Speech Manager

This chapter describes the Speech Manager, the part of the Macintosh system software that provides a standardized method for Macintosh applications to generate synthesized speech.

You need to read this chapter if you want your application to be able to generate speech. For example, you may want your application to incorporate the capability to speak its dialog box messages to the user. A word-processing application might use the Speech Manager to implement a command that speaks a selected section of a document to the user. A multimedia application might use the Speech Manager to provide a narration of a QuickTime movie instead of including sampled-sound data on a movie track. Because sound samples can take up large amounts of room on disk, using text in place of sampled sound is extremely efficient.

If you are developing an application that needs only to generate speech from strings, then the information on speech contained in the chapter "Introduction to Sound on the Macintosh" in this book might be sufficient. If, however, you need to be able to manipulate the speech output or customize it to make it easier for your users to understand, you should read this chapter.

The Speech Manager is not available in all system software versions. It was introduced with the Macintosh computers with audio visual capabilities in the summer of 1993. It will continue to be incorporated into future versions of system software. You should use the `Gestalt` function to ensure that the speech services you need are available before calling them. See the discussion in the section "Checking for Speech Manager Capabilities" beginning on page 4-12 for details.

The Speech Manager and the Sound Manager adopt many of the same metaphors in the processes of sound production and speech generation. You should be aware that the Speech Manager's approach often differs in subtle but important ways from that of the Sound Manager. Reading the chapter "Sound Manager" in this book might help you to learn to use the Speech Manager, but it is not required.

Also, while the Speech Manager uses the Sound Manager, your application should not attempt to directly access any Sound Manager data structures used by the Speech Manager. Because the Speech Manager is likely to be a rapidly evolving portion of system software, relying on Speech Manager data structures not explicitly documented in this chapter is likely to pose compatibility problems for your application.

This chapter begins with an introduction to the speech generation process and then discusses how you can

- check for the availability of the Speech Manager
- create and dispose of speech channels
- generate speech with different voices
- obtain information about and change speech channel settings
- start and stop speech production
- synchronize speech production with other activities by using callback procedures
- embed Speech Manager commands within text to make it more understandable

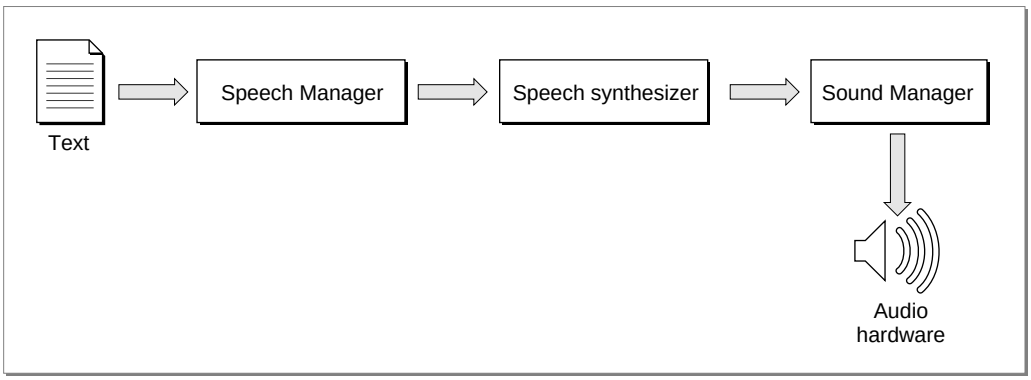
- convert text into phonemes and allow the user to enter phonetic text directly
- create, install, and manipulate customized pronunciation dictionaries

About the Speech Manager

You can use the Speech Manager to incorporate synthesized speech into your application. This section provides an overview and describes the basic concepts of the Speech Manager, and it outlines the process that the Speech Manager uses to convert text into speech. The Speech Manager converts text into sound data, which it passes to the Sound Manager to play through the current sound output device. The Speech Manager's interaction with the Sound Manager is transparent to your application, so you don't need to be familiar with the Sound Manager to take advantage of the Speech Manager's capabilities.

Figure 4-1 illustrates the speech generation process. Your application can initiate speech generation by passing a string or a buffer of text to the Speech Manager. The Speech Manager is responsible for sending the text to a **speech synthesizer,** a component that contains executable code that manages all communication between the Speech Manager and the Sound Manager. A synthesizer is usually contained in a resource in a file within the System Folder. A synthesizer is like a speech engine. It uses built-in dictionaries and pronunciation rules to help determine how to pronounce text. You can provide custom pronunciation dictionaries as described in the section "Including Pronunciation Dictionaries" beginning on page 4-36.

Figure 4-1 The speech generation process



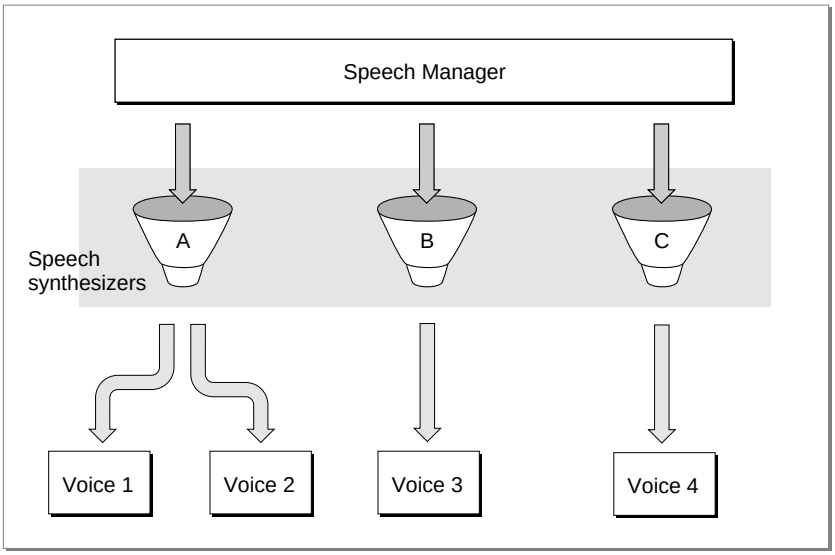
As Figure 4-1 suggests, the Speech Manager is simply a dispatch mechanism that allows your application to take advantage of the capabilities of whatever speech synthesizers, voices, and hardware are installed. The Speech Manager itself does not do any of the work of converting text into speech; it just provides a convenient programming interface that manages access to speech synthesizers and, indirectly, to the sound hardware. The

Speech Manager uses the Component Manager to access whatever speech synthesizers are available and allows applications to take maximum advantage of a computer's speech facilities without knowing what those facilities are. Because the Speech Manager's routines work on all voices and synthesizers, you will not need to rewrite your application to take advantage of improvements in speech technology.

Voices

Your application can use the system default voice to generate speech or it can specify that the Speech Manager use a particular voice that is available on the current computer system. A **voice** is a set of characteristics defined in parameters that specify a particular quality of speech. Just as different people's voices have different tonal qualities, so too can different voices have different qualities. A synthesized voice might sound male or female and might sound like an adult or a child. Some voices sound distinctively synthetic, while others sound more like real people. Figure 4-2 shows how the Speech Manager uses speech channels to synthesize speech with different voices.

Figure 4-2 The Speech Manager and multiple voices



As speech-synthesizing technology develops, the voices that your application can access are likely to sound more and more human. Each voice is designed to work with a particular speech synthesizer and can be customized in specific ways to create different effects.

Voices are usually stored in one of three places. The Speech Manager will first look in the application's resources file chain when attempting to locate a voice specification record. Then the Speech Manager will look in the System Folder and then the Extensions folder. Voices stored in the System Folder or Extensions folder are normally available to all

applications. Voices stored in the resource fork of an application file are private to that application and will not work if the synthesizers they depend on are not installed on a user's system.

Most of the time, your application designates the voice that speaks text, and usually that is the default voice. Based on the needs of your users and the way in which you expect them to use voices in your application you can provide access to voices in a number of different ways. You could include access to selecting voices in a dialog box that is available from a menu item such as Voices... Any application that allows users to choose among voices requires additional information about the available voices beyond the information provided by a voice specification record (described in detail on page 4-46), whose data should never be presented to the user. Such additional information might include the name of the voice as well as what script and language it supports.

Applications can use the `GetVoiceDescription` function (described in detail on page 4-66) with a voice specification record to obtain such information in a voice description record (described in detail on page 4-47). You might provide access to voices through a control panel. For information about implementing control panels, see *Inside Macintosh: More Macintosh Toolbox*. Or, you could implement a voices menu in your application's main menu bar, if you think that users will want to change the voice often and you have the room available. It's not a good idea to implement a hierarchical Voices menu since hierarchical menus are harder to use. For more information about choosing a user interface for your application, see *Macintosh Human Interface Guidelines*.

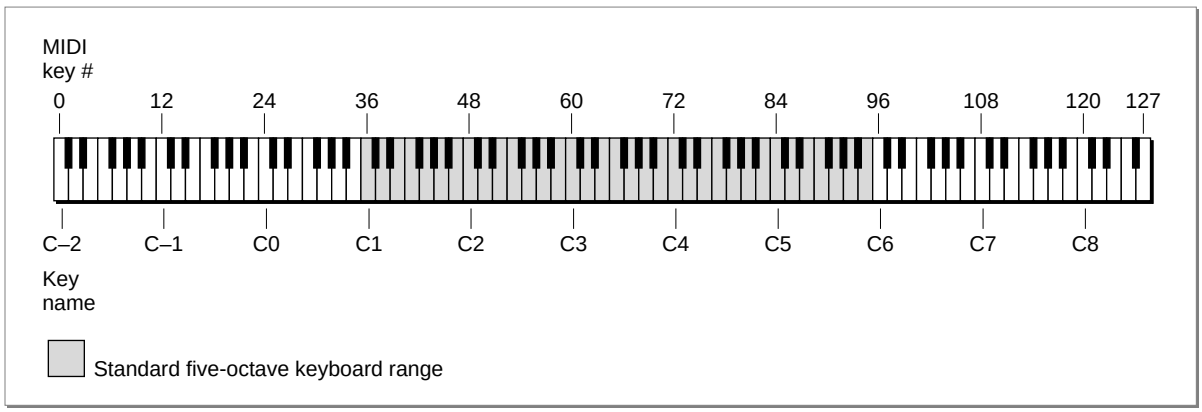
Speech Attributes

Any given person has only one voice, but can alter the characteristics of his or her speech in a number of different ways. For example, a person can speak slowly or quickly and with a low or a high pitch. Similarly, the Speech Manager provides routines that allow you to modify these and other speech attributes, regardless of which voice is in use. A **speech attribute** is a setting defined for a class of voices or for all voices that affects the quality of speech produced by the Speech Manager. The Speech Manager provides routines to directly alter two speech attributes—speech rate and speech pitch. These routines are described in the section "Changing Speech Attributes" beginning on page 4-73. You can change two other speech attributes—pitch modulation and speech volume—by using the mechanism of speech information selectors, which is described in the section "Speech Information Selectors" beginning on page 4-39.

The **speech rate** of a speech channel is the approximate number of words of text that the synthesizer should say in one minute. Slower speech rates make the speech easier to understand, but can be annoyingly tedious to listen to. Some applications, such as aids for the visually impaired, require very fast speech rates. Speech rates are expressed as fixed-point values. Each speech synthesizer determines it own range of speech rates. The **speech pitch** of a speech channel represents the middle pitch of the voice, roughly corresponding to the key in which a song is played. It is a fixed-point value in the range of 0.000 through 127.000, where 60.000 corresponds to middle C on a conventional piano. Each 1.000-unit change in a value corresponds to a musical half step. This is the same scale used in specifying MIDI note values, as described in the chapter "Sound Manager"

in this book. Figure 4-3 shows a piano keyboard with the corresponding MIDI note values.

Figure 4-3 MIDI note values and corresponding piano keys



MIDI note values differ from speech pitch values in that they are always integral and have a wider range than speech pitch values. On the scale used to measure both MIDI note values and speech pitches, a change of +12 units corresponds to doubling the frequency (an increase of one octave), while a change of –12 units corresponds to halving the frequency (a decrease of one octave). A **frequency** is a precise indication of the number of hertz of a sound wave at any instant. If you need to convert between speech pitches and hertz, note that a speech pitch of 60.000 corresponds to 261.625 Hz. Meanwhile, when a speech pitch value rises by one unit, the corresponding hertz value is multiplied by the twelfth root of 2, defined by the Sound Manager constant `twelfthRootTwo`. The following formula thus converts a speech pitch into hertz:

hertz = `twelfthRootTwo` $^{(\text{pitch} - 60.000)}$ * 261.625

In order to calculate speech pitch in terms of hertz, you can use the following formula:

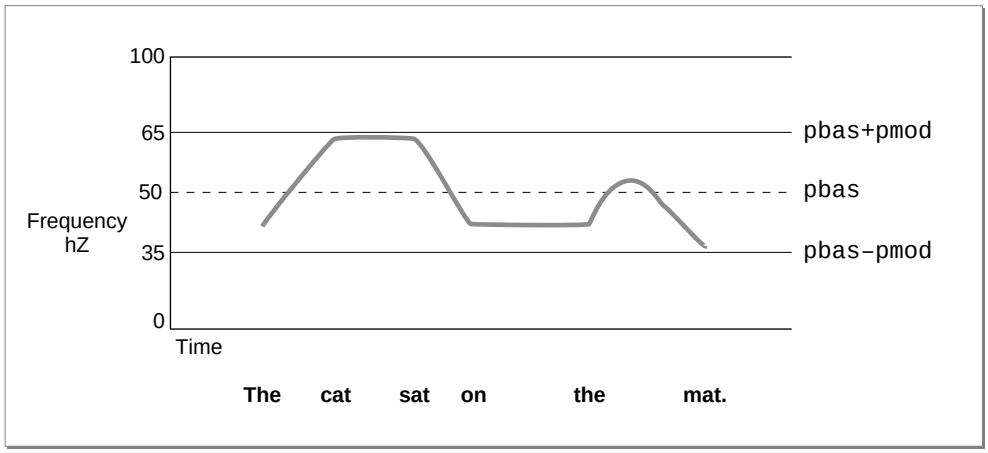
pitch = 60 + (ln(hertz) – ln(261.625)) / ln(`twelfthRootTwo`)

Typical voice frequencies might range from around 90 hertz for a low-pitched male voice to about 300 hertz for a high-pitched child's voice. These frequencies correspond to approximate pitch values in the ranges of 30.000 to 40.000 and 55.000 to 65.000, respectively.

You can determine the current speech pitch on a speech channel by calling the `GetSpeechPitch` function, described on page 4-75. You can change the current pitch by calling the `SetSpeechPitch` function, described on page 4-76. You can also determine the current speech rate and change it by using the `GetSpeechRate` function, described on page 4-73 and the `SetSpeechRate` function, described on page 4-74. Changes in speech pitch and speech rate are effective immediately (as soon as the synthesizer can respond), even if they occur in the middle of a word.

Pitch is the listener's subjective interpretation of speech's average frequency. The speech pitch specified is a baseline value corresponding to a particular frequency, from which the actual frequency of generated speech varies with the rises and falls of the intonation of speech. When a person speaks, there is a tune to the speech. Often you are more aware of the singsong quality, or change in the range of speech pitch, of a language that you don't know rather than one that you speak. The synthesizer must generate this tune in order to sound more human-like. Speech pitch is always described by a set of numbers that specify the range of pitch of the tune a synthesizer generates. This set of numbers can be the middle pitch and how far to deviate from that pitch or it can be the set of pitches within which the semi-tones of the tune can vary. Figure 4-4 shows an example of the range of pitches produced as the phrase "The cat sat on the mat." is spoken.

Figure 4-4 An example of pitch range for a voice



To simulate the variability in frequency of human speech, the Speech Manager defines the speech attribute of pitch modulation. The **pitch modulation** of a speech channel is the maximum amount by which the actual frequency of speech generated may deviate from the speech pitch.

Pitch modulation is also expressed as a fixed-point value in the range of 0.000 to 100.000. A pitch modulation value of 0.000 corresponds to a monotone in which all speech is generated at the frequency corresponding to the speech pitch. Speech generated at this pitch modulation would sound unnaturally robotic. Given a speech pitch value of 46.000, a pitch modulation of 2.000 would mean that the widest possible range of pitches corresponding to the actual frequency of generated text would be 44.000 to 48.000.

In some synthesizers, the actual pitch modulation may be restricted to a certain range. For example, if a synthesizer supported the full range of pitch modulations, a pitch modulation of 100.000 would result in unintelligible speech. In fact, however, some synthesizers, even with such a setting, produce speech that sounds virtually monotone. Even within a synthesizer, different voices might have different valid pitch modulation ranges. The Speech Manager provides no mechanism for obtaining the range of valid

pitch modulations, although some synthesizers may allow applications designed to work with those synthesizers to obtain such ranges.

You can obtain the pitch modulation by using the `GetSpeechInfo` function with the `soPitchMod` speech information selector, and you can change the pitch modulation by using the `SetSpeechInfo` function with the same selector. Speech information selectors are described in "Speech Information Selectors" beginning on page 4-39.

The **speech volume** of a speech channel is the average amplitude at which the channel generates speech. Volumes are expressed in fixed-point units ranging from 0.0 through 1.0. A value of 0.0 corresponds to silence, and a value of 1.0 corresponds to the maximum possible volume. Volume units lie on a scale that is linear with amplitude or voltage. A doubling of perceived loudness corresponds to a doubling of the volume.

Note that just as a speech synthesizer does not generate speech at a constant frequency, it does not generate speech at a constant amplitude. Even when the speech rate is high, brief pauses break up a steady stream of speech. The speech volume is, like speech pitch, an indicator of an average. There is no way to determine or change the modulation of speech volume.

A final speech attribute is **prosody,** the rhythm, modulation, and emphasis patterns of speech.There is no simple mechanism for your application to determine what rhythmic patterns a speech synthesizer is applying to speech. However, you can exert some control over prosody by using prosodic control symbols, discussed in "Prosodic Control Symbols" on page 4-34. Also, you can disable **ending prosody,** the modulation that distinguishes the end of a sentence or statement in normal speech, by using the `SpeakBuffer` function, described on page 4-57.

Speech Channels

To indicate to the Speech Manager which voice or attributes you would like it to use in generating speech, your application must use a speech channel. A **speech channel** is a data structure that the Speech Manager uses when processing text; it can be associated with a particular voice and particular speech attributes. Because multiple speech channels can coexist, your application can create several different vocal environments (to simulate a conversation, for example). Because a synthesizer can be associated with only one language and region, your application would need to create a separate speech channel to process each language in bilingual or multilingual text. (Currently, however, only English-producing synthesizers are available.)

Different speech channels can even generate speech simultaneously, subject to processor capabilities and Sound Manager limitations. This capability should be used with restraint, however, because it can be hard for the user to understand any speech when more than one channel is generating speech at a time. In general, your application should generate speech only at the specific request of the user and should allow the user to turn off speech output. At the very least, your application should include an option that allows the user to view text instead of hearing it. Some users might have trouble understanding speech generated by the Speech Manager, and others might have a

hearing deficit. Even users who are able to clearly understand computer-synthesized speech might prefer to read rather than hear.

Using the Speech Manager, you can identify how many voices are available and sort through an index of the voices to get information about a specified voice such as its gender, age, or the synthesizer with which it is associated. In general, your application does not need to know which speech synthesizer it is using, and in most cases, you do not need to be concerned with which speech synthesizer a voice is associated. Sometimes, however, a speech synthesizer may provide special capabilities beyond that provided by the Speech Manager. For example, a speech synthesizer might allow you to select an option to speak numbers in a nonstandard way. The Speech Manager allows you to determine which synthesizer is associated with a voice for these circumstances and provides hooks that allow your application to take advantage of synthesizer-specific capabilities.

In general, your application can achieve the best results by not making assumptions about which synthesizers might be available. The user of a 2 MB Macintosh Classic might use a synthesizer with low RAM requirements, while the user of a 20 MB Macintosh Quadra 950 might take advantage of a synthesizer that provides better audio quality at the expense of memory usage. The Speech Manager makes it easy to accommodate both kinds of users. Currently there are three synthesizers available with the Speech Manager. Each synthesizer has its own RAM requirements. To be compatible with all three synthesizers, you must reserve enough space in your application's heap to accommodate their requirements. In general, reserving around 250 KB per channel that you anticipate using provides enough space for the MacinTalk Pro synthesizer.

Callback Routines

The Speech Manager allows you to implement callback routines. With callback routines, you can synchronize speech with other actions. You can use callback routines to obtain information about when a synthesizer has finished speaking a phoneme, reaches a word ending, or finishes speaking. Using this feature, you could highlight text as it is being spoken or synchronize the speech production with a QuickTime movie or animation of a mouth speaking.

You can also customize speech that your application generates with the Speech Manager by embedding commands in text strings stored in resources in your application or by programmatically embedding commands in commonly spoken text.

The next section of this chapter shows you how to implement the most commonly used features of the Speech Manager. It demonstrates how you use the `SpeakString` function to convert a text string into speech without allocating a speech channel, how you can customize speech, how you can obtain more control over speech by allocating speech channels, and how you can make speech easier to understand by embedding commands within text strings. It also shows how to install a custom dictionary to provide more accurate pronunciation of less common words such as names.

Using the Speech Manager

You can use the Speech Manager simply to convert Pascal-style strings into speech. This simple technique is described in the chapter "Introduction to Sound on the Macintosh" in this book. This section shows how you can take advantage of more features of the Speech Manager.

Before you can generate synthetic speech on a Macintosh computer, you need to make sure that the Speech Manager is installed. "Checking for Speech Manager Capabilities" beginning on page 4-12 shows how to check for the availability of the Speech Manager. It also demonstrates how to use the `SpeakString` function to generate synthesized speech in the most straightforward way.

To take advantage of most of the Speech Manager's features, you must allocate a speech channel to pass to Speech Manager functions and dispose of the speech channel when you are finished using it. "Creating, Using, and Disposing of a Speech Channel" beginning on page 4-13 demonstrates how you do this and shows how you can use the `SpeakText` function to start speech generation from a buffer of text. Some applications permit users to choose a voice from those available to be used for speech generation. The `CountVoices`, `GetIndVoice`, and `GetVoiceDescription` functions support this capability. "Working With Different Voices" beginning on page 4-14 shows how you can use these functions to choose among available voices.

You can also use the `SpeakText` function to customize some attributes of speech generation. "Adjusting Speech Attributes" beginning on page 4-16 shows how you can do this. When you start synthesizing speech, you may need a way to stop speech from being generated. You can use the `StopSpeech` function to stop speech immediately, or you can use the `StopSpeechAt` function to choose exactly where you want speech stopped. You can stop speech temporarily and then resume it again using the `PauseSpeechAt` and `ContinueSpeech` functions. "Pausing Speech" beginning on page 4-18 shows how to pause or stop speech production and begin it again.

You might need to synchronize speech generation with other activities. For example, your application might include an on screen animation that must be synchronized with speech generation, or your application might need to determine when the Speech Manager has finished processing text on a speech channel so that it can unlock a handle or release some memory. "Implementing Callback Procedures" beginning on page 4-19 shows how you can accomplish these goals.

If your application uses embedded speech commands to obtain exacting control over speech generation, you should read "Writing Embedded Speech Commands" beginning on page 4-23. This section describes the complete syntax of embedded commands, and provides a guide to all embedded commands supported by the Speech Manager.

The Speech Manager allows you to enter phonemic text directly. If your application speaks only text that the user writes, this feature is unlikely to be useful to you, because you cannot anticipate what the user might enter. However, if there are a few or many sentences that your application frequently converts into speech, it might be useful to

represent parts of these sentences phonemically rather than textually. "Phonemic Representation of Speech" beginning on page 4-32 describes how to convert text to phonemes.

Some applications might allow the user to use pronunciation dictionaries to override the default pronunciations of certain words. "Including Pronunciation Dictionaries" beginning on page 4-36 explains how you can create a new pronunciation dictionary resource or install an existing pronunciation dictionary resource into a speech channel. The section also explains how you can provide the user with the default phonemic pronunciation of text by using the `TextToPhonemes` function.

Checking for Speech Manager Capabilities

Because the Speech Manager is not available in all system software versions, you should always check for speech capabilities before attempting to use them. Listing 4-1 defines a function that determines whether the Speech Manager is available.

Listing 4-1 Checking for speech generation capabilities

```
FUNCTION MySpeechMgrPresent: OSErr;
VAR
   myErr:      OSErr;
   myFeature:  LongInt;                          {feature being tested}
BEGIN
   {Test Speech Manager present bit.}
   myerr := Gestalt(gestaltSpeechAttr, myFeature);
   IF (myErr = noErr) AND (BTst(myFeature, gestaltSpeechMgrPresent)) THEN
   BEGIN
      myErr := SpeakString('The Speech Manager is working and');
      {Wait until synthesizer is done speaking.}
      WHILE (SpeechBusy <> 0) DO
      BEGIN                                      {do nothing}
      END;

      myErr := SpeakString('is almost done.');
      {Wait until synthesizer is done speaking.}
      WHILE (SpeechBusy <> 0) DO
      BEGIN                                      {do nothing}
      END;
      MySpeechMgrPresent := myErr;
   END;
END;
```

The `MySpeechMgrPresent` function defined in Listing 4-1 uses the `Gestalt` function to determine whether the Speech Manager is available. The `MySpeechMgrPresent`

function tests the `gestaltSpeechMgrPresent` bit, and, if the Speech Manager is present, the `MySpeechMgrPresent` function speaks the string passed to the `SpeakString` function. If the `Gestalt` function cannot obtain the desired information and returns a result code other than `noErr`, the `MySpeechMgrPresent` function assumes that the Speech Manager is not available.

The `SpeakString` function uses an implied speech channel, that is, the speech channel is automatically created and disposed of by the Speech Manager. The `SpeakString` function is useful when you need to synthesize Pascal-style strings of fewer than 256 characters. If you need to process text that is longer than 255 characters, then you must allocate a speech channel and use one of the routines that can generate speech in a channel such as the `SpeakText` or `SpeakBuffer` function. These routines are much more flexible in that they allow you to speak more text, customize the speech using speech selectors, or alter the generated speech by changing its modulation, pitch, rate, or voice.

Creating, Using, and Disposing of a Speech Channel

To take advantage of most of the Speech Manager's capabilities, you must pass a speech channel to Speech Manager functions. You use the `NewSpeechChannel` function to create a speech channel. After you are done using a speech channel, you must dispose of it by using the `DisposeSpeechChannel` function. Listing 4-2 shows how to create a speech channel, start speaking text with the `SpeakText` function, stop speaking text with the `StopSpeech` function, and then dispose of the speech channel when the speaking is finished.

Listing 4-2 Speaking text with a speech channel

```
FUNCTION MyUseSpeechChannel: OSErr;
VAR
   myErr:   OSErr;
   myErr2:  OSErr;
   myStr:   Str255;                                    {text to be spoken}
BEGIN
   myStr := 'Hold the mouse button down to stop speech.';
   myErr := NewSpeechChannel(NIL, gChannel);           {create the channel}
   IF (myErr = noErr) THEN
   BEGIN                                               {speak the string}
      myErr := SpeakText(gChannel, @myStr[1], Length(myStr));
      WHILE (SpeechBusy <> 0) DO                {wait until speaking is done}
      BEGIN
         IF (Button) THEN
         myErr := StopSpeech(gChannel);         {stop speech at mouse down}
      END;
      IF (gChannel <> NIL) THEN
```

```
        myErr2 := DisposeSpeechChannel(gChannel);{get rid of channel}
   END;
   IF (myErr = noErr) THEN
      MyUseSpeechChannel := myErr2
   ELSE
      MyUseSpeechChannel := myErr;
END;
```

The `MyUseSpeechChannel` function defined in Listing 4-2 creates a default speech channel using the default system voice. You pass `NIL` in the first parameter to use the system default voice. You must also pass a global variable to `NewSpeechChannel` in which is returned a valid speech channel. Once the channel exists, then you can use the `SpeakText` function to generate speech. To generate synthesized speech, you pass in the channel allocated by `NewSpeechChannel` in the first parameter, and then you pass a pointer to the text that you want to speak as well as the length of the text that you want the Speech Manager to attempt to speak. That is, you can pass a pointer to a buffer of text that is 500 bytes long, but specify that only the first 10 bytes get spoken. Then `MyUseSpeechChannel` uses the `SpeechBusy` function in a `WHILE` loop to allow the text to be completely spoken before disposing of the channel.

When the designated action to stop the speaking occurs, which in this example is the user pressing the mouse button, `MyUseSpeechChannel` halts speech production. In this case, the `StopSpeech` function stops the speech immediately (as soon as the synthesizer can). You need to pass `StopSpeech` the variable that identifies the channel on which the speech is currently being synthesized. If you want to have more control over when the speech is stopped, you can use the `StopSpeechAt` function, which allows you to stop speech immediately, at the end of a word, or at the end of a sentence. See the description of the `StopSpeechAt` function on page 4-60 for more information.

Once you are done using the speech channel that was created with `NewSpeechChannel`, you must dispose of it. The `MyUseSpeechChannel` function calls `DisposeSpeechChannel` with the global variable that identifies the channel currently in use.

Working With Different Voices

When you work with speech channels, you can set a voice for a particular channel. When you set a voice, you may want to filter out certain of its characteristics in order to identify the one you want. For example, in an educational software application for elementary school students, you may want to use only children's voices. In order to choose the voice you want, you get a **voice description record** that contains information about a voice such as the size of the voice, the name of the voice, the age and gender of the voice, and the synthesizer with which it works. You can get the number of available voices using the `CountVoices` function. You can cycle through the available voices and identify the one you want to use by using the `GetIndVoice` function. Then you fill out a voice description record using the `GetVoiceDescription` function. Listing 4-3 shows how to get identifying information about a voice.

Listing 4-3 Getting a description of a voice

```
FUNCTION MyInstallBoysVoice: OSErr;
VAR
   myErr:         OSErr;
   myIndex:       Integer;
   myNumVoices:   Integer;
   myVoice:       VoiceSpec;
   myFound:       VoiceSpec;
   myInfo:        VoiceDescription;
BEGIN
   myFound := NIL;
   myErr := CountVoices(myNumVoices);           {count voices}
   IF myErr = noErr THEN
   BEGIN
      FOR myIndex := 0 to myNumVoices DO        {loop through all voices}
      BEGIN
         myErr := GetIndVoice(myIndex, @myVoice);
         IF myErr = noErr THEN
         BEGIN
            myErr := GetVoiceDescription(@myVoice, @myInfo, sizeof(myInfo));
            IF myErr = noErr THEN               {check if a boy's voice}
               IF (myVoice.age < 16) AND (myVoice.gender = kMale) THEN
                  myFound := myVoice;
         END;
      END; {FOR}
      IF myFound <> NIL THEN                    {install boy's voice}
         myErr := NewSpeechChannel(@myFound, gChannel);
   END;
   MyInstallBoysVoice := myErr;                 {return result code}
END;
```

The `MyGetVoiceInfo` function checks to see how many voices are available. Once you have identified the list of available voices, you can index through the voices to select one about which you want to get information. You pass the number of the voice index in the first parameter of the `GetIndVoice` function. (This number cannot be larger than the number of voices.) `GetIndVoice` returns a **voice specification record** in the location specified in the second parameter— in this case, in the location of the pointer `@myVoice`. This sample cycles through the available voices looking for a male child's voice.

The voice specification record contains two identifiers: the creator identification of the required synthesizer and the voice identification of the voice.In order to get specific information about the voice you want to use, you need to call the `GetVoiceDescription` function. You need to pass a pointer to the voice specification record in the first parameter of the `GetVoiceDescription` function.

`GetVoiceDescription` returns the voice description record in the location pointed to in the second parameter, `@info`. The voice description record contains information about the voice such as its age or gender.

To specify which voice you want to use, you pass a pointer to the voice specification record as the first parameter to `NewSpeechChannel`. In this case, when the male child's voice is identified, it's voice specification record is passed to `NewSpeechChannel`, which allocates a channel with the specified voice. Note that this sample code contains limited error checking.

Adjusting Speech Attributes

Speech attributes are settings defined for a class of voices or for all voices that affect the quality of speech produced by the Speech Manager. In general, an application should not try to second-guess the developers of a voice or synthesizer by arbitrarily setting a speech attribute. However, there are some cases in which you would want to adjust the rate of speech (how many words per minute are spoken) or the speech pitch (the listener's subjective interpretation of speech's average frequency). Listing 4-4 shows how to adjust the speech pitch and speech rate of a particular channel.

Listing 4-4 Changing the speech rate and pitch

```
FUNCTION MyAdjustSpeechAttributes: OSErr;
VAR
   myErr:      OSErr;
   myErr2:     OSErr;
   myPitch:    Fixed;
   myRate:     Fixed;
   myStr:      Str255;
BEGIN
   myStr := 'This is the old pitch and rate.';
   myErr := NewSpeechChannel(NIL, gChannel);    {allocate a channel}
   IF myErr = noErr THEN
   BEGIN                                        {speak a string}
      myErr := SpeakText(gChannel, @myStr[1], Length(myStr));
      WHILE (SpeechBusy <> 0) DO                {wait for speech to finish}
      BEGIN
      END;
      {Find the current speech pitch.}
      myErr := GetSpeechPitch(gChannel, @myPitch);
      myPitch := myPitch * 2;                          {double the pitch}
      IF myErr = noErr THEN
         myErr := SetSpeechPitch(gChannel, myPitch);  {change the pitch}
```

```
      {Find the current speech rate.}
      IF myErr = noErr THEN
         myErr := GetSpeechRate(gChannel, @myRate);
      myRate := myRate * 2;                          {double the rate}
      IF myErr = noErr THEN
         myErr := SetSpeechRate(gChannel, myRate);    {change the rate}
      {Speak a string with new attributes.}
      myStr := 'This is the new pitch and rate.';
      myErr := SpeakText(gChannel, @myStr[1], Length(myStr));
      WHILE (SpeechBusy <> 0) DO          {wait for speech to finish}
      BEGIN
      END;
      {Dispose of the speech channel.}
      IF gChannel <> NIL THEN
         myErr2 := DisposeSpeechChannel(gChannel);
   END;
   IF myErr = noErr THEN
      MyAdjustSpeechAttributes := myErr2
   ELSE
      MyAdjustSpeechAttributes := myErr;
END;
```

The `MyAdjustSpeechAttributes` function first allocates a speech channel, as demonstrated previously. Then the `MyAdjustSpeechAttributes` function speaks a string to demonstrate the default speech rate and pitch for the default system voice. After the speech synthesis is finished, `MyAdjustSpeechAttributes` calls the `GetSpeechPitch` function with a valid speech channel and a pointer to a fixed-point value in which the value of the current speech pitch is returned. Then `MyAdjustSpeechAttributes` doubles the value of the speech pitch by multiplying and passes the new value to the `SetSpeechPitch` function.

`MyAdjustSpeechAttributes` repeats this sequence to determine the speech rate using the `GetSpeechRate` function, doubles the rate, and sets a new speech rate by passing the new rate value to the `SetSpeechRate` function. Next, `MyAdjustSpeechAttributes` calls `SpeakText` again to demonstrate the new speech pitch and rate. Creating a loop with the `SpeechBusy` function allows the synthesizer to finish speaking its text, and then `MyAdjustSpeechAttributes` disposes of the active channel.

When you set a rate value, each synthesizer may or may not be able to support that exact value. A synthesizer will attempt to set the value you specify, but it may substitute a value that it can support that is the closest it can come to your value. Don't be alarmed if `GetSpeechRate` returns a value other than the one you thought you set. The value returned is the closest value to the one set that the synthesizer is capable of reproducing.

Pausing Speech

When you start synthesizing speech, you may need a way to stop speech that is being generated. For example, your application might support a Stop Speech menu command to let users stop speech when they want to. Also, you should usually stop speech when you receive a suspend event. You can use `StopSpeech` to stop speech immediately, or you can use `StopSpeechAt` to choose exactly where you want speech stopped. You can also stop speech temporarily and then resume it again using the `PauseSpeechAt` and `ContinueSpeech` functions. Listing 4-5 shows how you might do this.

Listing 4-5 Pausing and continuing speech production

```
FUNCTION MyPauseAndContinueSpeech: OSErr;
VAR
   myErr, myErr2:    OSErr;
   myStr:            Str255;
BEGIN
   gChannel := NIL;
   myStr := 'Hold the mouse button down to test pause speech at immediate.';
   myErr := NewSpeechChannel(NIL, gChannel);    {open speech channel}
   IF myErr = noErr THEN
   BEGIN                                        {speak some text}
      myErr := SpeakText(gChannel, @myStr[1], Length(myStr));
      WHILE (SpeechBusy <> 0) DO                {wait for speech to finish}
         IF (Button) THEN
         BEGIN                                  {stop speech immediately}
            myErr := PauseSpeechAt(gChannel, kImmediate);
            IF myErr = noErr THEN
               WHILE (Button) DO    {while mouse button is down, do nothing}
               BEGIN
               END;                 {on mouse up, resume speaking}
               myErr := ContinueSpeech(gChannel);
      END;
      IF gChannel <> NIL THEN       {dispose of channel}
         myErr2 := DisposeSpeechChannel(gChannel);
   END;
   IF myErr = noErr THEN
      MyPauseAndContinueSpeech := myErr2
   ELSE
      MyPauseAndContinueSpeech := myErr;
END;
```

The `MyPauseAndContinueSpeech` function defined in Listing 4-5 begins by allocating a speech channel using the default system voice. It then begins to speak some text.

`MyPauseAndContinueSpeech` uses a busy loop to allow the speech to be completely spoken before finishing the subroutine. Then, when the designated action occurs, in this case the mouse button being depressed by a user, `MyPauseAndContinueSpeech` calls `PauseSpeechAt` with the currently active channel and a constant that defines where to stop the speech. This example uses the constant `kImmediate` to indicate that the speech should cease wherever it us currently being processed by the synthesizer. There are also constants that define the end of a word and the end of a sentence as appropriate stopping places.

When the mouse button is released, `MyPauseAndContinueSpeech` calls the `ContinueSpeech` function with the variable identifying the paused speech channel. When paused immediately, the synthesizer resumes speaking at the beginning of the word that was interrupted. While the speech is being generated, `MyPauseAndContinueSpeech` continues to call `SpeechBusy` to determine if the channel is still being used to process speech. When the channel is no longer busy, `MyPauseAndContinueSpeech` calls `DisposeSpeechChannel` to release the memory used by the speech channel.

Implementing Callback Procedures

The Speech Manager makes it easy for you to synchronize other activities to speech generation by allowing you to install various types of callback procedures on a speech channel. A **callback procedure** is a procedure that executes whenever a certain type of event is about to occur or has occurred. For example, you might use a word callback procedure to ensure that whenever the Speech Manager is about to speak a word, the word is visible onscreen. Callback procedures also allow you to synchronize more mundane activities with the Speech Manager; for example, you might need to know when you can dispose of a certain text buffer that you had asked the Speech Manager to speak. This section provides an overview of the different callback procedures that you can define.

The `soTextDoneCallBack` and `soSpeechDoneCallBack` speech information selectors allow you to designate text-done and speech-done callback procedures. A **text-done callback procedure** executes whenever the Speech Manager finishes processing a buffer of text to be spoken. This procedure usually executes before the Speech Manager has finished generating speech from the text and indeed often before it has started. The text-done callback procedure provides a mechanism that allows you to specify to the Speech Manager an additional buffer of text to be spoken, so that speech is generated continuously. Once your text-done callback procedure executes, you can release the memory occupied by the text buffer processed. A **speech-done callback procedure** does not execute until after the Speech Manager has completed generating speech from a buffer of text.

If your application uses or supports embedded speech commands, it may need to use the `soSyncCallBack` and `soErrorCallBack` speech information selectors to designate a synchronization callback procedure or an error callback procedure. A **synchronization callback procedure** executes whenever the Speech Manager encounters a synchronization command embedded within a text buffer to be spoken.

An **error callback procedure** executes whenever the Speech Manager encounters an error when attempting to process an embedded speech command. The Speech Manager passes information about the synchronization message or type of error to your callback procedure. If your application does not use synchronization or error callback procedures, it can obtain information about synchronization or error messages by continually polling the speech channel by using the `GetSpeechInfo` function with the `soErrors` or `soRecentSync` selectors.

The `soPhonemeCallBack` and `soWordCallBack` speech information selectors allow you to designate a phoneme callback procedure and a word callback procedure, respectively. A **phoneme callback procedure** executes whenever a phoneme is about to be spoken on a speech channel. A **word callback procedure** executes whenever a word is about to be spoken on a speech channel.

Since callback procedures execute at interrupt time they face several restrictions, as discussed in detail in *Inside Macintosh: Processes*. Most significantly, your callback procedure must not allocate or move memory or call any Toolbox or Operating System routine that might do so. Thus, typically a callback procedure simply sets a flag variable; for example, a phoneme callback procedure might change a variable that indicates which phoneme is being spoken. Your application can then poll this flag variable each time through its main event loop and perform whatever activity is desired if it finds that the flag variable has changed. Remember to design callback procedures to execute quickly.

Because they execute at interrupt time, callback procedures also cannot access application global variables unless the A5 register contains the value of the application's A5, as discussed in *Inside Macintosh: Memory*. Fortunately, the Speech Manager provides a mechanism that makes it easy to ensure that A5 is set correctly. Your application can call the `SetSpeechInfo` function with the `soCurrentA5` selector to pass the application's A5 in the `speechInfo` parameter to the Speech Manager. The Speech Manager will then set the A5 register to the passed value whenever it executes an application-defined callback procedure for that speech channel.

Sometimes your application might wish to provide a callback procedure with additional information beyond that which can be provided by examining application global variables. For example, a callback procedure might need to know from which document speech is being generated. Your application can use the `SetSpeechInfo` function with the `soRefCon` selector to specify a 4-byte reference constant value—for example, a handle to a document record—that the Speech Manager passes to all callback procedures on a particular speech channel. Your application can use the same callback procedure on multiple speech channels, for each of which the Speech Manager can pass a different value to the callback procedure. Thus, as long as your application never uses a single speech channel to generate speech on multiple documents simultaneously, it can use the reference constant value mechanism to pass document-specific information to a callback procedure. Typically, you use the reference constant to contain a pointer or handle to more extensive information that the callback procedure would require.

Listing 4-6 shows how you can indicate to the Speech Manager both the value to which it should set the A5 register when it executes a callback procedure on a particular speech channel and the reference constant value to pass to that callback procedure.

Listing 4-6 Setting up a speech channel for callbacks

```
FUNCTION MySetupCallbacks (chan: SpeechChannel; refCon: LongInt): OSErr;
VAR
   myA5:       LongInt;               {application's A5}
   myErr:      OSErr;
BEGIN
   myA5 := SetCurrentA5;              {get application's A5}

   {Pass A5 value to speech channel.}
   myErr := SetSpeechInfo(chan, soCurrentA5, Ptr(myA5));
   IF myErr = noErr THEN              {set the reference constant}
      myErr := SetSpeechInfo(chan, soRefCon, Ptr(refCon));

   MySetupCallbacks := myErr;
END;
```

The `MySetupCallbacks` function defined in Listing 4-6 uses the `SetSpeechInfo` function with both the `soCurrentA5` and the `soRefCon` selectors to prepare a specific speech channel for callbacks. Note that your application can call `MySetupCallbacks` as many times as desired for any particular speech channel; you might do this if you want to change the reference constant value to be passed to the speech channel.

Unlike other selectors, the `soCurrentA5` and `soRefCon` selectors do not require that you pass a pointer to the information you are specifying in the `speechInfo` parameter. Because an application's A5 value and a speech channel's reference constant value are always each 4 bytes long (the same size as the `speechInfo` parameter), your application passes these values directly, casting them to pointer values.

After your application sets up the A5 register and defines a reference constant value, it can install the appropriate type or types of callback procedure. Listing 4-7 shows how you might install a word callback procedure.

Listing 4-7 Installing a word callback procedure

```
PROCEDURE MyInstallWordCallback (chan: SpeechChannel; callbackProc: ProcPtr;
                                 refCon: LongInt);
VAR
   myErr:      OSErr;
BEGIN
   myErr := MySetupCallbacks(chan,  refCon);    {set up callbacks}
   myErr := SetSpeechInfo(chan, soWordCallBack, callbackProc);
   IF myErr <> noErr THEN
      DoError(myErr);                           {respond to an error}
END;
```

The `MyInstallWordCallback` procedure defined in Listing 4-7 first prepares for callbacks by calling the `MySetupCallbacks` function defined in Listing 4-6 for the speech channel and reference constant value specified by the `chan` and `refCon` parameters, respectively. Then it installs the callback procedure specified by the `callbackProc` parameter by using the `SetSpeechInfo` function with the `soWordCallBack` speech information selector. If, for example, you want to pass to your word callback procedure a pointer to the window containing the document being used for speech generation, you might call the `MyInstallWordCallback` procedure like this:

```
MyInstallWordCallback(mySpeechChan, @MyWordCallBack, LongInt(myWindow));
```

Listing 4-8 defines a simple word callback procedure.

Listing 4-8 A typical word callback procedure

```
PROCEDURE MyWordCallback (chan: SpeechChannel; refCon: LongInt;
                          wordPos: LongInt; wordLen: Integer);
BEGIN
   gWindowBeingRead := WindowPtr(refCon);
   gWordPos := wordPos;
   gWordLen := wordLen;
END;
```

▲ **WARNING**
Callback procedures are called at interrupt time and therefore must not attempt to allocate, move, or dispose of memory; dereference an unlocked handle; or call other routines that do so. Also, a callback procedure is a Pascal procedure and must preserve all registers other than A0–A1 and D0–D2. ▲

Because of the restrictions on callback procedures, a typical callback procedure usually just sets global flag variables based on the information passed to it. In Listing 4-8, the callback procedure copies information from the `refCon`, `wordPos`, and `wordLen` parameters to the three global variables `gWindowBeingRead`, `gWordPos`, and `gWordLen`. You can then call a routine to check the values of these global variables once each time through your application's event loop and respond appropriately if the `gWindowBeingRead` global variable is not `NIL`. (Your application would have to initialize the variable to `NIL`.) For example, the routine might ensure that the word about to be spoken is visible onscreen and scroll the document appropriately if it is not.

Although they have different uses, speech-done callback procedures, synchronization callback procedures, error callback procedures, and phoneme callback procedures are typically defined in ways similar to that of the word callback procedure in Listing 4-8. See "Application-Defined Routines" beginning on page 4-82 for complete information on callback routines.

Text-done callback procedures are usually more complex than the other types. You can use a text-done callback procedure simply to determine when the Speech Manager has completed processing a buffer of input text. The callback procedure can just set a global flag variable that is inspected once each time through the application's main event loop; when the flag variable indicates that the input buffer processing is complete, you can dispose of the input buffer.

Writing Embedded Speech Commands

Embedded speech commands allow you to customize the quality of speech output by fine tuning it. You can make speech much easier to understand than the default way in which text is spoken by a synthesizer. An **embedded speech command** is a command embedded within a text buffer to be spoken by the Speech Manager that causes the Speech Manager to take a certain action. For example, you could use an embedded speech command to emphasize a particular word in a text string to make it stand out to the user.

An advantage of this technique is that your application needs to call only the standard functions that generate speech: `SpeakString`, `SpeakText`, or `SpeakBuffer`. To change the way a phrase is generated, you do not need to change any of your application's code; you merely need to change the embedded command text. Your application can also use embedded speech commands even if it speaks text created by the user, as opposed to a limited set of phrases. Before passing text to the Speech Manager, your application could embed various commands within the text. For example, a word-processing application might embed commands that tell the Speech Manager to put extra emphasis around words that the user has boldfaced or underlined.

Embedded Command Delimiters

When processing input text data, speech synthesizers look for special sequences of characters called **command delimiters.** These character sequences are usually defined to be unusual pairings of printable characters that would not normally appear in the text. When a begin command delimiter string is encountered in the text, the following characters are assumed to contain one or more commands. The synthesizer will attempt to parse and process these commands until an end command delimiter string is encountered. By default, the begin command delimiter string is "`[[`", and the end command delimiter string is "`]]`". You can change the command delimiters if necessary, but you should be sure to use printable characters that are not in common use. Be sure to change the default delimiters back to the assigned characters when you are done with the speech processing for which you changed the delimiters. For example, if your application needs to speak text that naturally contains the default delimiter characters, then it should temporarily change the delimiters to sequences not included in the text. Or, if your application does not wish to support embedded speech commands, then it can disable such processing by setting both the begin command delimiter and the end command delimiter to 2 `NIL` bytes.

Syntax of Embedded Speech Commands

This section describes the syntax of embedded speech commands in detail. All embedded speech commands must be enclosed by the begin command delimiter and the end command delimiter, as follows:

```
[[emph +]]
```

All speech commands require parameters immediately following the speech command. The parameter to the speech emphasis command above is the plus sign. The format of the parameter depends on the command issued. Numeric type parameters include fixed-point numbers, bytes, integers, and 32-bit values. Hexadecimal numbers may be entered using either Pascal or C syntax; $1A22 and 0x1A22 are both acceptable.

A common type of parameter is an operating-system type parameter, used generally to specify a particular selector. For example,

```
[[inpt PHON]]
```

changes the text-processing mode so that the Speech Manager interprets text to be composed of phonemes.

Some commands allow you to specify an absolute value by including just a number as the parameter or to specify a relative value by adding a + or – character. For example, the following command raises the speech volume by 0.1:

```
[[volm +0.1]]
```

Your application can place multiple commands within a single set of delimiters by using semicolons–for example:

```
[[volm 0.3 ; rate 165]]
```

It is suggested that you precede all other embedded speech commands by a format version command. This command indicates to speech synthesizers the format version to be used by all subsequent embedded speech commands. The current format version is 1. You could write a format version command for the current format version like this:

```
[[vers $00000001]]
```

Table 4-1 provides a formalization of the embedded command syntax structure, subject to these conventions:

- Items enclosed in angle brackets (< and >) represent logical units that either are defined further below in the table or are atomic units that should be self-explanatory, in which case the explanations are provided in *italic* type. All logical units are listed in the first column.
- Items enclosed in single brackets ([and]) are optional.
- Items followed by an ellipsis (...) may be repeated one or more times.
- For items separated by a vertical bar (|), any one of the listed items may be used.

- Multiple space characters between tokens may be used if desired.
- Multiple commands within a single set of parameters should be separated by semicolons.

Table 4-1 The embedded command syntax structure

Identifier	Syntax
CommandBlock	**<***BeginDelimiter***> <***CommandList***> <***EndDelimiter***>**
BeginDelimiter	**<***String1***> | <***String2***>**
EndDelimiter	**<***String1***> | <***String2***>**
CommandList	**<***Command***> [; <***Command***>]...**
Command	**<***CommandSelector***> [***parameter***]...**
CommandSelector	**<***OSType***>**
Parameter	**<***OSType***> | <***String1***> | <***String2***> | <***StringN***> | <***FixedPointValue***> | <***32BitValue***> | <***16BitValue***> | <***8BitValue***>**
String1	**<***Character***>**
String2	**<***Character***> <***Character***>**
StringN	**[<***Character***>...]**
OSType	**<***Character***> <***Character***> <***Character***> <***Character***>**
32BitValue	**<***OSType***> | <***LongInt***> | <***HexLongInt***>**
16BitValue	**<***Integer***> |<***HexInteger***>**
8BitValue	**<***Byte***> | <***HexByte***>**
FixedPointValue	**<***Decimal number: 0.0000 ≤ N ≤ 65,535.9999***>**
LongInt	**<***Decimal number: 0 ≤ N ≤ 4,294,967,295***>**
HexLongInt	**<***Hex number: 0x00000000 ≤ N ≤ 0xFFFFFFFF***>**
Integer	**<***Decimal number: 0 ≤ N ≤ 65,535***>**
HexInteger	**<***Hex number: 0x0000 ≤ N ≤ 0xFFFF***>**
Character	**<***Any printable character (for example, A, b, *, # , x)***>**
Byte	**<***Decimal number: 0 ≤ N ≤ 255***>**
HexByte	**<***Hex number: 0x00 ≤ N ≤ 0xFF***>**

Table 4-2 outlines the set of currently defined embedded speech commands in alphabetical order and uses the same syntax conventions as Table 4-1. Note that when writing embedded speech commands, you omit the symbols like angle brackets and ellipses that are used here for explanatory purposes.

Table 4-2 Embedded speech commands

Command and selector	Command syntax and description
Character mode (`char`)	`char NORM | LTRL` The character mode command sets the word-speaking mode of the speech channel. When `NORM` mode is selected, the synthesizer attempts to automatically convert words into speech. This is the most basic function of the text-to-speech synthesizer. When `LTRL` mode is selected, the synthesizer speaks every word, number, and symbol character by character. Embedded command processing continues to function normally, however. This embedded speech command is analogous to the `soCharacterMode` speech information selector.
Comment (`cmnt`)	`cmnt [<`*Character*`>...]` The comment command is ignored by speech synthesizers. It enables a developer to insert a comment that will not be spoken into a text stream for documentation purposes. Note that all characters following the `cmnt` selector up to **<***EndDelimiter***>** are part of the comment.
Delimiter (`dlim`)	`dlim <`*BeginDelimiter*`> <`*EndDelimiter*`>` The delimiter command changes the character sequences that mark the beginning and end of all subsequent commands to the character sequences specified. The new delimiters take effect after the command list containing this command has been completely processed. If the delimiter strings are empty, an error is generated. This embedded speech command is analogous to the `soCommandDelimiter` speech information selector.
Emphasis (`emph`)	`emph + | -` The emphasis command causes the next word to be spoken with either greater emphasis or less emphasis than would normally be used. Using + will force added emphasis, while using – will force reduced emphasis. For an illustration of using the emphasis command, see the section "Examples of Embedded Speech Commands" beginning on page 4-30.

Table 4-2 Embedded speech commands (continued)

Command and selector	Command syntax and description
Input mode (`inpt`)	`inpt	TEXT	PHON`  The input mode command switches the input-processing mode to either normal text mode or phoneme mode. Passing `TEXT` sets the mode to text mode; passing `PHON` sets the mode to phoneme mode. Some speech synthesizers might define additional speech input mode selectors. In phoneme mode, characters are interpreted as representing phonemes, as described in "Phonemic Representation of Speech" on page 4-32.  This embedded speech command is analogous to the `soInputMode` speech information selector.
Number mode (`nmbr`)	`nmbr NORM	LTRL`  The number mode command sets the number-speaking mode of the speech synthesizer. When `NORM` mode is selected, the synthesizer attempts to automatically speak numeric strings as intelligently as possible. When `LTRL` mode is selected, numeric strings are spoken digit by digit. When the word-speaking mode is set to literal via the character mode command or the `soCharacterMode` speech information selector, numbers are spoken digit by digit regardless of the current number-speaking mode.  This embedded speech command is analogous to the `soNumberMode` speech information selector.
Baseline pitch (`pbas`)	`pbas [+	-] <`*FixedPointValue*`>` The baseline pitch command changes the current speech pitch for the speech channel to the fixed point value specified. If the pitch number is preceded by a + or – character, the speech pitch is adjusted relative to its current value. Base pitch values are always positive numbers in the range from 1.000 to 127.000. This embedded speech command is analogous to the `soPitchBase` speech information selector. For a discussion of speech pitch, see the section "Speech Attributes" beginning on page 4-6.

continued

Table 4-2 Embedded speech commands (continued)

Command and selector	Command syntax and description
Pitch modulation (`pmod`)	`pmod [+ | -] <`*FixedPointValue*`>` The pitch modulation command changes the modulation range for the speech channel based on the modulation depth fixed-point value specified. The actual pitch of generated speech might vary from the baseline pitch up or down as much as the modulation depth. If the modulation depth number is preceded by a + or – character, the pitch modulation is adjusted relative to its current value. Speech pitches fall in the range of 0.000 to 127.000. This embedded speech command is analogous to the `soPitchMod` speech information selector. For a discussion of speech pitch, see the section "Speech Attributes" beginning on page 4-6.
Speech rate (`rate`)	`rate [+ | -] <`*FixedPointValue*`>` The speech rate command sets the speech rate in words per minute on the speech channel to the fixed-point value specified. If the rate value is preceded by a + or – character, the speech rate is adjusted relative to its current value. Speech rates fall in the range 0.000 to 65535.999, which translate into 50 to 500 words per minute. Normal human speech rates are around 180 to 220 words per minute. This embedded speech command is analogous to the `soRate` speech information selector. For a discussion of speech rate, see the section "Speech Attributes" beginning on page 4-6.
Reset (`rset`)	`rset <`*32BitValue*`>` The reset command will reset the speech channel's voice and speech attributes back to default values. The parameter has no effect; it should be set to 0. This embedded speech command is analogous to the `soReset` speech information selector.
Silence (`slnc`)	`slnc <`*32BitValue*`>` The silence command causes the synthesizer to generate silence for the number of milliseconds specified. The timing of the silence will vary widely between synthesizers. For an illustration of using the silence command, see the section "Examples of Embedded Speech Commands" beginning on page 4-30.

Table 4-2 Embedded speech commands (continued)

Command and selector	Command syntax and description
Synchronization (`sync`)	`sync <`*32BitValue*`>` The synchronization command causes the application's synchronization callback procedure to be executed. The callback is made as the audio corresponding to the next word begins to sound. The callback procedure is passed the 32-bit value specified in the command. Synchronization callback procedures are described in "Synchronization Callback Procedure" beginning on page 4-85.
Format version (`vers`)	`vers <`*32BitValue*`>` The format version command informs the speech synthesizer of the format version that subsequent embedded speech commands will use. This command is optional but is recommended to ensure that embedded speech commands are compatible with all versions of the Speech Manager. The current format version is $0001.
Speech volume (`volm`)	`volm [+ | -] <`*FixedPointValue*`>` The speech volume command changes the speech volume on the speech channel to the fixed-point value specified. If the volume value is preceded by a + or – character, the speech volume is adjusted relative to its current value. Volumes are expressed in fixed-point units ranging from 0.000 through 1.000. A value of 0.0 corresponds to silence, and a value of 1.0 corresponds to the maximum possible volume. Volume units lie on a scale that is linear with amplitude or voltage. A doubling of perceived loudness corresponds to a doubling of the volume. This embedded speech command is analogous to the `soVolume` speech information selector.
Synthesizer-specific (`xtnd`)	`xtnd <`*OSType*`> [<`*Parameter*`>…]` The synthesizer-specific command enables synthesizer-specific commands to be embedded in the input text stream. Synthesizer-specific speech commands are processed by the speech synthesizer whose creator ID is specified in the first parameter and by other speech synthesizers that support commands aimed at the synthesizer with the specified creator ID. The format of the data following the parameter is entirely dependent on the synthesizer being used. This embedded speech command is analogous to the `soSynthExtension` speech information selector, described in "Speech Information Selectors" beginning on page 4-39.

While embedded speech commands are being processed, several types of errors might be detected and reported to your application. If you have enabled error callbacks by

using the `SetSpeechInfo` function with the `soErrorCallBack` selector, the error callback procedure will be executed once for every error that is detected, as described in "Error Callback Procedure" beginning on page 4-86. If you have not enabled error callbacks, you can still obtain information about the errors encountered by calling the `GetSpeechInfo` function with the `soErrors` selector. The following errors might be detected during processing of embedded speech commands:

`badParmVal`	–245	Parameter value is invalid
`badCmdText`	–246	Embedded command syntax or parameter problem
`unimplCmd`	–247	Embedded command is not implemented on synthesizer
`unimplMsg`	–248	Unimplemented message
`badVoiceID`	–250	Specified voice has not been preloaded
`badParmCount`	–252	Incorrect number of embedded command arguments

Examples of Embedded Speech Commands

If you use just a few of the embedded speech commands, you can markedly increase the understandability of text spoken by your application. Your application knows more about the speech being produced than a speech synthesizer does. A synthesizer speaks text according to a predetermined set of rules about language production. Therefore, the voices available on a Macintosh computer with the Speech Manager installed sound very synthetic and sometimes robotic because the pronunciation rules are formalized. You can make the speech produced by the synthesizer sound a lot more human by observing some simple rules of human speech and embedding speech commands in text according to these conventions. The techniques presented in this section could be applied when your application is having a dialog with the user or speaking some error messages or announcements.

The most common technique humans use in speaking is to emphasizing or deemphasizing words in a sentence. This change in emphasis marks for the listener new and important information by highlighting it vocally, making it easier for the listener to recognize important or different words in a sentence. For example, in a calendar-scheduling program, your application might speak a list of appointments for a day. The following text strings would all be spoken with the same tune and rhythm.

```
At 4pm you have a meeting with Kim Silver.
At 6pm you have a meeting with Tim Johnson.
At 7pm you have a meeting with Mark Smith.
```

The example that follows shows how you use embedded speech commands to deemphasize repeated words in similar sentences and highlight new information in a sentence. The first sentence of the following example sounds fairly acceptable. The second sentence deemphasizes the repeated words *have* and *meeting* to point out the new information—with whom the meeting is. The choice of which words to emphasize or deemphasize is based on what was spoken in the preceding sentence.To use the embedded command `emph` (emphasis), you insert it followed by a plus or minus sign before the word you want emphasized or deemphasized. The `emph` command lasts for a duration of one word.

```
At 4:15 you have a meeting with Ray Chiang.
At 6:30, you [[emph -]] have a [[emph -]] meeting with
William Ortiz.
At 7pm, you [[emph -]] have a [[emph -]] meeting with
Eric Braz Ford.
```

As shown in the next example, you can further enhance this text by spelling out the numbers so that you can emphasize changes in increments of time. For example, the following sentences deemphasize the repeated word *six* to highlight the difference between the meetings; which both occur between six and seven o'clock.

```
At four fifteen you have a meeting with Lori Kaplan.
At six [[emph -]] fifteen, you [[emph -]] have a [[emph -]]
meeting with Tim Monroe.
At [[emph -]] six thirty, you [[emph -]] have a [[emph -]]
meeting with Michael Abrams.
```

Another use of the emphasis embedded command is to make confusing, boring, or mechanical sounding text more understandable. One example of this is strings of nouns that refer to one entity (called complex nominals) that when spoken differently have a different meeting.

```
1a. Steel warehouse.
1b. Steel [[emph -]] warehouse.

2a. French teachers.
2b. French [[emph -]] teachers.
```

In the first example, phrase 1a, *steel warehouse*, refers to a warehouse made of steel, in which anything could be stored. But phrase 1b describes a warehouse of unspecified construction in which steel is stored. In the second example, phrase 2a, *French teachers*, refers to teachers from France who teach any subject. In the same example, phrase 2b specifies people from anywhere who teach French classes. You can use this technique of deemphasizing words in phrases to help users correctly understand the meaning of text spoken from your application.

You use the `emph` command to emphasize words in order to contrast them. You contrast words that are similar to words found later in a sentence to help distinguish new information.

```
You have [[emph +]] 3 text [[emph -]] messages, two fax [[emph
-]] messages, and [[emph +]] one [[emph +]] voice [[emph -]]
message.
```

This example emphasizes the words related to the number of messages and type of messages to help the listener discern the different kinds of information being presented.

Another common speaking technique that humans use is to pause before starting to speak about a new idea or before beginning a new paragraph. Adding an `slnc` (silence) command before beginning to speak a new idea or paragraph makes the synthetic voice sound like a person does when taking a breath in between ideas. This technique works best if you also raise the pitch range (using the `pmod` and `pbas` embedded commands) of the first sentence of the new paragraph. You must remember to lower the pitch range to achieve the desired effect.

```
[[emph -; pmod +1; pbas +1]] Good morning! [[pmod -1; pbas -1]]
This is a [[emph +]] newer [[emph -]] version of Apple's speech
synthesis. The previous [[emph -]] version has already been
[[emph -]] adopted by many developers. Users have sent us many
positive [[emph +]] reports.

[[slnc 500; pmod +1; pbas +1]]
This newer [[emph -]] version has better signal [[emph -]]
processing [[pmod -1; pbas -1]], new pitch [[emph -]] contours,
and a new compression. It still doesn't [[emph -]] sound perfect,
but people find it easier to understand.
```

This example deemphasizes the first word of the utterance, but raises the pitch to make the greeting sound more like a human would speak it. Then words are emphasized or deemphasized according to the techniques discussed previously. Silence is introduced before the new paragraph to signal a change in thought process. The pitch is raised and then lowered again after the first phrase. Note that you don’t have to wait a full sentence before changing the pitch back to its previous value. It’s best to work with these techniques until you find the most human-sounding utterances.

Phonemic Representation of Speech

The Speech Manager allows your application to process text phonemically. If your application speaks only text that the user writes, this feature is unlikely to be useful to you, because you cannot anticipate what the user might enter. However, if there are a few or many sentences that your application frequently converts into speech, it might be useful to represent parts of these sentences phonemically rather than textually.

It might be useful to convert your text into phonemes during application development in order to be able to reduce the amount of memory required to speak. If your application does not require the text-to-phoneme conversion portion of the speech synthesizer, significantly less RAM might be required to speak with some synthesizers.

Additionally, you might be able to use a higher quality text-to-phoneme conversion process (even one that does not work in real time) to generate precise phonemic information. This data can then be used with any speech synthesizer to produce better speech. For example, you might convert textual to phonemic data on a future version of the Speech Manager that performs such conversions more accurately than the Speech Manager currently does; that phonemic data could then be used to generate speech with

any version of the Speech Manager. The Speech Manager's `TextToPhonemes` function provides an easy method for converting text into its default phonemic equivalent.

To help the Speech Manager differentiate a textual representation of a word from a phonemic representation, you must embed commands in text that inform the Speech Manager to change into a mode in which it interprets a buffer of text as a phonemic representation of speech, in which particular combinations of letters represent particular phonemes. (You can also use the `SetSpeechInfo` function to change to phoneme mode.) To indicate to the Speech Manager that subsequent text is a phonemic representation of text to be spoken, embed the `[[inpt PHON]]` command within a string or buffer that your application passes to one of the `SpeakString`, `SpeakText`, or `SpeakBuffer` functions. To indicate that the Speech Manager should revert to textual interpretation of a text buffer, embed the `[[inpt TEXT]]` command. For example, passing the string

```
Hello, I am [[inpt PHON]]mAYkAXl[[inpt TEXT]], the talking
computer.
```

to `SpeakString`, `SpeakText`, or `SpeakBuffer` would result in the generation of the sentence, "Hello, I am Michael, the talking computer."

Some, but not all, speech synthesizers allow you to embed a command that causes the Speech Manager to interpret a buffer of text as a series of allophones.

Phonemic Symbols

Table 4-3 summarizes the set of standard phonemes recognized by American English speech synthesizers. Other languages and dialects require different phoneme inventories. Phonemes divide into two groups: vowels and consonants. All vowel symbols are pairs of uppercase letters. For simple consonants the symbol is that lowercase consonant; for blends and complex consonants, the symbol is in uppercase. Within the example words, the individual sounds being exemplified appear in boldface.

Table 4-3 American English phoneme symbols

Symbol	Example	Opcode	Symbol	Example	Opcode
%	silence	0	D	**th**em	21
@	breath intake	1	f	**f**in	22
AE	b**a**t	2	g	**g**ain	23
EY	b**ai**t	3	h	**h**at	24
AO	c**au**ght	4	J	**j**ump	25
AX	**a**bout	5	k	**k**in	26
IY	b**ee**t	6	l	**l**imb	27
EH	b**e**t	7	m	**m**at	28

continued

Table 4-3 American English phoneme symbols (continued)

Symbol	Example	Opcode	Symbol	Example	Opcode
IH	b**i**t	8	n	**n**at	29
AY	b**i**te	9	N	ta**ng**	30
IX	ros**e**s	10	p	**p**in	31
AA	c**o**t	11	r	**r**an	32
UW	b**oo**t	12	s	**s**in	33
UH	b**oo**k	13	S	**sh**in	34
UX	b**u**d	14	t	**t**in	35
OW	b**oa**t	15	T	**th**in	36
AW	b**ou**t	16	v	**v**an	37
OY	b**oy**	17	w	**w**et	38
b	**b**in	18	y	**y**et	39
C	**ch**in	19	z	**z**en	40
d	**d**in	20	Z	mea**s**ure	41

You can obtain information similar to that in Table 4-3 for whatever language a synthesizer supports by using the `GetSpeechInfo` function on a channel using the synthesizer with the `soPhonemeSymbols` selector. The information is returned in a phoneme descriptor record, whose structure is described on page 4-53.

Prosodic Control Symbols

The symbols listed in Table 4-4 are recognized as modifiers to the basic phonemes described in the preceding section. You can use them to more precisely control the quality of speech that is described in terms of raw phonemes.

Table 4-4 Prosodic control symbols

Type	Symbol	Symbol name	Description or illustration of effect
Lexical stress:			Marks stress within a word (optional)
Primary stress	1		AEnt2IHsIXp1EYSAXn (“anticipation”)
Secondary stress	2		
Syllable breaks:			Marks syllable breaks within a word (optional)
Syllable mark	=	(equal)	AEn=t2IH=sIX=p1EY=SAXn (“an-ti-ci-pa-tion”)
Word prominence:			Placed before the affected word
Destressed	~	(asciitilde)	Used for words with minimal informational content

Table 4-4 Prosodic control symbols (continued)

Type	Symbol	Symbol name	Description or illustration of effect
Normal stress	_	(underscore)	Used for information-bearing words
Emphatic stress	+	(plus)	Used for words requiring special emphasis
Prosodic:			Placed before the affected phoneme
Pitch rise	/	(slash)	Pitch will rise on the following phoneme
Pitch fall	\	(backslash)	Pitch will fall on the following phoneme
Lengthen phoneme	>	(greater)	Lengthens the duration of the following phoneme
Shorten phoneme	<	(less)	Shortens the duration of the following phoneme

Note
Like all other phonemes, the "silence" phoneme (%) and the "breath intake" phoneme (@) can be lengthened or shortened using the > and < symbols. ◆

The prosodic control symbols (/, \, <, and >) can be concatenated to provide exaggerated or cumulative effects. The specific nature of the effect is dependent on the speech synthesizer. Speech synthesizers also often extend or enhance the controls described in the table.

Table 4-5 indicates the effect of punctuation marks on sentence prosody. In particular, the table shows the effect of punctuation marks on speech pitch and indicates to what extent the punctuation marks cause a pause. Note that because some languages might not use these punctuation marks, some synthesizers might not interpret them correctly. In general, speech synthesizers strive to mimic the pauses and changes in pitch of actual speakers in response to punctuation marks, so to obtain best results, you can punctuate according to standard grammatical rules.

Table 4-5 Effect of punctuation marks on English-language synthesizers

Symbol	Symbol name	Effect of punctuation mark	Effect on Timing
&	(ampersand)	Forces no addition of silence between phonemes	No additional effect
:	(colon)	End of clause, no change in pitch	Short pause follows
,	(comma)	Continuation rise in pitch	Short pause follows
…	(ellipsis)	End of clause, no change in pitch	Pause follows
!	(exclam)	End-of-sentence sharp fall in pitch	Pause follows
-	(hyphen)	End of clause, no change in pitch	Short pause follows
(	(parenleft)	Start reduced pitch range	Short pause precedes

continued

Table 4-5 Effect of punctuation marks on English-language synthesizers (continued)

Symbol	Symbol name	Effect of punctuation mark	Effect on Timing
)	(parenright)	End reduced pitch range	Short pause follows
.	(period)	End-of-sentence fall in pitch	Pause follows
?	(question)	End-of-sentence rise in pitch	Pause follows
“ ‘	(quotedblleft, quotesingleleft)	Varies depending on context	Varies
” ’	(quotedblright, quotesingleright)	Varies depending on context	Varies
;	(semicolon)	Continuation rise in pitch	Short pause follows

Specific pitch contours associated with these punctuation marks might vary according to other considerations in the analysis of the text. For example, if a question is rhetorical or begins with a word recognized by the synthesizer to be a question word, the pitch might fall at the question mark. Consequently the above effects should be regarded as only guidelines and not absolute. This also applies to the timing effects, which will vary according to the current rate setting.

Including Pronunciation Dictionaries

No matter how sophisticated a speech synthesis system is, there will always be words that it does not automatically pronounce correctly. A clear instance of words that are often mispronounced is the class of proper nouns (names of people, place names, and so on). The Speech Manager supports pronunciation dictionaries which allow applications to override the default pronunciations of words. A **pronunciation dictionary** is a list of words along with their associated pronunciations stored in a resource of resource type `'dict'`.

The application is free to store dictionaries in either the resource fork or the data fork of a file. The application is responsible for loading the individual dictionaries into RAM and then passing a handle to the dictionary data to the Speech Manager. The initial release of the Speech Manager, however, does not include any routines that can add entries to dictionaries or manipulate them in other ways. The Speech Manager does include a routine, the `UseDictionary` function, that you can use to install one or more pronunciation dictionaries in a speech channel.

A multimedia application might store such a pronunciation dictionary resource in its own resource fork to specify the pronunciations of selected words used in a narration. A word-processing application, meanwhile, could allow a user to add words to a pronunciation dictionary stored in the resource fork of a text file. Or, a text-services application dedicated to speech generation might include large specialized dictionaries—for example, of medical terms—to specify pronunciation of words in particular subject

areas. Because the Speech Manager allows your application to install as many pronunciation dictionaries as desired in a speech channel, it can use pronunciation dictionaries in one or more of these ways.

Note
The Dictionary Manager, described in *Inside Macintosh: Text*, cannot be used with pronunciation dictionaries. ◆

Whenever a speech synthesizer needs to determine the proper phonemic representation for a particular word, it first looks for the word in its pronunciation dictionaries. Pronunciation dictionary entries contain information that enables precise conversion between text and the correct phoneme codes, as described in "Phonemic Representation of Speech" beginning on page 4-32. Pronunciation dictionary entries also provide stress, intonation, and other information to help speech synthesizers produce more natural speech, as described in "Prosodic Control Symbols" beginning on page 4-34. Note that you cannot use punctuation marks (as described in Table 4-5) in pronunciation dictionaries.

A single pronunciation dictionary entry cannot be used to specify the pronunciation of an entire phrase, because the Speech Manager checks its pronunciation dictionary on a word-by-word basis. Thus, the textual portion of a pronunciation dictionary entry must not contain any spaces.

If the pronunciation dictionaries installed in a speech channel do not include an indication of how a word should be pronounced, then the Speech Manager uses its own pronunciation rules and internal dictionary to pronounce the words. In general, you need to create a dictionary only for unusual words that your application requires but the Speech Manager ordinarily pronounces incorrectly. You might also allow a user who is not pleased with the default pronunciation of a word to add the correct pronunciation to a pronunciation dictionary. You can create a dictionary using MPW Rez or another appropriate tool. See "The Pronunciation Dictionary Resource" beginning on page 4-89 for a discussion of the format of the pronunciation dictionary resource and the meaning of it fields.

To install a pronunciation dictionary resource in a speech channel, you must read the resource into memory and pass it to the `UseDictionary` function. Because the `UseDictionary` function requires that you specify a speech channel, you might need to reinstall the dictionary whenever your application allocates a new speech channel or whenever it resets an existing speech channel. Listing 4-9 shows how you can use the `UseDictionary` function to install a pronunciation dictionary resource in a speech channel.

Listing 4-9 Installing a pronunciation dictionary resource into a speech channel

```
PROCEDURE MyUseDictionary (chan: SpeechChannel; resID: Integer);
VAR
   myDict:     Handle;                          {handle to dictionary data}
   myErr:      OSErr;
```

```
BEGIN
   myDict := GetResource('dict', resID);        {load the dictionary}
   IF (myDict <> NIL) AND (ResError = noErr) THEN
   BEGIN
      myErr := UseDictionary(chan, myDict);     {install the dictionary}
      IF myErr <> noErr THEN
         DoError(myErr);                        {respond to an error}
      ReleaseResource(myDict);                  {release the resource}
   END;
END;
```

The `MyUseDictionary` procedure defined in Listing 4-9 attempts to find a resource of resource type `'dict'` with resource ID `resID` and uses the Resource Manager to read it into memory. If your application stores pronunciation dictionaries in the data fork of files, it can instead use analogous File Manager routines to read the data. If the data is read in correctly, `MyUseDictionary` calls the `UseDictionary` function to install the dictionary on the specified speech channel. Because the speech synthesizer copies all necessary data from the dictionary to its internal buffers, the application is free to release the memory occupied by the dictionary, as illustrated by the `ReleaseResource` call.

The pronunciation dictionary resource in Listing 4-10 consists of pronunciation dictionary entries in Rez format. Each entry specifies a word in textual format and its phonemic equivalent.

Listing 4-10 A sample pronunciation dictionary resource

```
resource 'dict' (1, "TestDict") {
   smRoman, langEnglish, verUS, ThisSecond,
   {
      pron, {tx, "ROOSEVELT",    ph, "_1EHf_d1IY_1AAr"},
      pron, {tx, "CHELSEA",      ph, "_C1EHls2IY"},
      pron, {tx, "AMHERST",      ph, "_2UXmAXrst"},
      pron, {tx, "REDSOX",       ph, "_r1EHd_s1AAks"},
      pron, {tx, "HALLOWEEN",    ph, "_h1AAl2OW_w1IYn"},
      pron, {tx, "FELIX",        ph, "_f1IYl2IHks_D2UX_k1AEt"},
      pron, {tx, "WEDNESDAY",    ph, "_m1IHd_w1IYk"},
   },
};
```

Note that you are not restricted to using pronunciations similar to those of the words listed. Typically, however, pronunciation dictionaries contain entries for words that the Speech Manager pronounces unsatisfactorily.

Also, note that a pronunciation dictionary's entries need not be in any particular order. In particular, you should not assume that a pronunciation dictionary is in alphabetical order unless your application creates the dictionary and maintains that order.

The pronunciation dictionary resource header consists of nine fields, of which four must be explicitly defined in a Rez definition such as the one in Listing 4-10. The first three of these fields specify the script, language, and region code of the language for which the pronunciation dictionary is designed. Note that you must create a separate pronunciation dictionary for each region, language, or script. The fourth field of a pronunciation dictionary is the date the pronunciation dictionary was last modified, in terms of seconds since midnight, January 1, 1904. In Listing 4-10, it is assumed that the constant `ThisSecond` is defined to be such a date. For information on obtaining information about the current date in this format, see *Inside Macintosh: Operating System Utilities*.

Speech Manager Reference

This section describes the constants, data structures, routines, and resources that are specific to the Speech Manager.

The section "Constants" describes the available speech information selectors.

The section "Data Structures" beginning on page 4-45 shows all of the Speech Manager's Pascal data structures, including those for the voice specification and description records, the speech status information record, and the phoneme information and descriptor records.

The section "Speech Manager Routines" beginning on page 4-54 describes the Speech Manager functions that allow you to generate speech, use voices, manage and control speech channels, convert text to phonemes, and use pronunciation dictionaries.

The section "Application-Defined Routines" beginning on page 4-82 describes the kinds of callback procedures you can implement.

The section "Resources" beginning on page 4-89 describes the format of pronunciation dictionary resources.

Constants

This section describes the available speech information selectors.

Speech Information Selectors

This section describes the **speech information selectors** that you can pass in the `selector` parameter of the `GetSpeechInfo` and `SetSpeechInfo` functions.

```
CONST
   soCharacterMode         = 'char';   {get or set character-processing mode}
   soCommandDelimiter      = 'dlim';   {set embedded command delimiters}
   soCurrentA5             = 'myA5';   {set A5 on callbacks}
```

```
soCurrentVoice          = 'cvox';   {set speaking voice}
soErrorCallBack         = 'ercb';   {set error callback}
soErrors                = 'erro';   {get error information}
soInputMode             = 'inpt';   {get or set text-processing mode}
soNumberMode            = 'nmbr';   {get or set number-processing mode}
soPhonemeCallBack       = 'phcb';   {set phoneme callback}
soPhonemeSymbols        = 'phsy';   {get phoneme symbols and example }
                                    { words}
soPitchBase             = 'pbas';   {get or set baseline pitch}
soPitchMod              = 'pmod';   {get or set pitch modulation}
soRate                  = 'rate';   {get or set speech rate}
soRecentSync            = 'sync';   {get most recent synchronization }
                                    { message information}
soRefCon                = 'refc';   {set reference constant value}
soReset                 = 'rset';   {set channel back to default state}
soSpeechDoneCallBack    = 'sdcb';   {set speech-done callback}
soStatus                = 'stat';   {get status of channel}
soSyncCallBack          = 'sycb';   {set synchronization callback}
soSynthExtension        = 'xtnd';   {get or set synthesizer-specific }
                                    { information}
soSynthType             = 'vers';   {get synthesizer information}
soTextDoneCallBack      = 'tdcb';   {set text-done callback}
soVolume                = 'volm';   {get or set speech volume}
soWordCallBack          = 'wdcb';   {set word callback}
```

Constant descriptions

`soCharacterMode`

Get or set the speech channel's character-processing mode. Two constants are currently defined for the processing mode, `modeNormal` and `modeLiteral`. When the character-processing mode is `modeNormal`, input characters are spoken as you would expect to hear them. When the mode is `modeLiteral`, each character is spoken literally, so that the word "cat" would be spoken "C–A–T". The `speechInfo` parameter points to a variable of type `OSType`, which is the character-processing mode.

This selector works with `GetSpeechInfo` and `SetSpeechInfo` and does not move memory.

`soCommandDelimiter`

Set the embedded speech command delimiter characters to be used for the speech channel. By default the opening delimiter is "`[[`" and the closing delimiter is "`]]`". Your application might need to change these delimiters temporarily if those character sequences occur naturally in a text buffer that is to be spoken. Your application can also disable embedded command processing by passing empty delimiters (2 `NIL` bytes). The `speechInfo` parameter is a pointer to a delimiter information record, described on page 4-54.

This selector works with the `SetSpeechInfo` function and does not move memory.

`soCurrentA5` Set the value that the Speech Manager assigns to the A5 register before invoking any application-defined callback procedures for the speech channel. The A5 register must be set correctly if the callback procedures are to be able to access application global variables. For more information on the A5 register, see *Inside Macintosh: Memory*. The `speechInfo` parameter should be set to the pointer contained in the A5 register at a time when the application is not executing interrupt code or to `NIL` if your application wishes to clear a value previously set with the `soCurrentA5` selector.

This selector works with the `SetSpeechInfo` function and does not move memory. See Listing 4-6 on page 4-21 for an illustration of the use of this selector.

`soCurrentVoice`

Set the current voice on the current speech channel to the specified voice. The `speechInfo` parameter is a pointer to a voice specification record. Your application should create the record by calling the `MakeVoiceSpec` function, described on page 4-64. `SetSpeechInfo` will return an `incompatibleVoice` error if the specified voice is incompatible with the speech synthesizer associated with the speech channel. If you have a speech channel open using a voice from a particular synthesizer and you try to switch to a voice that works with a different synthesizer, you receive an `incompatibleVoice` error. You need to create a new channel to use with the new voice.

This selector works with only `SetSpeechInfo` and might move memory. Your application should not invoke it at interrupt time.

`soErrorCallBack`

Set the callback procedure to be called when an error is encountered during the processing of an embedded command. The callback procedure might also be called if other conditions (such as insufficient memory) arise during the speech conversion process. When a Speech Manager function returns an error directly, the error callback procedure is not called. The callback procedure is passed information about the most recent error; it can determine information about the oldest pending error by using the speech information selector `soErrors`. The `speechInfo` parameter is a pointer to an application-defined error callback procedure, whose syntax is described on page 4-86. Passing `NIL` in `speechInfo` disables the error callback procedure.

This selector works with the `SetSpeechInfo` function and does not move memory.

`soErrors` Get saved error information for the speech channel and clear its error registers. This selector lets you poll for various run-time errors that occur during speaking, such as the detection of badly formed embedded commands. Errors returned directly by Speech Manager functions are not reported here. If your application defines an error callback procedure, the callback should use the `soErrors` selector

to obtain error information. The `speechInfo` parameter is a pointer to a speech error information record, described on page 4-49.

This selector works with the `GetSpeechInfo` function and does not move memory.

`soInputMode` Get or set the speech channel's current text-processing mode. The returned value specifies whether the channel is currently in text input mode or phoneme input mode. The `speechInfo` parameter is a pointer to a variable of type `OSType`, which specifies a text-processing mode. The following constants specify the available text-processing modes:

```
CONST
   modeText             = 'TEXT';
   modePhonemes         = 'PHON';
```

The `modeText` constant indicates that the speech channel is in text-processing mode. The `modePhonemes` constant indicates that the speech channel is in phoneme-processing mode. When in phoneme-processing mode, a text buffer is interpreted to be a series of characters representing various phonemes and prosodic controls, as discussed in "Phonemic Representation of Speech" on page 4-32 and "Prosodic Control Symbols" on page 4-34. Some synthesizers might support additional input-processing modes and define constants for these modes.

This selector works with both the `GetSpeechInfo` and `SetSpeechInfo` functions. It might move memory only when used in conjunction with the `SetSpeechInfo` function.

`soNumberMode` Get or set the speech channel's current number-processing mode. Two `OSType` constants are currently defined, `modeNormal` and `modeLiteral`. When the number-processing mode is `modeNormal`, the synthesizer assembles digits into numbers (so that 12 is spoken as "twelve"). When the mode is `modeLiteral`, each digit is spoken literally (so that 12 is spoken as "one, two"). The `speechInfo` parameter is a pointer to a variable of type `OSType`, which specifies the number-processing mode.

This selector works with both the `GetSpeechInfo` and `SetSpeechInfo` functions and does not move memory.

`soPhonemeCallBack`
Set the callback procedure to be called every time the Speech Manager is about to generate a phoneme on the speech channel. The `speechInfo` parameter is a pointer to an application-defined phoneme callback procedure, whose syntax is described on page 4-87. Passing `NIL` in `speechInfo` disables the phoneme callback procedure.

This selector works with the `SetSpeechInfo` function and does not move memory.

`soPhonemeSymbols`
Get a list of phoneme symbols and example words defined for the

speech channel's synthesizer. Your application might use this information to show the user what symbols to use when entering phonemic text directly. The `speechInfo` parameter is a pointer to a variable of type `Handle` that, on exit from the `GetSpeechInfo` function, is a handle to a phoneme descriptor record, described on page 4-53.

This selector works with the `GetSpeechInfo` function and might move memory. Your application should not invoke it at interrupt time.

`soPitchBase` Get or set the speech channel's baseline speech pitch. This selector is intended for use by the Speech Manager; ordinarily, an application uses the `GetSpeechPitch` and `SetSpeechPitch` functions, described on page 4-75 and page 4-76, respectively. The `speechInfo` parameter is a pointer to a variable of type `Fixed`.

This selector works with both the `GetSpeechInfo` and `SetSpeechInfo` functions and does not move memory.

`soPitchMod` Get or set a speech channel's pitch modulation. The `speechInfo` parameter is a pointer to a variable of type `Fixed`. Pitch modulation is also expressed as a fixed-point value in the range of 0.000 to 127.000. These values correspond to MIDI note values, where 60.000 is equal to middle C on a piano scale. The most useful speech pitches fall in the range of 40.000 to 55.000. A pitch modulation value of 0.000 corresponds to a monotone in which all speech is generated at the frequency corresponding to the speech pitch. Given a speech pitch value of 46.000, a pitch modulation of 2.000 would mean that the widest possible range of pitches corresponding to the actual frequency of generated text would be 44.000 to 48.000.

This selector works with both the `GetSpeechInfo` and `SetSpeechInfo` functions and does not move memory.

`soRate` Get or set a speech channel's speech rate. The `speechInfo` parameter is a pointer to a variable of type `Fixed`. The possible range of speech rates is from 0.000 to 65535.65535. The range of supported rates is not predefined by the Speech Manager; each speech synthesizer provides its own range of speech rates. Average human speech occurs at a rate of 180 to 220 words per minute.

This selector works with both the `GetSpeechInfo` and `SetSpeechInfo` functions and does not move memory.

`soRecentSync` Get the message code for the most recently encountered synchronization command. If no synchronization command has been encountered, 0 is returned. The `speechInfo` parameter is a pointer to a variable of type `OSType`.

This selector works with the `GetSpeechInfo` function and does not move memory.

`soRefCon` Set a speech channel's reference constant value. The reference constant value is passed to application-defined callback procedures and might contain any value convenient for the application. The `speechInfo` parameter is a long integer containing the reference

constant value. In contrast with other selectors, this selector does not require that the `speechInfo` parameter's value be a pointer value. Typically, however, an application does use this selector to pass a pointer or handle value to callback procedures.

This selector works with the `SetSpeechInfo` function and does not move memory. See Listing 4-6 on page 4-21 for an illustration of the use of this selector.

`soReset` Set a speech channel back to its default state. For example, speech pitch and speech rate are set to default values. The `speechInfo` parameter should be set to `NIL`.

This selector works with the `SetSpeechInfo` function and does not move memory.

`soSpeechDoneCallBack`
Set the callback procedure to be called when the Speech Manager has finished generating speech on the speech channel. The `speechInfo` parameter is a pointer to an application-defined speech-done callback procedure, whose syntax is described on page 4-84. Passing `NIL` in `speechInfo` disables the speech-done callback procedure.

This selector works with the `SetSpeechInfo` function and does not move memory.

`soStatus` Get a speech status information record for the speech channel. The `speechInfo` parameter is a pointer to a speech status information record, described on page 4-48.

This selector works with the `GetSpeechInfo` function and does not move memory.

`soSyncCallBack`
Set the callback procedure to be called when the Speech Manager encounters a synchronization command within an embedded speech command in text being processed on the speech channel. The `speechInfo` parameter is a pointer to an application-defined synchronization callback procedure, whose syntax is described on page 4-85. Passing `NIL` in `speechInfo` disables the synchronization callback procedure.

This selector works with the `SetSpeechInfo` function and does not move memory.

`soSynthExtension`
Get or set synthesizer-specific information or settings. The `speechInfo` parameter is a pointer to a speech extension data record, described on page 4-53. Your application should set the `synthCreator` field of this record before calling `GetSpeechInfo` or `SetSpeechInfo`. Ordinarily, your application must pass additional information to the synthesizer in the `synthData` field.

This selector works with both the `GetSpeechInfo` and `SetSpeechInfo` functions. Whether it moves memory depends on the synthesizer being used and the information passed to the synthesizer.

`soSynthType` Get a speech version information record for the speech synthesizer being used on the specified speech channel. The `speechInfo` parameter is a pointer to a speech version information record, described on page 4-50.

This selector works with the `GetSpeechInfo` function and does not move memory.

`soTextDoneCallBack` Set the callback procedure to be called when the Speech Manager has finished processing speech being generated on the speech channel. The `speechInfo` parameter is a pointer to an application-defined text-done callback procedure, whose syntax is described on page 4-84. Passing `NIL` in `speechInfo` disables the text-done callback procedure.

This selector works with the `GetSpeechInfo` function and does not move memory.

`soVolume` Get or set the speech volume for a speech channel. The `speechInfo` parameter is a pointer to a variable of type `Fixed`. Volumes are expressed in fixed-point units ranging from 0.0 through 1.0. A value of 0.0 corresponds to silence, and a value of 1.0 corresponds to the maximum possible volume. Volume units lie on a scale that is linear with amplitude or voltage. A doubling of perceived loudness corresponds to a doubling of the volume.

This selector works with both the `GetSpeechInfo` and `SetSpeechInfo` functions and does not move memory.

`soWordCallBack` Set the callback procedure to be called every time the Speech Manager is about to generate a word on the speech channel. The `speechInfo` parameter is a pointer to an application-defined word callback procedure, whose syntax is described on page 4-87. Passing `NIL` in `speechInfo` disables the word callback procedure.

This selector works with the `SetSpeechInfo` function and does not move memory. See Listing 4-7 on page 4-21 for an illustration of the use of this selector.

Data Structures

This section describes the data structures defined by the Speech Manager.

The speech channel record contains information internal to the Speech Manager. Speech channels, which process Speech Manager text and commands, are defined as pointers to Speech Manager records.

A voice specification record provides a unique specification of a voice. You can create such a record with the `MakeVoiceSpec` function and then pass it to the `GetVoiceDescription` function to obtain information about the voice. This information is contained in a voice description record. Or, you can use the `GetVoiceInfo` function to obtain information about the file that stores a voice. This information is contained in a voice file information record.

By using the `GetSpeechInfo` function, you can obtain information about a speech channel, as well as information about its synthesizer. Such information is returned in speech status information records, speech error information records, and speech version information records.

The `GetSpeechInfo` function also allows you to obtain information about the phonemes defined for a synthesizer. Information about a single phoneme is contained in a phoneme information record. A phoneme descriptor record contains phoneme information records for all of the phonemes that a synthesizer supports.

Synthesizers that use the `GetSpeechInfo` or `SetSpeechInfo` function to allow exploitation of synthesizer-specific features often require that data passed to it be formatted in a particular way. The speech extension data record allows your application to exchange data in any format with a synthesizer.

The `SpeakString`, `SpeakText`, and `SpeakBuffer` functions can process both text and commands embedded in that text. So that commands can be distinguished from text, the commands must be enclosed by command delimiters. The delimiter information record allows your application to change the command delimiters.

Voice Specification Records

A voice specification record provides a unique specification that you must use to obtain information about a voice. You also must use a voice specification record if you wish to create a speech channel that generates speech in a voice other than the current system default voice. The `VoiceSpec` data type defines a voice specification record. In Pascal, the `VoiceSpecPtr` data type defines a pointer to a voice specification record. The `VoiceSpecPtr` data type is not defined in the interface files for C programmers. If you are programming in C and you need to pass a variable of type `VoiceSpecPtr` to a Speech Manager routine, simply pass a pointer to a voice specification record instead.

```
TYPE VoiceSpec =
RECORD
   creator:    OSType;        {ID of required synthesizer}
   id:         OSType;        {ID of voice on the synthesizer}
END;
```

Field descriptions

`creator` The synthesizer that is required to use the voice. This is equivalent to the value contained in the `synthManufacturer` field of a speech version information record and that contained in the `synthCreator` field of a speech extension data record. The set of `OSType` values specified entirely by space characters and lowercase letters is reserved.

`id` The voice ID of the voice for the synthesizer. Every voice on a synthesizer has a unique ID.

IMPORTANT

To ensure compatibility with future versions of the Speech Manager, you should never fill in the fields of a voice specification record yourself. Instead, you should create a voice specification record by using the `MakeVoiceSpec` function. ▲

Voice Description Records

By calling the `GetVoiceDescription` function, you can obtain information about a voice in a voice description record. The `VoiceDescription` data type defines a voice description record.

```
TYPE VoiceDescription =
RECORD
   length:     LongInt;    {size of record}
   voice:      VoiceSpec;  {voice synthesizer and ID info}
   version:    LongInt;    {version number of voice}
   name:       Str63;      {name of voice}
   comment:    Str255;     {text information about voice}
   gender:     Integer;    {neuter, male, or female}
   age:        Integer;    {approximate age in years}
   script:     Integer;    {script code of text voice can process}
   language:   Integer;    {language code of voice output}
   region:     Integer;    {region code of voice output}
   reserved1:  LongInt;    {always 0--reserved for future use}
   reserved2:  LongInt;    {always 0--reserved for future use}
   reserved3:  LongInt;    {always 0--reserved for future use}
   reserved4:  LongInt;    {always 0--reserved for future use}
END;
```

Field descriptions

`length` The size of the voice description record, in bytes.

`voice` A voice specification record that uniquely identifies the voice.

`version` The version number of the voice.

`name` The name of the voice, preceded by a length byte. Names must be 63 characters or less.

`comment` Additional text information about the voice. The information might indicate how much memory the voice requires. Some synthesizers use this field to store a phrase that can be spoken.

`gender` The gender of the individual represented by the voice. The value in this field must be one of the following constants:

```
CONST
   kNeuter     = 0;     {neuter voice}
   kMale       = 1;     {male voice}
   kFemale     = 2;     {female voice}
```

A neuter voice is a voice that is not distinctively male or female.

`age` The approximate age in years of the individual represented by the voice.

`script` The script code of text that the voice can process.

`language` A code that indicates the language of voice output.

`region` A code that indicates the region represented by the voice.

`reserved1` Reserved.

`reserved2` Reserved.

`reserved3` Reserved.

`reserved4` The four reserved fields are reserved for use by Apple.

Voice File Information Records

A **voice file information record** specifies the file in which a voice is stored and the resource ID of the voice within that file. You can use the `GetVoiceInfo` function to obtain a voice file information record for a voice. The `VoiceFileInfo` data type defines a voice file information record. In Pascal, the `VoiceFileInfoPtr` data type defines a pointer to a voice file information record.

```
TYPE VoiceFileInfo =
RECORD
   fileSpec:   FSSpec;     {volume, dir, and name of file}
   resID:      Integer;    {resource ID of voice in the file}
END;
```

Field descriptions

`fileSpec` A file system specification record that contains the volume, directory, and name of the file containing the voice. Generally, files containing a single voice are of type `kTextToSpeechVoiceFileType`, and files containing multiple voices are of type `kTextToSpeechVoiceBundleType`.

`resID` The resource ID of the voice in the file. Voices are stored in resources of type `kTextToSpeechVoiceType`.

Speech Status Information Records

By calling the `GetSpeechInfo` function with the `soStatus` selector, you can find out information about the status of a speech channel. This information is stored in a **speech status information record,** which the `SpeechStatusInfo` data type defines.

```
TYPE SpeechStatusInfo =
RECORD
   outputBusy:       Boolean;    {TRUE if audio is playing }
                                 { or text is being processed}
   outputPaused:     Boolean;    {TRUE if channel is paused}
   inputBytesLeft:   LongInt;    {bytes of text left to process}
   phonemeCode:      Integer;    {opcode for current phoneme}
END;
```

Field descriptions

`outputBusy` Whether the speech channel is currently producing speech. A speech channel is considered to be producing speech even at some times when no audio data is being produced through the Macintosh speaker. This occurs, for example, when the Speech Manager is processing an input buffer but has not yet initiated speech or when speech output is paused.

`outputPaused` Whether speech output in the speech channel has been paused by a call to the `PauseSpeechAt` function.

`inputBytesLeft`
The number of input bytes of the text that the speech channel must still process. When `inputBytesLeft` is 0, the buffer of input text passed to one of the `SpeakText` or `SpeakBuffer` functions may be disposed of. (Note that when you call the `SpeakString` function, the Speech Manager stores a duplicate of the string to be spoken in an internal buffer; thus, you may delete the original string immediately after calling `SpeakString`.)

`phonemeCode` The opcode for the phoneme that the speech channel is currently processing.

Speech Error Information Records

By calling the `GetSpeechInfo` function with the `soErrors` selector, you can obtain a **speech error information record,** which shows what Speech Manager errors occurred while processing a text buffer on a given speech channel. The `SpeechErrorInfo` data type defines a speech error information record.

```
TYPE SpeechErrorInfo =
RECORD
   count:      Integer;       {number of errors since last check}
   oldest:     OSErr;         {oldest unread error}
   oldPos:     LongInt;       {character position of oldest error}
   newest:     OSErr;         {most recent error}
   newPos:     LongInt;       {character position of newest error}
END;
```

Field descriptions

`count`	The number of errors that have occurred in processing the current text buffer since the last call to the `GetSpeechInfo` function with the `soErrors` selector. Of these errors, you can find information about only the first and last error that occurred.
`oldest`	The error code of the first error that occurred after the previous call to the `GetSpeechInfo` function with the `soErrors` selector.
`oldPos`	The character position within the text buffer being processed of the first error that occurred after the previous call to the `GetSpeechInfo` function with the `soErrors` selector.
`newest`	The error code of the most recent error.
`newPos`	The character position within the text buffer being processed of the most recent error.

Speech error information records never include errors that are returned by Speech Manager routines. Instead, they reflect only errors encountered directly in the processing of text, and, in particular, in the processing of commands embedded within text.

The speech error information record keeps track of only the most recent error and the first error that occurred after the previous call to the `GetSpeechInfo` function with the `soErrors` selector. If your application needs to keep track of all errors, then you should install an error callback procedure, as described in "Error Callback Procedure" beginning on page 4-86.

Speech Version Information Records

By calling the `GetSpeechInfo` function with the `soSynthType` selector, you can obtain a **speech version information record,** which provides information about the speech synthesizer currently being used. The `SpeechVersionInfo` data type defines a speech version information record.

```
TYPE SpeechVersionInfo =
RECORD
   synthType:           OSType;        {general synthesizer type}
   synthSubType:        OSType;        {specific synthesizer type}
   synthManufacturer:   OSType;        {synthesizer creator ID}
   synthFlags:          LongInt;       {synthesizer feature flags}
   synthVersion:        NumVersion;    {synthesizer version number}
END;
```

Field descriptions

`synthType`	The general type of the synthesizer. For the current version of the Speech Manager, this field always contains the value `kTextToSpeechSynthType`, indicating that the synthesizer converts text into speech.
`synthSubType`	The specific type of the synthesizer. Currently, no specific types of synthesizer are defined. If you define a new type of synthesizer, you

should register the four-character code for your type with Developer Technical Support.

`synthManufacturer`

A unique identification of a synthesizer engine. If you develop synthesizers, then you should register a different four-character code for each synthesizer you develop with Developer Technical Support. The `creatorID` field of the voice specification record and the `synthCreator` field of a speech extension data record should each be set to the value stored in this field for the desired synthesizer.

`synthFlags`

A set of flags indicating which synthesizer features are activated. The following constants define the bits in this field whose meanings are defined for all synthesizers:

```
CONST
   kNoEndingProsody     = 1;
   kNoSpeechInterrupt   = 2;
   kPreflightThenPause  = 4;
```

The `kNoEndingProsody` flag bit is used to control whether or not the speech synthesizer automatically applies ending prosody, the speech tone and cadence that normally occur at the end of a statement. Under normal circumstances (for example, when the flag bit is not set), ending prosody is applied to the speech when the end of the `textBuf` data is reached. This default behavior can be disabled by setting the `kNoEndingProsody` flag bit.

Some synthesizers do not speak until the `kNoEndingProsody` flag bit is reset, or they encounter a period in the text, or `textBuf` is full.

The `kNoSpeechInterrupt` flag bit is used to control the behavior of `SpeakBuffer` when called on a speech channel that is still busy. When the flag bit is not set, `SpeakBuffer` behaves similarly to `SpeakString` and `SpeakText`. Any speech currently being produced on the specified speech channel is immediately interrupted, and then the new text buffer is spoken. When the `kNoSpeechInterrupt` flag bit is set, however, a request to speak on a channel that is still busy processing a prior text buffer will result in an error. The new buffer is ignored and the error `synthNotReady` is returned. If the prior text buffer has been fully processed, the new buffer is spoken normally. One way of achieving continuous speech without using callback procedures is to continually call `SpeakBuffer` with the `kNoSpeechInterrupt` flag bit set until the function returns `noErr`. The function will then execute as soon as the first text buffer has been processed.

The `kPreflightThenPause` flag bit is used to minimize the latency experienced when the speech synthesizer is attempting to speak. Ordinarily, whenever a call to `SpeakString`, `SpeakText`, or `SpeakBuffer` is made, the speech synthesizer must perform a certain amount of initial processing before speech output is heard. This startup latency can vary from a few milliseconds to several

seconds depending upon which speech synthesizer is being used. Recognizing that larger startup delays might be detrimental to certain applications, a mechanism is provided to allow the synthesizer to perform any necessary computations at noncritical times. Once the computations have been completed, the speech is able to start instantly. When the `kPreflightThenPause` flag bit is set, the speech synthesizer will process the input text as necessary to the point where it is ready to begin producing speech output. At this point, the synthesizer will enter a paused state and return to the caller. When the application is ready to produce speech, it should call the `ContinueSpeech` function to begin speaking.

`synthVersion` The version number of the synthesizer.

Phoneme Information Records

Information about a phoneme is stored in a **phoneme information record.** Ordinarily, you use a phoneme information record to show the user how to enter text to represent a particular phoneme when the `'PHON'` input mode is activated. The `PhonemeInfo` data type defines a phoneme information record.

```
TYPE PhonemeInfo =
RECORD
   opCode:        Integer;    {opcode for the phoneme}
   phStr:         Str15;      {corresponding character string}
   exampleStr:    Str31;      {word that shows use of phoneme}
   hiliteStart:   Integer;    {offset from beginning of word }
                              { to beginning of phoneme sound}
   hiliteEnd:     Integer;    {offset from beginning of word }
                              { to end of phoneme sound}
END;
```

Field descriptions

`opCode` The opcode for the phoneme. For a list of English-language opcodes, see Table 4-3 on page 4-33.

`phStr` The string used to represent the phoneme. The string does not necessarily have a phonetic connection to the phoneme, but might simply be an abstract textual representation of it.

`exampleStr` An example word that illustrates use of the phoneme.

`hiliteStart` The number of characters in the example word that precede the portion of that word representing the phoneme.

`hiliteEnd` The number of characters between the beginning of the example word and the end of the portion of that word representing the phoneme.

You might use the information contained in the `hiliteStart` and `hiliteEnd` fields to highlight the characters in the example word that represent the phoneme.

Note that in order to obtain a phoneme information record for an individual phoneme, you must obtain a list of phonemes through a phoneme descriptor record, described next.

Phoneme Descriptor Records

By calling the `GetSpeechInfo` function with the `soPhonemeSymbols` selector, you can obtain a **phoneme descriptor record,** which describes all phonemes defined for the current synthesizer. The `PhonemeDescriptor` data type defines a phoneme descriptor record.

```
TYPE PhonemeDescriptor =
RECORD
   phonemeCount:     Integer;    {number of phonemes defined by current }
                                 { synthesizer}
   thePhonemes:      ARRAY[0..0] OF PhonemeInfo;
                                 {list of phoneme information records}
END;
```

Field descriptions

`phonemeCount` The number of phonemes that the current synthesizer defines. Typically, this will correspond to the number of phonemes in the language supported by the synthesizer.

`thePhonemes` An array of phoneme information records.

A common use for a phoneme descriptor record is to provide a graphical display to the user of all available phonemes. Note that such a list would be useful only for a user entering phonemic data directly rather than just entering text.

Speech Extension Data Records

The **speech extension data record** allows you to use the `GetSpeechInfo` and `SetSpeechInfo` functions with selectors defined by particular synthesizers. By requiring that you pass to one of these functions a pointer to a speech extension data record, synthesizers can permit the exchange of data in any format. The `SpeechXtndData` data type defines a speech extension data record.

```
TYPE SpeechXtndData =
RECORD
   synthCreator:     OSType;     {synthesizer creator ID}
                                 {data used by synthesizer}
   synthData:        PACKED ARRAY[0..1] OF Char;
END;
```

Field descriptions

`synthCreator` The synthesizer's creator ID, identical to the value stored in the `synthManufacturer` field of a speech version information record.

	You should set this field to the appropriate value before calling `GetSpeechInfo` or `SetSpeechInfo`.
`synthData`	Synthesizer-specific data. The size and format of the data in this field may vary.

Delimiter Information Records

A **delimiter information record** defines the characters used to indicate the beginning and end of a command embedded in text. A delimiter can be one or two characters. The `DelimiterInfo` data type defines a delimiter information record.

```
TYPE DelimiterInfo =
RECORD
   startDelimiter:   PACKED ARRAY[0..1] OF Char;
   endDelimiter:     PACKED ARRAY[0..1] OF Char;
END;
```

Field descriptions

`startDelimiter`	The start delimiter for an embedded command. By default, the start delimiter is "`[[`".
`endDelimiter`	The end delimiter for an embedded command. By default, the end delimiter is "`]]`".

Ordinarily, applications that support embedded speech commands should not change the start or end delimiters. However, if for some reason you must change the delimiters, you can use the `SetSpeechInfo` function with the `soCommandDelimiter` selector. For example, you might do this if a text buffer naturally includes the delimiter strings. Before passing such a buffer to the Speech Manager, you can change the delimiter strings to some two-character sequences not used in the buffer and then change the delimiter strings back once processing of the buffer is complete.

If a single-byte delimiter is desired, it should be followed by a `NIL` (0) byte. If the delimiter strings both consist of two `NIL` bytes, embedded command processing is disabled.

Speech Manager Routines

This section describes the routines provided by the Speech Manager. You can use these routines to

- generate speech and then pause or stop it
- obtain information about an individual voice or all voices
- create and dispose of speech channels
- obtain the Speech Manager's version and status
- change the rate or pitch of speech

- convert textual into phonetic data
- install a pronunciation dictionary into a speech channel

With the exception of the `SpeechManagerVersion`, `SpeechBusy`, and `SpeechBusySystemWide` functions, all Speech Manager routines return a result code to indicate whether an error has occurred.

The section "Application-Defined Routines" beginning on page 4-82 describes the syntax and operation of application-defined callback procedures.

Starting, Stopping, and Pausing Speech

You can use the `SpeakString` function to generate speech from strings of fewer than 256 characters. The `SpeakText` function also generates speech, but through a speech channel through which you can exert control over the generated speech. The `SpeakBuffer` function includes all the capabilities of `SpeakText` and allows you to set certain flags that control speech behavior.

To stop speech, use the `StopSpeech` function or the `StopSpeechAt` function. The latter provides control over when speech is stopped. To pause and later resume speech, use the `PauseSpeechAt` and `ContinueSpeech` functions.

SpeakString

You can use the `SpeakString` function to have the Speech Manager speak a text string.

```
FUNCTION SpeakString (s: Str255): OSErr;
```

`s` The string to be spoken.

DESCRIPTION

The `SpeakString` function attempts to speak the Pascal-style text string contained in the string `s`. Speech is produced asynchronously using the default system voice. When an application calls this function, the Speech Manager makes a copy of the passed string and creates any structures required to speak it. As soon as speaking has begun, control is returned to the application. The synthesized speech is generated asynchronously to the application so that normal processing can continue while the text is being spoken. No further interaction with the Speech Manager is required at this point, and the application is free to release the memory that the original string occupied.

If `SpeakString` is called while a prior string is still being spoken, the sound currently being synthesized is interrupted immediately. Conversion of the new text into speech is then begun. If you pass a zero-length string (or, in C, a `null` pointer) to `SpeakString`, the Speech Manager stops any speech previously being synthesized by `SpeakString` without generating additional speech. If your application uses `SpeakString`, it is often a good idea to stop any speech in progress whenever your application receives a

suspend event. (Note, however, that calling `SpeakString` with a zero-length string has no effect on speech channels other than the one managed internally by the Speech Manager for the `SpeakString` function.)

The text passed to the `SpeakString` function may contain embedded speech commands.

SPECIAL CONSIDERATIONS

Because the `SpeakString` function moves memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SpeakString` function are

Trap macro	Selector
`_SoundDispatch`	$0220000C

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory to speak
`synthOpenFailed`	–241	Could not open another speech synthesizer channel

SpeakText

You can use the `SpeakText` function to have the Speech Manager speak a buffer of text.

```
FUNCTION SpeakText (chan: SpeechChannel; textBuf: Ptr;
                    textBytes: LongInt): OSErr;
```

`chan`	The speech channel through which speech is to be spoken.
`textBuf`	A pointer to the first byte of text to spoken.
`textBytes`	The number of bytes of text to spoken.

DESCRIPTION

The `SpeakText` function converts the text stream specified by the `textBuf` and `textBytes` parameters into speech using the voice and control settings for the speech channel `chan`, which should be created with the `NewSpeechChannel` function. The speech is generated asynchronously. This means that control is returned to your application before the speech has finished (and probably even before it has begun). The maximum length of the text buffer that can be spoken is limited only by the available RAM.

If `SpeakText` is called while the channel is currently busy speaking the contents of a prior text buffer, it immediately stops speaking from the prior buffer and begins speaking from the new text buffer as soon as possible. If you pass a zero-length string (or, in C, a `null` pointer) to `SpeakText`, the Speech Manager stops all speech currently being synthesized by the speech channel specified in the `chan` parameter without generating additional speech.

▲ **WARNING**
The text buffer must be locked in memory and must not move while the Speech Manager processes it. This buffer is read at interrupt time, and moving it could cause a system crash. If your application defines a text-done callback procedure, then it can move the text buffer or dispose of it once the callback procedure is executed. ▲

SPECIAL CONSIDERATIONS

Because the `SpeakText` function moves memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SpeakText` function are

Trap macro	Selector
`_SoundDispatch`	$0624000C

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

SpeakBuffer

You can use the `SpeakBuffer` function to have the Speech Manager speak a buffer of text, using certain flags to control speech behavior.

```
FUNCTION SpeakBuffer (chan: SpeechChannel; textBuf: Ptr;
                      textBytes: LongInt;
                      controlFlags: LongInt): OSErr;
```

`chan` The speech channel through which speech is to be spoken.

`textBuf` A pointer to the first byte of text to spoken.

`textBytes` The number of bytes of text to spoken.

`controlFlags`
Control flags to customize speech behavior.

DESCRIPTION

The SpeakBuffer function behaves identically to the SpeakText function, but allows control of several speech parameters by setting values of the controlFlags parameter. The controlFlags parameter relies on the following constants, which may be applied additively:

```
CONST
   kNoEndingProsody        = 1;  {disable prosody at end of sentences}
   kNoSpeechInterrupt      = 2;  {do not interrupt current speech}
   kPreflightThenPause     = 4;  {compute speech without generating}
```

Each constant specifies a flag bit of the controlFlags parameter, so by passing the constants additively you can enable multiple capabilities of SpeakBuffer. If you pass 0 in the controlFlags parameter, SpeakBuffer works just like SpeakText. By passing kNoEndingProsody + kNoSpeechInterrupt in the controlFlags parameter, SpeakBuffer works like SpeakText except that the kNoEndingProsody and kNoSpeechInterrupt features have been selected. Future versions of the Speech Manager may define additional constants.

The kNoEndingProsody flag bit is used to control whether or not the speech synthesizer automatically applies ending prosody, the speech tone and cadence that normally occur at the end of a statement. Under normal circumstances (for example, when the flag bit is not set), ending prosody is applied to the speech when the end of the textBuf data is reached. This default behavior can be disabled by setting the kNoEndingProsody flag bit.

Some synthesizers do not speak until the kNoEndingProsody flag bit is reset, or they encounter a period in the text, or textBuf is full.

The kNoSpeechInterrupt flag bit is used to control the behavior of SpeakBuffer when called on a speech channel that is still busy. When the flag bit is not set, SpeakBuffer behaves similarly to SpeakString and SpeakText. Any speech currently being produced on the specified speech channel is immediately interrupted, and then the new text buffer is spoken. When the kNoSpeechInterrupt flag bit is set, however, a request to speak on a channel that is still busy processing a prior text buffer will result in an error. The new buffer is ignored and the error synthNotReady is returned. If the prior text buffer has been fully processed, the new buffer is spoken normally. One way of achieving continuous speech without using callback procedures is to continually call SpeakBuffer with the kNoSpeechInterrupt flag bit set until the function returns noErr. The function will then execute as soon as the first text buffer has been processed.

The kPreflightThenPause flag bit is used to minimize the latency experienced when the speech synthesizer is attempting to speak. Ordinarily, whenever a call to SpeakString, SpeakText, or SpeakBuffer is made, the speech synthesizer must perform a certain amount of initial processing before speech output is heard. This startup latency can vary from a few milliseconds to several seconds depending upon which speech synthesizer is being used. Recognizing that larger startup delays might be detrimental to certain applications, a mechanism exists to allow the synthesizer to

perform any necessary computations at noncritical times. Once the computations have been completed, the speech is able to start instantly. When the `kPreflightThenPause` flag bit is set, the speech synthesizer will process the input text as necessary to the point where it is ready to begin producing speech output. At this point, the synthesizer will enter a paused state and return to the caller. When the application is ready to produce speech, it should call the `ContinueSpeech` function to begin speaking.

When the `controlFlags` parameter is set to 0, `SpeakBuffer` behaves identically to `SpeakText`.

SPECIAL CONSIDERATIONS

Because the `SpeakBuffer` function might move memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SpeakBuffer` function are

Trap macro	Selector
`_SoundDispatch`	$0828000C

RESULT CODES

`noErr`	0	No error
`synthNotReady`	–242	Speech channel is still busy speaking
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

StopSpeech

You can use the `StopSpeech` function to terminate speech immediately on a specified channel.

```
FUNCTION StopSpeech (chan: SpeechChannel): OSErr;
```

`chan` The speech channel on which speech is to be stopped.

DESCRIPTION

The `StopSpeech` function immediately terminates speech on the channel specified by the `chan` parameter. After returning from `StopSpeech`, your application can safely release any text buffer that the speech synthesizer has been using. You can call `StopSpeech` for an already idle channel without ill effect.

You can also stop speech by passing a zero-length string (or, in C, a `null` pointer) to one of the `SpeakString`, `SpeakText`, or `SpeakBuffer` functions. Doing this stops speech

only in the specified speech channel (or, in the case of `SpeakString`, in the speech channel managed internally by the Speech Manager).

SPECIAL CONSIDERATIONS

Because the `StopSpeech` function might move or purge memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `StopSpeech` function are

Trap macro	Selector
`_SoundDispatch`	$022C000C

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

SEE ALSO

Before calling the `StopSpeech` function, you can use the `SpeechBusy` function, which is described on page 4-72, to determine if a synthesizer is still speaking. If you are working with multiple speech channels, you can use the status selector with the routine `GetSpeechInfo` which is described on page 4-77, to determine if a specific channel is still speaking.

StopSpeechAt

You can use the `StopSpeechAt` function to terminate speech delivery on a specified channel either immediately or at the end of the current word or sentence.

```
FUNCTION StopSpeechAt (chan: SpeechChannel; whereToStop: LongInt)
                       : OSErr;
```

`chan` The speech channel on which speech is to be stopped.

`whereToStop`
A constant indicating when speech processing should stop. Pass the constant `kImmediate` to stop immediately, even in the middle of a word. Pass `kEndOfWord` or `kEndOfSentence` to stop speech at the end of the current word or sentence, respectively.

DESCRIPTION

The `StopSpeechAt` function halts the production of speech on the channel specified by `chan` at a specified point in the text. This routine returns immediately, although speech output continues until the specified point has been reached.

▲ **WARNING**
If you call the `StopSpeechAt` function before the Speech Manager finishes processing input text, then the function might return before some input text has yet to be spoken. Thus, before disposing of the text buffer, your application should wait until its text-done callback procedure has been called (if one has been defined), or until it can determine (by, for example obtaining a speech status information record) that the Speech Manager is no longer processing input text. ▲

If the end of the input text buffer is reached before the specified stopping point, the speech synthesizer stops at the end of the buffer without generating an error.

SPECIAL CONSIDERATIONS

Because the `StopSpeechAt` function might move or purge memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `StopSpeechAt` function are

Trap macro	Selector
`_SoundDispatch`	$0430000C

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

PauseSpeechAt

You can use the `PauseSpeechAt` function to pause speech on a speech channel.

```
FUNCTION PauseSpeechAt (chan: SpeechChannel; whereToStop: LongInt)
                        : OSErr;
```

`chan` The speech channel on which speech is to be paused.

`whereToStop`
A constant indicating when speech processing should be paused. Pass the constant `kImmediate` to pause immediately, even in the middle of a word. Pass `kEndOfWord` or `kEndOfSentence` to pause speech at the end of the current word or sentence, respectively.

DESCRIPTION

The `PauseSpeechAt` function makes speech production pause at a specified point in the text. `PauseSpeechAt` returns immediately, although speech output will continue until the specified point.

You can determine whether your application has paused speech output on a speech channel by obtaining a speech status information record through the `GetSpeechInfo` function. While a speech channel is paused, the speech status information record indicates that `outputBusy` and `outputPaused` are both `TRUE`.

If the end of the input text buffer is reached before the specified pause point, speech output pauses at the end of the buffer.

The `PauseSpeechAt` function differs from the `StopSpeech` and `StopSpeechAt` functions in that a subsequent call to `ContinueSpeech`, described next, causes the contents of the current text buffer to continue being spoken.

▲ **WARNING**
If you plan to continue speech synthesis from a paused speech channel, the text buffer being processed must remain available at all times and must not move while the channel is in a paused state. ▲

SPECIAL CONSIDERATIONS

Because the `PauseSpeechAt` function might move or purge memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PauseSpeechAt` function are

Trap macro	Selector
`_SoundDispatch`	$0434000C

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

ContinueSpeech

You can use the `ContinueSpeech` function to resume speech paused by the `PauseSpeechAt` function.

```
FUNCTION ContinueSpeech (chan: SpeechChannel): OSErr;
```

`chan` The paused speech channel on which speech is to be resumed.

DESCRIPTION

At any time after the `PauseSpeechAt` function is called, the `ContinueSpeech` function can be called to continue speaking from the beginning of the word in which speech paused. Calling `ContinueSpeech` on a channel that is not currently in a paused state has no effect on the speech channel or on future calls to the `PauseSpeechAt` function. If you call `ContinueSpeech` on a channel before a pause is effective, `ContinueSpeech` cancels the pause.

If the `PauseSpeechAt` function stopped speech in the middle of a word, the Speech Manager will start speaking that word from the beginning when you call `ContinueSpeech`.

SPECIAL CONSIDERATIONS

Because the `ContinueSpeech` function moves memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `ContinueSpeech` function are

Trap macro	Selector
`_SoundDispatch`	$0238000C

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

Obtaining Information About Voices

Specification of a voice requires a voice specification record. When you already know the creator and ID for a voice, you should use the `MakeVoiceSpec` function to create such a record rather than filling in the fields of one directly. To obtain information about all available voices, use the `CountVoices` function to determine how many voices are available, and the `GetIndVoice` function to obtain a voice specification record corresponding to each voice.

Having created a voice specification record, you can obtain information about the voice to which it corresponds. The `GetVoiceDescription` function provides information about a voice in the form of a voice description record. In addition to duplicating the capabilities of the `GetVoiceDescription` function, the `GetVoiceInfo` function allows you to obtain information about where on disk a voice is stored.

MakeVoiceSpec

To set the fields of a voice specification record, you should use the `MakeVoiceSpec` function. You should never set the fields of such a record directly.

```
FUNCTION MakeVoiceSpec (creator: OSType; id: OSType;
                        voice: VoiceSpecPtr): OSErr;
```

`creator` The ID of the synthesizer that your application requires.

`id` The ID of the voice on the synthesizer specified by the `creator` parameter.

`voice` A pointer to the voice specification record whose fields are to be filled in.

DESCRIPTION

A voice specification record is a unique voice ID used by the Speech Manager. Most voice management routines expect to be passed a pointer to a voice specification record. When you already know the creator and ID for a voice, you should use the `MakeVoiceSpec` function to create such a record rather than filling in the fields of one directly. On exit, the voice specification record pointed to by the `voice` parameter contains the appropriate values.

SPECIAL CONSIDERATIONS

You can call the `MakeVoiceSpec` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `MakeVoiceSpec` function are

Trap macro	Selector
`_SoundDispatch`	$0604000C

RESULT CODES

`noErr`	0	No error

CountVoices

You can determine how many voices are available by calling the `CountVoices` function.

```
FUNCTION CountVoices (VAR numVoices: Integer): OSErr;
```

`numVoices` On exit, the number of voices that the application can use.

DESCRIPTION

The `CountVoices` function returns, in the `numVoices` parameter, the number of voices available. The application can then use this information to call the `GetIndVoice` function, described next, to obtain voice specification records for one or more of the voices.

Each time `CountVoices` is called, the Speech Manager searches for new voices.

SPECIAL CONSIDERATIONS

Because the `CountVoices` function moves memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `CountVoices` function are

Trap macro	Selector
`_SoundDispatch`	$0108000C

RESULT CODES

`noErr`	0	No error

GetIndVoice

You can obtain a voice specification record for a voice by passing an index to the `GetIndVoice` function.

```
FUNCTION GetIndVoice (index: Integer; voice: VoiceSpecPtr): OSErr;
```

`index` The index of the voice for which to obtain a voice specification record. This number must range from 1 to the total number of voices, as returned by the `CountVoices` function.

`voice` A pointer to the voice specification record whose fields are to be filled in.

DESCRIPTION

The `GetIndVoice` function returns, in the voice specification record pointed to by the `voice` parameter, a specification of the voice whose index is provided in the `index` parameter. Your application should make no assumptions about the order in which voices are indexed.

▲ **WARNING**
Your application should not add, remove, or modify a voice and then call the `GetIndVoice` function with an index value other than 1. To allow the Speech Manager to update its information about voices, your application should always either call the `CountVoices` function or call the `GetIndVoice` function with an index value of 1 after adding, removing, or modifying a voice or after a time at which the user might have done so. ▲

If you specify an index value beyond the number of available voices, the `GetIndVoice` function returns a `voiceNotFound` error.

SPECIAL CONSIDERATIONS

Because the `GetIndVoice` function moves memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetIndVoice` function are

Trap macro	Selector
`_SoundDispatch`	$030C000C

RESULT CODES

`noErr`	0	No error
`voiceNotFound`	–244	Voice resource not found

GetVoiceDescription

You can obtain a description of a voice by using the `GetVoiceDescription` function.

```
FUNCTION GetVoiceDescription (voice: VoiceSpecPtr;
                              info: VoiceDescriptionPtr;
                              infoLength: LongInt): OSErr;
```

`voice` A pointer to the voice specification record identifying the voice to be described, or `NULL` to obtain a description of the system default voice.

`info` A pointer to a voice description record. If this parameter is `NULL`, the function does not fill in the fields of the voice description record; instead, it simply determines whether the `voice` parameter specifies an available voice and, if not, returns a `voiceNotFound` error.

`infoLength`

The length, in bytes, of the voice description record. In the current version of the Speech Manager, the voice description record contains 362 bytes. However, you should always use the `SizeOf` function to determine the length of this record.

DESCRIPTION

The `GetVoiceDescription` function fills out the voice description record pointed to by the `info` parameter with the correct information for the voice specified by the `voice` parameter. It fills in the `length` field of the voice description record with the number of bytes actually copied. This value will always be less than or equal to the value that your application passes in `infoLength` before calling `GetVoiceDescription`. This scheme allows applications targeted for the current version of the Speech Manager to work on future versions that might have longer voice description records; it also allows you to write code for future versions of the Speech Manager that will also run on computers that support only the current version.

If the voice specification record does not identify an available voice, `GetVoiceDescription` returns a `voiceNotFound` error.

SPECIAL CONSIDERATIONS

Because the `GetVoiceDescription` function moves memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetVoiceDescription` function are

Trap macro	Selector
`_SoundDispatch`	$0610000C

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Parameter error
`memFullErr`	–108	Not enough memory to load voice into memory
`voiceNotFound`	–244	Voice resource not found

GetVoiceInfo

You can use the `GetVoiceInfo` function to obtain the same information about a voice that the `GetVoiceDescription` function provides or to determine in which file and

resource a voice is stored. This function is intended primarily for use by synthesizers, but an application can call it too.

```
FUNCTION GetVoiceInfo (voice: VoiceSpecPtr; selector: OSType;
                       voiceInfo: Ptr): OSErr;
```

`voice` A pointer to the voice specification record identifying the voice about which your application requires information, or `NIL` to obtain information on the system default voice.

`selector` A specification of the type of data being requested. For current versions of the Speech Manager, you should set this field either to `soVoiceDescription`, if you would like to use the `GetVoiceInfo` function to mimic the `GetVoiceDescription` function, or to `soVoiceFile`, if you would like to obtain information about the location of a voice on disk.

`voiceInfo` A pointer to the appropriate data structure. If the selector is `soVoiceDescription`, then `voiceInfo` should be a pointer to a voice description record, and the `length` field of the record should be set to the length of the voice description record. If the selector is `soVoiceFile`, then `voiceInfo` should be a pointer to a voice file information record.

DESCRIPTION

The `GetVoiceInfo` function accepts a selector in the `selector` parameter that determines the type of information you wish to obtain about the voice specified in the `voice` parameter. The function then fills the fields of the data structure appropriate to the selector you specify in the `voiceInfo` parameter.

If the voice specification is invalid, `GetVoiceInfo` returns a `voiceNotFound` error. If there is not enough memory to load the voice into memory to obtain information about it, `GetVoiceInfo` returns the result code `memFullErr`.

SPECIAL CONSIDERATIONS

Because the `GetVoiceInfo` function might move memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetVoiceInfo` function are

Trap macro	Selector
`_SoundDispatch`	$0614000C

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory to load voice into memory
`voiceNotFound`	–244	Voice resource not found

Managing Speech Channels

To take advantage of any but the most rudimentary of the Speech Manager's capabilities, you need to use speech channels. However, you cannot create a speech channel simply by declaring a variable of type `SpeechChannel`. Before your application calls any routine that requires a speech channel as a parameter, you must call the `NewSpeechChannel` function to allow the Speech Manager to allocate memory associated with the speech channel. Later, you can release the memory occupied by a speech channel by calling the `DisposeSpeechChannel` function. In general, it is a good idea to create a speech channel just before you need it and then dispose of it as soon as you have finished processing speech through it.

NewSpeechChannel

You can use the `NewSpeechChannel` function to create a new speech channel.

```
FUNCTION NewSpeechChannel (voice: VoiceSpecPtr;
                           VAR chan: SpeechChannel): OSErr;
```

`voice` A pointer to the voice specification record corresponding to the voice to be used for the new speech channel. Pass `NIL` to create a speech channel using the system default voice.

`chan` On exit, a valid speech channel.

DESCRIPTION

The `NewSpeechChannel` function allocates memory for a speech channel record and sets the speech channel variable pointed to by the `chan` parameter to point to this speech channel record. The Speech Manager automatically locates and opens a connection to the proper synthesizer for the voice specified by the `voice` parameter.

There is no predefined limit to the number of speech channels an application can create. However, system constraints on available RAM, processor loading, and number of available sound channels limit the number of speech channels actually possible.

▲ **WARNING**
Your application should not attempt to manipulate the data pointed to by a variable of type `SpeechChannel`. The internal format that the Speech Manager uses for speech channel data is not documented and may change in future versions of system software. ▲

SPECIAL CONSIDERATIONS

Because the `NewSpeechChannel` function allocates memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `NewSpeechChannel` function are

Trap macro	Selector
`_SoundDispatch`	$0418000C

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory to open speech channel
`synthOpenFailed`	–241	Could not open another speech synthesizer channel
`voiceNotFound`	–244	Voice resource not found

DisposeSpeechChannel

You can use the `DisposeSpeechChannel` function to dispose of an existing speech channel.

```
FUNCTION DisposeSpeechChannel (chan: SpeechChannel): OSErr;
```

`chan` The speech channel to dispose of.

DESCRIPTION

The `DisposeSpeechChannel` function disposes of the speech channel specified in the `chan` parameter and releases all memory the channel occupies. If the speech channel specified is producing speech, then the `DisposeSpeechChannel` function immediately stops speech before disposing of the channel. If you have defined a text-done callback procedure or a speech-done callback procedure, the procedure will not be called before the channel is disposed of.

The Speech Manager releases any speech channels that have not been explicitly disposed of by an application when the application quits. In general, however, your application should dispose of any speech channels it has created whenever it receives a suspend event. This ensures that other applications can take full advantage of Speech Manager and Sound Manager capabilities.

SPECIAL CONSIDERATIONS

Because the `DisposeSpeechChannel` function might purge memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DisposeSpeechChannel` function are

Trap macro	Selector
`_SoundDispatch`	$021C000C

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

Obtaining Information About Speech

Once you have determined with the Gestalt Manager that the Speech Manager is present, you can use the `SpeechManagerVersion` function to determine what version is available.

To determine how many speech channels are currently processing speech in your application, you can use the `SpeechBusy` function. To determine how many are processing speech in your application and other processes, you can use the `SpeechBusySystemWide` function.

SpeechManagerVersion

You can use the `SpeechManagerVersion` function to determine the current version of the Speech Manager installed in the system.

```
FUNCTION SpeechManagerVersion: NumVersion;
```

DESCRIPTION

The `SpeechManagerVersion` function returns the version of the Speech Manager installed in the system, in the format of the first 4 bytes of a `'vers'` resource. You can use this call to determine whether your program can access features of the Speech Manager that are included in some Speech Manager releases but not in earlier ones. Note, however, that because this chapter documents the initial release of the Speech Manager, all features and techniques described in this chapter should be available in all versions of the Speech Manager.

SPECIAL CONSIDERATIONS

You can call the `SpeechManagerVersion` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SpeechManagerVersion` function are

Trap macro	Selector
`_SoundDispatch`	$0000000C

SpeechBusy

You can use the `SpeechBusy` function to determine whether any channels of speech are currently synthesizing speech.

```
FUNCTION SpeechBusy: Integer;
```

DESCRIPTION

The `SpeechBusy` function returns the number of speech channels that are currently synthesizing speech in the application. This is useful when you want to ensure that an earlier speech request has been completed before having the system speak again. Note that paused speech channels are counted among those that are synthesizing speech.

The speech channel that the Speech Manager allocates internally in response to calls to the `SpeakString` function is counted in the number returned by `SpeechBusy`. Thus, if you use just `SpeakString` to initiate speech, `SpeechBusy` always returns 1 as long as speech is being produced. When `SpeechBusy` returns 0, all speech has finished.

SPECIAL CONSIDERATIONS

You can call the `SpeechBusy` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SpeechBusy` function are

Trap macro	Selector
`_SoundDispatch`	$003C000C

SpeechBusySystemWide

You can use the `SpeechBusySystemWide` function to determine if any speech is currently being synthesized in your application or elsewhere on the computer.

```
FUNCTION SpeechBusySystemWide: Integer;
```

DESCRIPTION

The `SpeechBusySystemWide` function returns the total number of speech channels currently synthesizing speech on the computer, whether they were initiated by your application or process's code or by some other process executing concurrently. Note that paused speech channels are counted among those channels that are synthesizing speech.

This function is useful when you want to ensure that no speech is currently being produced anywhere on the Macintosh computer before initiating speech. Although the Speech Manager allows different applications to produce speech simultaneously, this can be confusing to the user. As a result, it is often a good idea for your application to check that no other process is producing speech before producing speech itself. If the difference between the values returned by `SpeechBusySystemWide` and the `SpeechBusy` function is 0, no other process is producing speech.

SPECIAL CONSIDERATIONS

You can call the `SpeechBusySystemWide` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SpeechBusySystemWide` function are

Trap macro	Selector
`_SoundDispatch`	$0040000C

Changing Speech Attributes

To determine the rate and pitch at which a speech channel is processing text, you can use the `GetSpeechRate` and `GetSpeechPitch` functions. The `SetSpeechRate` and `SetSpeechPitch` functions allow you to change rate and pitch.

The most robust of the Speech Manager's routines are the `GetSpeechInfo` and `SetSpeechInfo` functions. These allow you to obtain many types of information about a speech channel and to change many settings of a speech channel. To specify the operation that you wish to perform, you must pass `GetSpeechInfo` or `SetSpeechInfo` a selector. A full list of selectors is provided in "Speech Information Selectors" beginning on page 4-39.

GetSpeechRate

You use the `GetSpeechRate` function to obtain a speech channel's current speech rate.

```
FUNCTION GetSpeechRate (chan: SpeechChannel; VAR rate: Fixed)
                        : OSErr;
```

`chan` The speech channel whose rate you wish to determine.

`rate` On exit, the speech channel's speech rate, expressed as a fixed-point, words-per-minute value.

DESCRIPTION

The `GetSpeechRate` function returns, in the `rate` parameter, the speech rate of the speech channel specified by the `chan` parameter.

SPECIAL CONSIDERATIONS

You can call the `GetSpeechRate` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetSpeechRate` function are

Trap macro	Selector
`_SoundDispatch`	$0448000C

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

SetSpeechRate

You can set the speech rate of a designated speech channel with the `SetSpeechRate` function.

```
FUNCTION SetSpeechRate (chan: SpeechChannel; rate: Fixed): OSErr;
```

`chan` The speech channel whose rate you wish to set.

`rate` The new speech rate for the speech channel, expressed as a fixed-point, words-per-minute value.

DESCRIPTION

The `SetSpeechRate` function adjusts the speech rate on the speech channel specified by the `chan` parameter to the rate specified by the `rate` parameter. As a general rule, typical speaking rates range from around 150 words per minute to around 180 words per minute. It is important to keep in mind, however, that users will differ greatly in their ability to understand synthesized speech at a particular rate based upon their level of experience listening to the voice and their ability to anticipate the types of utterances they will encounter.

SPECIAL CONSIDERATIONS

You can call the `SetSpeechRate` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SetSpeechRate` function are

Trap macro	Selector
`_SoundDispatch`	$0444000C

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

GetSpeechPitch

You can determine a speech channel's current speech pitch by using the `GetSpeechPitch` function.

```
FUNCTION GetSpeechPitch (chan: SpeechChannel; VAR pitch: Fixed)
                            : OSErr;
```

`chan` The speech channel whose pitch you wish to determine.

`pitch` On exit, the current pitch of the voice in the speech channel, expressed as a fixed-point frequency value.

DESCRIPTION

The `GetSpeechPitch` function returns, in the `pitch` parameter, the pitch of the speech channel specified by the `chan` parameter. Typical voice frequencies range from around 90 hertz for a low-pitched male voice to perhaps 300 hertz for a high-pitched child's voice. These frequencies correspond to approximate pitch values in the ranges of 30.000 to 40.000 and 55.000 to 65.000, respectively. For information about the mathematical relationship between pitches and frequencies expressed in hertz, see "Speech Attributes" beginning on page 4-6.

SPECIAL CONSIDERATIONS

You can call the `GetSpeechPitch` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetSpeechPitch` function are

Trap macro	Selector
`_SoundDispatch`	$0450000C

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

SetSpeechPitch

You can use the `SetSpeechPitch` function to set the speech pitch on a designated speech channel.

```
FUNCTION SetSpeechPitch (chan: SpeechChannel; pitch: Fixed)
                            : OSErr;
```

`chan`	The speech channel whose pitch you wish to set.
`pitch`	The new pitch for the speech channel, expressed as a fixed-point frequency value.

DESCRIPTION

The `SetSpeechPitch` function changes the current speech pitch on the speech channel specified by the `chan` parameter to the pitch specified by the `pitch` parameter. Typical voice frequencies range from around 90 hertz for a low-pitched male voice to perhaps 300 hertz for a high-pitched child's voice. These frequencies correspond to approximate pitch values in the ranges of 30.000 to 40.000 and 55.000 to 65.000, respectively. For information about the mathematical relationship between pitches and frequencies expressed in hertz, see "Speech Attributes" beginning on page 4-6. Although fixed-point values allow you to specify a wide range of pitches, not all synthesizers will support the full range of pitches. If your application specifies a pitch that a synthesizer cannot handle, it may adjust the pitch to fit within an acceptable range.

SPECIAL CONSIDERATIONS

You can call the `SetSpeechPitch` function at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SetSpeechPitch` function are

Trap macro	Selector
`_SoundDispatch`	$044C000C

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

GetSpeechInfo

You can use the `GetSpeechInfo` function to obtain information about a designated speech channel.

```
FUNCTION GetSpeechInfo (chan: SpeechChannel; selector: OSType;
                        speechInfo: Ptr): OSErr;
```

`chan` The speech channel about which information is being requested.

`selector` A speech information selector that indicates the type of information being requested.

`speechInfo`
A pointer whose meaning depends on the speech information selector specified in the `selector` parameter.

DESCRIPTION

The `GetSpeechInfo` function returns, in the data structure pointed to by the `speechInfo` parameter, the type of information requested by the `selector` parameter as it applies to the speech channel specified in the `chan` parameter.

The format of the data structure specified by the `speechInfo` parameter depends on the selector you choose. For example, a selector might require that your application allocate a block of memory of a certain size and pass a pointer to that block. Another selector might require that `speechInfo` be set to the address of a handle variable. In this case, the `GetSpeechInfo` function would allocate a relocatable block of memory and change the handle variable specified to reference the block.

SPECIAL CONSIDERATIONS

You can call the `GetSpeechInfo` function at interrupt time only if the speech information selector specified in the `selector` parameter does not move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetSpeechInfo` function are

Trap macro	Selector
`_SoundDispatch`	$0658000C

RESULT CODES

`noErr`	0	No error
`siUnknownInfoType`	–231	Feature is not implemented on synthesizer
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

SEE ALSO

For a complete list of speech information selectors, see "Speech Information Selectors" beginning on page 4-39. This list indicates how your application should set the `speechInfo` parameter for each selector and indicates which selectors might cause memory to be moved or purged.

SetSpeechInfo

You can use the `SetSpeechInfo` function to change a setting of a particular speech channel.

```
FUNCTION SetSpeechInfo (chan: SpeechChannel; selector: OSType;
                        speechInfo: Ptr): OSErr;
```

`chan` The speech channel for which your application wishes to change a setting.

`selector` A speech information selector that indicates the type of information being changed.

`speechInfo`
A pointer whose meaning depends on the speech information selector specified in the `selector` parameter.

DESCRIPTION

The `SetSpeechInfo` function changes the type of setting indicated by the `selector` parameter in the speech channel specified by the chan parameter, based on the data your application provides via the `speechInfo` parameter.

The format of the data structure specified by the `speechInfo` parameter depends on the selector you choose. Ordinarily, a selector requires that `speechInfo` be a pointer to a data structure that specifies a new setting for the speech channel.

SPECIAL CONSIDERATIONS

You can call the `SetSpeechInfo` function at interrupt time only if the speech information selector specified in the `selector` parameter does not move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SetSpeechInfo` function are

Trap macro	Selector
`_SoundDispatch`	$0654000C

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Parameter value is invalid
`siUnknownInfoType`	–231	Feature is not implemented on synthesizer
`incompatibleVoice`	–245	Specified voice cannot be used with synthesizer
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

SEE ALSO

For a complete list of speech information selectors, see "Speech Information Selectors" beginning on page 4-39. This list indicates how your application should set the `speechInfo` parameter for each selector and indicates which selectors might cause memory to be moved or purged.

Converting Text To Phonemes

The Speech Manager provides a utility routine, the `TextToPhonemes` function, to convert textual data into phonetic data. This is particularly useful during application development, when you might wish to adjust phrases that your application generates to produce smoother speech. By first converting the target phrase into phonemes, you can see what the synthesizer will try to speak. Then you need correct only the parts that would not have been spoken the way you want.

TextToPhonemes

You can use the `TextToPhonemes` function to convert textual data into phonemic data.

```
FUNCTION TextToPhonemes (chan: SpeechChannel; textBuf: Ptr;
                         textBytes: LongInt; phonemeBuf: Handle;
                         VAR phonemeBytes: LongInt): OSErr;
```

`chan` A speech channel whose associated synthesizer and voice are to be used for the conversion process.

`textBuf` A pointer to a buffer of text to be converted.

`textBytes` The number of bytes of text to be converted.

`phonemeBuf`
A handle to a buffer to be used to store the phonemic data. The `TextToPhonemes` function may resize the relocatable block referenced by this handle.

`phonemeBytes`
On exit, the number of bytes of phonemic data written to the handle.

DESCRIPTION

The `TextToPhonemes` function converts the `textBytes` bytes of textual data pointed to by the `textBuf` parameter to phonemic data, which it writes into the relocatable block specified by the `phonemeBuf` parameter. If necessary, `TextToPhonemes` resizes this relocatable block. The `TextToPhonemes` function sets the `phonemeBytes` parameter to the number of bytes of phonetic data actually written.

▲ **WARNING**
If the textual data is contained in a relocatable block, a handle to that block must be locked before the `TextToPhonemes` function is called. ▲

The data returned by `TextToPhonemes` corresponds precisely to the phonemes that would be spoken had the input text been sent to `SpeakText` instead. All current mode settings for the speech channel specified by `chan` are applied to the converted speech. No callbacks are generated while the `TextToPhonemes` routine is generating its output.

SPECIAL CONSIDERATIONS

Because the `TextToPhonemes` function might move memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `TextToPhonemes` function are

Trap macro	Selector
`_SoundDispatch`	$0A5C000C

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Parameter value is invalid
`nilHandleErr`	–109	Handle argument is `NIL`
`siUnknownInfoType`	–231	Feature not implemented on synthesizer
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

Installing a Pronunciation Dictionary

Pronunciation dictionaries allow your application to override the default Speech Manager pronunciations of individual words, such as names with quirky spellings. The `UseDictionary` function allows your application to load a pronunciation dictionary into a speech channel.

UseDictionary

You can use the `UseDictionary` function to install a designated dictionary into a speech channel.

```
FUNCTION UseDictionary (chan: SpeechChannel; dictionary: Handle)
                        : OSErr;
```

`chan` The speech channel into which a dictionary is to be installed.

`dictionary`
A handle to the dictionary data. This is often a handle to a resource of type `'dict'`.

DESCRIPTION

The `UseDictionary` function attempts to install the dictionary data referenced by the `dictionary` parameter into the speech channel referenced by the `chan` parameter. The synthesizer will use whatever elements of the dictionary resource it considers useful to the speech conversion process. Some speech synthesizers might ignore certain types of dictionary entries.

After the `UseDictionary` function returns, your application is free to release any storage allocated for the dictionary handle. The search order for application-provided dictionaries is last-in, first-searched.

All details of how an application-provided dictionary is represented within the speech synthesizer are dependent on the specific synthesizer implementation and are private to the synthesizer.

SPECIAL CONSIDERATIONS

Because the `UseDictionary` function might move memory, you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `UseDictionary` function are

Trap macro	Selector
`_SoundDispatch`	$0460000C

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory to use new dictionary
`badDictFormat`	–246	Pronunciation dictionary format error
`invalidComponentID`	–3000	Speech channel is uninitialized or bad

SEE ALSO

For a description of the format of a pronunciation dictionary, see "The Pronunciation Dictionary Resource" on page 4-89. For a discussion of how you might manipulate a dictionary in memory, see "Including Pronunciation Dictionaries" beginning on page 4-36.

Application-Defined Routines

The Speech Manager allows you to define callback procedures that execute

- when text input processing is complete (but not necessarily after speech has stopped)
- when text has been completely processed and spoken
- whenever the Speech Manager encounters an embedded synchronization command
- whenever the Speech Manager encounters an error in processing embedded speech commands
- whenever a phoneme is about to be spoken
- whenever a word is about to be spoken

▲ **WARNING**
When the Speech Manager executes a callback procedure, the Speech Manager sets the A5 register to the value specified by the most recent call to the `SetSpeechInfo` function with the `soCurrentA5` selector. However, if the most recent value specified with the `soCurrentA5` selector is `NIL` or if your application has not yet specified a value, then the Speech Manager leaves the A5 register unchanged. In this case, the callback procedure cannot access application global variables because it executes at interrupt time. For code showing how to use the `soCurrentA5` selector to ensure that the A5 register is set to your application's A5, see Listing 4-6 on page 4-21. ▲

Text-Done Callback Procedure

You can specify a text-done callback procedure by passing the `soTextDoneCallBack` selector to the `SetSpeechInfo` function.

MyTextDoneCallback

A text-done callback procedure has the following syntax:

```
PROCEDURE MyTextDoneCallback
                              (chan: SpeechChannel; refCon: LongInt;
                              VAR nextBuf: Ptr; VAR byteLen: LongInt;
                              VAR controlFlags: LongInt);
```

`chan`	The speech channel that has finished processing input text.
`refCon`	The reference constant associated with the speech channel.
`nextBuf`	On exit, a pointer to the next buffer of text to process or `NIL` if your application has no additional text to be spoken. This parameter is mostly for internal use by the Speech Manager.
`byteLen`	On exit, the number of bytes of the text buffer pointed to by the `nextBuf` parameter.
`controlFlags`	On exit, the control flags to be used in generating the next buffer of text.

DESCRIPTION

If a text-done callback procedure is installed in a speech channel, then the Speech Manager calls this procedure when it finishes processing a buffer of text. The Speech Manager might not yet have completed finishing speaking the text and indeed might not have started speaking it.

A common use of a text-done callback procedure is to alert your application once the text passed to the `SpeakText` or `SpeakBuffer` function can be disposed of (or, when the text is contained within a locked relocatable block, when the relocatable block can be unlocked). The Speech Manager copies the text you pass to the `SpeakText` or `SpeakBuffer` function into an internal buffer. Once it has finished processing the text, you may dispose of the original text buffer, even if speech is not yet complete. However, if you wish to write a callback procedure that executes when speech is completed, see the definition of a speech-done callback procedure below.

Although most applications won't need to, your callback procedure can indicate to the Speech Manager whether there is another buffer of text to speak. If there is another buffer, your callback procedure should reference it by setting the `nextBuf` and `byteLen` parameters to appropriate values. (Your callback procedure might also change the control flags to be used to process the speech by altering the value in the `controlFlags` parameter.) Setting these parameters allows the Speech Manager to generate uninterrupted speech. If there is no more text to speak, your callback procedure should set `nextBuf` to `NIL`. In this case, the Speech Manager ignores the `byteLen` and `controlFlags` parameters.

If your text-done callback procedure does not change the values of the `nextBuf` and `byteLen` parameters, the text buffer just spoken will be spoken again.

SPECIAL CONSIDERATIONS

Because your callback procedure executes at interrupt time, you must not call any routines that might move or purge memory. If you are writing a callback procedure so that your application will know when it can dispose of a text buffer, then use the callback procedure to set a global flag variable. Your application's main event loop can check this flag and dispose of the text buffer if it is set.

Your callback procedure is able to access application global variables only if the A5 register is properly set. The Speech Manager sets A5 to the proper value if you provide

your application's A5 value by calling the `SetSpeechInfo` function with the `soCurrentA5` selector.

ASSEMBLY-LANGUAGE INFORMATION

Because a callback procedure is called at interrupt time, it must preserve all registers other than A0–A2 and D0–D2.

Speech-Done Callback Procedure

You can specify a speech-done callback procedure by passing the `soSpeechDoneCallBack` selector to the `SetSpeechInfo` function.

MySpeechDoneCallback

A speech-done callback procedure has the following syntax:

```
PROCEDURE MySpeechDoneCallback (chan: SpeechChannel;
                                refCon: LongInt);
```

`chan`	The speech channel that has finished processing input text.
`refCon`	The reference constant associated with the speech channel.

DESCRIPTION

If a speech-done callback procedure is installed in a speech channel, then the Speech Manager calls this procedure when it finishes speaking a buffer of text.

You might use a speech-done callback procedure if you need to update some visual indicator that shows what text is currently being spoken. For example, suppose your application passes text buffers to the Speech Manager one paragraph at a time. Your speech-done callback procedure might set a global flag variable to indicate to the application that the Speech Manager has finished speaking a paragraph. When a routine called by your application's main event loop checks the global flag variable and determines that it has been set, the routine might ensure that the next paragraph of text is visible.

You might use a speech-done callback procedure to set a flag variable that alerts the application that it should pass a new buffer of text to the Speech Manager. If you do so, however, there might be a noticeable pause as the Speech Manager switches from processing one text buffer to another. Ordinarily, it is easier to achieve this goal by using a text-done callback procedure, as described earlier.

SPECIAL CONSIDERATIONS

Because your callback procedure executes at interrupt time, you must not call any routines that might move or purge memory.

Your callback procedure is able to access application global variables only if the A5 register is properly set. The Speech Manager sets A5 to the proper value if you provide your application's A5 value by calling the `SetSpeechInfo` function with the `soCurrentA5` selector.

ASSEMBLY-LANGUAGE INFORMATION

Because a callback procedure is called at interrupt time, it must preserve all registers other than A0–A2 and D0–D2.

Synchronization Callback Procedure

You can specify a synchronization callback procedure by passing the `soSyncCallBack` selector to the `SetSpeechInfo` function and embedding a synchronization command within a text buffer passed to the `SpeakText` or `SpeakBuffer` function.

MySynchronizationCallback

A synchronization callback procedure has the following syntax:

```
PROCEDURE MySynchronizationCallback (chan: SpeechChannel;
                                     refCon: LongInt;
                                     syncMessage: OSType);
```

`chan` The speech channel that has finished processing input text.

`refCon` The reference constant associated with the speech channel.

`syncMessage`
The synchronization message passed in the embedded command. Usually, you use this message to distinguish between several different types of synchronization commands, but you can use it any way you wish.

DESCRIPTION

The Speech Manager calls a speech channel's synchronization callback procedure whenever it encounters a synchronization command embedded in a text buffer. You might use the synchronization callback procedure to provide a callback not ordinarily provided. For example, you might inset synchronization commands at the end of every sentence in a text buffer, or you might enter synchronization commands after every numeric value in the text. However, to synchronize your application with phonemes or words, it makes more sense to use the built-in phoneme and word callback procedures,

defined in "Phoneme Callback Procedure" on page 4-87 and "Word Callback Procedure" on page 4-88.

SPECIAL CONSIDERATIONS

Because your callback procedure executes at interrupt time, you must not call any routines that might move or purge memory. If you need to make a visual change in response to a synchronization command, then use the callback procedure to set a global flag variable. Your application's main event loop can check this flag and update the screen display if it is set.

Your callback procedure is able to access application global variables only if the A5 register is properly set. The Speech Manager sets A5 to the proper value if you provide your application's A5 value by calling the `SetSpeechInfo` function with the `soCurrentA5` selector.

ASSEMBLY-LANGUAGE INFORMATION

Because a callback procedure is called at interrupt time, it must preserve all registers other than A0–A2 and D0–D2.

Error Callback Procedure

You can specify an error callback procedure by passing the `soErrorCallBack` selector to the `SetSpeechInfo` function.

MyErrorCallback

An error callback procedure has the following syntax:

```
PROCEDURE MyErrorCallback (chan: SpeechChannel; refCon: LongInt;
                           error: OSErr; bytePos: LongInt);
```

`chan`	The speech channel that has finished processing input text.
`refCon`	The reference constant associated with the speech channel.
`error`	The error that occurred in processing an embedded command.
`bytePos`	The number of bytes from the beginning of the text buffer being spoken to the error encountered.

DESCRIPTION

The Speech Manager calls a speech channel's error callback procedure whenever it encounters a syntax error within a command embedded in a text buffer it is processing. This can be useful during application debugging, to detect problems with commands that you have embedded in text buffers that your application speaks. It can also be

useful if your application allows users to embed commands within text buffers. Your application might display an alert indicating that the Speech Manager encountered a problem in processing an embedded command.

Ordinarily, the error information that the Speech Manager provides the error callback procedure should be sufficient. However, if your application needs information about errors that occurred before the error callback procedure was enabled, the application (including the error callback procedure) can call the `GetSpeechInfo` function with the `soErrors` selector.

SPECIAL CONSIDERATIONS

Because your callback procedure executes at interrupt time, you must not call any routines that might move or purge memory. If you need to display an alert box to the user, then use the callback procedure to set a global flag variable. Your application's main event loop can check this flag and display the alert box if it is set.

Your callback procedure is able to access application global variables only if the A5 register is properly set. The Speech Manager sets A5 to the proper value if you provide your application's A5 value by calling the `SetSpeechInfo` function with the `soCurrentA5` selector.

ASSEMBLY-LANGUAGE INFORMATION

Because a callback procedure is called at interrupt time, it must preserve all registers other than A0–A2 and D0–D2.

Phoneme Callback Procedure

You can specify a phoneme callback procedure by passing the `soPhonemeCallBack` selector to the `SetSpeechInfo` function.

MyPhonemeCallback

A phoneme callback procedure has the following syntax:

```
PROCEDURE MyPhonemeCallback (chan: SpeechChannel; refCon: LongInt;
                             phonemeOpcode: Integer);
```

`chan` The speech channel that has finished processing input text.

`refCon` The reference constant associated with the speech channel.

`phonemeOpcode`
The phoneme about to be pronounced.

DESCRIPTION

The Speech Manager calls a speech channel's phoneme callback procedure just before it pronounces a phoneme. For example, your application might use such a callback procedure to enable mouth synchronization. In this case, the callback procedure would set a global flag variable to indicate that the phoneme being pronounced is changing and another global variable to `phonemeOpcode`. A routine called by your application's main event loop could detect that the phoneme being pronounced is changing and update a picture of a mouth to reflect the current phoneme. In practice, providing a visual indication of the pronunciation of a phoneme requires several consecutive pictures of mouth movement to be rapidly displayed. Consult the linguistics literature for information on mouth movements associated with different phonemes.

SPECIAL CONSIDERATIONS

Because your callback procedure executes at interrupt time, you must not call any routines that might move or purge memory.

Your callback procedure is able to access application global variables only if the A5 register is properly set. The Speech Manager sets A5 to the proper value if you provide your application's A5 value by calling the `SetSpeechInfo` function with the `soCurrentA5` selector.

ASSEMBLY-LANGUAGE INFORMATION

Because a callback procedure is called at interrupt time, it must preserve all registers other than A0–A2 and D0–D2.

Word Callback Procedure

You can specify a word callback procedure by passing the `soWordCallBack` selector to the `SetSpeechInfo` function.

MyWordCallback

A word callback procedure has the following syntax:

```
PROCEDURE MyWordCallback (chan: SpeechChannel; refCon: LongInt;
                          wordPos: LongInt; wordLen: Integer);
```

`chan`	The speech channel that has finished processing input text.
`refCon`	The reference constant associated with the speech channel.
`wordPos`	The number of bytes between the beginning of the text buffer and the beginning of the word about to be pronounced.
`wordLen`	The length in bytes of the word about to be pronounced.

DESCRIPTION

The Speech Manager calls a speech channel's word callback procedure just before it pronounces a word. You might use such a callback procedure, for example, to draw the word about to be spoken in a window. In this case, the callback procedure would set a global flag variable to indicate that the word being spoken is changing and another two global variables to `wordPos` and `wordLen`. A routine called by your application's main event loop could detect that the word being spoken is changing and draw the word in a window.

SPECIAL CONSIDERATIONS

Because your callback procedure executes at interrupt time, you must not call any routines that might move or purge memory.

Your callback procedure is able to access application global variables only if the A5 register is properly set. The Speech Manager sets A5 to the proper value if you provide your application's A5 value by calling the `SetSpeechInfo` function with the `soCurrentA5` selector.

ASSEMBLY-LANGUAGE INFORMATION

Because a callback procedure is called at interrupt time, it must preserve all registers other than A0–A2 and D0–D2.

Resources

This section describes the format of a pronunciation dictionary resource, which the Speech Manager uses to override its default pronunciation of words. The Speech Manager uses pronunciation rules as well as an internal dictionary (not stored in the same format as pronunciation dictionary resources) to determine how to pronounce words not included in a speech channel's installed pronunciation dictionaries. For an introduction to the use of and examples showing how your application can install and manipulate pronunciation dictionaries, see "Including Pronunciation Dictionaries" beginning on page 4-36.

This section does not describe the format of voice resources or speech synthesizer resources, because you should not need to access them directly.

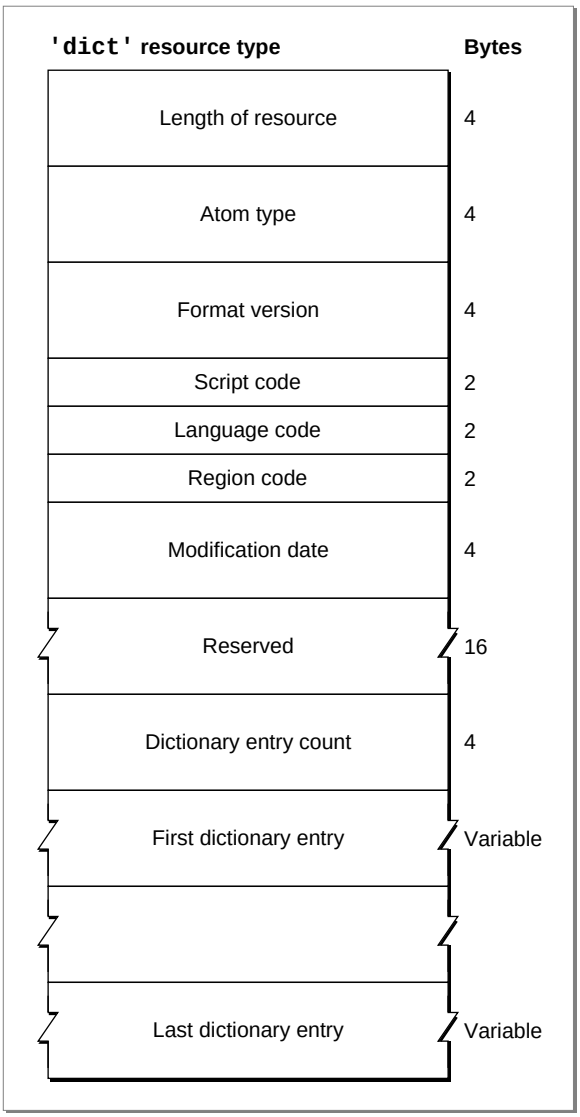
The Pronunciation Dictionary Resource

You can store a list of words and their associated pronunciations in a resource of resource type `'dict'`. You can associate any number of dictionary resources with a speech channel. Before using its internal rules to pronounce a word, the Speech Manager searches the dictionary resources that your application has associated with the speech channel in a last-in, first-searched order.

Note
Because your application is responsible for loading data from a pronunciation dictionary into memory, you can, if desired, store pronunciation information in the data fork of a file rather than in the resource fork. Also, you can devise your own format in which to store pronunciation data, as long as you convert that data into the format described in this section before calling the `UseDictionary` function. ◆

Figure 4-5 shows the format of a pronunciation dictionary resource.

Figure 4-5 Format of a pronunciation dictionary resource



Note
Some synthesizers might use resources (such as resources of type `'ttsd'`) to store their internal pronunciation dictionaries. These internal dictionaries are not necessarily in the same format as the pronunciation dictionaries described here. ◆

To define a dictionary resource, you ordinarily use a resource of type `'dict'`. Such a resource contains a pronunciation dictionary resource header, which is at the start of the resource and defines characteristics of the dictionary as a whole, and any number of pronunciation dictionary entries. Each pronunciation dictionary entry corresponds to one word and contains one or more pronunciation dictionary entry fields. Each pronunciation dictionary entry field contains one piece of information about the word being described in the entry; for example, a dictionary entry would include a field with a textual representation of the word.

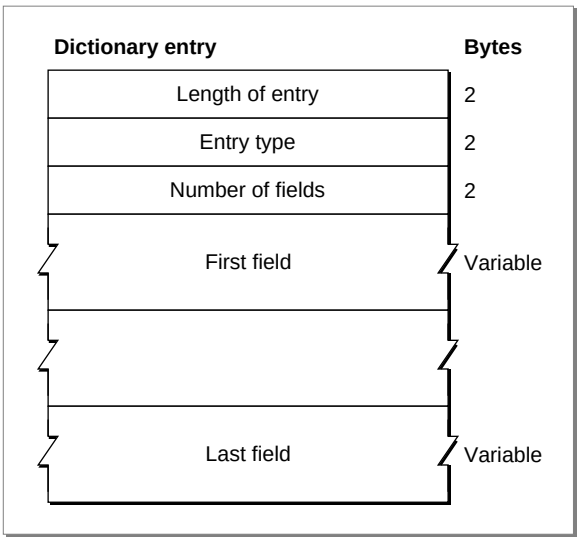
The pronunciation dictionary resource header includes the following:

- Total byte length. The total number of bytes of the dictionary, including the entire pronunciation dictionary resource header in addition to the dictionary's entries.
- Atom type. The currently defined atom type is `'dict'`. Future versions of the Speech Manager might define additional atom types for other types of dictionaries.
- Format version. The currently defined format version is 1. Future versions of the Speech Manager might support additional format versions for the `'dict'` atom type.
- Script code. The script code of words defined in the pronunciation dictionary (for example, `smRoman`). All words in a dictionary must be in the same script.
- Language code. The language code of words defined in the pronunciation dictionary (for example, `langEnglish`). All words in a dictionary must be in the same language.
- Region code. The region code of pronunciations in the dictionary (for example, `verUS`). All words in a dictionary must target the same region.
- Date last modified. The number of seconds between midnight, January 1, 1904, and the modification time. You can use the `GetDateTime` procedure to determine the number of seconds between midnight, January 1, 1904, and the current time. For more information, see *Inside Macintosh: Operating System Utilities*.
- Reserved. These 16 bytes are reserved for future use. You should set them to 0.
- Entry count. The number of dictionary entries.

Immediately following the pronunciation dictionary resource header is a list of the pronunciation dictionary entries.

Figure 4-6 shows the format of a pronunciation dictionary entry.

Figure 4-6 Format of a dictionary entry in a dictionary resource



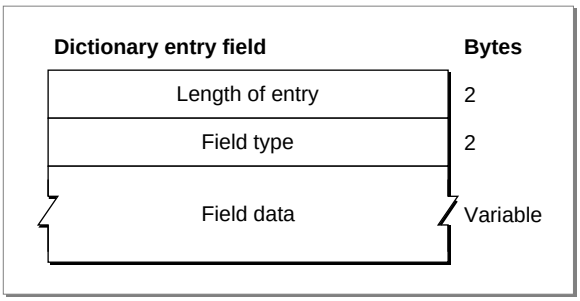
Each pronunciation dictionary entry consists of the following:

- Entry byte length. The total number of bytes in the entry, including this word.
- Entry type. A code for the type of pronunciation dictionary entry. The code $0000 represents a null entry, and codes $0001 through $0020 are reserved for future use by Apple Computer, Inc. You should thus ordinarily fill in this field with $0021, which is the code for a pronunciation entry, or $0022, which is the code for an abbreviation entry. In the current version of the Speech Manager, abbreviation entries work just like pronunciation entries.
- Field count. The number of pronunciation dictionary entry fields contained within this entry.

Immediately following the field count indicator are the fields themselves. Typically, a pronunciation entry always includes a field containing the word in textual format and a field containing the phonetic pronunciation of the word.

Each field within a dictionary entry has the format illustrated in Figure 4-7.

Figure 4-7 Format of a dictionary entry field



The three parts of a dictionary entry field are as follows:

- Field byte length. The total number of bytes in the pronunciation entry field, not including the pad byte of the field data when applicable.
- Field type. A code for the format of the pronunciation dictionary entry field's data. The code $0000 represents a null entry field, and Apple reserves codes $0001 through $0020 as well as code $0023 for future use. Code $0021 represents a textual representation of the word being described in the entry. Code $0022 represents a phonetic pronunciation of the word, including a complete set of syllable, lexical stress, word prominence, and prosodic marks, all represented in textual format.
- Field data. If the field type is $0021 or $0022, then this field contains characters representing the word textually or phonetically, respectively. The characters are not preceded by a length byte and are not followed by a null character. However, if there are an odd number of characters, then a byte must be added as padding to ensure that fields align on word boundaries. The pad byte need not be set to a particular value.

Summary of the Speech Manager

Pascal Summary

Constants

```
CONST
   {Gestalt selector and response bits for speech attributes}
   gestaltSpeechAttr          = 'ttsc';   {speech attributes selector}
   gestaltSpeechMgrPresent    = 0;        {Speech Manager is present}
   gestaltSpeechHasPPCGlue    = 1;        {native glue for PowerPC present}

   {Operating System types}
   kTextToSpeechSynthType     = 'ttsc';   {synthesizer component type}
   kTextToSpeechVoiceType     = 'ttvd';   {voice resource type}
   kTextToSpeechVoiceFileType = 'ttvf';   {voice file type}
   kTextToSpeechVoiceBundleType
                              = 'ttvb';   {voice bundle file type}

   {masks for SpeakBuffer and text-done callback control flags}
   kNoEndingProsody        = 1;        {disable prosody at end of sentences}
   kNoSpeechInterrupt      = 2;        {do not interrupt current speech}
   kPreflightThenPause     = 4;        {compute speech without generating}

   {constants for StopSpeechAt and PauseSpeechAt}
   kImmediate              = 0;        {stop immediately}
   kEndOfWord              = 1;        {stop at end of word}
   kEndOfSentence          = 2;        {stop at end of sentence}

   {GetSpeechInfo and SetSpeechInfo selectors}
   soCharacterMode         = 'char';   {get or set character-processing mode}
   soCommandDelimiter      = 'dlim';   {set embedded command delimiters}
   soCurrentA5             = 'myA5';   {set A5 on callbacks}
   soCurrentVoice          = 'cvox';   {set speaking voice}
   soErrorCallBack         = 'ercb';   {set error callback}
   soErrors                = 'erro';   {get error information}
   soInputMode             = 'inpt';   {get or set text-processing mode}
   soNumberMode            = 'nmbr';   {get or set number-processing mode}
   soPhonemeCallBack       = 'phcb';   {set phoneme callback}
```

```
   soPhonemeSymbols        = 'phsy';   {get phoneme symbols and sample words}
   soPitchBase             = 'pbas';   {get or set baseline pitch}
   soPitchMod              = 'pmod';   {get or set pitch modulation}
   soRate                  = 'rate';   {get or set speech rate}
   soRecentSync            = 'sync';   {get most recent synchronization }
                                       { message information}
   soRefCon                = 'refc';   {set reference constant value}
   soReset                 = 'rset';   {set channel back to default state}
   soSpeechDoneCallBack    = 'sdcb';   {set speech-done callback}
   soStatus                = 'stat';   {get status of channel}
   soSyncCallBack          = 'sycb';   {set synchronization callback}
   soSynthExtension        = 'xtnd';   {get or set synthesizer-specific }
                                       { information}
   soSynthType             = 'vers';   {get synthesizer information}
   soTextDoneCallBack      = 'tdcb';   {set text-done callback}
   soVolume                = 'volm';   {get or set speech volume}
   soWordCallBack          = 'wdcb';   {set word callback}

   {input mode constants}
   modeText             = 'TEXT';
   modePhonemes         = 'PHON';

   {character and number mode constants}
   modeNormal           = 'NORM';
   modeLiteral          = 'LTRL';

   {GetVoiceInfo selectors}
   soVoiceDescription   = 'info';      {get basic voice information}
   soVoiceFile          = 'fref';      {get voice file reference information}

   {genders}
   kNeuter              = 0;
   kMale                = 1;
   kFemale              = 2;
```

Data Structures

Speech Channel Record

```
TYPE
   SpeechChannelRecord  = LongInt;                {speech channel record}
   SpeechChannel        = ^SpeechChannelRecord;   {speech channel}
   SpeechChannelPtr     = ^SpeechChannel;         {speech channel pointer}
```

Voice Specification Record

```
VoiceSpec =
RECORD
   creator:          OSType;        {ID of required synthesizer}
   id:               OSType;        {ID of voice on the synthesizer}
END;
VoiceSpecPtr = ^VoiceSpec;
```

Voice Description Record

```
VoiceDescription =
RECORD
   length:           LongInt;       {size of record--set by application}
   voice:            VoiceSpec;     {voice synthesizer and ID info}
   version:          LongInt;       {version number of voice}
   name:             Str63;         {name of voice}
   comment:          Str255;        {text information about voice}
   gender:           Integer;       {neuter, male, or female}
   age:              Integer;       {approximate age in years}
   script:           Integer;       {script code of text voice can }
                                    { process}
   language:         Integer;       {language code of voice output}
   region:           Integer;       {region code of voice output}
   reserved1:        LongInt;       {always 0--reserved for future use}
   reserved2:        LongInt;       {always 0--reserved for future use}
   reserved3:        LongInt;       {always 0--reserved for future use}
   reserved4:        LongInt;       {always 0--reserved for future use}
END;
VoiceDescriptionPtr = ^VoiceDescription;
```

Voice File Information Record

```
VoiceFileInfo =
RECORD
   fileSpec:         FSSpec;        {volume, dir, and name of file}
   resID:            Integer;       {resource ID of voice in the file}
END;
VoiceFileInfoPtr = ^VoiceFileInfo;
```

Speech-Status Information Record

```
SpeechStatusInfo =
RECORD
   outputBusy:       Boolean;       {TRUE if audio is playing}
   outputPaused:     Boolean;       {TRUE if channel is paused}
   inputBytesLeft:   LongInt;       {bytes of text left to process}
   phonemeCode:      Integer;       {opcode for current phoneme}
END;
SpeechStatusInfoPtr = ^SpeechStatusInfo;
```

Speech Error Information Record

```
SpeechErrorInfo =
RECORD
   count:            Integer;       {number of errors since last check}
   oldest:           OSErr;         {oldest unread error}
   oldPos:           LongInt;       {character position of oldest error}
   newest:           OSErr;         {most recent error}
   newPos:           LongInt;       {character position of newest error}
END;
```

Speech Version Information Record

```
SpeechVersionInfo =
RECORD
   synthType:        OSType;        {general synthesizer type}
   synthSubType:     OSType;        {specific synthesizer type}
   synthManufacturer:
                     OSType;        {synthesizer creator ID}
   synthFlags:       LongInt;       {synthesizer feature flags}
   synthVersion:     NumVersion;    {synthesizer version number}
END;
SpeechVersionInfoPtr = ^SpeechVersionInfo;
```

Phoneme Information Record

```
PhonemeInfo =
RECORD
   opCode:           Integer;       {opcode for the phoneme}
   phStr:            Str15;         {corresponding character string}
   exampleStr:       Str31;         {word that shows use of phoneme}
   hiliteStart:      Integer;       {offset from beginning of word }
                                    { to beginning of phoneme sound}
```

```
   hiliteEnd:        Integer;       {offset from beginning of word }
                                    { to end of phoneme sound}
END;
```

Phoneme Descriptor Record

```
PhonemeDescriptor =
RECORD
   phonemeCount:     Integer;       {number of phonemes defined by }
                                    { current synthesizer}
                                    {list of phoneme information records}
   thePhonemes:      ARRAY[0..0] OF PhonemeInfo;
END;
```

Speech Extension Data Record

```
SpeechXtndData =
RECORD
   synthCreator:     OSType;        {synthesizer creator ID}
                                    {data used by synthesizer}
   synthData:        PACKED ARRAY[0..1] OF Char;
END;
```

Delimiter Information Record

```
DelimiterInfo =
RECORD
   startDelimiter:   PACKED ARRAY[0..1] OF Char;   {start delimiter}
   endDelimiter:     PACKED ARRAY[0..1] OF Char;   {end delimiter}
END;
```

Speech Manager Routines

Starting, Stopping, and Pausing Speech

```
FUNCTION SpeakString        (s: Str255): OSErr;
FUNCTION SpeakText          (chan: SpeechChannel; textBuf: Ptr;
                             byteLen: LongInt): OSErr;
FUNCTION SpeakBuffer        (chan: SpeechChannel; textBuf: Ptr;
                             byteLen: LongInt; controlFlags: LongInt):
                             OSErr;
FUNCTION StopSpeech         (chan: SpeechChannel): OSErr;
```

```
FUNCTION StopSpeechAt       (chan: SpeechChannel; whereToStop: LongInt):
                             OSErr;
FUNCTION PauseSpeechAt      (chan: SpeechChannel; whereToStop: LongInt):
                             OSErr;
FUNCTION ContinueSpeech     (chan: SpeechChannel): OSErr;
```

Obtaining Information About Voices

```
FUNCTION MakeVoiceSpec      (creator: OSType; id: OSType;
                             voice: VoiceSpecPtr): OSErr;
FUNCTION CountVoices        (VAR numVoices: Integer): OSErr;
FUNCTION GetIndVoice        (index: Integer; voice: VoiceSpecPtr): OSErr;
FUNCTION GetVoiceDescription
                            (voice: VoiceSpecPtr;
                             info: VoiceDescriptionPtr; infoLength: LongInt)
                             : OSErr;
FUNCTION GetVoiceInfo       (voice: VoiceSpecPtr; selector: OSType;
                             voiceInfo: Ptr): OSErr;
```

Managing Speech Channels

```
FUNCTION NewSpeechChannel   (voice: VoiceSpecPtr; VAR chan: SpeechChannel):
                             OSErr;
FUNCTION DisposeSpeechChannel
                            (chan: SpeechChannel): OSErr;
```

Obtaining Information About Speech

```
FUNCTION SpeechManagerVersion
                            : NumVersion;
FUNCTION SpeechBusy         : Integer;
FUNCTION SpeechBusySystemWide
                            : Integer;
```

Changing Speech Attributes

```
FUNCTION GetSpeechRate      (chan: SpeechChannel; VAR rate: Fixed): OSErr;
FUNCTION SetSpeechRate      (chan: SpeechChannel; rate: Fixed): OSErr;
FUNCTION GetSpeechPitch     (chan: SpeechChannel; VAR pitch: Fixed): OSErr;
FUNCTION SetSpeechPitch     (chan: SpeechChannel; pitch: Fixed): OSErr;
FUNCTION GetSpeechInfo      (chan: SpeechChannel; selector: OSType;
                             speechInfo: Ptr): OSErr;
FUNCTION SetSpeechInfo      (chan: SpeechChannel; selector: OSType;
                             speechInfo: Ptr): OSErr;
```

Converting Text to Phonemes

```
FUNCTION TextToPhonemes     (chan: SpeechChannel; textBuf: Ptr;
                             textBytes: LongInt; phonemeBuf: Handle;
                             VAR phonemeBytes: LongInt): OSErr;
```

Installing a Pronunciation Dictionary

```
FUNCTION UseDictionary      (chan: SpeechChannel; dictionary: Handle)
                             : OSErr;
```

Application-Defined Routines

```
PROCEDURE MyTextDoneCallback
                            (chan: SpeechChannel; refCon: LongInt;
                             VAR nextBuf: Ptr; VAR byteLen: LongInt;
                             VAR controlFlags: LongInt);
PROCEDURE MySpeechDoneCallback
                            (chan: SpeechChannel; refCon: LongInt);
PROCEDURE MySynchronizationCallback
                            (chan: SpeechChannel; refCon: LongInt;
                             syncMessage: OSType);
PROCEDURE MyErrorCallback   (chan: SpeechChannel; refCon: LongInt;
                             error: OSErr; bytePos: LongInt);
PROCEDURE MyPhonemeCallback
                            (chan: SpeechChannel; refCon: LongInt;
                             phonemeOpcode: Integer);
PROCEDURE MyWordCallback    (chan: SpeechChannel; refCon: LongInt;
                             wordPos: LongInt; wordLen: Integer);
```

C Summary

Constants

```
/*Gestalt selector and response bits for speech attributes*/
#define gestaltSpeechAttr  'ttsc'   /*speech attributes selector*/
enum {
   gestaltSpeechMgrPresent    = 0   /*Speech Manager is present*/
   gestaltSpeechHasPPCGlue    = 1   /*native glue for PowerPC present*/
};
```

```
/*Operating System types*/
#define kTextToSpeechSynthType         'ttsc'   /*synthesizer component */
                                                /* type*/
#define kTextToSpeechVoiceType         'ttvd'   /*voice resource type*/
#define kTextToSpeechVoiceFileType     'ttvf'   /*voice file type*/
#define kTextToSpeechVoiceBundleType   'ttvb'   /*voice bundle file type*/

/*masks for SpeakBuffer and text-done callback control flags*/
enum {
   kNoEndingProsody        = 1,     /*disable prosody at end of sentences*/
   kNoSpeechInterrupt      = 2,     /*do not interrupt current speech*/
   kPreflightThenPause     = 4      /*compute speech without generating*/
};

/*constants for StopSpeechAt and PauseSpeechAt*/
enum {
   kImmediate        = 0,           /*stop immediately*/
   kEndOfWord        = 1,           /*stop at end of word*/
   kEndOfSentence    = 2            /*stop at end of sentence*/
};

/*GetSpeechInfo and SetSpeechInfo selectors*/
#define soCharacterMode       'char'   /*get or set character-processing */
                                       /* mode*/
#define soCommandDelimiter    'dlim'   /*set embedded command delimiters*/
#define soCurrentA5           'myA5'   /*set A5 on callbacks*/
#define soCurrentVoice        'cvox'   /*set speaking voice*/
#define soErrorCallBack       'ercb'   /*set error callback*/
#define soErrors              'erro'   /*get error information*/
#define soInputMode           'inpt'   /*get or set text-processing mode*/
#define soNumberMode          'nmbr'   /*get or set number-processing mode*/
#define soPhonemeCallBack     'phcb'   /*set phoneme callback*/
#define soPhonemeSymbols      'phsy'   /*get phoneme symbols and sample*/
                                       /* words*/
#define soPitchBase           'pbas'   /*get or set baseline pitch*/
#define soPitchMod            'pmod'   /*get or set pitch modulation*/
#define soRate                'rate'   /*get or set speech rate*/
#define soRecentSync          'sync'   /*get most recent synchronization */
                                       /* message information*/
#define soRefCon              'refc'   /*set reference constant value*/
#define soReset               'rset'   /*set channel back to default state*/
#define soSpeechDoneCallBack  'sdcb'   /*set speech-done callback*/
#define soStatus              'stat'   /*get status of channel*/
#define soSyncCallBack        'sycb'   /*set synchronization callback*/
```

```
#define soSynthExtension      'xtnd'   /*get or set synthesizer-specific */
                                       /* information*/
#define soSynthType           'vers'   /*get synthesizer information*/
#define soTextDoneCallBack    'tdcb'   /*set text-done callback*/
#define soVolume              'volm'   /*get or set speech volume*/
#define soWordCallBack        'wdcb'   /*set word callback*/

/*input mode constants*/
#define modeText        'TEXT'
#define modePhonemes    'PHON'

/*character and number mode constants*/
#define modeNormal      'NORM'
#define modeLiteral     'LTRL'

/*GetVoiceInfo selectors*/
enum {
   soVoiceDescription   = 'info',   /*get basic voice information*/
   soVoiceFile          = 'fref'    /*get voice file reference information*/
};

/*genders*/
enum {
   kNeuter = 0,
   kMale,
   kFemale
};
```

Data Types

Speech Channel Record

```
typedef struct SpeechChannelRecord {
   long data[1];                        /*used internally*/
} SpeechChannelRecord;

typedef SpeechChannelRecord *SpeechChannel;
```

Voice Specification Record

```
typedef struct VoiceSpec {
   OSType      creator;             /*ID of required synthesizer*/
   OSType      id;                  /*ID of voice on the synthesizer*/
} VoiceSpec;
```

Voice Description Record

```
typedef struct VoiceDescription {
   long        length;             /*size of structure--set by application*/
   VoiceSpec   voice;              /*voice synthesizer and ID info*/
   long        version;            /*version number of voice*/
   Str63       name;               /*name of voice*/
   Str255      comment;            /*text information about voice*/
   short       gender;             /*neuter, male, or female*/
   short       age;                /*approximate age in years*/
   short       script;             /*script code of text voice can process*/
   short       language;           /*language code of voice output*/
   short       region;             /*region code of voice output*/
   long        reserved[4];        /*always 0--reserved for future use*/
} VoiceDescription;
```

Voice File Information Record

```
typedef struct VoiceFileInfo {
   FSSpec      fileSpec;           /*volume, dir, and name of file*/
   short       resID;              /*resource ID of voice in the file*/
} VoiceFileInfo;
```

Speech Status Information Record

```
typedef struct SpeechStatusInfo {
   Boolean     outputBusy;         /*TRUE if audio is playing*/
   Boolean     outputPaused;       /*TRUE if channel is paused*/
   long        inputBytesLeft;     /*bytes of text left to process*/
   short       phonemeCode;        /*opcode for current phoneme*/
} SpeechStatusInfo;
```

Speech Error Information Record

```
typedef struct SpeechErrorInfo {
   short       count;              /*number of errors since last check*/
   OSErr       oldest;             /*oldest unread error*/
   long        oldPos;             /*character position of oldest error*/
   OSErr       newest;             /*most recent error*/
   long        newPos;             /*character position of newest error*/
} SpeechErrorInfo;
```

Speech Version Information Record

```
typedef struct SpeechVersionInfo {
   OSType      synthType;           /*general synthesizer type*/
   OSType      synthSubType;        /*specific synthesizer type*/
   OSType      synthManufacturer;   /*synthesizer creator ID*/
   long        synthFlags;          /*synthesizer feature flags*/
   NumVersion  synthVersion;        /*synthesizer version number*/
} SpeechVersionInfo;
```

Phoneme Information Record

```
typedef struct PhonemeInfo {
   short       opcode;              /*opcode for the phoneme*/
   Str15       phStr;               /*corresponding character string*/
   Str31       exampleStr;          /*word that shows use of phoneme*/
   short       hiliteStart;         /*offset from beginning of word */
                                    /* to beginning of phoneme sound*/
   short       hiliteEnd;           /*offset from beginning of word */
                                    /* to end of phoneme sound*/
} PhonemeInfo;
```

Phoneme Descriptor Record

```
typedef struct PhonemeDescriptor {
   short          phonemeCount;     /*number of phonemes defined by */
                                    /* current synthesizer*/
   PhonemeInfo    thePhonemes[1];   /*list of phoneme information records*/
} PhonemeDescriptor;
```

Speech Extension Data Record

```
typedef struct SpeechXtndData {
   OSType      synthCreator;        /*synthesizer creator ID*/
   Byte        synthData[2];        /*data used by synthesizer*/
} SpeechXtndData;
```

Delimiter Information Record

```
typedef struct DelimiterInfo {
   Byte        startDelimiter[2];   /*start delimiter*/
   Byte        endDelimiter[2];     /*end delimiter*/
} DelimiterInfo;
```

Speech Manager Routines

Starting, Stopping, and Pausing Speech

```
pascal OSErr SpeakString    (StringPtr s);
pascal OSErr SpeakText      (SpeechChannel chan, Ptr textBuf,
                             long textBytes);
pascal OSErr SpeakBuffer    (SpeechChannel chan, Ptr textBuf,
                             long textBytes, long controlFlags);
pascal OSErr StopSpeech     (SpeechChannel chan);
pascal OSErr StopSpeechAt   (SpeechChannel chan, long whereToStop);
pascal OSErr PauseSpeechAt  (SpeechChannel chan, long whereToPause);
pascal OSErr ContinueSpeech
                            (SpeechChannel chan);
```

Obtaining Information About Voices

```
pascal OSErr MakeVoiceSpec  (OSType creator, OSType id, VoiceSpec *voice);
pascal OSErr CountVoices    (short *numVoices);
pascal OSErr GetIndVoice    (short index, VoiceSpec *voice);
pascal OSErr GetVoiceDescription
                            (VoiceSpec *voice, VoiceDescription *info,
                             long infoLength);
pascal OSErr GetVoiceInfo   (VoiceSpec *voice, OSType selector,
                             void *voiceInfo);
```

Managing Speech Channels

```
pascal OSErr NewSpeechChannel
                            (VoiceSpec *voice, SpeechChannel *chan);
pascal OSErr DisposeSpeechChannel
                            (SpeechChannel chan);
```

Obtaining Information About Speech

```
pascal NumVersion SpeechManagerVersion
                            (void);
pascal short SpeechBusy     (void);
pascal short SpeechBusySystemWide
                            (void);
```

Changing Speech Attributes

```
pascal OSErr GetSpeechRate  (SpeechChannel chan, Fixed *rate);
```

```
pascal OSErr SetSpeechRate  (SpeechChannel chan, Fixed rate);
pascal OSErr GetSpeechPitch
                            (SpeechChannel chan, Fixed *pitch);
pascal OSErr SetSpeechPitch
                            (SpeechChannel chan, Fixed pitch);
pascal OSErr GetSpeechInfo  (SpeechChannel chan, OSType selector,
                             void *speechInfo);
pascal OSErr SetSpeechInfo  (SpeechChannel chan, OSType selector,
                             void *speechInfo);
```

Converting Text to Phonemes

```
pascal OSErr TextToPhonemes
                            (SpeechChannel chan, Ptr textBuf,
                             long textBytes, Handle phonemeBuf,
                             long *phonemeBytes);
```

Installing a Pronunciation Dictionary

```
pascal OSErr UseDictionary  (SpeechChannel chan, Handle dictionary);
```

Application-Defined Routines

```
#pragma procname SpeechTextDone
typedef pascal void (*SpeechTextDoneCBPtr)
                            (SpeechChannel, long, Ptr *, long *, long *);
typedef SpeechTextDoneProcPtr SpeechTextDoneCBPtr;
#pragma procname SpeechDone
typedef pascal void (*SpeechDoneCBPtr)
                            (SpeechChannel, long);
typedef SpeechDoneProcPtr SpeechDoneCBPtr;
#pragma procname SpeechSync
typedef pascal void (*SpeechSyncCBPtr)
                            (SpeechChannel, long, OSType);
typedef SpeechSyncProcPtr SpeechSyncCBPtr;
#pragma procname SpeechError
typedef pascal void (*SpeechErrorCBPtr)
                            (SpeechChannel, long, OSErr, long);
typedef SpeechErrorProcPtr SpeechErrorCBPtr;
#pragma procname SpeechPhoneme
typedef pascal void (*SpeechPhonemeCBPtr)
                            (SpeechChannel, long, short);
typedef SpeechPhonemeProcPtr SpeechPhonemeCBPtr;
```

```
#pragma procname SpeechWord
typedef pascal void (*SpeechWordCBPtr)
                              (SpeechChannel, long, long, short);
typedef SpeechWordProcPtr SpeechWordCBPtr;
```

Assembly-Language Information

Data Structures

Voice Specification Data Structure

0	creator	4 bytes	ID of required synthesizer
4	id	4 bytes	ID of voice on the synthesizer

Voice Description Data Structure

0	length	long	size of structure—set by application
4	voice	8 bytes	voice specification record
12	version	long	version number of voice
16	name	64 bytes	name of voice; preceded by length byte
80	comment	256 bytes	text information about voice; preceded by length byte
336	gender	short	neuter (0), male (1), or female (2)
338	age	short	approximate age in years
340	script	short	script code of text voice can process
342	language	short	language code of text voice can process
344	region	short	region code of voice output
346	reserved	16 bytes	always set to 0—reserved for future use

Voice File Information Data Structure

0	fileSpec	70 bytes	volume, directory, and name of file
70	resID	word	resource ID of voice in the file

Speech Status Information Data Structure

0	outputBusy	byte	1 if audio is playing
1	outputPaused	byte	1 if channel is paused
2	inputBytesLeft	long	bytes of text left to process
6	phonemeCode	short	opcode for current phoneme

Speech Error Information Data Structure

0	`count`	word	number of errors since last check
2	`oldest`	long	oldest unread Operating System error
6	`oldPos`	long	character position of oldest error
10	`newest`	long	newest Operating System error
14	`newPos`	long	character position of newest error

Speech Version Information Data Structure

0	`synthType`	4 bytes	always `'TTSC'`
4	`synthSubType`	4 bytes	synthesizer type
8	`synthManufacturer`	4 bytes	synthesizer creator ID
12	`synthFlags`	long	synthesizer feature flags
16	`synthVersion`	long	synthesizer version number

Phoneme Information Data Structure

0	`opcode`	word	opcode for the phoneme
2	`phStr`	16 bytes	corresponding character string; preceded by length byte
18	`exampleStr`	32 bytes	word that shows use of phoneme
50	`hiliteStart`	word	offset from beginning of word to beginning of phoneme sound
52	`hiliteEnd`	word	offset from beginning of word to end of phoneme sound

Phoneme Descriptor Data Structure

0	`phonemeCount`	word	number of phonemes defined by current synthesizer
2	`thePhonemes`	variable	list of phoneme information records

Speech Extension Data Structure

0	`synthCreator`	4 bytes	synthesizer creator ID
4	`synthData`	variable	data used by synthesizer

Delimiter Information Data Structure

0	`startDelimiter`	2 bytes	start embedded command characters; defaults to "`[[`"
2	`endDelimiter`	2 bytes	end embedded command characters; defaults to "`]]`"

Trap Macros

Trap Macro Requiring Routine Selectors

_SoundDispatch

Selector	Routine
$0000000C	SpeechManagerVersion
$003C000C	SpeechBusy
$0040000C	SpeechBusySystemWide
$0108000C	CountVoices
$021C000C	DisposeSpeechChannel
$0220000C	SpeakString
$022C000C	StopSpeech
$0238000C	ContinueSpeech
$030C000C	GetIndVoice
$0418000C	NewSpeechChannel
$0430000C	StopSpeechAt
$0434000C	PauseSpeechAt
$0444000C	SetSpeechRate
$0448000C	GetSpeechRate
$044C000C	SetSpeechPitch
$0450000C	GetSpeechPitch
$0460000C	UseDictionary
$0604000C	MakeVoiceSpec
$0610000C	GetVoiceDescription
$0614000C	GetVoiceInfo
$0624000C	SpeakText
$0654000C	SetSpeechInfo
$0658000C	GetSpeechInfo
$0828000C	SpeakBuffer
$0A5C000C	TextToPhonemes

Result Codes

`noErr`	0	No error
`paramErr`	–50	Parameter error
`memFullErr`	–108	Not enough memory to speak
`nilHandleErr`	–109	Handle argument is `NIL`
`siUnknownInfoType`	–231	Feature not implemented on synthesizer
`noSynthFound`	–240	Could not find the specified speech synthesizer
`synthOpenFailed`	–241	Could not open another speech synthesizer channel
`synthNotReady`	–242	Speech synthesizer is still busy speaking
`bufTooSmall`	–243	Output buffer is too small to hold result
`voiceNotFound`	–244	Voice resource not found
`incompatibleVoice`	–245	Specified voice cannot be used with synthesizer
`badDictFormat`	–246	Pronunciation dictionary format error
`badPhonemeText`	–247	Raw phoneme text contains invalid characters
`unimplMsg`	–248	Unimplemented message
`badVoiceID`	–250	Specified voice has not been preloaded
`badParmCount`	–252	Incorrect number of embedded command arguments
`invalidComponentID`	–3000	Speech channel is uninitialized or bad