Speech-Enabling Visual Basic 6 Applications with Microsoft SAPI 5.1

Summary

Prerequisites

You need no previous experience with speech recognition, but you should understand fundamentals of Visual Basic®. The article and samples are written specifically for Microsoft Visual Basic 6.0. However, since SAPI 5.1 supports OLE Automation, any users of a language and a compiler supporting Automation standards (including but not limited to C# and JScript®) may find this article useful.

Difficulty Level

1 (Simple or beginner)

Summary Paragraph

Microsoft's new speech recognition technology called Speech Application Programming Interface (SAPI) vastly reduces the complexity of speech recognition (SR) or text-to-speech (TTS) programming. You can now speech-enable applications in as few as five lines. While the October 2000 release of SAPI 5.0 was written for C/C++ programmers, the summer 2001 release of SAPI 5.1 allows Visual Basic programmers access to speech technology by adding OLE Automation support. Now, your Visual Basic applications can use SR or TTS with the same simplicity that C/C++ developers enjoy.

Introduction

Microsoft Speech Application Interface (SAPI) brings speech recognition to the desktop in new and simple ways. The SAPI 5.0 version released in the fall of 2000 radically redesigned the product from the ground up. The new design introduces many advantages such as, faster and higher quality speech engines, engine independence for applications, and reliance on component object model (COM) technology. However, simplicity is the most important advantage in programming speech applications. C/C++ applications previously requiring 200 lines of code are now speech-enabled in as few as five lines. The recent release of SAPI 5.1 extends this simplicity to Visual Basic through OLE Automation. Visual Basic applications can add speech technology in only two lines of code.

Now that speech technology is no longer considered complex or esoteric, you are encouraged to add speech to your applications. This article discusses fundamental principles for speech recognition and, along with several examples, demonstrates the simplicity of programming with SAPI 5.1.

SAPI Overview

Speech programming has two general aspects: text-to-speech (TTS) and speech recognition (SR). TTS uses a synthesized voice to speak text. This enables applications to speak to you as well as

reading text and files. TTS extends your experience by supplementing existing visual feedback methods. This could include spoken verification of actions, alerting you to specific situations, or providing a screen reader.

Speech recognition (SR) converts human speech into text. This is further broken down into two types of SR: dictation, and command and control. Dictation is the traditional perception of speech recognition. The speech engine matches your words to its dictionary and converts them to text on the screen. The best example is this type of SR is dictating a letter where you have no restrictions on vocabulary. In contrast, command and control limits recognition to a smaller list of words and can associate specific words to specific actions. As an example, an application's menu bar may be speech-enabled. When you speak the word "file" or "save as," the File menu could drop down or the Save As window might appear. As a result, command and control is efficient and effective at processing commands. Command and control also extends the user experience by supplementing traditional input methods. Keeping with the menu example, you can speak the name of the menu or menu item to activate it rather than moving the mouse to a different part of the screen.

Figure 1 demonstrates the relationships between all components for speech. In general, SAPI is a set of three layers or components. The topmost layer represents your application and is the layer of most interest to developers. Your application initiates speech commands and ultimately the application receives the results of the other two layers. The second layer represents SAPI runtime files. As a system-level component, SAPI intercepts all speech commands and either processes the speech itself or forwards the commands to the next layer. Because the next layer SAPI is between the application and the speech engines, it is often called middleware. Finally the speech engines are below SAPI. Two special device drivers (collectively called speech engines) separate voice commands into either SR or TTS. When the respective engine processes the command, the result is passed back up to SAPI, which in turn, passes it on to the application.

Two sets of well-defined interfaces connect adjacent components. Between the application and SAPI run-time files is the application programming interface (also called an API). For Visual Basic developers, the Automation chapter in the SAPI SDK 5.1 documentation is important. Nearly 400 API methods and properties that compose the API are explained in detail. Between SAPI and the engines is the device driver interface (DDI). This DDI is intended for low-level access to engine information or even for writing your own speech engine. These layers in architecture allow for great flexibility in speech management in that each layer can be independent of the other two. The only restriction is that the connecting interfaces conform to Microsoft's SAPI standards. By conforming, each layer seamlessly interacts with another layer. As a result, the application designer does not need to know which manufacturer's engines have been installed. In short, SAPI is the conversion tool between applications and speech engines.

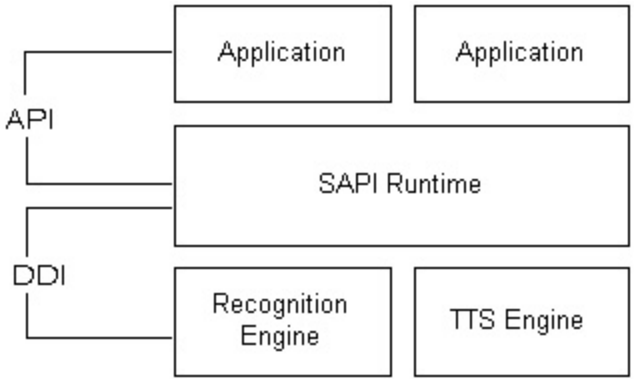

Figure 1: The architecture of SAPI 5.1 showing the three layers.

Getting Started

SAPI 5.1 is a system component and must be loaded prior to using speech operations. SAPI 5.1 is supported on Windows NT 4.0 or later version or on Windows 98 or later version. Although SAPI 5.1 may be installed through other means (see Deployment below), it is recommended that you load the complete SAPI software development kit (SDK) 5.1. It is available through the Microsoft Speech.NET web site (http://www.microsoft.com/speech). The SDK as well as redistributable installation files are provided at no cost. In addition to installing SAPI 5.1 run-time files and engines, the SDK is also installed. The SDK contains reference material for the methods and other programming support, examples in three languages supporting Automation (Visual Basic, JScript, and C#), and tutorials. Windows XP Professional or Home Editions include SAPI 5.1 but only install TTS. However, Microsoft Office XP uses SAPI 5.0 and cannot use Automation for speech. End users of your applications can install SAPI 5.1 in other ways; see Deployment below.

In order for you to hear the TTS voice, your computer needs speakers. SR requires an input device (microphone). A high quality head set is recommended for optimal speech recognition, although most microphones work adequately.

SAPI 5.0 and SAPI 5.1 generally use smart defaults during installation. That is, default values are assigned using the computer's existing values whenever possible. These defaults assign active speakers and microphones among other things. As a result, SAPI 5.1 is ready for immediate use after installation without explicitly setting or configuring any additional features. It is recommended that you spend at least 15 minutes training the computer. Training allows the speech recognizer to adapt to the sound of your voice, word pronunciation, accent, speaking manner, and even new or idiomatic words. The more training you do, the higher your recognition accuracy will be.

The examples and discussion in this article use Microsoft Visual Basic 6.0. However, you can use any language and compiler supporting Automation. In those cases, the principles of SAPI programming and design will be the same as those discussed with Visual Basic, although minor code modifications may be required. Visual Basic projects must link to the SAPI type library (or "reference" in Visual Basic terms). This reference file allows Visual Basic applications to access the SAPI 5.1 features and commands. Add the reference in Visual Basic using the Project->References menu item and select Microsoft Speech Object Library. Some computers may have other reference files with "speech" or "voice" in the name but these reference files are not related to SAPI 5.1. To use the examples below, create a new project each time, ensuring that the SAPI reference is added

to the project. Each sample describes the control items and their names needed for each form. Paste or type the sample in the code section of the project.

Text-to-Speech (TTS)

TTS is the simplest use of SAPI. The example below creates a synthesized voice which speaks text supplied to it, in this case a fixed string of "Hello, world." Create a project with a form without any controls and paste the sample into the code section.

```
Private Sub Form_Load()
    Dim MyVoice As SpVoice
    Set MyVoice = New SpVoice
    MyVoice.Speak "Hello, world"
End Sub
```

Although this is a small application of three lines, programmers not accustomed to object programming may notice three features. First, the variable MyVoice is explicitly declared but it is of type SpVoice. This type is unique to SAPI and it is dependent on the reference Microsoft Speech Object Library. Second, the object is instantiated with the keywords "set" and "new." This keyword combination not only allocates memory for the object, but also often fills in required values or properties. While the keywords are actually a Visual Basic mechanism and not related to SAPI, SAPI initializes as much information for a newly created object as possible. In many cases, SAPI objects do not require further initializing. As a result, the object MyVoice is ready for immediate use. The object sets the characteristics of the TTS voice. These characteristics include voice, speed, and pitch. Fortunately, most of SAPI's functionality using objects and using these keywords is common. The third and most interesting note is that the Speak method of MyVoice actually speaks the text. By creating a single object and issuing a Speak command, the application is now speech-enabled.

SAPI would be too restrictive if it spoke only hard-coded text. The next example allows greater speaking latitude by using a text box. Text may be typed or pasted into the text box and, when directed, it will be spoken. Create a project with a form having two controls: a text box named Text1, and a command button named Command1. Paste the sample into the code section, completely replacing any existing code. While the example is slightly larger than the previous one, there are still only two SAPI-related lines; the other lines accommodate Visual Basic.

```
Dim gMyVoice As SpVoice

Private Sub Form_Load()
    Set gMyVoice = New SpVoice
    TextField.Text = "Hello, world"
End Sub

Private Sub SpeakItBtn_Click()
    gMyVoice.Speak TextField.Text
End Sub
```

Speech Recognition (SR): Dictation

The next example introduces SR. It is a simple application (displaying your spoken comments in a text box) but introduces three fundamental SR concepts. To run the sample, create a project with a

form having a single text box control named Text1 Paste the sample into the code section, completely replacing any existing code. After loading and running the application, speak into the microphone and the text will display your spoken words.

```
Public WithEvents myRecognizer As SpSharedRecoContext
Public myGrammar As ISpeechRecoGrammar

Private Sub Form_Load()
    Set myRecognizer = New SpSharedRecoContext
    Set myGrammar = myRecognizer.CreateGrammar
    myGrammar.DictationLoad
    myGrammar.DictationSetState SGDSActive
End Sub

Private Sub myRecognizer_Recognition(ByVal StreamNumber As Long, ByVal StreamPosition
As Variant, ByVal RecognitionType As SpeechLib.SpeechRecognitionType, ByVal Result As
SpeechLib.ISpeechRecoResult)
    Text1.Text = Result.PhraseInfo.GetText
End Sub
```

The first concept for SR is that of the recognition context. In the example, the SpSharedRecoContext object of **`myRecognizer`** represents the recognition context. It is the primary means by which an application interacts with SAPI for speech recognition. A recognition context is an object that enables an application to start and stop recognition, and receive information back from both SAPI and the speech engine. In short, SR must be assigned to a specific recognition context. This sample creates only one recognition context and by default no restrictions are placed on the vocabulary. You may say almost any word, and to the best of the speech engine's capability, it will translate your words into text. Compare this to the command and control example shown later, where the list of words that can be recognized is limited. This difference helps to understand recognition contexts. Using a dictation application, you could say, "close" for instance, and no action will be taken other than translating it. The same word in a command and control scenario could actually initiate an action such as closing a window. In this regard, the word "close" has two different uses or contexts. A recognition context simply helps contain the speech.

The second concept is that of a grammar. A grammar is the set of words available for recognition. In either dictation or command and control, the grammar must be explicitly created and then loaded. For dictation, a default dictionary is available. This default dictionary allows users access to all the words in a language. The American English default dictionary, for example, contains about 65,000 words. A command and control grammar, on the other hand, contains a limited number of words. Its purpose is not to provide access to all the word in a language but rather to provide access a set of word that the application needs. A command and control grammar performs two functions. First, it customizes the word list. For example, the application may need to recognize only a single menu's worth of words such as *open*, *close*, or *exit*. In this case, requiring 65,000 words wastes computer processing time and memory. Second, the grammar associates a word or phrase with a command. For instance, the grammar could associates the statements "open", "please open', and "I'd like to open" to the same label such as OpenFile. Regardless of the variation actually said, the application would receive only one event. The code sample above creates the dictation grammar object with **`CreateGrammar`**, and activates the dictionary with **`DictationSetState`**. This two-step process implies that applications can have more than one grammar loaded. In this way, several grammars may be loaded at one time (during application start up for instance) and switched as you change recognition contexts. However, only one grammar can be active at a time. Command and control

uses a similar mechanism for loading grammars. Words for a command and control grammar must exist in a list you create, and this word list must be stored. Commonly a file is used to store this list. However, the list can also be stored dynamically (in memory). For that reason, the command and control grammar must take an additional step and indicate which list to use and where the list is located.

The third important concept is that of events. Events inform the application of activity that might be significant to the application. For example, when you strike a key, or click the mouse, the operating system sends an event to the application which can take appropriate action. In Visual Basic, clicking a command button is a common event: you click the button and the application receives a button event. While the code for the command button looks like a normal function or method, in truth it is actually an event. In the same way, events may be generated from outside the application. The operating system may generate an event when incoming e-mail arrives, for instance. That event could be trapped by an application that might subsequently beep indicating arrived mail. This kind of external event is actually very common and is the basis of OLE Automation.

SAPI uses this eventing mechanism to send information back to the application. In the code sample above, when a successful recognition occurs, SAPI passes an event back to the application. The event's parameters contain information about the recognition. Specifically, the recognition's text information is in the parameter **`Result`** (of type **`ISpeechRecoResult)`**. Use the Object Browser to explore **`ISpeechRecoResult`**. Result itself is composed of four interfaces including phrase information (the actual words in the recognized phrase), recognition context data (since more than one recognition context could be available), the audio stream format, and timing such as when the sound started and ended in the audio stream. Since the information is not always obvious, the method **`GetText`** quickly retrieves recognition's text.

Although SAPI only has two interfaces returning events (one for SR and another for TTS), there is a suite of 28 events. These include starting and stopping recognition or speech attempts. There is even an event for viseme changes (notification that the word being spoken requires changing mouth positions if you are using animation). However, two commonly used events for SR include Recognition and FalseRecognition. Recognition indicates that the spoken phrase was matched to text and that the match has a high confidence rating; that is, the level of certainty that the spoken phrase was interpreted correctly. Clear enunciation and lack of interfering background noise generally produce a high confidence rating. FalseRecognition is just the opposite. The spoken phrase did not have a high enough confidence rating to be considered successfully matched. Even so, a FalseRecognition returns text most closely matching the phrase. In other words, a FalseRecognition is a best guess.

Speech Recognition (SR): Command and control

Command and control introduces slightly more complexity. See Figure 2 for the next code sample. The application displays the contents of a single menu; however, individual menu items cannot be accessed. To run the sample, create a project with a form containing a menu bar with a single menu named File. At least one menu item should be added under it. Paste the sample into the code section, completely replacing any existing code. After loading and running the application, speak the command "file." The menu, along with any menu items under it, is displayed.

As mentioned in the previous dictation discussion, command and control uses a list for recognition; words or phrases not in the list will not be recognized. Storing words to and retrieving words from this list represents an extra step needed by command and control. There are two types of command and control grammars: static and dynamic. A static grammar stores the words in a file, usually in XML format. There are many instances in which you know ahead of time what commands are available. A travel booking application, for instance, may always have "departure" and "arrival" cities; a coffee service application may have a fixed list of available coffee drinks. In these cases, a static list could be used. A dynamic grammar is created while the application is running. Listing 1 uses a dynamic list. Since menus can be created, deleted, or modified while the application is running, dynamic grammars make sense. You may not know before the application runs, which items will or will not be available.

As with dictation, all command and control grammars must be created and activated before using. Unlike dictation, the application must have a way to identify and label specific words or phrases. Command and control uses a rule-based grammar. This means, similar or related commands may be grouped or named as rules. It is possible to nest rules inside other ones. A high-level rule is a special case and is a rule not contained within any other rule. Nesting these rules helps organize content by making it logical and easier to maintain. For example, the coffee ordering application may have several top-level rules, such as for coffee drinks, soft drinks, or snack types. Furthermore, soft drinks itself may contain two rules for carbonated and non-carbonated drinks. For simplicity, the Listing 1 sample creates one grammar (GrammarMenu), with one rule (menu) and one command (file).

Specifically, the following lines set up a dynamic grammar:

```
Dim TopRule As ISpeechGrammarRule
Set TopRule = GrammarMenu.Rules.Add("MENU", SRATopLevel Or SRADynamic, 1)
TopRule.InitialState.AddWordTransition Nothing, "file"
GrammarMenu.Rules.Commit
GrammarMenu.CmdSetRuleState "MENU", SGDSActive
```

The second line actually creates the rule, declaring it as both a dynamic and a top-level rule. Individual commands are added with **`TopRule.InitialState.AddWordTransition`**. Another menu item could be added by repeating that line with "save" as the final parameter. The grammar is then notified of all the changes with the call **`GrammarMenu.Rules.Commit`**. After that call, the new commands may be used. The grammar is activated only after all changes are complete.

During a recognition attempt, the speech engine notifies SAPI of a possible recognition. SAPI tries to match the recognized word or phrase with a rule. For instance, if you said, "file," SAPI would check the open grammar to match the word. An unsuccessful match will not return a recognition event. If successfully matched, the recognition is passed back to the application. In the **`RecoContextTopMenu_Recognition`** event, the property **`Result.PhraseInfo.GrammarId`** contains the grammar associated with the word. The application now has enough information to complete the command since it recognizes the word spoken and which grammar it is associated with. Logical decisions such as If or Select statements can process the information.

Often recognition contexts assume a greater role when you use command and control. Each application in the two SR samples above has only one recognition context each. For dictation, all speech is applied to the text window, and the command and control example listens only for the

word “file.” Neither application is compelling by itself. However, you may want to combine the two. In this scenario, a second recognition context can be added. The function of the two grammars (dictation and the command and control for the menu) are, thus, very different. The design of the new application would need to incorporate two features, each of which is relatively simple to implement. The first is a mechanism switching between the two modes. It would not be satisfactory to dictate the word “file” resulting in the File menu opening unexpectedly. A simple solution is a button toggling between the two modes. The other consideration is to activate the other grammar when the new mode is selected. Only one grammar may be active at a time. It is important to note that more than one grammar may be open at a time, but not active. Keeping multiple grammars open saves time and effort from reloading a grammar (if it is stored in a file) or reconstructing it (if it is created dynamically).

Design Consideration

Consider the design of the speech features before including speech in an application. Early versions of speech-enabling attempted to convert the entire user interface, with predictably awkward results. In general, approach speech as one of many elements in the user experience. Speech enable an application where it makes sense. For example, since requirements for mouse or keyboard navigation in word processors are minimal, a word processor could be fully speech-enabled with only a minor mechanism needed to switch between dictation, and command and control modes. On the other hand, speech may be more successful if supplementing traditional user interface methods. To reduce the number the times you move the mouse away from the drawing area while using a drafting or drawing application, speech could be used to enter coordinates, change colors, or switch between windows. Lastly, because of the lag time inherent to all speech recognition processes, using voice for game commands requiring instantaneous results would be less effective; keyboard commands or mouse clicks produce faster responses. Yet, the same game may benefit from speech but for other commands.

Looking ahead to the next generation of computers, devices such as hand-held computers and smart phones will certainly have their own user interface requirements. Programmers and designers will be able to adopt speech in equally new and creative ways.

Deployment

After compilation, the application may need to be deployed on other computers. The target computers must have SAPI 5.1. There are two options for deploying SAPI 5.1. The first option is to write your own installer. This is usually the best option as it guarantees that you have all the required files. The SAPI SDK 5.1 describes setup requirements for an installer. It is recommended that you use a full-featured installation package, such as the Microsoft Installer, as the deployment package since SAPI SDK 5.1 requires merge modules. Merge modules allow you to selectively install SAPI components In addition, the merge modules install automatically on supported platforms; thus, overwriting or deleting files by mistake is unlikely. However, as a result of requiring merge modules, the Visual Basic Package and Deployment Wizard cannot be used. The second option is to use existing SAPI 5.1 installations. Windows XP Professional and Home Editions have SAPI 5.1 already installed, but only for TTS. If you require SR, you must explicitly load this feature from a custom installer. Office .NET will support SAPI 5.1 for both TTS and SR although its release date has not yet been announced.

Download Samples

The download files contain three examples: one each for TTS, SR dictation, and SR command and control. The samples are intended to run with Visual Basic 6. The samples need the SAPI SDK installed. To run properly the speech engines must be installed. While the code listed in this article is based on the samples, the samples are much more complete. They include detailed comments, thorough error checking, proper application termination, and they are more robust. You are encouraged to use and modify the samples as needed.

Conclusion

Last year's redesign of SAPI removed much of the complication, and hopefully, intimidation of speech programming. The intent is to make speech processing simple and convenient. As seen, TTS can be introduced in as few as two lines of code and SR in as few as five lines. In addition, SAPI is no longer limited to C/C++ programming. SAPI 5.1 is flexible enough to use different languages such as Visual Basic, JScript and the new C#. Your understanding of several concepts presented in this article helps create the speech foundation for your application. Certainly not all the SAPI concepts were discussed, and, of course, each concept has more details than presented here. SAPI is technically sophisticated to allow developers new and exciting speech possibilities. For instance, SAPI 5.1 integrates with telephony and Internet communication protocols, or custom audio objects can be written to address challenges not explicitly covered by existing means.

Using SAPI 5.1 as a foundation, Microsoft is creating the next generation speech applications. As part of Microsoft's new Speech .NET Technologies, Web applications may be speech-enabled. Realizing the proliferation of mobile devices such as Web-enabled phones or handheld PCs, the need for speech is even greater. Small keyboards, limiting screen size, or just the convenience of speaking commands all make compelling cases.

Figure 2

```
Dim Recognizer As SpSharedRecognizer
Dim WithEvents RecoContextTopMenu As SpSharedRecoContext

Dim GrammarMenu As ISpeechRecoGrammar

Public GID_MENU As Integer

Private Sub Form_Load()
    GID_MENU = 0

    Set Recognizer = New SpSharedRecognizer
    Set RecoContextTopMenu = Recognizer.CreateRecoContext

    Set GrammarMenu = RecoContextTopMenu.CreateGrammar(GID_MENU)
    GrammarMenu.DictationLoad
    Dim TopRule As ISpeechGrammarRule
    Set TopRule = GrammarMenu.Rules.Add("MENU", SRATopLevel Or SRADynamic, 1)
    TopRule.InitialState.AddWordTransition Nothing, "file"
    GrammarMenu.Rules.Commit
    GrammarMenu.CmdSetRuleState "MENU", SGDSActive
End Sub

Private Sub RecoContextTopMenu_Recognition(ByVal StreamNumber As Long, ByVal
StreamPosition As Variant, ByVal RecognitionType As SpeechRecognitionType, ByVal
Result As ISpeechRecoResult)
```

```
    HandleRecoEvent Result
End Sub

Public Sub HandleRecoEvent(Result As SpeechLib.ISpeechRecoResult)
    Dim strText As String

    Select Case Result.PhraseInfo.GrammarId
        Case GID_MENU
            Select Case Result.PhraseInfo.GetText
                Case "file"
                    SendKeys "%F"
            End Select
    End Select
End Sub
```