signature,
                   char        *cepslan_data );
```

source Input - implementation dependent initialization data from init_str field of audio source string.

client_name
Input - Reserved

handle Output - Reserved, must return -1.

sample_rate
Input - Identifies sample rate determined by initialized domain.

file_format
Input - Identifies cep or pcm file format. Derived from type field of audio source string.

data_size Output - Returns block data size. Must be greater than 28 and less than 30000.

byte_order Output - Returns block byte order. Valid values are PCM_BYTEORDER_LOBYTE_FIRST and PCM_BYTEORDER_HIBYTE_FIRST (see SMAUDIO.H).

signature Output - Returns audio format signature which will be written to file header of output pcm files. Must be alphanumeric and no longer than 8 characters.

cepslan_data
Input - Reserved

Any non-zero return code returned by this function will be mapped to SM_RC_BAD_AUDIO and returned via the SmConnect reply.

17.1.2 AudioCreate

Called as a result of an SmConnect SMAPI request. Used for initialization or instantiation of audio class in an object design audio library. This function will be called before any other audio library function.

```
int AudioCreate ( const char *channel_name,
                  FILE       *file_handle,
                  int         audio_log_level );
```

channel_name
: Input - Reserved

file_handle Input - File pointer to system log file.

audio_log_level
: Input - Defines level of logging in audio library. Set with audio_log_level tag in audio stanza of engine.cfg.

Return code is not passed back to application through SMAPI.

17.1.3 AudioDestroy

Called as a result of an SmDisconnect SMAPI request. Used for termination. Audio class destructors can be invoked by this function in an object design audio library. No other audio library functions will be called after this one until a new SmConnect request is made and the AudioCreate function is called.

```
void AudioDestroy ( void );
```

17.1.4 AudioDisconnect

Called as a result of an SmDisconnect SMAPI request. This function will break an established connection between the engine and the audio source.

```
int AudioDisconnect ( void );
```

Return code is not passed back to application through SMAPI.

17.1.5 AudioGetPCM

Called repeatedly as a result of an SmMicOn request. This function will return audio data from the audio source to the engine.

```
int AudioGetPCM ( char  *block_buffer,
                  long   max_bytes,
                  long  *new_bytes,
                  int   *end_of_pcm );
```

block_buffer
: Input - Points to buffer for audio data.

max_bytes
: Input - Size of buffer for audio data.

new_bytes
: Output - Number of bytes of data returned in buffer.

end_of_pcm
: Output - Indicates if more data available. 0 = more data, 1 = no more data.

Return codes other than SM_RC_OK will be passed back to application via SM_REPORT_ENGINE_ERROR SMAPI message.

17.1.6 AudioPutPCM

Called repeatedly as a result of an SmPlayMessage, SmPlayUtterance, or SmPlayWords SMAPI request. This function will send audio data from the engine to the audio sink.

```
int AudioPutPCM ( char *block_buffer,
                  long  number_of_bytes );
```

block_buffer
: Input - Points to buffer containing audio data.

number_of_bytes
: Input - Specifies number of bytes of audio data.

Return code is not passed back to application through SMAPI.

17.1.7 AudioStartPlayback

Called as a result of an SmPlayMessage, SmPlayUtterance, or SmPlayWords SMAPI request. This function will begin the flow of audio data from the engine to the audio sink.

```
int AudioStartPlayback ( void );
```

Return code is not passed back to application through SMAPI.

17.1.8 AudioStartRecording

Called as a result of an SmMicOn SMAPI request. This function will begin the flow of audio data from the audio source to the engine.

```
int AudioStartRecording ( unsigned long *time_zero,
                          int            report_errors );
```

time_zero Output - Returns timestamp indicating when pcm begins to flow. This wall clock timestamp is the number of milliseconds since the operating system was started.

report_errors
Input - Indicates whether to report audio overruns.

Return code will be passed back to application via SmMicOn reply.

17.1.9 AudioStopPlayback

Called as a result of an SmCancelPlayback SMAPI request. This function will stop the flow of audio data from the engine to the audio sink.

```
int AudioStopPlayback ( int flush_output );
```

flush_output
Input - Indicates whether to flush playback buffers in the audio library.

Return code is not passed back to application through SMAPI.

17.1.10 AudioStopRecording

Called as a result of an SmMicOff SMAPI request. This function will stop the flow of audio data from the audio source to the engine.

```
int AudioStopRecording ( void );
```

Return code is not passed back to application through SMAPI.

17.2 Optional Functions

The following ten functions are not required but are supported by the engine to allow additional communication between an application (or the engine) and the audio library.

AudioGetHandle Function
AudioQueryConfig Function
AudioQueryDevices Function
AudioQuerySource Function
AudioSetDevice Function
AudioSetInput Function
AudioSetInputGain Function
AudioSetOutput Function
AudioSetOutputGain Function
AudioSetSource Function

17.2.1 AudioGetHandle

Called as a result of an SmConnect SMAPI request. This function is used to return an audio device handle to the engine.

```
int AudioGetHandle ( void );
```

returns audio handle via return value.

Return code is not passed back to application through SMAPI.

17.2.2 AudioQueryConfig

Called as a result of an SmQuery SMAPI request with an item value of SM_AUDIO_CONFIGURATION. This function is used to return a bit map representing the current audio configuration. (ie input mic/line, input gain, output speaker/line, output gain etc.)

```
int  AudioQueryConfig ( long  *configuration );
```

configuration

Output - Returns bitmap of audio device configuration. Definition is implementation dependent and is not used by engine.

Return code will be passed back to application via SmQuery reply.

17.2.3 AudioQueryDevices

Called as a result of an SmQuery SMAPI request with an item value of SM_AVAILABLE_AUDIO_DEVICES
This function is used to return a bit map of available audio devices in the system.

```
int AudioQueryDevices ( long *available_devices );
```

available_devices

Output - Returns bitmap of available devices. Definition is implementation dependent and is not used by engine.

Return code will be passed back to application via SmQuery reply.

17.2.4 AudioQuerySource

Called as a result of an SmQueryBinary SMAPI request with an item value of SM_AUDIO_SOURCE. This function is used to query the value of data passed to the audio library via SMAPI. It should return the value set via a call to AudioSetSource.

```
int AudioQuerySource ( short *length,
                       void **value );
```

length Output - Returns length of data.

value Output - Returns audio source data.

Return code will be passed back to application via SmQueryBinary reply.

17.2.5 AudioSetDevice

Called as a result of an SmSet SMAPI request with an item value of SM_AUDIO_DEVICE. This function is used to select an audio device for recording and playback.

```
int AudioSetDevice ( long device );
```

device Input - Identifies audio device to use. Definition is implementation dependent and is not used by engine.

Return code will be passed back to application via SmSet reply.

17.2.6 AudioSetInput

Called as a result of an SmSet SMAPI request with an item value of SM_AUDIO_INPUT_MODE. This function is used to specify how audio is input to the selected audio device.

```
int  AudioSetInput ( long input  );
```

input Input - Identifies instance audio input (i.e., mic, line left, line right, etc). Definition is implementation dependent and is not used by engine.

Return code will be passed back to application via SmSet reply.

17.2.7 AudioSetInputGain

Called as a result of an SmSet SMAPI request with an item value of SM_AUDIO_INPUT_GAIN. This function is used to set the input gain level.

```
int  AudioSetInputGain ( long  gain  );
```

gain Input - Identifies input gain. See smlimits.h for valid values.

Return code will be passed back to application via SmSet reply.

17.2.8 AudioSetOutput

Called as a result of an SmSet SMAPI request with an item value of SM_AUDIO_OUTPUT_MODE. This function is used to specify how audio is output from the selected audio device

```
int  AudioSetOutput ( long output );
```

output Input - Identifies audio output (i.e., speaker, line, etc)

Return code will be passed back to application via SmSet reply.

17.2.9 AudioSetOutputGain

Called as a result of an SmSet SMAPI request with an item value of SM_AUDIO_OUTPUT_GAIN. This function is used to set the output gain level.

```
int  AudioSetOutputGain ( long gain );
```

gain Input - Identifies output gain. See smlimits.h for valid values.

Return code will be passed back to application via SmSet reply.

17.2.10 AudioSetSource

Called as a result of an SmSetBinary SMAPI request with an item value of SM_AUDIO_SOURCE. This function is used to pass data to the audio library from SMAPI. This function can be used, for instance, to pass a file name to the audio library when performing file audio.

```
int AudioSetSource ( short length,
                     void *value );
```

length Input - Specifies length of data.

value Input - Points to audio source data.

Return code will be passed back to application via SmSetBinary reply.

Appendix A Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program or service is not intended to state or imply that only that IBM product, program, or service may be used.

Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service.

The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

```
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armand, NY 10504-1785
USA
```

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department T01B, 3039 Cornwallis, Research Triangle Park, NC 27709-2195, USA. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

A.1 Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

IBM

ViaVoice

VoiceType

Visual Age

Adobe Acrobat is a trademark or registered trademark of Adobe Systems Incorporated.

Intel and Pentium are trademarks or registered trademarks of Intel Corporation in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Other company, product, and service names may be trademarks or service marks of others.

Appendix B Glossary

This glossary defines terms and abbreviations used in this publication. If you do not find the term you are looking for, refer to the IBM Dictionary of Computing, SC20-1699.

This glossary includes terms and definitions from:

- The IBM Dictionary of Computing, New York: McGraw-Hill, copyright 1994 by International Business Machines Corporation. Copies may be purchased from McGraw-Hill or in bookstores.
- The American National Standard Dictionary for Information Systems, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Definitions are identified by the symbol (A) after the definition.
- The Information Technology Vocabulary, developed by the Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of the vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among participating National Bodies of SC1.

A

API Application program interface.

application

1. A practical use for an information processing system, such as a payroll application, an airline reservation application, or a network application.
2. A collection of software components used to perform specific types of work on a computer.

application program

1. A program that is specific to the solution of an application problem. Synonymous with "application software."(T)
2. A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll.
3. A program used to connect and communicate with stations in a network, enabling users to perform application-oriented activities.

application programming interface (API)

1. A functional interface, supplied by the operating system or by a separately licensed program, that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

2. The interface through which an application program interacts with an access method.

C

coexistence
The ability of multiple speech-aware applications to interact with the speech engine and the user on the same system.

command mode
The state of a speech-recognition system in which the user can speak commands. The system interprets recognized words as voice commands. Contrast with "dictation mode".

continuous speech
Speech spoken normally, during which the user does not pause between words, but instead speaks words at a natural pace. Contrast with "isolated speech" and "discrete speech".

customize To make a personal version of something, for example, a voice model.

D

dictation application
The window used to review and edit words recognized from dictation mode. The words can be transferred to other applications. In ViaVoice, this is the SpeakPad application.

dictation mode
The state of a speech-recognition system in which the user dictates text. The system interprets all decoded words (except dictation voice commands) as text. Contrast with "command mode".

discrete speech
Speech spoken with momentary pauses between words. Same as isolated speech. Contrast with "continuous speech".

domain

1. In ViaVoice, an industry-specific set of vocabularies which have an associated set of pronunciations. When the term task is used in function, attribute, and result code names, it is synonymous with domain.
2. A set of vocabularies and word-usage models designed to support a particular speech application. The ViaVoice product is shipped with a general office domain for each language that ViaVoice supports; additional domains are available.

domain ID
Domain identification. A short name (or identifier) that is uniquely associated with a domain.

dynamic vocabulary
: In ViaVoice, a vocabulary created for the duration of the speech engine connection. The words in a dynamic vocabulary already have pronunciations either in the domain or in the user's personal vocabulary.

E

enrollment
: The two-part process for identifying a person to the system. During the first part, the person specifies an enrollment identifier and speaks a set of predefined sentences into a microphone; these sentences are recorded. During the second part, the speech engine analyzes the recorded sentences and creates a unique personal voice model. This second part is also referred to as training.

enrollment ID
: Enrollment identification. A unique name (or identifier) associated with a person's enrollment and to the speaker model created from it.

enrollment script
: The set of predefined sentences that a person speaks into a microphone as part of enrollment.

enrollment session
: An invocation of the enrollment process, during which a user reads an enrollment script and the speech system records it.

F

firm word
: A word definitely recognized during dictation and selected by the speech engine as the best match for a spoken word. Contrast with "infirm word".

G

general office domain
: A set of 20,000 to 30,000 words and their pronunciations, representative of the office environment provided with ViaVoice. The general office domain is intended for general dictation purposes. There is a general office domain provided for each language supported by ViaVoice.

I

infirm word
: A word that has been tentatively recognized during dictation and selected by the speech engine as a preliminary match for a spoken word. The word could be changed by the language model when subsequent words are recognized. Contrast with "firm word".

isolated speech
: A speech recognition method that requires the user to pause briefly between each spoken word. Same as "discrete speech". Contrast with "continuous speech".

M

macro
: A "voice" shortcut. For example, instead of saying each element of your home address, you could define a dictation macro called "home-address" which, when spoken, would display your home address. ViaVoice supports macros for both command and control and dictation.

N

navigator session
: Special command-recognition speech session that can operate without acquiring speech focus. This permits the Navigator to handle commands for both speech-enabled and speech-aware applications.

notification
: Session-related (asynchronous) status messages generated by the speech engine and sent to subscribing applications.

P

parallel sessions
: A single speech-aware application has multiple connections to the speech engine. See also "shared sessions".

personal voice model
: The characteristics of an individual user's speech patterns.

phoneme
: A unit of sound distinguished by linguists and also found in pronunciation dictionaries. A phoneme has a variable duration of up to several seconds.

pronunciation
: In ViaVoice, a possible phonetic representation of a word, stored in the speech engine and referenced by one or more vocabulary words. A pronunciation is a string of phonemes (units of sound) that represents how a given word is pronounced. A word may have several pronunciations; for example, the word "tomato" may have pronunciations "toe-MAH-toe" and "toe-MAY-toe".

R

recognition
: In ViaVoice, when spoken words are understood by the speech system as text or commands.

S

shared sessions
When several speech-aware applications are connected to a single speech engine at the same time. For example, shared sessions enable IBM's navigator to work on the desktop at the same time as your speech-aware application. See also "parallel sessions".

speaker-dependent
A recognition technique in which the voice model is customized for a particular user's voice for best performance. Contrast with speaker-independent.

speaker-independent
A recognition technique in which the voice model is not customized for a particular user's voice. Contrast with speaker-dependent.

speech-aware application
An application that has been designed or modified to respond to voice, such as spoken commands or dictation. A speech-aware application directly accesses the ViaVoice API to process speech. See also "speech-enabled application".

speech-enabled application
An application that relies on a separate speech-aware application to process its speech input.

speech focus
In ViaVoice, the application that receives decoded audio from the engine.

speech engine
Part of ViaVoice; an application that performs all speech processing tasks and maintains the acoustic and word-usage models.

T

template A type of dictation macro which can contain predefined fields into which a user can dictate. Can be thought of as a form which can be filled in by speech.

training The second part of the enrollment process, during which the engine builds a personal voice model using the voice data collected during the first part of enrollment. See also "enrollment".

U

user ID User identification. A short name (or identifier) that is uniquely associated with a user of ViaVoice.

user model
In ViaVoice, the word-usage- and user-dependent parameters created during an enrollment session and used thereafter by the engine to recognize a user's voice.

utterance In ViaVoice, any stream of speech between two periods of silence.

V

vocabulary A set of words that can be made part of the active vocabulary. The words of a vocabulary often share a common word-usage model.

voice model
A mathematical model that describes how phonetic units are pronounced. Used by the recognition module of the speech engine. See also "speaker-dependent" and "speaker-independent".

W

window

1. A portion of the display surface used to present images that pertain to a particular application. Different applications can be displayed simultaneously in different windows.
2. An area of the screen, with visible boundaries, that displays information. A window can be smaller than, or the same size as, the screen.
3. A division of the screen in which one of several programs being executed concurrently can display information.

word-usage model
A database used by the speech engine to improve recognition performance during dictation. A word-usage model is composed of word sequences that occur with representative frequency in the written language. The speech engine uses these word sequences during the process of matching sounds to the speaker's vocabulary to increase the probability that a word is correctly recognized within the context of the dictation.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W