Speech Synthesis Programming Guide

Developer

Contents

Introduction to Speech Synthesis Programming Guide 5
Organization of This Document 5
See Also 6

Speech Synthesis in OS X 7
Why Use Synthesized Speech? 7
Spoken Output and Accessibility 8
Speech Synthesis Concepts and Components 8
- The Speech Generation Process 10
- Representations of Speech 11
- Voices 12
- Speech Channels 13
- Notifications, Callbacks, and Speech Synchronization 14
- The MacinTalk Synthesizer 14
- Attributes of Synthesized Speech 15

Opportunities for the Customization of Synthesized Speech 18
- Use Different Voices 18
- Adjust Speech Attributes and Control Speech Production Using the Speech Synthesis APIs 19
- Control Speech Quality Using Embedded Speech Commands 19

Designing and Implementing an Application That Speaks 21
Strategies for Incorporating Synthesized Speech 21
- Take Advantage of System-Provided Functionality 21
- Provide Some Customization 22
- Provide Advanced Customization 22

User Interface Design Guidelines for Speech 23
Carbon and Cocoa Speech Synthesis APIs Compared 24
- Overview of the Cocoa Speech Synthesis API 25
- Overview of the Carbon Speech Synthesis API 26

Implementing Basic Speech Synthesis Tasks Using Cocoa and Carbon 27
- Generating Speech Using the Cocoa Speech Synthesis API 27
- Generating Speech Using the Carbon Speech Synthesis API 30

Using AppleScript to Produce Spoken Output 34

Techniques for Customizing Synthesized Speech 36
Adjust Speech Channel Settings Using the Carbon Speech Synthesis API 36
Use Embedded Speech Commands to Fine-Tune Spoken Output 39
    Embedded Speech Command Delimiters 39
    Overview of Embedded Speech Command Syntax 40
    The OS X Embedded Speech Commands 41
    Embedded Speech Command Errors 49
Use Phoneme Modifiers to Adjust Pronunciation 49
Use Punctuation Correctly 52
Use the TUNE Format to Supply Complex Pitch Contours 53
Synchronize Speech with Application-Specific Actions 55
Avoid Cross-Talk 56
Four Ways to Improve Spoken Output 58
    Adjust the Pronunciation of Troublesome Words 58
    Let the User Catch Up 59
    Focus the User's Attention 60
    Liven It Up! 60

Syntax of Embedded Speech Commands 62

Phonemes 64

Glossary 67

Document Revision History 68

Figures, Tables, and Listings

Speech Synthesis in OS X 7

Figure 1-1 The speech generation process 10
Figure 1-2 Multiple synthesizers and their voices 12
Figure 1-3 Pitch variations in the phrase "Hello, world" 17

Designing and Implementing an Application That Speaks 21

Listing 2-1 Generating speech using the Cocoa speech synthesis API 27
Listing 2-2 Using delegate methods to respond to speech events 29
Listing 2-3 Generating speech using the Carbon speech synthesis API 30
Listing 2-4 Using a Carbon callback procedure to respond to a speech event 33
Listing 2-5 Using AppleScript to produce spoken output 34
Listing 2-6 Using AppleScript to save spoken output to a file 35

Techniques for Customizing Synthesized Speech 36

Table 3-1 Embedded speech commands 42
Table 3-2 Prosodic control symbols and descriptions 50
Table 3-3 Effects of punctuation marks on synthesized speech 52

Syntax of Embedded Speech Commands 62

Table A-1 Embedded command syntax structure 62

Phonemes 64

Table B-1 North American English phoneme symbols 64

Introduction to Speech Synthesis Programming Guide

Speech synthesis, also called text-to-speech, is the generation of synthetic speech. An application or other process sends text to a speech synthesizer, which creates a spoken version that can be output through the audio hardware or saved to a file.

This document covers speech synthesis support in OS X. It provides an overview of speech components and the speech synthesis process, and it describes how to incorporate and manipulate synthesized speech in your application.

Note: This document does not discuss the use of assistive applications, such as VoiceOver. If you need to know how to make your application accessible to users with disabilities, read *Accessibility Overview for OS X* to find out about accessibility support in OS X.

Note: Although speech synthesis and speech recognition are closely related technologies, this document focuses on speech synthesis. For some information on speech recognition, see *Speech Recognition Manager Reference*.

You should read this document to learn about speech synthesis in OS X and about how you can customize your application's spoken output. If you're unfamiliar with the concepts of synthesized speech, be sure to read "Speech Synthesis in OS X" (page 7) for an overview of how speech fits into the operating system and of the speech generation process. Carbon, Cocoa, and AppleScript provide APIs to produce spoken output. Be sure to read the API overviews in "Designing and Implementing an Application That Speaks" (page 21) to find out which programming language provides the features you need.

Organization of This Document

Speech Synthesis Programming Guide contains the following chapters and appendixes:

- "Speech Synthesis in OS X" (page 7) provides an overview of the speech generation process and its components, and describes ways you can use and customize spoken output in your application.
- "Designing and Implementing an Application That Speaks" (page 21) describes some design strategies and guidelines for providing spoken output in your application. It also outlines the speech synthesis APIs available in Cocoa, Carbon, and AppleScript, and provides some sample code.

- "Techniques for Customizing Synthesized Speech" (page 36) details several methods for customizing your application's spoken output, and describes a handful of ways to improve synthesized speech.
- "Syntax of Embedded Speech Commands" (page 62) describes the formal syntax of embedded speech commands.
- "Phonemes" (page 64) lists the North American English phoneme symbols the MacinTalk synthesizer recognizes.
- "Glossary" (page 67) contains definitions of speech-related terms used in this document. Every term you see displayed in bold font, such as **speech synthesizer**, appears in the Glossary.
- "Revision History" (page 68) lists the changes made to this document.

See Also

In addition to this document, the Reference Library > User Experience > Speech Technologies contains several other resources to help you take advantage of synthesized speech in your application.

- *Speech Synthesis Manager Reference* describes the Carbon speech synthesis API.
- *NSSpeechSynthesizer Class Reference* describes the Cocoa speech synthesis API.
- *Speech Programming Topics* briefly describes some of the tasks you can accomplish using the Cocoa speech synthesis API.
- *CocoaSpeechSynthesisExample* is a sample Cocoa application that uses the Carbon speech synthesis API.
- The speech developers mailing list (speech-dev) is an excellent place to discuss issues related to speech synthesis and recognition.

Speech Synthesis in OS X

OS X includes an advanced speech synthesizer that provides high-quality synthesized speech and comprehensive speech synthesis APIs that allow developers to create and customize spoken output.

This chapter discusses some of the benefits of using speech synthesis in your application and describes the components of the OS X Speech Synthesis framework. In addition, this chapter provides an overview of the ways in which you can customize the speech your application generates. You should read this chapter if you're unfamiliar with the concepts of speech synthesis or if you're wondering how to take advantage of synthesized speech in your application.

Why Use Synthesized Speech?

Although people have learned to communicate with computers and applications using display screens and various input devices, these methods represent an effort on the part of users to conform to the computer's communication paradigm, not vice versa. When an application produces synthesized speech, however, it communicates with users in human terms, in a natural and efficient way. Using speech, an application can communicate an almost infinite range of information to the user. Because it is not limited to producing a small set of sounds users must learn to associate with specific conditions or actions, an application that generates speech can give users precise information about complex subjects and conditions.

Consider, for example, a home accounting application in which users enter data about their expenditures for the month. If the application speaks each number as it is entered, users know immediately when they've entered an incorrect number without ever having to look at the display screen. Another example is an email program that tells users not only when a new message arrives but also from whom.

Applications can also customize spoken output to meet specific requirements. For example, a language-learning application can customize speech to produce accurately pronounced words and phrases users can mimic. Games and other entertainment applications can use speech customization to emphasize the individuality of different onscreen characters.

Of course, an application that generates speech might also benefit from allowing users to speak to it, using a technology called speech recognition. However, this document focuses on the speech synthesis side of the computer-user conversation. You can find reference documentation on the OS X speech recognition APIs in *Speech Recognition Manager Reference* .

Spoken Output and Accessibility

It's important to understand that adding synthesized speech to an application and making an application accessible to all users (a process called access enabling) are different processes with different goals. In particular, adding support for synthesized speech to your application is not the same as meeting accessibility requirements, such as those set by section 508 of the United States Rehabilitation Act of 1973.

Although both application-generated speech and the speech produced by a screen reader or other assistive application might sound the same (and use much of the same underlying technology), they perform different functions. Synthesized speech enhances an application's user interface and helps accomplish application-specific tasks, such as describing error conditions or providing verbal feedback on users's actions. In contrast, speech generated by an assistive application enables users to access all parts of the operating system and drive the user interfaces of other applications without using the mouse or display screen. Because an assistive application must be able to help users access all applications they might run, it focuses on providing access to the features all applications have in common, such as menus, buttons, and text-input fields.

To illustrate the difference between the roles of an application's spoken output and the speech generated by an assistive application, consider an access-enabled chess application. For the purposes of this discussion, assume that this chess application produces spoken output that describes the moves taken by both the user and the application. Using an assistive application, visually impaired users can run this chess application and activate all the buttons and other controls in its user interface. However, an assistive application cannot describe the move the chess application makes in its turn, because that information arises from a change in the internal state of the chess application, not from a button click or menu-item selection. If the chess application did not produce its own spoken output, visually impaired users would be able to move their own chess pieces but would not be able to find out how the application responded.

In OS X v10.3 Apple introduced VoiceOver, an alternative way of interacting with the Macintosh that allows visually impaired users to use applications and OS X itself using only the keyboard. Because VoiceOver and many other assistive applications generate spoken output, they use the same OS X speech synthesizer your application uses when it generates spoken output. While VoiceOver is running, therefore, users may experience interruptions in your application's speech or cross-talk (overlapping speech). To find out how VoiceOver interacts with your application's spoken output and how to avoid interrupting the spoken output of other applications, see "Avoid Cross-Talk" (page 56).

Speech Synthesis Concepts and Components

In OS X, the Speech Synthesis framework supports the conversion of text into speech, using a common API for managing voices and synthesizers. This architecture supports multiple, plug-in synthesizers and languages from different vendors, as well as multiple voices for each synthesizer. Application developers interact with the Speech Synthesis framework using the C-based API defined in the Application Services framework, the

Objective-C API defined in the Application Kit, or the AppleScript `say` command. Developers of command-line tools and other processes can link with the Application Services framework to produce spoken output, because there is no graphical user interface inherent in synthesized speech. Even if you don't plan to offer any customized speech features, your application or process benefits from the systemwide feature that allows users to hear spoken aloud nearly any text they can select.

The Speech Synthesis framework includes:

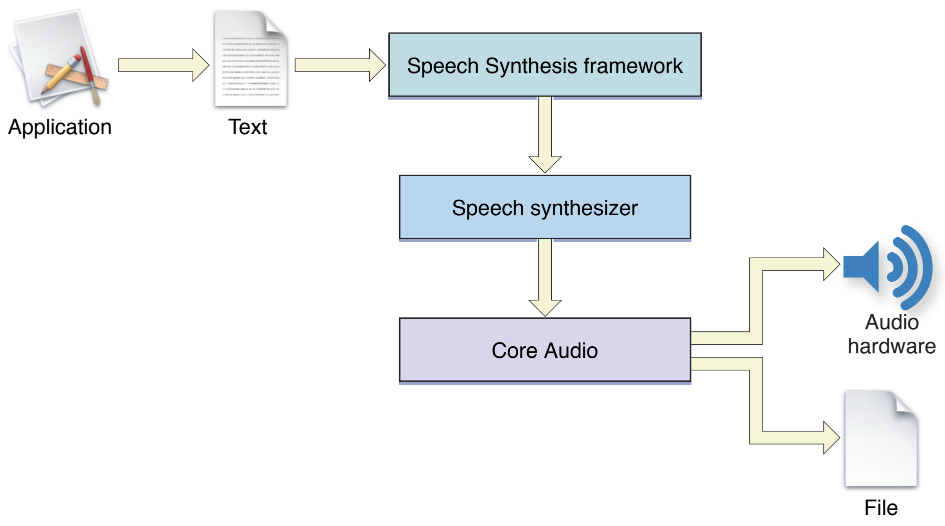
- The Carbon speech synthesis API (also called the Speech Synthesis Manager), which is defined in the Speech Synthesis subframework in the Application Services framework. The Carbon speech synthesis API provides extensive control over speech synthesis to applications that can link with the Application Services framework.
- The `NSSpeechSynthesizer` class, which is defined in the Application Kit framework. The `NSSpeechSynthesizer` class provides basic speech-synthesis functionality to Cocoa applications.
- Speech synthesizers, which are contained in loadable bundles and which reside in `/System/Library/Speech/Synthesizers`. Synthesizers perform the conversion of text to speech and contain code that performs lexical analysis and determines pronunciations. Apple's built-in synthesizer is the MacinTalk synthesizer, which is described in "The MacinTalk Synthesizer" (page 14).
- Speech voices, which are bundles that contain individual voice characteristics and, sometimes, code. Apple provides more than 20 built-in voices, which reside in `/System/Library/Speech/Voices`. For more information about voices and their relationship to synthesizers, see "Voices" (page 12).

The following sections outline the OS X speech generation process and describe components of the Speech Synthesis framework and concepts of speech generation. The information in these sections is applicable to any application or process that produces synthesized speech, regardless of the speech synthesis API it uses.

The Speech Generation Process

Essentially, the Speech Synthesis framework is a dispatch mechanism that allows your application to take advantage of the capabilities of whatever speech synthesizers, voices, and hardware are installed on a user's computer. The Speech Synthesis framework provides a convenient programming interface that manages access to the speech synthesizers and, indirectly, to the sound hardware. Figure 1-1 illustrates the speech generation process at a high level.

Figure 1-1 The speech generation process



As outlined in Figure 1-1, your application initiates speech generation by passing a string or buffer of text to the Speech Synthesis framework, via the appropriate API. The Speech Synthesis framework is responsible for sending the text to a **speech synthesizer**, a component containing executable code that manages all communication between the Speech Synthesis framework and Core Audio.

A synthesizer contains a set of built-in dictionaries and pronunciation rules that it uses to determine how to pronounce text. The synthesizer receives text from an application and converts it to phonemes (described in "Representations of Speech" (page 11)), and sends the result, including optional pronunciation directives, to a voice. Each synthesizer can work with only those voices that are designed for it; it cannot use voices designed for other synthesizers, even if the voices are installed in the computer.

As shown in Figure 1-1, Core Audio receives digital sound-wave input from the synthesizer and sends this data to the current sound output device or to a file. Because all communication between the Speech Synthesis framework and Core Audio is transparent to your application, you do not need to be concerned with potential changes to the underlying technology or implementations in this area.

An application can participate in the speech generation process at different levels, ranging from simple to complex. At one end of the spectrum, an application can be completely passive, allowing users to use system-supplied speech features to choose when to hear the application's text spoken aloud and with which voice. At the other end, an application can supply the Speech Synthesis framework with precise information about how the speech should be produced and with which voice it should be spoken. For more on the ways you can use synthesized speech in your application, see "Opportunities for the Customization of Synthesized Speech" (page 18).

Representations of Speech

There are two ways your application can represent speech: textually and phonemically. Textual representation consists of a sequence of standard, human-readable words in a string or buffer. Phonemic representation is text converted into **phonemes**, which are distinct units that distinguish one word from another. Different languages have different sets of phonemes. For example, in English, the words "pad" and "bad" are distinguished by the phonemes "p" and "b." Each phoneme is represented by a unique symbol, which consists of single or paired upper-case or lower-case letters (for a complete list of North American English phoneme symbols recognized by the MacinTalk synthesizer, see "Phonemes" (page 64)). For example, the phonemic representation of the word "pad" is "pAEd," where the phoneme symbols "p," "AE," and "d" stand for "p," the short "a" sound, and "d," respectively.

A speech synthesizer always converts text to phonemes before sending it to a voice because the phonemic representation allows it to encode the precise pronunciation of each word. The Speech Synthesis framework provides a function that allows your application to convert text into phonemes before it is sent to the synthesizer. In applications that speak only text that users enter this feature is of limited usefulness, because you can't anticipate what a user might type. However, if your application speaks a finite set of words or phrases that you create, it can be useful to represent at least some of that text phonemically to ensure its desired pronunciation.

Performing your own text-to-phoneme conversion has the following advantages:

- You can use a text-to-phoneme conversion process that might be of higher quality than that provided by the available synthesizer. You can then use the phonemic data you generate in this way with any speech synthesizer to produce better speech.
- You can use phoneme modifiers to adjust the pronunciation of words, giving you a very high degree of control over the spoken output. For example, you can change the placement of the primary stress within a word.
- You can use the TUNE format to shape the overall melody and timing of an utterance. The TUNE format (described in "Use the TUNE Format to Supply Complex Pitch Contours" (page 53)) allows you to create a template of pitch and rate changes and apply it to the phonemic representation of a word or phrase. For example, you can use the TUNE format to make an utterance sound as if it is spoken with emotion.

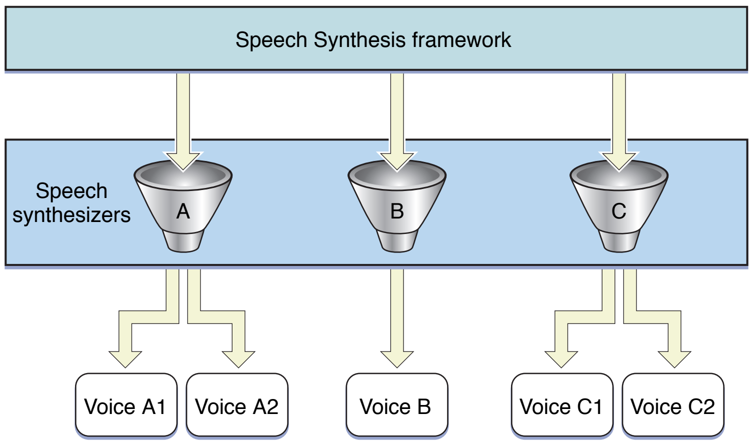
The Speech Synthesis framework also allows you to intersperse phonemic representations of specific words and phrases in a buffer of text. This is useful if the text that your application needs to speak contains words with nonstandard pronunciations, such as proper names, or words you want to be spoken in a particular way. To combine textual and phonemic representations of speech in this way, you must use embedded speech commands (described in "Control Speech Quality Using Embedded Speech Commands" (page 19)).

Voices

A **voice** is a set of characteristics that exhibit particular qualities of speech, such as pitch and tone. Just as each person's voice has unique tonal qualities, so too does each synthesized voice. A synthesized voice might sound male or female and might sound like an adult or a child. Some synthesized voices sound distinctively synthetic, while others sound more natural. To explore the range of voices that come installed in OS X, go to the Speech pane of System Preferences, click the Text to Speech tab, and listen to the voices listed in the System Voice menu. Your application can use the default system voice to generate speech, or it can use the speech synthesis API to select (or allow users to select) one of the other voices available on the user's system.

Although a single voice supports only one language and region, a synthesizer may contain any number of voices, each of which can support a different language. Figure 1-2 shows how different synthesizers and their voices can coexist on a computer.

Figure 1-2 Multiple synthesizers and their voices



The Speech Synthesis framework defines a data structure, called a voice description record, that holds information about a voice, such as its name, gender, age, language, and the synthesizer with which it's associated. The framework provides functions that allow you to identify how many voices are currently available in the user's system and to get the information in a voice description record for a specific voice. Although most of the information in voice description records should not be exposed to users, you can display some of it, such as the voice name, to help users make informed choices.

Note that, in general, your application does not need to know which speech synthesizer it is using or with which speech synthesizer a given voice is associated. However, some speech synthesizers provide special capabilities in addition to those provided by the Speech Synthesis framework. For example, a speech synthesizer might allow you to select an option to speak all numbers in a nonstandard way, such as digit-by-digit. For these circumstances, the Speech Synthesis framework provides APIs that allow you to determine which synthesizer is associated with a voice and provides hooks that allow your application to take advantage of synthesizer-specific capabilities.

As speech technology continues to develop, it's likely that the voices your application can access will sound increasingly human. When you use the OS X speech synthesis APIs, you automatically benefit from any improvements made to the voices built into the system. Regardless of the voice used to speak the output, you can customize the way it speaks your text, using techniques outlined in "Adjust Speech Attributes and Control Speech Production Using the Speech Synthesis APIs" (page 19) and "Control Speech Quality Using Embedded Speech Commands" (page 19).

Speech Channels

To send text to a synthesizer and to specify which voice or attributes you would like it to use, your application uses a speech channel. Conceptually, a **speech channel** is the conduit between your application and the Speech Synthesis framework. Your application acquires a speech channel, sends through it the text to be spoken, and, optionally, sets speech-channel attributes that affect the synthesized speech.

Precisely how your application interacts with a speech channel is defined by the API it uses. The Carbon speech synthesis API includes functions you use to create and manage speech channels, as well as functions that allow you to get and set speech-channel attributes. On the other hand, in the Cocoa speech synthesis API, speech-channel management is transparent to you. When you use the Cocoa API to generate spoken output, the necessary speech channels are created, used, and destroyed automatically. Similarly, the AppleScript `say` command does not expose the use of speech channels. Whichever API you use, however, it's useful to understand the role of speech channels in the speech generation process. The remainder of this section describes this role and how some applications might need to create multiple speech channels.

At any point in time, a speech channel is associated with a particular voice and specific speech attributes. However, multiple speech channels can coexist in a single application, which allows your application to create more than one vocal environment to, for example, simulate a dialogue among different characters in a game. Alternatively, you can use a single speech channel and switch to different voices when necessary, but this approach can be inefficient. An example of an application that requires multiple speech channels is one that needs to generate speech in more than one language. As mentioned in "Voices" (page 12), a voice is associated with only one language and region, so an application that needs to produce spoken output in a bilingual or multilingual environment would need a separate speech channel for each language.

Separate speech channels in a single application can generate speech simultaneously, subject to processor capabilities. However, this capability should be used with restraint, because it is very difficult for users to make sense of speech when more than one channel is generating speech at the same time. Of course, different speech channels created by different applications may also produce speech simultaneously; for this reason, it's a good idea to implement an arbitration scheme in your application (for more information on how to do this, see "Avoid Cross-Talk" (page 56)).

Notifications, Callbacks, and Speech Synchronization

The Speech Synthesis framework allows you to receive notifications of certain events during the speech generation process. Using these notifications, you can synchronize speech with actions in your application, such as highlighting the word being spoken or animating a character's mouth to correspond to the phoneme being pronounced.

Not surprisingly, the Cocoa and Carbon speech synthesis APIs support different sets of notifications and implement them differently (AppleScript does not support synchronization of speech with application actions). The Cocoa API defines delegate methods you can implement; the Carbon API defines a large number of callbacks for which you can provide handler functions.

Some of the notifications you can receive tell you when:

- A word is about to be spoken
- A phoneme is about to be spoken
- Speaking has finished
- The text has been processed, but not necessarily spoken yet (available only in the Carbon API)
- A `sync` embedded speech command is encountered (available only in the Carbon API). For more information about this command, see "The OS X Embedded Speech Commands" (page 41).

For more information about the specific notifications available and how to use them in your application, see "Synchronize Speech with Application-Specific Actions" (page 55).

The MacinTalk Synthesizer

The **MacinTalk synthesizer** is the built-in synthesizer in OS X. It generates North American English from unrestricted text, and supports the addition of a number of text-embedded commands to control pronunciation and intonation. The output of the MacinTalk synthesizer can be played through the computer's speakers or saved to a file.

In general, a synthesizer produces the most natural-sounding speech when it combines its built-in text processing rules with pronunciation hints provided by the author of the text. The MacinTalk synthesizer contains a sophisticated lexical analyzer that allows it to make a "best guess" at how a human might speak a given sample of text. But the MacinTalk synthesizer (like all synthesizers) does a better job when you provide precise pronunciation information. Whether synthesized speech is an optional feature or constitutes the centerpiece of your application's functionality, you should consider using the customization strategies described in "Opportunities for the Customization of Synthesized Speech" (page 18) to ensure the production of high-quality speech that meets your specifications.

Attributes of Synthesized Speech

Any given person has only one voice, but can significantly alter the characteristics and meaning of his or her speech by varying the pitch, volume, and speed of delivery. People instinctively respond to these vocal attributes and rely on them to provide layers of meaning in addition to the semantic meaning of the words they hear. The Speech Synthesis framework supplies functions that allow you to manipulate speech attributes, such as pitch and speed, to achieve the effects you want.

A **speech attribute** is a setting defined on a speech channel that affects the quality of the spoken output for a specific subset of voices, or for all voices associated with a particular synthesizer. At any single point in time, there is a one-to-one correspondence between a voice and a speech channel, so you can think of a speech attribute as applying to either a voice or to a speech channel. Using functions in the Carbon speech synthesis API, you can alter four speech attributes: rate, pitch, pitch modulation, and volume. Alternatively, you can use embedded speech commands to set these four attributes, plus the prosody attribute, on a per-word basis, regardless of the programming language you're using. For more information on how to use embedded commands, see "Use Embedded Speech Commands to Fine-Tune Spoken Output" (page 39).

Speech Rate

The **speech rate** of a speech channel is the approximate number of words of text that the synthesizer speaks in one minute. Although a slower speech rate can make the speech easier to understand, listening to words that are spoken too slowly can be tedious. Be sure to test your application to determine the optimum speech rate for your target audience, so you can ship your application with a reasonable default setting. Visually impaired users, for example, are often comfortable listening to much faster speech rates than sighted users.

Speech rates are expressed as real values. For example, typical, conversational speech is at a rate of about 180 words per minute, whereas some visually impaired users can comfortably listen to VoiceOver at rates of up to 500 words per minute. Each speech synthesizer determines its own range of speech rates that can be applied to the voices it uses. The Carbon speech synthesis API includes functions that allow you to get and change the current speech rate on a speech channel (for more information on how to do this, see "Adjust Speech Channel Settings Using the Carbon Speech Synthesis API" (page 36)).

Speech Pitch, Frequency, and Pitch Modulation

Pitch is a combination of the average speaking frequency and its variations around that average. When you listen to a voice speaking, you're aware of variations in pitch that create a sort of melody. Often, you're more aware of this musical quality when you listen to conversations in a language you don't speak, because you're not focused on the semantic meaning of what you're hearing. To produce human-like speech, therefore, a synthesizer must try to replicate these pitch variations in its voices.

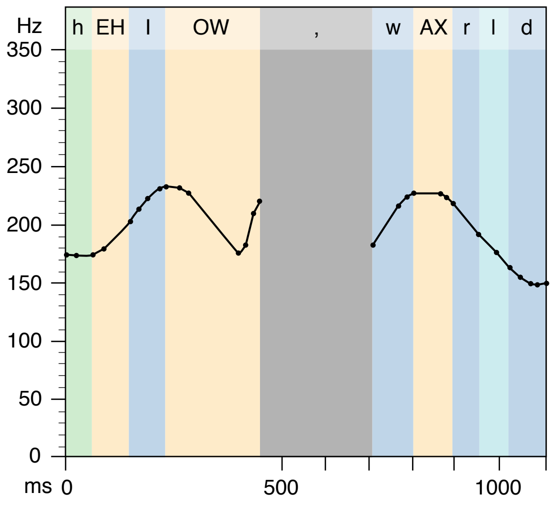
The **speech pitch** of a speech channel represents the middle pitch of a voice, from which the actual pitches of the speech can vary with rising and falling tunes. You can think of speech pitch as roughly corresponding to the key in which a song is played. A speech pitch is expressed as a real value 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 speech-pitch value corresponds to a musical half-step. You may notice that this is the same scale that is used to specify MIDI note values. Although the scale is the same, however, speech-pitch values differ from MIDI note values in two fundamental ways: speech-pitch values do not have to be integral and they occupy a narrower range than MIDI note values.

On this scale, a change of +12 units corresponds to a doubling of frequency (an increase of one octave) and a change of -12 units corresponds to a halving of frequency (a decrease of one octave). A **frequency** is a precise indication of the number of hertz (Hz) of a sound wave at any instant. Typical voice frequencies might range from about 75 Hz for a low-pitched male voice to about 300 Hz for a high-pitched child's voice. These frequencies correspond to approximate speech-pitch values in the ranges of 30.000 to 40.000 and 55.000 to 65.000, respectively. If you need to convert between speech pitches and hertz, note that a speech pitch of 60.000 corresponds to 261.625 Hz.

The Carbon speech synthesis API includes functions to determine the current speech pitch on a speech channel and to change the speech pitch (see "Adjust Speech Channel Settings Using the Carbon Speech Synthesis API" (page 36) for more information on how to do this).

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

Figure 1-3 Pitch variations in the phrase "Hello, world"



Pitch modulation is expressed as a real value in the range of 0.000 through 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 sounds unnaturally robotic.

Speech Volume

The **speech volume** of a speech channel is the average amplitude at which the channel generates speech. Speech volumes are expressed as real values 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 volume that can be produced by the available audio hardware. Volume units lie on a scale that is linear with amplitude or voltage; therefore, a doubling of the speech-volume value corresponds to a doubling of perceived loudness.

Just as a synthesizer does not usually generate speech at a constant frequency, it does not generate speech at a constant amplitude. Even when the speech rate is high, brief occurrences of silence (such as pauses between phrases) break up a steady stream of speech. The speech volume, like speech pitch, is an indicator of an average. The Carbon speech synthesis API provides a function you can use to set the volume of the current speech channel (see "Adjust Speech Channel Settings Using the Carbon Speech Synthesis API" (page 36) to find out how to do this).

Prosody

The most complex speech attribute is prosody. The **prosody** speech attribute describes the rhythm, modulation, and emphasis patterns of speech, such as word and syllable stress and the pitch at the end of a sentence. Although there is no simple mechanism for your application to determine what rhythmic patterns a speech synthesizer automatically applies to speech, you can exert some control over this aspect of spoken output by using the `emph` embedded speech command (described in "The OS X Embedded Speech Commands" (page 41)). In addition, you can use functions in the Carbon speech synthesis API to enable or disable ending prosody, which is the pitch modulation that a speech synthesizer applies to the end of a sentence.

The primary way you can affect the prosody of your application's spoken output is by using the TUNE format to supply pitch and rate specifications for individual words or phrases. For more information on how to do this, see "Use the TUNE Format to Supply Complex Pitch Contours" (page 53).

Perhaps more than with other speech attributes, you can spend a lot of time fine-tuning the prosody of the speech your application generates. If you have a limited set of strings your application needs to speak, however, it's well worth the effort to adjust the prosody (along with the other speech attributes) to achieve your goal. For some other ways to produce better-sounding speech, see "Four Ways to Improve Spoken Output" (page 58).

Opportunities for the Customization of Synthesized Speech

The Speech Synthesis framework supports many techniques for customizing the speech your application generates, ranging from simple to complex. This section outlines the various options available to you.

When you're ready to begin designing your application to include some or all of the customizations described in this section, you should read "Designing and Implementing an Application That Speaks" (page 21) for a survey of the available APIs, guidance on design considerations, and information on implementing basic speech synthesis tasks. Then, read "Techniques for Customizing Synthesized Speech" (page 36) for in-depth customization information.

Use Different Voices

One of the first things users notice about the speech your application produces is the voice that speaks it. Consequently, using a specific voice is an easy way to customize the spoken output of your application.

If the voice itself is not an important feature in your application, you can simply use the system default voice (note that users can set the default voice in the Speech System Preferences). However, you may want to designate a specific voice (or voices) or give your users the ability to choose a voice. For example, if you're developing a game that displays more than one distinct character, you need to be able to give each character its own voice. If, on the other hand, you're developing an interactive application for children, you might want to give them a selection of entertaining voices from which to choose.

Designating a specific voice or set of voices requires you to find out which voices are available on the user's system, examine individual voice descriptions to determine which ones you want, and tell the synthesizer which voice to use. Both the Cocoa and Carbon speech synthesis APIs provide programmatic ways to do this (see the code listings in "Implementing Basic Speech Synthesis Tasks Using Cocoa and Carbon" (page 27) for some examples).

Adjust Speech Attributes and Control Speech Production Using the Speech Synthesis APIs

As described in "Attributes of Synthesized Speech" (page 15), the Speech Synthesis framework defines several attributes that describe aspects of speech, such as volume and pitch. The Speech Synthesis framework provides functions that allow you to adjust rate and pitch. Note that these functions act on speech channels, not on the text itself. This means that, for example, changing the speech rate of a speech channel effectively changes the rate of all speech that channel produces. In addition, you cannot assume that such a change will persist if you change the voice on the channel. This is because telling a channel to use a different voice can cause all the channel's parameters to be reset to default values. In addition, the new voice may not support some of the attribute settings. However, there may be cases in which it makes sense for your application to change the speech rate or pitch, such as in response to a user request. For an example of how to do this, see "Adjust Speech Channel Settings Using the Carbon Speech Synthesis API" (page 36).

You can exert control over the production of spoken output by using speech synthesis functions to stop, pause, and continue speech. For example, you might allow users to select a Stop Speaking menu item or click a Pause button to control the spoken output. For more information on the functions and methods you can use to control the flow of speech and examples of how to use them, see "Implementing Basic Speech Synthesis Tasks Using Cocoa and Carbon" (page 27).

Although you can use speech synthesis functions to adjust speech attributes, such as volume and pitch, you cannot use them to successfully adjust the pronunciation of words. To fine-tune the pronunciation or prosody of individual words and phrases, you need to use embedded speech commands (described in "Control Speech Quality Using Embedded Speech Commands").

Control Speech Quality Using Embedded Speech Commands

An **embedded speech command** allows you to control the quality of spoken output with great precision, because you associate it with an individual word or phrase you want to affect. Embedded commands can be used in buffers (or strings) of both textual and phonemic representations of speech. In fact, you can combine phonemic representations of specific words or phrases with textual representations in the same string or buffer.

Embedded commands allow you to make precise adjustments to the pronunciation of words, the way words are emphasized in a sentence, and the overall cadence of the speech. You can use embedded commands to make speech easier to understand and more human-sounding or to mimic particular pronunciations and intonations. In addition, using this technique confers a significant advantage: you do not need to make any changes to the API your application calls to generate speech, because the embedded commands are contained in the text your application passes to the synthesizer. Your application need only call the standard functions or methods that begin the speech generation process, such as `SpeakString` or `startSpeakingString:` (for more information on these, see the examples in "Implementing Basic Speech Synthesis Tasks Using Cocoa and Carbon" (page 27)).

Although embedded commands are most useful for controlling the speech you create, you can also add embedded commands to control the speech generated from text users enter. For example, a word processing application might embed commands that tell the synthesizer to emphasize the pronunciation of words the user has boldfaced or underlined. For a description of the available embedded commands and examples of how to use them, see "Use Embedded Speech Commands to Fine-Tune Spoken Output" (page 39).

One embedded command, the `[[inpt <mode>]]` command, tells the synthesizer to interpret the content following the command in the mode designated by the value of the `<mode>` parameter, until it reaches another `[[inpt <mode>]]` command. For example, to supply a precise, phonemic representation of a word that appears in a buffer of text, you precede the word with the `[[inpt PHON]]` command, which tells the synthesizer to interpret the following content phonemically, and insert the `[[inpt TEXT]]` command after the word to signal the return to textual representation.

The `[[inpt <mode>]]` command also includes a mode that allows you to take advantage of the TUNE format, which is an input format that encodes a precise intonation for a word or phrase. You can use the TUNE format to replicate the intonation and timing of a particular utterance. For more information on this format and how to use it, see "Use the TUNE Format to Supply Complex Pitch Contours" (page 53).

Designing and Implementing an Application That Speaks

This chapter gathers together some strategies to consider and guidelines to follow as you design (or retrofit) an application to produce spoken output. It begins with a survey of different implementation strategies you should consider to find the one that meets your goals. It then provides user-interface guidelines you should keep in mind as you design your application. Finally, this chapter outlines the speech synthesis APIs available to you and provides some examples that show how to get started.

Strategies for Incorporating Synthesized Speech

Spoken output is a natural enhancement for a broad range of applications, from games to productivity applications to educational applications. For example, if you're designing an application for language-learning, it's clear you need to provide accurately pronounced speech users can emulate. If you're developing a game, you probably want to provide a large set of expressive phrases your characters can speak. But synthesized speech can also enhance an application that doesn't have such obvious reasons to produce spoken output, because it can provide users with a more convenient and more enjoyable way to interact with the application.

As you design your application, look for ways synthesized speech can enhance the user interface. A few suggestions are included in "User Interface Design Guidelines for Speech" (page 23). The following sections describe ways you can use synthesized speech in your application, divided into three categories that roughly correspond to the levels of effort required to implement them.

Take Advantage of System-Provided Functionality

Even if you do not include any speech-specific code in your application, users will be able to hear most of the text displayed in your application spoken aloud by a system voice. In the Text to Speech pane of Speech preferences, users can create a key combination to use when they want to hear the text they've selected in any application. In the same preference pane, users can also choose to hear the text of alerts spoken aloud (this is a feature known as Talking Alerts) and to be told when an application requires attention.

You do not have to do anything special to allow your users to benefit from these features; to the contrary, if you use standard, system-supplied APIs and technologies, it comes for free. Selectable text that appears in your application, including user-supplied text, can be spoken aloud when users press their designated key combination or when they select Speech > Start Speaking Text from the Services menu item. (Note that the Services menu item is included by default in Cocoa and Carbon applications; for more information, see *Services*

Implementation Guide and *Setting Up Your Carbon Application to Use the Services Menu* .) When your application uses system-provided mechanisms for displaying alerts, the Talking Alerts feature automatically speaks the alert text.

You may find that these built-in features meet your application's speech needs. If, however, you want to enhance and customize the spoken output in your application to differentiate it from competing products, read the following sections to explore ways you can do this.

Provide Some Customization

In addition to allowing users to select text to hear spoken aloud, your application can speak when it encounters specific conditions or performs specific tasks. For example, your application could guide new users by describing the steps required to accomplish common tasks. The speech synthesis APIs provide functions and methods you can use to associate spoken output with application-specific tasks and events (for more information on how to do this, see "Synchronize Speech with Application-Specific Actions" (page 55)).

If you want to have more control over the production of speech in your application, you can override some of the default behaviors of the synthesizer. One way to do this is to use Carbon speech synthesis functions to change speech-channel attributes, such as speech rate and pitch. Another way to do this is to use embedded speech commands (described in "Control Speech Quality Using Embedded Speech Commands" (page 19)) and insert them as needed in the text to be spoken. The synthesizer uses these commands to alter the intonation of words and phrases by controlling the pitch, word emphasis, and pause length, among other attributes. This technique is especially useful if you want ensure the correct pronunciation of a proper noun (such as your company name) or if the spoken content must conform to specific requirements (such as in a language-learning application or other educational software). Embedded speech commands are available regardless of the programming language you're using.

Note: Even though user-supplied text does not contain embedded commands, you can add them when it makes sense. For example, you might place extra emphasis on words or phrases users surround with asterisks or underscores.

Provide Advanced Customization

The phonemic and TUNE input-processing modes allow you to make fine-grained adjustments to spoken output. For example, you can stipulate the pronunciation of a word by giving the synthesizer the individual phonemes that comprise the word.

Using the TUNE input-processing mode, you can reproduce all the minute variations in pitch and rate of an actual utterance, allowing your application to produce speech that replicates some of the subtleties of human speech. If you want your application to produce speech that follows such exact specifications, see "Use Phoneme Modifiers to Adjust Pronunciation" (page 49) and "Use the TUNE Format to Supply Complex Pitch Contours" (page 53).

User Interface Design Guidelines for Speech

As described in "Why Use Synthesized Speech?" (page 7), there are many ways to enhance your application by providing spoken output. For example, you can use speech to notify users of something that happened in the background, such as "Your download is finished" or "You have a meeting in 15 minutes." Essentially, spoken output is another facet of the user interface and, as such, it should follow most of the high-level guidelines in *OS X Human Interface Guidelines* . In addition to those guidelines, keep in mind this section's design considerations and speech-specific guidelines as you design your application.

Consider providing spoken confirmation of information users enter or selections they make. For example, a user may not be looking at the screen when typing in data from another source, and spoken confirmation of the input would be welcome. Similarly, if a user inadvertently selects the wrong item from a long list, spoken confirmation of each choice would immediately alert the user to any mistakes.

When using speech to notify users that an event has occurred, consider pausing for a few seconds between the visual display of the event (such as a dialog) and the spoken message. Speech is an effective way to get users's attention if they are not already looking at the screen, but if they are, the spoken notification might seem redundant. Inserting a delay between the visual and aural notification gives users the opportunity to respond to the event without hearing any speech. If such a pause makes sense in your application, be sure to provide a way for users to customize its length.

To provide a consistent and enjoyable speech experience to your users, follow these guidelines:

- Provide a way for users to turn on and off the spoken output your application produces. Hearing-impaired users will not benefit from the speech your application generates, and other users may not like it or may use your application in situations that require silence (such as in a library). Consider your target market carefully before you decide to make speech the default setting.
- Be sure to provide a visual alternative to all spoken (or aural) communication your application produces. Users should be able to choose between spoken and visual communication without losing access to any information or functionality.
- As much as possible, provide ways for users to customize the speech they hear. For example, users should be able to choose a pleasing voice and to set the speech rate and pitch to comfortable levels.

- When testing your application, be sure to listen to all spoken output your application generates. This way, you'll be able to catch incorrect and ambiguous pronunciations and fix them with embedded speech commands and phoneme modifiers.
- As with written alerts, spoken alerts should tell the user what happened, why it happened, and what the user can do about it. For more guidelines on writing effective alert messages, see *OS X Human Interface Guidelines* .
- As much as possible, express complicated ideas in a few short sentences instead of one long sentence. It's much harder for users to process complicated spoken information because they can't go back and reread the parts they missed or didn't understand. Effective use of punctuation can also help make complex sentences easier to understand. This is because the synthesizer, like a human speaker, pauses when it encounters commas, semicolons, and periods. For more information on how to draw users's attention to the important information within a sentence, see "Four Ways to Improve Spoken Output" (page 58).
- Be sure to document the speech capabilities of your application. If users don't know what features are available and how they can customize them, they might not ever use them.

Carbon and Cocoa Speech Synthesis APIs Compared

Important: As long as your application can access the Application Services framework, it can use the Carbon speech synthesis API, regardless of the programming language you're developing in. This means that you can create a Cocoa application, or even a command-line tool, include the `ApplicationServices.framework`, and call Carbon speech synthesis functions to implement speech-generation tasks.

Before you begin designing your application with synthesized speech in mind, note that the type of customization you plan to do has some impact on your choice of API. Both Carbon and Cocoa supply basic speech-synthesis functionality, but the Carbon API provides more programmatic control over speech attributes. Unlike the Carbon speech synthesis API, the `NSSpeechSynthesizer` class defined in the Application Kit does not support the ability to convert text to phonemes or to change speech attributes. If you don't plan to take advantage of the programmatic features now or in a future version of your application, you can use the Cocoa API without worrying about having to redesign and recode the application later. If, however, you want to support advanced capabilities (or there's a chance that you might do so in the future), you should consider using the Carbon API from the beginning.

Although you can mix the Cocoa and Carbon speech synthesis APIs in a single application, you may experience a few difficulties because of differences in implementation. For example, if you specify a voice that the current speech synthesizer doesn't support, in Carbon you must explicitly close the current speech channel and open a new one to use the new voice, whereas in Cocoa this process is automatic. You may find that your best option is to develop the application in Cocoa, but use the Carbon speech synthesis API for all speech-related tasks.

Before you choose an API, bear in mind that you can accomplish a great deal of speech customization by adding embedded commands to the text your application passes to the synthesizer. However, a potential disadvantage to using embedded commands is that you must add the appropriate embedded commands to every occurrence of a particular word to specify its pronunciation. Contrast this with calling a function that sets a speech attribute for all spoken output that passes through a speech channel. Depending on your circumstances, however, you may decide that this disadvantage is outweighed by the finer-grained control that comes with using embedded commands.

The remainder of this section provides brief overviews of the Cocoa and Carbon speech synthesis APIs. For in-depth reference information on these APIs, see *NSSpeechSynthesizer Class Reference* and *Speech Synthesis Manager Reference* .

Overview of the Cocoa Speech Synthesis API

The Cocoa API includes the `NSSpeechSynthesizer` class, which handles a number of speech synthesis tasks in a way native to Objective-C. When you create and initialize an instance of `NSSpeechSynthesizer`, a speech channel is created and a voice (either the default system voice or one you designate in the initialization method) is associated with the object. The `NSSpeechSynthesizer` object is your application's conduit to the Speech Synthesis framework.

The `NSSpeechSynthesizer` class defines methods that allow you to:

- Get information about a voice (such as age and gender)
- Change the voice used for spoken output
- Determine if another application is currently speaking
- Start and stop speech
- Manage delegates

To make your application speak a word or phrase, you use an instance method to send text to your `NSSpeechSynthesizer` object (alternatively, you can use an instance method to cause the sound output to be saved to a file). Using the delegate methods defined by the `NSSpeechSynthesizer` class, you can also perform application-specific actions just before a word or phoneme is spoken or just after the synthesizer finishes speaking a string. You might use these methods to, for example, change the state of a start/stop speaking button or to synchronize the animation of a character's mouth with the spoken output.

Although you can use a class method to get the attributes for a specific voice, the `NSSpeechSynthesizer` class does not define methods that allow you to get or change the attributes of a speech channel. In addition, the `NSSpeechSynthesizer` class does not support the programmatic conversion of text to phonemes. To do these things, you must use functions in the Carbon speech synthesis API.

Overview of the Carbon Speech Synthesis API

The Carbon speech synthesis API (also called the Speech Synthesis Manager) includes functions that allow you to:

- Create and manage speech channels
- Adjust speech attributes on a speech channel
- Convert text to phonemes
- Get information about speech channels and voices
- Start, stop, and pause speech
- Create, invoke, and dispose of universal procedure pointers that point to functions you supply to synchronize speech with application-specific actions

In addition to these functions, the Carbon speech synthesis API defines constants that describe voice and speech-channel attributes, data types (such as phoneme and voice description structures), and a large number of selectors that operate on speech channels.

Even though the Carbon speech synthesis API is not object-oriented, it may help to think of a speech channel (a structure of type `SpeechChannel`) as analogous to an instance of the `NSSpeechSynthesizer` class. This is because a speech channel is the primary conduit between your application and the Speech Synthesis framework, and you must create one to perform most speech-related tasks, such as getting information about a voice, sending text to be spoken, or adjusting speech attributes. The one exception to this is the `SpeakString` function, which does not require you to create a speech channel. When you pass a string to the `SpeakString` function, the Speech Synthesis Manager automatically creates and manages the structures required to speak.

Using selectors that you can pass to the `SetSpeechInfo` function, you can replicate some of the functionality you get when you use embedded speech commands. For example, you can change the input-processing mode on the speech channel by passing the `soInputMode` selector. This has the same effect as the `[[inpt <mode>]]` embedded speech command, except that it operates on the speech channel as a whole, not on a portion of the text. Table 3-1 (page 42) pairs each embedded speech command with its analogous selector, if one exists. Other selectors allow you to set speech channel attributes or to associate a callback function with a speech channel. See *Speech Synthesis Manager Reference* for a complete list of available selectors.

Implementing Basic Speech Synthesis Tasks Using Cocoa and Carbon

This section describes how to use the Cocoa and Carbon APIs to perform basic set-up tasks, such as getting a speech channel, designating a specific voice, starting and stopping speech, and responding to speech events.

Generating Speech Using the Cocoa Speech Synthesis API

To generate speech using the Cocoa speech synthesis API, you must instantiate an `NSSpeechSynthesizer` object and send to it the text to speak. The code in Listing 2-1 shows how to use this object to get information about available voices and how to respond to some speech events. This code is a simplified version of the NSSpeechSynthesizerExample example project located in `/Developer/Examples/Speech/Synthesis`. The code in Listing 2-1 does not show how to create a pop-up menu of available voices or manage text selection and it does not implement any error handling.

The code in Listing 2-1 shows an implementation of an `NSObject` subclass called `ExampleWindow`. It uses a simple window that contains the following items:

- A text view (declared as `NSTextView * _textView`) that displays the text to be spoken
- A pop-up menu (declared as `NSPopUpButton * _voicePop`) that displays the available voices from which the user can choose
- A button (declared as `NSButton * _speakButton`) the user clicks to start and stop speech

Listing 2-1 Generating speech using the Cocoa speech synthesis API

```
@implementation ExampleWindow
/* Instantiate an NSSpeechSynthesizer object when the application starts */
- (void)awakeFromNib
{
    _speechSynthesizer  = [NSSpeechSynthesizer new];
    /* Make the ExampleWindow object the responder to NSSpeechSynthesizer delegate
 methods */
    [_speechSynthesizer setDelegate:self];


    /* Call a custom method to populate the pop-up menu of available voices
(implementation not shown) */
    [self getSpeechVoices];
}


/* When the user clicks the Start Speaking button, invoke the custom
startSpeakingTextView method to retrieve (or create) the text and speak it */
```

```
- (IBAction) speakTextButtonSelected:(id)sender
{
  [self startSpeakingTextView];
}


- (void)startSpeakingTextView
{
    if([_speechSynthesizer isSpeaking]) {
        [_speechSynthesizer stopSpeaking];
    }
    else {
         NSString *    theViewText;
        /* If the user chooses to hear the default system voice, get the text to
speak from the window (either the default text or user-supplied) */
        if ([_voicePop indexOfSelectedItem] == 0) {
            [_speechSynthesizer setVoice:NULL];
            theViewText = [_textView string];
        }
       /* Otherwise, get the user's chosen voice, create a string using the voice's
 demo text, and speak it */
        else {
            [_speechSynthesizer setVoice:[[NSSpeechSynthesizer availableVoices]
objectAtIndex:[_voicePop indexOfSelectedItem] - kNumOfFixedMenuItemsInVoicePopup]];
            /* Get the attributes of the chosen voice */
            NSDictionary * attributes = [NSSpeechSynthesizer
attributesForVoice:[_speechSynthesizer voice]];
            /* Get the value of the voice's name attribute */
            NSString * theName = [attributes objectForKey:NSVoiceName];
           /* Build a string using the voice's name and demo text in this format:
 "This is <name>. <Demo text.>" */
            theViewText = [NSString stringWithFormat:@"This is %@. %@",
theName,[attributes objectForKey:NSVoiceDemoText]];
            /* Display this new string in the window */
            [_textView setString:theViewText];
         }
        /* Send string to synthesizer object */
        [_speechSynthesizer startSpeakingString:theViewText];
        /* Change button name to reflect current state */
        [_speakButton setTitle:@"Stop Speaking"];
    }
}
@end
```

As shown in the `awakeFromNib` method in Listing 2-1, the `ExampleWindow` object will respond to delegate methods defined by the `NSSpeechSynthesizer` class. Listing 2-2 includes example implementations of two of these methods, showing how to perform application-specific actions that are synchronized with speech events.

Listing 2-2 Using delegate methods to respond to speech events

```
/* This delegate method is invoked when the NSSpeechSynthesizer object has finished
 speaking. This happens when there is no more text to speak or when the user clicks
 the Stop Speaking button. */


- (void)speechSynthesizer:(NSSpeechSynthesizer *)sender
didFinishSpeaking:(BOOL)finishedSpeaking
{
    /* Return cursor to beginning of line */
    [_textView setSelectedRange:NSMakeRange(0,0)];
    /* Reset button title to initial string */
    [_speakButton setTitle:@"Start Speaking")];
    [_speakButton setEnabled:YES];
    [_voicePop setEnabled:YES];
}


/* This delegate method is called when a word (defined by its character range
within the string) is about to be spoken. This implementation uses this information
 to highlight each word as it's being spoken. */


- (void)speechSynthesizer:(NSSpeechSynthesizer *)sender
willSpeakWord:(NSRange)characterRange ofString:(NSString *)string
{
    UInt32    selectionPosition = characterRange.location;
    UInt32    wordLength = characterRange.length;
    [_textView scrollRangeToVisible:NSMakeRange(selectionPosition, wordLength)];
    /* Highlight word about to be spoken */
    [_textView setSelectedRange:NSMakeRange(selectionPosition, wordLength)];
    [_textView display];
}
```

Generating Speech Using the Carbon Speech Synthesis API

To generate speech using the Carbon speech synthesis API, you must create a speech channel and send to it the text to speak. The example code in this section is modeled on the CocoaSpeechSynthesisExample example project (located in `/Developer/Examples/Speech/Synthesis`), which shows how to use the Carbon speech synthesis API within a Cocoa application. Much of the example application's infrastructure is provided by Cocoa's `NSDocument` class and the code that displays and manages the window and its contents is not reproduced in the following code listings. The code in the listings below shows how to use a handful of the Carbon speech synthesis functions; see the CocoaSpeechSynthesisExample application for a broader sampling.

The code in Listing 2-3 shows a partial implementation of an `NSDocument` subclass, called `SpeakingTextWindow`. `SpeakingWindow` contains the following instance variables:

- `fCurSpeechChannel` (of type `SpeechChannel`) to point to the current speech channel
- `fCurrentlySpeaking` (of type `BOOL`) to indicate the current speech state

Listing 2-3 Generating speech using the Carbon speech synthesis API

```
/* Callback function prototype: */
static pascal void     MyWordCallBackProc(SpeechChannel inSpeechChannel, long
inRefCon, long inWordPos, short inWordLen);


@implementation SpeakingTextWindow
- (void)awakeFromNib
{
    OSErr        theErr = noErr;
    short        numOfVoices;
    long         voiceIndex;
    BOOL        voiceFoundAndSelected = false;
    VoiceSpec    theVoiceSpec; /* VoiceSpec is a structure that contains the
identity of the synthesizer required to use a voice and the ID of a voice. */
```

```
    /* Get the number of voices on the system. Note that you do not need to get a
 speech channel to get information about available voices. */
    theErr = CountVoices(&numOfVoices); // Handle error if necessary.


    for (voiceIndex = 1; voiceIndex <= numOfVoices; voiceIndex++) {
        VoiceDescription    theVoiceDesc;
        /* Get the VoiceSpec structure for this voice. The structure fields will
be filled in by a call to GetVoiceDescription. */
        theErr = GetIndVoice(voiceIndex, &theVoiceSpec); // Handle error if
necessary.


        /* Fill in the fields of the theVoiceDesc VoiceDescription structure. */
        theErr = GetVoiceDescription(&theVoiceSpec, &theVoiceDesc,
sizeof(theVoiceDesc)); // Handle error if necessary.


        /* Add this voice name to the pop-up menu (not shown). */
    }


    /* If a speech channel already exists, dispose of it. */
    if (fCurSpeechChannel) {
        theErr = DisposeSpeechChannel(fCurSpeechChannel); // Handle error if
necessary.
        fCurSpeechChannel = NULL;
    }


    /* Create a speech channel. */
    theErr = NewSpeechChannel(NULL, &fCurSpeechChannel); // Handle error if
necessary.
   /* Set the refcon to the document controller object to ensure that the callback
 functions have access to it. */
   theErr = SetSpeechInfo(fCurSpeechChannel, soRefCon, (Ptr)self); // Handle error
 if necessary.
    /* Enable the Start/Stop and Pause/Continue buttons (not shown). */
}


- (IBAction)startStopButtonPressed:(id)sender
{
    /* This action method is called when a user clicks the Start/Stop speaking
button. */
    OSErr theErr = noErr;


    if (fCurrentlySpeaking) {
       /* If speech is currently being produced, stop it immediately. Alternatively,
 you could use the StopSpeechAt function to stop the speech at the end of a word
or sentence.*/
        theErr = StopSpeech(fCurSpeechChannel); // Handle error if necessary.
        fCurrentlySpeaking = false;
        /* Update the controls, based on current speaking state (the
updateSpeakingControlState method is not shown). */
        [self updateSpeakingControlState];
    }
    else {
        /* Call the method that sets up the callbacks on the speech channel and
sends the text to be spoken. */
        [self startSpeakingTextView];
    }
}


- (void)startSpeakingTextView
{
    /* This method sets up a callback that gets called when a word has been spoken.
 It also starts spoken output by calling the SpeakText function. */
    OSErr theErr = noErr;
    NSString * theViewText;


    /* Get the text from the window and store in theViewText (not shown). */
    /* Set up the word callback function. Other callback functions can be set up
in a similar way. */
    theErr = SetSpeechInfo(fCurSpeechChannel, soSpeechDoneCallBack,
MySpeechDoneCallBackProc); // Handle error if necessary.


    /* Convert the theViewText NSString object to a C string variable.*/
    char * theTextToSpeak = (char *)[theViewText lossyCString];
    /* Send the text to the speech channel. */
   theErr = SpeakText(fCurSpeechChannel, theTextToSpeak, strlen(theTextToSpeak));
 // Handle error if necessary.


   /* Update variables and control states (you might want to define other variables
 to hold the current pause state and the most recent error code). */
    fCurrentlySpeaking = true;
    [self updateSpeakingControlState];
}
```

As shown in Listing 2-3, the `startSpeakingTextView` method sets up a callback procedure on the speech channel. The CocoaSpeechSynthesisExample example application uses the callback procedure to call a function that highlights each word in the text as it's spoken.

The code in Listing 2-4 shows the callback procedure, which uses the `NSObject` method `performSelectorOnMainThread:withObject:waitUntilDone:` to call the routine that actually performs the processing associated with the callback. The reason `MyWordCallBackProc` doesn't perform the word highlighting itself is that all Carbon speech synthesis callbacks (except `SpeechTextDoneProcPtr`) call their associated functions on a thread other than the main thread. Unless you've indicated that your Cocoa application is multithreaded, this can cause problems if your callback routine touches the user interface or other application objects. To avoid these problems, use the `performSelectorOnMainThread:withObject:waitUntilDone:` method to ensure your callback processing routine is called on the main thread. Of course, this mechanism is unnecessary in a pure Carbon application.

Listing 2-4 Using a Carbon callback procedure to respond to a speech event

```
pascal void MyWordCallBackProc(SpeechChannel inSpeechChannel, long inRefCon, long
 inWordPos, short inWordLen)
{
    NSAutoreleasePool *    pool = [[NSAutoreleasePool alloc] init];


    /* Call the highlightWordWithParams: method to highlight each word as it's
spoken. highlightWordWithParams (not shown) receives a dictionary containing two
values: the number of bytes between the beginning of the text and the beginning
of the word about to be spoken and the length in bytes of that word. */


    [(SpeakingTextWindow *)inRefCon
performSelectorOnMainThread:@selector(highlightWordWithParams:)
withObject:[NSDictionary dictionaryWithObjectsAndKeys:[NSNumber
numberWithLong:inWordPos], kWordCallbackParamPosition, [NSNumber
numberWithLong:inWordLen], kWordCallbackParamLength, NULL] waitUntilDone:false];
```

```
    [pool release];
}
```

If you'd like to explore making your Cocoa application multithreaded, see *Threading Programming Guide*. If you're writing an application similar to CocoaSpeechSynthesisExample and you'd like to make it multithreaded, be sure to include the following line of code before you call any Carbon speech synthesis function for the first time:

```
[NSThread detachNewThreadSelector:@selector(self) toTarget:self withObject:nil];
```

After you've used the `detachNewThreadSelector:toTarget:withObject:` method to create a new thread, you can then perform the callback processing tasks within your callback procedures.

Using AppleScript to Produce Spoken Output

Using the AppleScript `say` command, you can cause text to be spoken aloud or saved to a file. The `say` command is one of the user interaction commands available in the Standard Additions scripting addition (available in `/System/Library/ScriptingAdditions`). To experiment with the script examples in this section, open the Script Editor application (located in `Applications/AppleScript`), type the script into the Script Editor window, and click Run.

The `say` command speaks the string that follows it (the string can be text enclosed in double quotes or text in a variable). Optionally, you can use the `using` parameter to tell the `say` command to use a specific voice and the `saving to` parameter to redirect the spoken output to an AIFF file. The `say` command also accepts two parameters that are ignored unless Speech Recognition is turned on. These two parameters (`displaying` and `waiting until completion`) are not described in this document. For more information on the syntax and usage of the `say` command, open `StandardAddition.osax` in Script Editor.

The following example uses the Switch to Finder script (located in `Applications/AppleScript/Example Scripts/Finder Scripts`) to show how you can add the `say` command to a script to produce spoken output.

Listing 2-5 Using AppleScript to produce spoken output

```
tell application "Finder"
    activate
    set visible of every process whose visible is true and name is not "Finder"
to false
    say "To see other application windows again, select Show All from the Finder
```

```
menu." using "Vicki"
end tell
```

If you save the spoken output to an AIFF file, you can use it in some other application or listen to it in iTunes (or download it to an iPod). The following example adds a second `say` command to the script in Listing 2-5, this one directing some of the spoken output to a file in the `/Users` folder.

Listing 2-6 Using AppleScript to save spoken output to a file

```
tell application "Finder"
    activate
    set visible of every process whose visible is true and name is not "Finder"
to false
    say "To see other application windows again, select Show All from the Finder
menu." using "Vicki"
    say "This is an example of using the AppleScript say command to save spoken
output to a file." saving to "Users:AppleScript_speech.aiff"
end tell
```

Techniques for Customizing Synthesized Speech

This chapter describes how to fine-tune the speech your application generates. It provides guidelines for using speech synthesis APIs and embedded speech commands and it includes a number of examples of specific tasks. This chapter also describes several ways you can improve your application's spoken output. If you need fine-grained control over the generated speech in your application, you should read this chapter to learn how to take advantage of the advanced features of the Speech Synthesis framework.

Some of the advanced techniques described in this chapter are supported only by the Carbon speech synthesis API, although any application that has access to the Application Services framework can use them. If you haven't decided which API to use, you should read "Carbon and Cocoa Speech Synthesis APIs Compared" (page 24) to find out which API supports the level of customization you want to implement. Other techniques described in this chapter involve the use of embedded speech commands and other text modifiers, which are available to all applications.

Adjust Speech Channel Settings Using the Carbon Speech Synthesis API

The Carbon speech synthesis API allows you to get and set speech attributes, such as rate and volume, and specify other settings on speech channels, such as input mode. In addition to a couple of functions that focus on specific attributes, the Carbon speech synthesis API defines the `GetSpeechInfo` and `SetSpeechInfo` functions, which act upon a speech channel using an attribute, setting, or other value specified in a selector parameter. This section describes how you can use the attribute-specific functions and the `SetSpeechInfo` function to adjust the speech attributes and other settings on speech channels.

The Carbon speech synthesis API defines the following functions to get and set the rate and pitch attributes on a speech channel:

- `GetSpeechRate`
- `SetSpeechRate`
- `GetSpeechPitch`
- `SetSpeechPitch`

For example, an application might get a new speech rate value from the user and use it to change the speech rate used by the speech channel, as shown below:

```
/* fRateField is associated with a button in the UI and fCurSpeechChannel is a
pointer to a speech channel structure created earlier in the application. */
Fixed theNewValue   = [fRateField doubleValue] * 65536.0;
theErr = SetSpeechRate(fCurSpeechChannel, theNewValue);
```

In a similar way, an application can use the `SetSpeechPitch` function to set a speech channel's pitch attribute to a new value. To get or set other speech attributes and settings on a speech channel, however, you use the `GetSpeechInfo` and `SetSpeechInfo` functions with the appropriate selectors. The one exception to this is the rate attribute, which can be retrieved and set using either the `GetSpeechRate` and `SetSpeechRate` functions mentioned above or the `SetSpeechInfo` function with the `soRate` selector, as shown below:

```
/* As above, fRateField is associated with a button in the UI and fCurSpeechChannel
 is a pointer to a speech channel structure created earlier in the application.
*/
Fixed theNewValue   = [fRateField doubleValue] * 65536.0;
theErr = SetSpeechInfo(fCurSpeechChannel, soRate, &theNewValue);
```

The selectors defined in the Carbon speech synthesis API act upon a wide range of properties associated with speech channels. The selectors divide into the following categories:

- Attribute selectors, which specify a speech attribute, such as volume or rate, that the speech channel should use when generating speech.

  All attribute selectors work with both the `GetSpeechInfo` and `SetSpeechInfo` functions. As an alternative to using an attribute selector with the `SetSpeechInfo` function, you can use an equivalent embedded speech command to set a speech attribute (see Table 3-1 (page 42) for more about these commands). The currently available selectors in this category are:

  - `soRate`
  - `soPitchBase`
  - `soPitchMod`
  - `soVolume`

- Input-mode selectors, which specify the mode, such as phoneme or character, that the speech channel should use when processing text.

  All input-mode selectors work with both the `GetSpeechInfo` and `SetSpeechInfo` functions. As an alternative to using a mode selector with the `SetSpeechInfo` function, you can use an equivalent embedded speech command to specify an input-processing mode (see Table 3-1 (page 42) for more about these commands). The currently available selectors in this category are:

  - `soInputMode`

- `soCharacterMode`
- `soNumberMode`

- Callback selectors, which associate with a speech channel an application-defined callback function to be called when a particular speech event occurs, such as the conclusion of speech.

  All callback selectors work with only the `SetSpeechInfo` function. One callback selector, `soSyncCallBack`, has a related embedded speech command you can use to trigger a callback to your synchronization callback procedure (see Table 3-1 (page 42) for more about this command). The currently available selectors in this category are:

  - `soTextDoneCallBack`
  - `soSpeechDoneCallBack`
  - `soSyncCallBack`
  - `soErrorCallBack`
  - `soPhonemeCallBack`
  - `soWordCallBack`

- Information selectors, which provide information about the current speech channel and the synthesizer associated with it, such as the set of phoneme symbols recognized by the synthesizer.

  All information selectors work only with the `GetSpeechInfo` function. There are no embedded speech commands equivalent to the information selectors. The currently available selectors in this category are:

  - `soStatus`
  - `soErrors`
  - `soSynthType`
  - `soRecentSync`
  - `soPhonemeSymbols`

- Setting selectors, which specify a speech channel setting, such as the current voice.

  Some setting selectors work with both the `GetSpeechInfo` and `SetSpeechInfo` functions, others with only the `SetSpeechInfo` function. One setting selector, `soCommandDelimiter`, has an equivalent embedded speech command you can use to change the default command delimiter strings (see Table 3-1 (page 42) for more about this command). The currently available selectors in this category are:

  - `soCurrentVoice`
  - `soCommandDelimiter`
  - `soReset`
  - `soRefCon`
  - `soSynthExtension`

- `soOutputToFileWithCFURL`

Use Embedded Speech Commands to Fine-Tune Spoken Output

As described in "Control Speech Quality Using Embedded Speech Commands" (page 19), you use embedded commands to fine-tune the pronunciation of individual words in the text your application passes to a synthesizer. Even if you use only a few of the embedded speech commands described in this section, you may significantly increase the understandability of your application's spoken output. This section provides an overview of embedded speech command syntax, lists the available commands, and illustrates how to use them to achieve different effects.

Note that some embedded speech commands have functional equivalents provided by the Carbon selector mechanism (for a complete list of available selectors, see *Speech Synthesis Manager Reference* .) This means that to achieve some effects, you can either insert the embedded command in the text, or you can pass the equivalent selector to the Carbon `SetSpeechInfo` function. If you use the `SetSpeechInfo` function (described in "Adjust Speech Channel Settings Using the Carbon Speech Synthesis API" (page 36)), the effect applies to all speech passing through the current speech channel, subject to synthesizer capabilities. If you use the embedded command to achieve the same effect, however, it applies only to the word immediately preceded by the embedded command.

Embedded Speech Command Delimiters

When processing an input string or buffer, speech synthesizers look for special strings of characters called command delimiters. These character strings are usually defined to be pairings of printable characters that do not typically appear in the text. One character string is defined as the begin command delimiter and another character string is defined as the end command delimiter. When the synthesizer encounters the begin command delimiter string, it interprets the characters following it as one or more embedded commands until it reaches the end command delimiter string.

The default begin and end command delimiter strings recognized by the MacinTalk synthesizer are "[[" and "]]," respectively. You can change these strings if necessary, but you should take care to use printable characters that you do not expect to see in the text your application processes. Also, if you change the default delimiters, be sure to change them back to the default characters when you have finished with the text, because the change is persistent for the current speech channel. For example, if you expect square brackets to appear in the text you'll be sending to the synthesizer, you can change the default command delimiters to strings containing other printable characters that do not naturally occur in your text.

You can disable the processing of all embedded commands by setting both the begin and end command delimiters to two NUL bytes. You might want to do this if your application speaks text over which you have no control and you're absolutely sure the text contains no embedded commands. To disable processing of embedded commands programmatically, use the `soCommandDelimiter` selector with the `SetSpeechInfo` function, as shown below:

```
// Create a structure to hold the new delimiter values
DelimiterInfo MyNewDelimiters;
MyNewDelimiters.startDelimiter[0] = 0;
MyNewDelimiters.startDelimiter[1] = 0;
MyNewDelimiters.endDelimiter[0] = 0;
MyNewDelimiters.endDelimiter[1] = 0;
SetSpeechInfo(CurrentSpeechChannel, soCommandDelimiter, &MyNewDelimiters);
```

Overview of Embedded Speech Command Syntax

Note: This section describes enough of the embedded command syntax for you to be able to understand the examples in this document. For a formal description of the syntax of embedded speech commands and their parameters, see "Syntax of Embedded Speech Commands" (page 62).

All embedded commands consist of a 4-character command code and a parameter, enclosed by the begin and end command delimiter strings. For example, the `emph` command requires a parameter that tells the synthesizer to increase or decrease the emphasis with which to speak the next word, as shown below:

`[[emph +]]` The + parameter tells the synthesizer to increase emphasis for the following word.

More than one command may occur within a single pair of delimiter strings if they are separated by semicolons, as shown below:

`[[emph +; rate 165]]` Together, these commands tell the synthesizer to speak the following word or phrase with increased emphasis and at a rate of 165 words per minute.

A parameter may consist of a string, a numeric type, or an operating-system type, and may be accompanied by the + or - characters (the exact format of a parameter depends on the command with which it's associated). Some commands allow you to use the parameter to specify either an absolute value or a relative value. For example, the `volm` command allows you to specify a particular volume or an amount by which to increase or decrease the current volume, as shown below:

`[[volm 0.3]]` This command sets the volume with which the following word is spoken to 0.3.

`[[volm +0.1]]` This command increases the volume with which the following word is spoken by 0.1.

The speech synthesizer ignores all whitespace within an embedded command, so you may insert as many spaces as you need to make your command text more readable.

In addition, this document uses the following characters to express the syntax of embedded speech commands (these characters do not appear in actual embedded speech commands):

- The < and > characters enclose items that represent logical units, such as string, character, integer, or real value. When you insert an embedded command in your text, you replace the logical unit with an actual value. For example, you might replace "`<RealValue>`" with `3.0`. For precise definitions of each logical unit, see the formal description of the syntax in "Syntax of Embedded Speech Commands" (page 62).
- The | character means "or" and appears between members in a list of possible items, any single one of which may be used. For example, the `emph` command accepts either the + character or the - character for its parameter. Therefore, the syntax of the `emph` command is expressed as `emph + | -`.
- The [and] characters enclose an optional item or list of items. For example, the `rate` command accepts the optional addition of the + or - character to its numerical parameter to indicate a change relative to the current value. Therefore, the syntax of the `rate` command is expressed as `rate [+ | -] <RealValue>`.
- Items followed by an ellipsis character (...) may be repeated one or more times.

The OS X Embedded Speech Commands

Table 3-1 describes the embedded speech commands, their parameters, equivalent speech information selectors (if they exist), and in which versions of OS X the commands are available. The syntax of each command in Table 3-1 is expressed using the conventions described in "Overview of Embedded Speech Command Syntax" (page 40).

Note: All embedded speech commands, except for `ctxt`, are available in OS X v10.0 and later. The `ctxt` command is available in OS X v10.4 and later.

Table 3-1 Embedded speech commands

Command	Syntax and description	Selector

`char`	`char NORM | LTRL`
	The character mode command sets the word-speaking mode of the speech channel. When the `NORM` parameter is used, the synthesizer attempts to automatically convert words into speech. This is the most basic function of the synthesizer. When the `LTRL` parameter is used, the synthesizer speaks the individual characters of every word, number, and symbol following the command (all other embedded commands are processed normally). For example, to cause the synthesizer to speak the word "cat" as "C-A-T," you would include the following in a text buffer or string:
	`[[char LTRL]] cat [[char NORM]]`
		`SoCharacterMode`

`cmnt`	`cmnt [<Character>...]`
	The comment command is ignored by speech synthesizers. It enables you to add arbitrary content to the text buffer that will never be included in the spoken output. Note that the comment text itself must be included within the begin and end command delimiters of the `cmnt` command.
	`[[cmnt This is a comment that will be ignored by the synthesizer.]]`
		None

Command	Syntax and description	Selector
`ctxt`		None

Command	Syntax and description	Selector

	`ctxt [WSKP | WORD | NORM | TSKP | TEXT]`
	
	The context command allows you to identify the context of a word to help the synthesizer generate the correct pronunciation of that word, even if no other words in the surrounding phrase or sentence are spoken. Because the pronunciation of words can be different depending on the context in which they appear, you can use the context command to specify the pronunciation used in a particular context.
	
	The context command recognizes two modes: word-by-word and text fragment. In both modes, you use the appropriate "skip" parameter (`WSKP` or `TSKP`) to identify the text that provides context and the `WORD` or `TEXT` parameter to identify the word or phrase whose pronunciation is affected by the context. The synthesizer parses the entire phrase or sentence to determine the correct pronunciation of the word or phrase, but does not speak the portions of the text marked as "skipped." Use the `[[ctxt NORM]]` command to signal a return to the default input-processing mode.
	
	In word-by-word mode, the synthesizer parses the complete text selection to determine the part of speech (such as noun or verb) of the specified word. The synthesizer pronounces the word according to its part of speech, but it does not make any intonation or duration adjustments to the pronunciation. For example, the word "coordinates" is pronounced differently depending on whether it is used as a noun or a verb. The two sentences below illustrate how to use the context command to tell the synthesizer which pronunciation of the word to use:
	
	`[[ctxt WSKP]] GPS provides [[ctxt WORD]] coordinates. [[ctxt NORM]]`
	
	`[[ctxt WSKP]] The post office [[ctxt WORD]] coordinates [[ctxt WSKP]] its deliveries. [[ctxt NORM]]`
	
	In text fragment mode, the synthesizer parses the complete text selection to determine the part of speech and the intonation and duration of the specified word or phrase. For example, the different pronunciations of the phrase "first step" are informed by the context provided by the surrounding words in the following two sentences:	

Command	Syntax and description	Selector

	`[[ctxt TSKP]] Your [[ctxt TEXT]] first step [[ctxt TSKP]] should be to relax. [[ctxt NORM]]`
	
	`[[ctxt TSKP]] To relax should be your [[ctxt TEXT]] first step. [[ctxt NORM]]`	

`dlim`	`dlim <BeginDelimiter> <EndDelimiter>`
	
	The delimiter command changes the character sequences that indicate the beginning and end of all subsequent embedded speech commands. The new delimiters take effect after the command list containing the `dlim` command has been completely processed. If the delimiter strings are empty, an error is generated. If you want to disable embedded command processing for the remainder of the text buffer, you can pass two NUL bytes in the `BeginDelimiter` and `EndDelimiter` parameters.
	
	`[[dlim $$ $$]`
		`soCommandDelimiter`

`emph`	`emph + | -`
	
	The emphasis command causes the synthesizer to speak the next word with greater or less emphasis than it is currently using. The + parameter increases emphasis and the - parameter decreases emphasis.
	
	For example, to emphasize the word "not" in the following phrase, use the `emph` command as follows:
	
	`Do [[emph +]] not [[emph -]] over tighten the screw.`
		None

Command	Syntax and description	Selector

`inpt`	`inpt TEXT | PHON | TUNE`
	
	The input mode command switches the input-processing mode to textual mode, phoneme mode, or TUNE format mode. Note that some synthesizers may define additional speech input modes you can use. The default input-processing mode is textual, and you should always use the `[[inpt TEXT]]` command to revert to textual mode after you're finished providing content in one of the other modes. In phoneme mode, the synthesizer interprets characters as representing phonemes (listed in "Phonemes" (page 64)). In the TUNE format mode, the synthesizer recognizes the same set of phonemes but also interprets additional information that specifies a precise spoken contour, or tune, for the words. For more information about the TUNE format, see "Use the TUNE Format to Supply Complex Pitch Contours" (page 53).
	
	For example, to supply the phonemic representation of a name that synthesizers frequently mispronounce, you can use the `inpt` command as follows:
	
	`My name is [[inpt PHON]] AY1yIY2SAX [[inpt TEXT]].`
		`soInputMode`

`nmbr`	`nmbr NORM | LTRL`
	
	The number mode command sets the number-speaking mode of the synthesizer. The `NORM` parameter causes the synthesizer to speak the number 46 as "forty-six," whereas the `LTRL` parameter causes the synthesizer to speak the same number as "four six."
	
	For example, to make it clear that the following 7-digit number is a phone number, you can use the `nmbr` command to tell the synthesizer to say each digit separately, as follows:
	
	`Please call me at [[nmbr LTRL]] 5551990 [[nmbr NORM]].`
		`soNumberMode`

Command	Syntax and description	Selector
`pbas`	`pbas [+ | −] <RealValue>` The baseline pitch command changes the current speech pitch for the speech channel to the specified real value. If the pitch value is preceded by the + or - character, the speech pitch is adjusted relative to its current value. Baseline pitch values are always positive numbers in the range of 1.000 to 127.000.	`soPitchBase`
`pmod`	`pmod [+ | −] <RealValue>` The pitch modulation command changes the modulation range for the speech channel, based on the specified modulation-depth real value.	`soPitchMode`
`rate`	`rate [+ | −] <RealValue>` The speech rate command sets the speech rate on the speech channel to the specified real value. Speech rates fall in the range 0.000 to 65535.999, which translates into a range of 50 to 500 words per minute. If the rate is preceded by a + or - character, the speech rate is increased or decreased relative to its current value.	`soRate`
`rset`	`rset <32BitValue>` The reset command resets the speech channel's voice and attributes to default values. The parameter has no effect; it should be set to `0`.	`soReset`
`slnc`	`slnc <32BitValue>` The silence command causes the synthesizer to generate silence for the specified number of milliseconds. You might want to insert extra silence between two sentences to allow listeners to fully absorb the meaning of the first one. Note that the precise timing of the silence will vary among synthesizers.	none

Command	Syntax and description	Selector

`sync`	`sync <32BitValue>`
	The synchronization command causes an application's synchronization callback procedure to be executed. The callback is made as the audio corresponding to the next word begins to sound. The 32-bit value is set by the application and is passed to the callback procedure.
	You can use the `sync` command to trigger a callback at times other than those defined by the built-in callbacks (such as the phoneme and speech-done callbacks). For example, you might want to perform some custom processing each time a date is spoken to highlight its place on a graphical timeline. To do this, you would define a synchronization callback procedure and refcon values, and insert a `sync` command after each date in the text, as follows:
	`In 1066 [[sync 0x000000A1]], William the Conqueror invaded England and by 1072 [[sync 0x000000A2]], the whole of England was conquered and united.`
		`soSyncCallback`

`vers`	`vers <32BitValue>` The format version command tells the speech synthesizer which embedded command format version will be used by all subsequent embedded speech commands.	none
`volm`	`volm [+ | -] <RealValue>` The speech volume command sets the speech volume on the current speech channel to the specified real value. If the volume value is preceded by a + or - character, the speech volume is increased or decreased relative to its current value.	`soVolume`
`xtnd`	`xtnd <OSType> [<Parameter> ...]` The synthesizer-specific `xtnd` command enables other synthesizer-specific commands to be embedded in the text. The first parameter (`OSType`) must be the creator ID of the synthesizer. The remaining optional parameters are synthesizer-specific.	`soSynthExtension`

Embedded Speech Command Errors

While embedded speech commands are being processed, errors might be detected and reported to your application. If you enable error callbacks using the `SetSpeechInfo` function with the `soErrorCallBack` selector, your error callback procedure will be executed once for every error that is detected (for more information on the error callback, see `SpeechErrorProcPtr`). If you don't enable error callbacks, you can still get information about these errors by calling the `GetSpeechInfo` function with the `soErrors` selector.

During processing of embedded speech commands, the following errors can be detected:

Result code	Value	Description
`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

Use Phoneme Modifiers to Adjust Pronunciation

As described in "Representations of Speech" (page 11), the Speech Synthesis framework allows you to represent some or all of the words in a string or buffer as phonemes. When you supply the phonemic representation of a word, you specify the precise combination of sounds you want the synthesizer to pronounce. In addition, you can add phoneme modifiers to increase or decrease the stress with which phonemes and words are pronounced.

Recall that phonemes are represented by combinations of uppercase or lowercase characters, such as OW for the long "o" sound in the English word "boat." (Other languages use different phonemes and phoneme symbols; this document focuses on the set of North American English phonemes the MacinTalk synthesizer recognizes.) The complete set of phonemes is listed in "Phonemes" (page 64).

Because a synthesizer has no reliable way to detect the difference between characters that represent phonemes and characters that represent words, you must state the appropriate mode. There are two ways you can do this:

- In your application, use the `soPhonemeMode` selector with the `SetSpeechInfo` function to tell the current speech channel to process subsequent text as phonemes. This is a mode change for the speech channel, which means that it will interpret *all* text as phonemes until you call the `SetSpeechInfo` function with the `soTextMode` selector, to return to the default, text-processing mode (or until the speech channel closes). For examples of how to use this function, see "Adjust Speech Channel Settings Using the Carbon Speech Synthesis API" (page 36).

- In your text buffer or string, precede the phonemic representation of a word or phrase with the `[[inpt PHON]]` embedded speech command and follow it with the `[[inpt TEXT]]` command. This causes the synthesizer to switch to the phoneme input-processing mode for the content after the `[[inpt PHON]]` command and switch back again when it encounters the `[[inpt TEXT]]` command.

Within the phonemic representation of a word or phrase, you can insert modifiers that allow you to adjust the stress the synthesizer places on words and syllables. These modifiers are called prosodic controls.

Unlike embedded speech commands, prosodic controls do not require command delimiter strings and they do not allow parameters. Because prosodic controls are valid only within the phonemic representation of text, the symbols that represent them consist of characters that are not used to represent phonemes. To use prosodic control symbols in the phonemic representation of your text, place the appropriate symbol before the phoneme you want to modify. The effect of the prosodic control symbol is limited to the phoneme that immediately follows it; it has no effect on any subsequent phonemes.

Table 3-2 lists the available prosodic control symbols and describes how they modify individual phonemes. If you'd like to listen to the spoken version of any of the examples in Table 3-2, you can copy it to a Text Edit document, precede it with the `[[inpt PHON]]` command, and select Speech > Start Speaking Text from the Services menu item.

Table 3-2 Prosodic control symbols and descriptions

Category	Action	Symbol	Description and example
Lexical stress	Primary stress	1	Marks the primary stress within a word For example, the word "developer" is pronounced with the primary stress on the second syllable, as shown below: `dIHv1EHlAXpAXr`
	Secondary stress	2	Marks the secondary stress within a word For example, the word "application" is pronounced with the primary stress on the third syllable and a secondary stress on the first syllable, as shown below: `2AEplIHk1EYSIXn`

Category	Action	Symbol	Description and example
Syllable breaks	Syllable mark	= (equal)	Marks syllable breaks within a word For example, the word "cheaply" is pronounced with a subtle syllable break between "cheap" and "ly." To ensure that a synthesizer pronounces this word correctly (and not with a syllable break between "chea" and "ply"), you can insert a syllable mark, as shown below: `C1IYp=lIY`
Word prominence	Destressed	~ (tilde)	Marks words that should be destressed in a sentence Words that carry minimal information can be destressed to lessen their prominence in a sentence. For example, in the sentence "What is in the bag?," the words "in" and "the" are unimportant, relative to "What," "is," and "bag." Therefore, "in" and "the" can be marked as not needing stress, as shown below: `_w1UXt _1IHz ~2IHn ~nAX _b1AEg?`
	Normal stress	_ (underscore)	Marks words that should receive normal stress Words that bear information should be spoken with normal stress to differentiate them from less important words. For example, in the sentence "What is in the bag?," the words "What," "is," and "bag" should be spoken with normal stress because they convey more information to the listener than the words "in" and "the." Therefore, these information-bearing words can be marked as needing normal stress, as shown below: `_w1UXt _1IHz ~2IHn ~nAX _b1AEg?`
	Emphatic stress	+ (plus)	Marks words that require special emphasis The most important words in a sentence should receive emphatic stress to make them stand out from the rest of the sentence. For example, in the sentence "Don't ever do that again!," the word "that" can be given extra emphasis to draw attention to it, as shown below: `~dOWnt ~1EHvAXr ~d1UW +DAEt _AXg1EHn!`

Note: As with other embedded commands, the exact nature of the effects you can achieve with prosodic control symbols is dependent on synthesizer capabilities.

Use Punctuation Correctly

Punctuation marks are not embedded commands, but they appear in text and can affect the prosody of synthesized speech in some similar ways. This section describes how English-language synthesizers are likely to interpret punctuation marks.

For the most part, punctuation marks affect the pitch of synthesized speech and the duration of pauses. For example, the period at the end of a sentence generally causes a synthesizer to lower the pitch and insert a pause. Most speech synthesizers strive to mimic the pauses and changes in pitch of human speakers in response to punctuation marks, so you'll obtain the best results by punctuating your text according to standard grammatical rules.

Table 3-3 lists the standard English punctuation marks and how they affect sentence prosody. Be aware that some languages do not use some of these punctuation marks, so synthesizers for other languages might not interpret them as described in Table 3-3, if at all.

Table 3-3 Effects of punctuation marks on synthesized speech

Symbol	Effect on pitch	Effect on timing
,	Rise in pitch	Short pause follows
(	Start range of reduced pitch	Short pause follows
)	End range of reduced pitch	Short pause follows
.	Fall in pitch	Pause follows
"	Expand pitch range	A short pause precedes an opening quote and follows a closing quote

Even among English-language synthesizers, the specific pitch contours associated with the punctuation marks listed in Table 3-3 might vary according to other considerations arising from analysis of the text. For example, if a synthesizer determines that a question is rhetorical, the pitch might fall at the question mark, instead of rise. Also, the timing effects associated with the punctuation marks can vary according to current speech rate settings. Consequently, you should view the information in Table 3-3 as guidance only; test your application's spoken output with a particular synthesizer to find out how the punctuation is actually interpreted.

Use the TUNE Format to Supply Complex Pitch Contours

In addition to supporting the phoneme input-processing mode, the MacinTalk synthesizer available in OS X v10.2 and later supports the TUNE input-processing mode. This mode accepts directives in the TUNE format, which allows you to supply a complex pitch contour, or tune, with which a word or phrase should be spoken. Such a tune can represent the pitch and speech-rate changes you hear when a person speaks in an expressive way. For example, adults speaking to small children often vary the pitch of their speech much more than they do when speaking to other adults. As described in "Use Phoneme Modifiers to Adjust Pronunciation" (page 49), phoneme modifiers can be used to adjust the stress placed on particular phonemes, but you cannot use them to cause multiple variations in pitch during the pronunciation of a single phoneme. To do this, you must use the TUNE format.

Apple provides the Repeat After Me developer tool to help you create the set of symbols that describe a tune. Using the Repeat After Me application (located in `/Developer/Applications/Utilities/Speech`), you can record an utterance that exhibits your desired pitch contour and use that to shape any other utterance in your application.

Similar to the way you enter and exit the phoneme input-processing mode, you use the `inpt` embedded command to turn on and off the TUNE input-processing mode. Specifically, you insert `[[inpt TUNE]]` before the content in the TUNE format and insert `[[inpt TEXT]]` after it. The TUNE format recognizes the same set of phoneme symbols used in the phoneme input-processing mode (see "Phonemes" (page 64) for a description of these symbols).

The TUNE format defines a command syntax you use to specify pitch and duration attributes for each phoneme. Each phoneme may be followed by a pair of braces, enclosing a single duration attribute, preceded by the symbol "D," and an arbitrary number of pitch attributes, preceded by the symbol "P." The duration attribute indicates the total duration of the phoneme in milliseconds. Each pitch attribute consists of a pair of numbers separated by a colon. The first number is decimal value that specifies a pitch in hertz (Hz) and the second number is an integer that specifies the location of that pitch within the phoneme, expressed as an integer percentage of the total duration of the phoneme.

To illustrate the syntax of the TUNE format, consider the sentence "Are you sure you brushed your teeth?" The default pronunciation of this sentence is perfectly understandable, but the intonation is uninteresting. (If you're reading this document in Safari, Preview, or Xcode, select "Are you sure you brushed your teeth?" and choose Speech > Start Speaking Text from the Services menu item to hear the default pronunciation.) Imagine that you want this sentence to be spoken as a parent might speak it to a child, with emphasis on "sure" and an exaggerated rise in pitch through the end of the sentence. Using the Repeat After Me application, you can record a person speaking the sentence in this way, apply the resulting pitch and duration information to the text, and get the representation in the TUNE format. Following this process, you might end up with something similar to the following:

```
[[inpt TUNE]]
~
AA {D 120; P 176.9:0 171.4:22 161.7:61}
r {D 60; P 166.7:0}
~
y {D 210; P 161.0:0}
UW {D 70; P 178.5:0}
_
S {D 290; P 173.3:0 178.2:8 184.9:19 222.9:81}
1AX {D 280; P 234.5:0 246.1:39}
r {D 170; P 264.2:0}
~
y {D 200; P 276.9:0 274.9:17 271.0:50}
UW {D 40; P 265.0:0 264.3:50}
_
b {D 140; P 263.6:0 263.5:13 263.3:60}
r {D 110; P 263.1:0 260.4:43}
1UX {D 30; P 256.8:0 256.8:6}
S {D 190; P 256.1:0}
t {D 20; P 252.0:0 253.6:47}
~
y {D 30; P 255.5:0 257.8:45}
AO {D 40; P 260.6:0 260.0:56}
r {D 40; P 259.5:0}
_
t {D 190; P 251.3:0 250.0:16 245.9:68}
1IY {D 260; P 243.4:0 248.1:8 286.1:72 288.5:84}
T {D 220; P 291.6:0 262.8:27 220.0:67 184.8:100}
? {D 300}
[[inpt TEXT]]
```

To listen to this version of the sentence, select the lines above (be sure to include the "`[[inpt TUNE]]`" at the beginning and the "`[[inpt TEXT]]`" at the end), copy them, and paste them into a Text Edit document. Make sure all the lines are still selected and then select Speech > Start Speaking Text from the Services menu item in the Text Edit menu.

The TUNE format also includes optional settings that describe the beginning value and range of the pitch, expressed in hertz, and the speech rate, expressed in words per minute. You can use these settings to state the pitch and rate conditions that were in effect when you created the tune. If either of these settings have nonzero values, the synthesizer will scale the pitch and duration attribute values you supply for the phonemes according to voice conditions in effect during synthesis. This is analogous to transposing a song to a different key and playing it at a different tempo. If both of these settings are missing, the synthesizer interprets the pitch and duration attribute values as literal values that should be reproduced exactly, which is analogous to playing a song in the key and time signature in which it was composed.

Synchronize Speech with Application-Specific Actions

As mentioned in "Notifications, Callbacks, and Speech Synchronization" (page 14), you can synchronize your application's spoken output with other tasks in your application. Both the Cocoa and the Carbon speech synthesis APIs provide mechanisms you can use to get notifications when, for example, a word or phoneme is about to be spoken or has just been spoken. This section describes how you can receive these notifications and some of the ways you might use them.

The `NSSpeechSynthesizer` class defines a few delegate methods you can use to synchronize tasks with speech-related actions. For example, you can implement the `speechSynthesizer:willSpeakWord:ofString` delegate method to highlight a word as it's being spoken. For an example of an implementation that does this, see Listing 2-2 (page 29). The `NSSpeechSynthesizer` class also defines the `speechSynthesizer:willSpeakPhoneme` and `speechSynthesizer:didFinishSpeaking` delegate methods you can use to find out when a phoneme is about to be spoken and when an `NSSpeechSynthesizer` object has finished speaking, respectively. Listing 2-2 (page 29) includes an implementation of the `speechSynthesizer:didFinishSpeaking` delegate method that resets the cursor to the beginning of the line of text and re-enables buttons in the application window.

The Carbon speech synthesis API defines several callback function types you can use to create and install callback functions in a speech channel. For each event to which you want to respond, you create a callback function that adheres to the prototype defined by the callback pointer (see *Speech Synthesis Manager Reference* for these prototypes). Then, you install each callback function in a speech channel by passing the appropriate selector to the `SetSpeechInfo` function, as shown below:

```
// Install MyWordCallback callback function in the current speech channel
error = SetSpeechInfo(currentSpeechChannel, soWordCallBack, MyWordCallback);
```

When the Speech Synthesis Manager encounters one of the events handled by these callbacks, it calls the callback function you've installed, allowing you to synchronize custom processing with that speech event. The six callback function types defined in the Carbon speech synthesis API are listed below, each accompanied by the selector you use to install the callback function:

Callback	Selector
`SpeechWordProcPtr`	`soWordCallBack`
`SpeechPhonemeProcPtr`	`soPhonemeCallBack`
`SpeechDoneProcPtr`	`soSpeechDoneCallBack`
`SpeechErrorProcPtr`	`soErrorCallBack`

Callback	Selector
`SpeechSyncProcPtr`	`soSyncCallBack`
`SpeechTextDoneProcPtr`	`soTextDoneCallBack`

The `SpeechWordProcPtr`, `SpeechPhonemeProcPtr`, and `SpeechDoneProcPtr` callbacks are triggered by the same events as the `NSSpeechSynthesizer` delegate methods. Therefore, you can use these to perform custom processing when a word or phoneme is about to be spoken and when speaking has stopped. See Listing 2-4 (page 33) for an example usage of the `SpeechWordProcPtr` callback.

The Carbon speech synthesis API uses the `SpeechErrorProcPtr` pointer to call a speech channel's error callback function when it encounters syntax errors in a text buffer's embedded commands (see "Embedded Speech Command Errors" (page 49) for a list of possible errors). In addition to helping you find such errors during application development, this callback allows you to display an alert or perform some other action if there are errors in the embedded commands users supply.

The `SpeechSyncProcPtr` defines a callback function you can implement to synchronize application-specific actions with the presence of the `sync` embedded speech command. When the Speech Synthesis Manager encounters a `sync` command in a string or buffer of text, it calls the callback function you've installed in the speech channel. You can use the parameter of the `sync` command to provide an arbitrary value that gets passed to your callback function, allowing you to distinguish among different usages of the command. Although you can use the `sync` command to trigger a callback when a word or phoneme is about to be spoken, it's best to use the provided callback mechanisms for these events, reserving the `sync` command for application-defined events.

The `SpeechTextDoneProcPtr` defines a callback function that gets called when the Speech Synthesis Manager finishes processing a buffer of text. This can happen before the synthesizer finishes speaking the text or before the synthesizer even starts speaking the text. You might supply a callback function for this event if you want to be able to dispose of the original text buffer as soon as the Speech Synthesis Manager finishes copying it.

Avoid Cross-Talk

Just as it's confusing to listen to more than one person talking at the same time, it's confusing for users to hear more than one application speaking at the same time. With the popularity of VoiceOver and an increasing number of applications capable of producing speech, the potential for overlapping or interrupted speech is significant. This section explains how VoiceOver implements speech arbitration and describes ways you can avoid interrupting the spoken output of other applications and processes.

While VoiceOver is running, there is an automatic arbitration mechanism in place that causes all other spoken output to stop when VoiceOver starts to speak. Because VoiceOver provides the accessibility interface to OS X and visually impaired users rely on it to navigate and control the system, it is appropriate to give it priority over other types of spoken output.

While VoiceOver is not running, however, there is no arbitration mechanism in place. For this reason, it's a good idea for your application to ascertain if another application or process is currently speaking before beginning to speak. Both the Carbon and Cocoa speech synthesis APIs provide a way to do this.

If you're using the Cocoa `NSSpeechSynthesizer` class to produce spoken output, you can invoke the `isAnyApplicationSpeaking` class method to find out if another application or a system component (such as VoiceOver) is currently producing speech. This method returns a Boolean value your application can use to decide when it's appropriate to speak. Depending on the needs of your application, you might use this method in the following way:

```
if ([NSSpeechSynthesizer isAnyApplicationSpeaking]) {
    // Wait.
} else {
    [_mySpeechSynthesizer startSpeakingString:myTextToSpeak];
}
```

If you're using the Carbon speech synthesis API, you use a combination of two functions to determine whether any other application or system component is currently speaking. First, use the `SpeechBusySystemWide` function to get the total number of speech channels (including paused speech channels) that are currently synthesizing speech on the computer. This includes the speech channels the Speech Synthesis Manager automatically creates in response to the `SpeakString` function and all speech channels your application is using. To find out if there are other applications or processes currently producing speech, therefore, you must subtract the speech channels your application is using from the number of speech channels you get from `SpeechBusySystemWide`. To get the total number of speech channels associated with your application, use the `SpeechBusy` function, as shown below:

```
short totalChannels, myTotalChannels;
totalChannels = SpeechBusySystemWide();
myTotalChannels = SpeechBusy();
if ((totalChannels - myTotalChannels) > 0) {
    // Wait.
} else {
    SpeakText(mySpeechChannel, myTextToSpeak, strlen(myTextToSpeak));
}
```

Four Ways to Improve Spoken Output

A synthesizer follows a predetermined set of rules about language production when it converts text to spoken output. But no matter how sophisticated and extensive those rules are, there will always be situations they don't cover. As the developer, you know a lot more about how your application's speech should sound than any synthesizer does, so you should take advantage of the available customization opportunities to produce the best possible spoken output.

If you're viewing this document in Safari, Preview, or Xcode, you can listen to any example in this section by selecting it and then choosing Speech > Start Speaking Text from the Services menu item in the Application menu. If you'd like to experiment with the samples, one way to do this is to type or copy and paste them into a Text Edit window. After you've made adjustments and you want to listen to the result, select it and choose Speech > Start Speaking Text from the Services menu item in the Text Edit menu.

Adjust the Pronunciation of Troublesome Words

As described in "Opportunities for the Customization of Synthesized Speech" (page 18), you can use embedded commands to adjust the pronunciation of words a synthesizer is likely to mispronounce, such as proper nouns. Another category of words a synthesizer may have difficulty with is words that are spelled the same but pronounced differently depending on semantic context. A common developer reaction to either of these situations is to deliberately misspell the word in an attempt to trick the synthesizer into pronouncing it correctly. Although this approach might work with a particular version of a synthesizer, it is ultimately unreliable. This is because future enhancements to a synthesizer can result in a more accurate pronunciation of the original word and an even worse pronunciation of the misspelled version. A much better approach is to represent the word phonemically and apply the appropriate prosodic controls.

Although you can select individual phonemes and create the phonemic representation of a word "by hand," it's usually more efficient to start with a synthesizer's default phonemic representation and adjust it as necessary. This is because a synthesizer often mispronounces only one or two phonemes in a word, which means the remaining phonemes are accurate.

For example, the default pronunciation of the name "Matthias" places the stress on the first syllable and pronounces the first "a" the same as the "a" in the English word "father." (The phonemic representation of this pronunciation is `m1AAtIYIXs`.) To hear the default pronunciation, listen to the spoken version of the following sentence:

```
My name is Matthias.
```

If you wanted to change the pronunciation so that the stress is on the second syllable and the first "a" sounds like the "a" in "about," you would change the phonemic representation of the name to `mAXt1IYIXs`. To hear how this changes the synthesizer's pronunciation, listen to the spoken version of the following sentence:

```
My name is [[inpt PHON]]mAXt1IYIXs[[inpt TEXT]].
```

Let the User Catch Up

Listening to speech is a mentally intensive process, whether the speech is produced by another person or generated by a synthesizer. For this reason, most human speakers naturally insert pauses into their speech to allow listeners enough time to absorb the content. Including pauses in the spoken output of an application is especially important, because the computer can't adjust its delivery in response to verbal or nonverbal feedback from the listener.

Adding pauses to synthesized speech is primarily a matter of inserting units of silence at specific places in the text. You can do this in any of the following ways:

- Use appropriate punctuation within sentences. The correct use of commas, colons, and semicolons is as important for listeners as it is for readers.

  Listen to both versions of the sentence below:

  ```
  Today I feel well yesterday I felt terrible.
  ```

  ```
  Today I feel well; yesterday I felt terrible.
  ```

  The second version conveys the juxtaposition of the two states of the speaker's condition much more clearly than the first version.

- Use short, declarative sentences when possible. Although complex sentences can be acceptable in text, they can be difficult to understand when spoken. The synthesizer automatically adds a noticeable pause between sentences, which helps users assimilate the information in one sentence before turning their attention to the next sentence. For this reason, an idea expressed in a couple of short sentences will include more silence than the same idea expressed in a single, long sentence.

  Listen to the following long sentence:

  ```
  After you insert a section break, you can use the layout tool (located in the
  Tools menu) to format the new section, which can have different margins and
  numbers of columns than other sections in the document.
  ```

  Although the synthesizer pauses briefly at the commas and the parentheses, the pauses that accompany the periods in the 3-sentence version of this information make it easier to absorb:

  ```
  After you insert a section break, you can use the layout tool to format the new
  section. The layout tool is located in the Tools menu. Each section can have
  different margins and numbers of columns than other sections in the document.
  ```

- Use the `slnc` (silence) embedded speech command. You can add an arbitrary amount of silence anywhere in the text by inserting the `[[slnc x]]` command (where `x` is a number of milliseconds).

  For example, inserting extra silence between the items in a list makes it easier for people to take note of each item. Listen to the following sentence, which lists four items, separated by commas:

```
Don't forget to bring your hat, sunglasses, sandals, and towel.
```

Now listen to the same sentence, with 400 milliseconds of silence inserted between the listed items, and notice that you hear each item more distinctly:

```
Don't forget to bring your hat, [[slnc 400]] sunglasses, [[slnc 400]] sandals,
[[slnc 400]] and towel.
```

Focus the User's Attention

Listen closely to people speaking and you'll notice that they tend to emphasize the words in a sentence that carry new and important information and deemphasize less important and repetitive words. These differences in emphasis make it easier for listeners to recognize the important ideas in a sentence. Adding appropriate emphasis (or deemphasis) to words in your application's speech can make the spoken output much easier for listeners to understand.

The following three sentences all follow the same pattern, but each provides different information. Without adjustments in emphasis, the sentences are very similar and it's hard to focus on the differences in the times and the names.

```
On May tenth, you have a meeting in Cupertino. On June tenth, you have a meeting
in Tokyo. On July tenth, you have a meeting in Paris.
```

Now listen to these three sentences with embedded commands that emphasize the important words and deemphasize the less-important, repetitive words:

```
On May tenth, you have a meeting in Cupertino. On [[emph +]] June [[emph -]] tenth,
you [[emph -]] have a [[emph -]] meeting in [[emph +]] Tokyo. On [[emph +]] July
[[emph -]] tenth, you [[emph -]] have a [[emph -]] meeting in [[emph +]] Paris.
```

Liven It Up!

People naturally express emotion in their speech to add other layers of meaning and to keep listeners engaged. Adding the illusion of emotion to synthesized speech is not as easy as inserting pauses and fine-tuning pronunciations, but you can achieve satisfactory results by carefully adjusting the pitch and timing of your spoken output.

For example, when people are sad or depressed, their speech is usually slower, more monotone, and often quieter than normal. Conversely, when people are happy or excited, their speech generally exhibits greater range in pitch and is often faster and louder than normal. You can use the TUNE format to approximate these qualities to give the impression of emotion to the speech your application generates.

For example, the default pronunciation of the sentence “Sorry, Dave, I can’t do that right now.“ is emotionally bland. To give listeners the impression that the speaker is perhaps a bit regretful, but nonetheless implacable, you might use the TUNE format to create the following utterance:

```
[[inpt TUNE]]
~
s {D 250; P 212.0:0 212.0:35 212.0:54 212.0:85 212.0:96}
1AA {D 190; P 232.0:0 218.0:35 222.0:80}
r {D 80; P 216.0:0}
IY {D 150; P 177.0:0 162.0:29 162.0:68 162.0:77 162.0:90 162.0:100}
, {D 20}
~
d {D 60; P 162.0:0 162.0:36 162.0:57 160.0:93}
1EY {D 350; P 162.0:0 150.0:27 150.0:41 150.0:70}
v {D 30; P 150.0:0 150.0:29 150.0:52 150.0:67 150.0:90 150.0:100}
, {D 510}
~
2AY {D 140; P 173.0:0 196.0:45}
~
k {D 100; P 196.0:0 196.0:95}
AE {D 180; P 198.0:0 232.0:56}
n {D 80; P 232.0:0}
t {D 20; P 232.0:0 232.0:38}
~
d {D 40; P 232.0:0 232.0:85 208.0:92}
1UW {D 180; P 210.0:0 232.0:32 253.0:60 245.0:76}
~
D {D 60; P 245.0:0 186.0:92}
AE {D 240; P 186.0:0 168.0:37}
t {D 30; P 155.0:0 155.0:60 155.0:93}
~
r {D 70; P 155.0:0 149.0:53}
1AY {D 180; P 157.0:0 137.0:61}
t {D 40; P 128.0:0 132.2:56 135.0:94}
~
n {D 80; P 129.0:0 153.0:31 147.0:94}
1AW {D 340; P 147.0:0 140.8:22 169.2:88 148.0:100}
. {D 780}
[[inpt TEXT]]
```

Syntax of Embedded Speech Commands

This appendix provides a formalization of the embedded command syntax structure, subject to the following conventions:

- Items enclosed in angle brackets (< and >) represent logical units that are listed and defined in another row in the table.
- Items enclosed in brackets ([and]) are optional.
- Items followed by an ellipsis (...) may be repeated one or more times.
- When two or more items are separated by a vertical bar (|), any one of the listed items may be used.

Table A-1 defines the identifiers used in embedded commands.

Table A-1 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>* | *<RealValue>* | *<32BitValue>* | *<16BitValue>* | *<8BitValue>*
String1	*<Character>*
String2	*<Character> <Character>*
StringN	[*<Character>* ...]
OSType	*<Character> <Character> <Character> <Character>*

Identifier	Syntax
32BitValue	<*OSType*> | <*LongInt*> | <*HexLongInt*>
16BitValue	<*Integer*> | <*HexInteger*>
8BitValue	<*Byte*> | <*HexByte*>
RealValue	<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, *, ~, \\)>
Byte	<Decimal number: 0 ≤ N ≤ 255>
HexByte	<Hex number: 0x00 ≤ N ≤ 0xFF>

Phonemes

This appendix lists the phoneme symbols for North American English that the MacinTalk synthesizer recognizes. Other languages and dialects use different phoneme collections. Whether you use a speech synthesis function to get the default phonemic representation of a word or you want to create it yourself, you need to know the symbols that represent individual phonemes. In addition, if you are developing a custom synthesizer that produces North American English speech, you must make sure it can interpret the phonemes listed in this appendix.

Phonemes divide into two groups: vowels and consonants. All phoneme symbols that represent vowels are pairs of uppercase letters. Consonants are represented by a single letter.

In Table B-1, a phoneme symbol is followed by a word in which a letter or combination of letters is displayed in bold font. The letter or letter combination in bold exemplifies the sound the phoneme symbol represents. The two exceptions to this are the first two symbols, which stand for silence and a breath intake.

Table B-1 North American English phoneme symbols

Phoneme symbol	Example of pronunciation
%	(*silence*)
@	(*breath intake*)
AE	b**a**t
EY	b**ai**t
AO	c**au**ght
AX	**a**bout
IY	b**ee**t
EH	b**e**t
IH	b**i**t
AY	b**i**te
IX	ros**e**s

Phoneme symbol	Example of pronunciation
AA	f**a**ther
UW	b**oo**t
UH	b**oo**k
UX	b**u**d
OW	b**oa**t
AW	b**ou**t
OY	b**oy**
b	**b**in
C	**ch**in
d	**d**in
D	**th**em
f	**f**in
g	**g**ain
h	**h**at
J	**j**ump
k	**k**in
l	**l**imb
m	**m**at
n	**n**ap
N	ta**ng**
p	**p**in
r	**r**an
s	**s**in
S	**sh**in

Phoneme symbol	Example of pronunciation
t	**t**in
T	**th**in
v	**v**an
w	**w**et
y	**y**et
z	**z**oo
Z	mea**s**ure

Glossary

embedded speech command A command embedded in textual input that the synthesizer interprets and applies to the pronunciation of the spoken version.

frequency The precise indication of the number of hertz of a sound wave at any instant.

MacinTalk synthesizer The built-in speech synthesizer available in OS X v10.0 and later. (Earlier versions of the MacinTalk synthesizer were available in earlier versions of Macintosh system software.)

phoneme A distinct unit that serves to distinguish between meanings of words.

phoneme modifier A symbol defined to adjust the pronunciation of an individual phoneme. Phoneme modifiers are also called prosodic control symbols.

pitch modulation The maximum amount by which the actual frequency of speech may deviate from the speech pitch.

prosodic control symbol See phoneme modifier.

prosody The rhythm, intonation, and lexical stress in speech.

speech attribute A setting defined on a speech channel that affects the characteristics of the spoken output for a subset of voices or for all voices associated with a particular synthesizer.

speech channel A structure through which an application communicates with a specific speech synthesizer and voice. An application may have more than one speech channel open at one time, but a speech channel may not be associated with more than one synthesizer and voice at one time.

Speech Manager Obsolete term for the Speech Synthesis Manager. The Speech Manager provides a C-based API that supports extensive control over speech synthesis.

speech pitch The middle pitch of a voice, from which the actual pitches of the speech can vary with rising and falling tunes. Pitch is a combination of the average speaking frequency and its variations around that average.

speech rate The approximate number of words of text that the synthesizer speaks in one minute.

speech synthesizer A component that converts text into speech. A speech synthesizer usually contains executable code, built-in dictionaries, and pronunciation rules that help it determine how to pronounce text. A speech synthesizer may also be called a speech engine.

speech volume The average amplitude at which the speech channel generates speech.

voice A component containing data and, optionally, executable code that helps to shape the sound of the synthesized speech.

Document Revision History

This table describes the changes to *Speech Synthesis Programming Guide* .

Date	Notes
2006-09-05	New document that describes OS X speech synthesis support and explains how to produce customized spoken output in your application.