INSIDE MACINTOSH

Interapplication Communication

Contents

Chapter 3   Introduction to Apple Events   3-1

Chapter 5   Creating and Sending Apple Events   5-1

Chapter 6   Resolving and Creating Object Specifier Records   6-1

Chapter 7   Introduction to Scripting   7-1

Chapter 11	Program-to-Program Communications Toolbox 11-1

Chapter 12   Data Access Manager   12-1

Glossary

Index

Figures, Tables, and Listings

Chapter 3   **Introduction to Apple Events**   3-1

xviii

Chapter 7   Introduction to Scripting   7-1

Chapter 8   Apple Event Terminology Resources   8-1

About This Book

This book, *Inside Macintosh: Interapplication Communication*, describes the interapplication communication architecture, which provides a standard and extensible mechanism for communication among Macintosh applications. This book also describes the system software routines that you can use to implement various forms of interapplication communication in your application.

If you are new to programming on the Macintosh computer, you should read *Inside Macintosh: Overview* for an introduction to general concepts of Macintosh programming; *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to use menus, windows, and controls in your application; and *Macintosh Human Interface Guidelines* for a complete discussion of user interface guidelines and principles that every Macintosh application should follow.

This book describes how to implement publish and subscribe features in your application, how to communicate with other applications using Apple events, how to respond to scripts, and how to exchange information with other applications using the PPC Toolbox. It also discusses how your application can use the Data Access Manager to access information from a database application or other data source.

For an overview of all the features provided by the interapplication communication architecture, see the chapter "Introduction to Interapplication Communication" in this book.

To provide support for publish and subscribe features in your application, see the chapter "Edition Manager" in this book. This chapter describes how your application can allow users to share dynamic data among many documents.

To communicate with other applications by using Apple events, first see the chapter "Introduction to Apple Events" for a general introduction to Apple events. For information on how to respond to the required Apple events, see the chapter "Responding to Apple Events." To create and send Apple events, see the chapter "Creating and Sending Apple Events."

You can choose to write your application so that it can recognize descriptions, in Apple events, of objects in the application such as words, paragraphs, shapes, or documents. To do so, see the chapter "Resolving and Creating Object Specifier Records."

In addition to supporting Apple events, you can make your application scriptable—that is, capable of responding to Apple events sent to it by a scripting component. By executing scripts, users of scriptable applications can automate repetitive tasks or conditional tasks that involve multiple

applications. For more general information about scripting, see the chapter "Introduction to Scripting." See the chapter "Apple Event Terminology Resources" for information on the resources your application needs to provide in order to be scriptable.

You can also make your application recordable, that is, capable of recording a user's actions for later playback. For more information, see the chapter "Recording Apple Events."

For information on how your application can execute a script with the aid of a scripting component, see the chapter "Scripting Components."

Although you'll usually want to use Apple events to communicate with other applications, if you need low-level control of communication between applications you can use the Program-to-Program Communications (PPC) Toolbox. For more information, see the chapter "Program-to-Program Communications Toolbox."

Applications can use the Data Access Manager to access information from a database application or other data source. For example, a user in San Francisco might use a spreadsheet application to request data from a company database in New York. The spreadsheet application can use the Data Access Manager to request the data from the database. The database application in New York sends back the requested data, and the spreadsheet application can then use this data to generate a graph of the information. For information on sending and retrieving information from a data source, see the chapter "Data Access Manager."

For definitions of specific Apple events and Apple event objects, see the *Apple Event Registry: Standard Suites*, available from APDA.

For information on handling files in your application and a description of aliases and alias records, see *Inside Macintosh: Files*.

For information on processes and process serial numbers, see *Inside Macintosh: Processes*.

Format of a Typical Chapter

Almost all chapters in this book follow a standard structure. For example, the chapter "Creating and Sending Apple Events" contains these sections:

n "Creating an Apple Event" and "Sending an Apple Event." These sections describe how your application can create and send Apple events. They describe the Apple Event Manager routines that you can use to accomplish these tasks, give related user interface information, and provide code samples and additional information.

n "Reference to Creating and Sending Apple Events." This section provides a complete reference to the Apple Event Manager routines you can use to create and send Apple events. Each routine description also follows a

standard format, which presents the routine declaration followed by a description of every parameter of the routine. Some routine descriptions also give additional descriptive information, such as assembly-language information or result codes.

n   "Summary of Creating and Sending Apple Events." This section provides the Pascal and C interfaces for the constants, data structures, routines, and result codes associated with the Apple Event Manager routines for creating and sending Apple events. It also includes relevant assembly-language interface information.

Conventions Used in This Book

Inside Macintosh uses various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain information, such as the contents of registers, use special formats so that you can scan them quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, fields, constants, parameters, and routines are shown in Courier (`this is Courier`).

Words that appear in **boldface** are key terms or concepts and are defined in the Glossary.

Types of Notes

There are several types of notes used in this book.

Note
A note like this contains information that is interesting but possibly not essential to an understanding of the main text. (An example appears on page 3-26.) u

IMPORTANT
A note like this contains information that is essential for an understanding of the main text. (An example appears on page 3-33.) s

s   **W A R N I N G**
Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. (An example appears on page page 4-10.) s

Assembly-Language Information

Some chapters provide additional assembly-language information. For example, *Inside Macintosh* provides information about the registers for specific routines like this:

Registers on entry

A0   Contents of register A0 on entry

Registers on exit

D0   Contents of register D0 on exit

In addition, *Inside Macintosh* provides information about the fields of a parameter block in this format:

```
inAndOut      Integer      Input/output parameter.
output1       Ptr          Output parameter.
input1        Ptr          Input parameter.
```

The arrow in the far left column indicates whether the field is an input parameter, output parameter, or both. You must supply values for all input parameters and input/output parameters. The routine returns values in output parameters and input/output parameters.

The second column shows the field name as defined in the MPW Pascal interface files; the third column indicates the Pascal data type of that field. The fourth column provides a brief description of the use of the field. For a complete description of each field, see the discussion that follows the parameter block or the description of the parameter block in the reference section of the chapter.

The Development Environment

The system software routines described in this book are available using Pascal, C, or assembly-language interfaces. How you access these routines depends on the development environment you are using. When showing system software routines, this book uses the Pascal interface available with the Macintosh Programmer's Workshop (MPW).

All code listings in this book are shown in Pascal (except for listings that describe resources, which are shown in Rez-input format). They show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and, in many cases, tested. However, Apple Computer, Inc., does not intend for you to use these code samples in your application. You can find the location of code listings in the list of figures, tables, and listings. If you know the name of a particular

routine (such as `MyHandleODoc` or `MyHandleQuit`) shown in a code listing, you can find the page on which the routine occurs by looking under the entry "sample routines" in the index of this book.

In order to make the code listings in this book more readable, they show only limited error handling. You need to develop your own techniques for handling errors.

This book occasionally illustrates concepts by reference to sample applications called *SurfWriter*, *SurfDB*, and *SurfCharter;* these are not actual products of Apple Computer, Inc.

APDA is Apple's worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most of our products. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	800-282-2732 (United States)
	800-637-0029 (Canada)
	716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDA
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information on registering signatures, file types, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-3T
Cupertino, CA 95014-6299

The Apple Event Registrar maintains the *Apple Event Registry: Standard Suites* and other information about the ongoing development of Apple event suites. For more information about Apple event suites, including those under development, send electronic mail to the AppleLink address REGISTRY.

Introduction to Interapplication Communication

Contents

This chapter describes the interapplication communication (IAC) architecture for Macintosh computers, summarizes how your application can take advantage of it, and tells you where in this book to find the information you need to perform specific tasks.

The Apple Event Manager, Event Manager, and Program-to-Program Communications (PPC) Toolbox underlie all the IAC tasks your application can perform. This chapter introduces the Apple Event Manager and the Program-to-Program Communications Toolbox. For information about the Event Manager, see *Inside Macintosh: Macintosh Toolbox Essentials.* For definitions of the standard Apple events available for use by all applications, see the *Apple Event Registry: Standard Suites.*

The IAC architecture includes the Open Scripting Architecture (OSA). The OSA provides a mechanism that allows users to control multiple applications by means of scripts, or sets of instructions, written in a variety of scripting languages. Each scripting language has a corresponding scripting component that is managed by the Component Manager. When a user executes a script, the scripting component sends Apple events to one or more applications to perform the actions the script describes.

This chapter introduces the OSA and describes how to make your application scriptable, or capable of responding to Apple events sent to it by a scripting component. For more information about using the Component Manager, see *Inside Macintosh: More Macintosh Toolbox.*

Overview of Interapplication Communication

The **interapplication communication (IAC) architecture** provides a standard and extensible mechanism for communication among Macintosh applications. The IAC architecture makes it possible for your application to

n   provide automated copy and paste operations between your application and other applications

n   be manipulated by means of scripts

n   send and respond to Apple events

n   send and respond to high-level events other than Apple events

n   read and write blocks of data between applications

The chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* describes how your application can use Event Manager routines to send and respond to high-level events. High-level events need not adhere to any specific protocol, so their interpretation is defined by each application that sends or receives them.

The most important requirement for high-level communication among all applications is a common vocabulary of events. To provide such a standard, Apple Computer, Inc., has defined a protocol called the **Apple Event Interprocess Messaging Protocol (AEIMP).** High-level events that conform to this protocol are called **Apple events.**

The vocabulary of publicly available Apple events is published in the *Apple Event Registry: Standard Suites,* which defines the standard Apple events that developers and Apple have worked out for use by all applications. To ensure that your application can communicate at a high level with other applications that support Apple events now and in the future, you should support the standard Apple events that are appropriate for your application.

Effective IAC requires close cooperation among applications at several levels. In addition to the format for high-level events and the standard vocabulary of Apple events, Apple has defined several other standards your application can use to communicate with other applications. These include standard methods for dealing with shared dynamic data, scripts, and low-level message blocks.

The IAC architecture comprises the following parts:

n The **Edition Manager** allows applications to automate copy and paste operations between applications, so that data can be shared dynamically.

n The **Open Scripting Architecture (OSA)** provides a standard mechanism, based on the Apple Event Manager, that allows users to control multiple applications by means of scripts written in a variety of scripting languages.

n The **Apple Event Manager** allows applications to send and respond to Apple events.

n The **Event Manager** allows applications to send and respond to high-level events other than Apple events.

n The **Program-to-Program Communications (PPC) Toolbox** allows applications to exchange blocks of data with each other by reading and writing low-level message blocks. It also provides a standard user interface that allows a user working in one application to select another application with which to exchange data.

Figure 1-1 shows the primary relationships among these parts. The managers and components toward the top of the figure rely on the managers beneath them. The Edition Manager uses the services of the Apple Event Manager to support dynamic data sharing. Scripting components manipulate and execute scripts with the aid of the Apple Event Manager. The Apple Event Manager in turn relies on the Event Manager to send Apple events as high-level events, and the Event Manager uses the services of the PPC Toolbox.

Figure 1-1 also shows the five principal means of communication provided by the IAC architecture. In addition to using the Edition Manager and scripting components to send Apple events on their behalf, applications can use the Apple Event Manager directly to send Apple events to other applications. All applications can use the Apple Event Manager to respond appropriately to Apple events, whether they are sent by the Edition Manager, a scripting component, or other applications. Applications can also use the Event Manager directly to send or receive high-level events other than Apple events, and the PPC Toolbox directly to send or receive blocks of data.

Figure 1-1   Principal methods of communication between applications
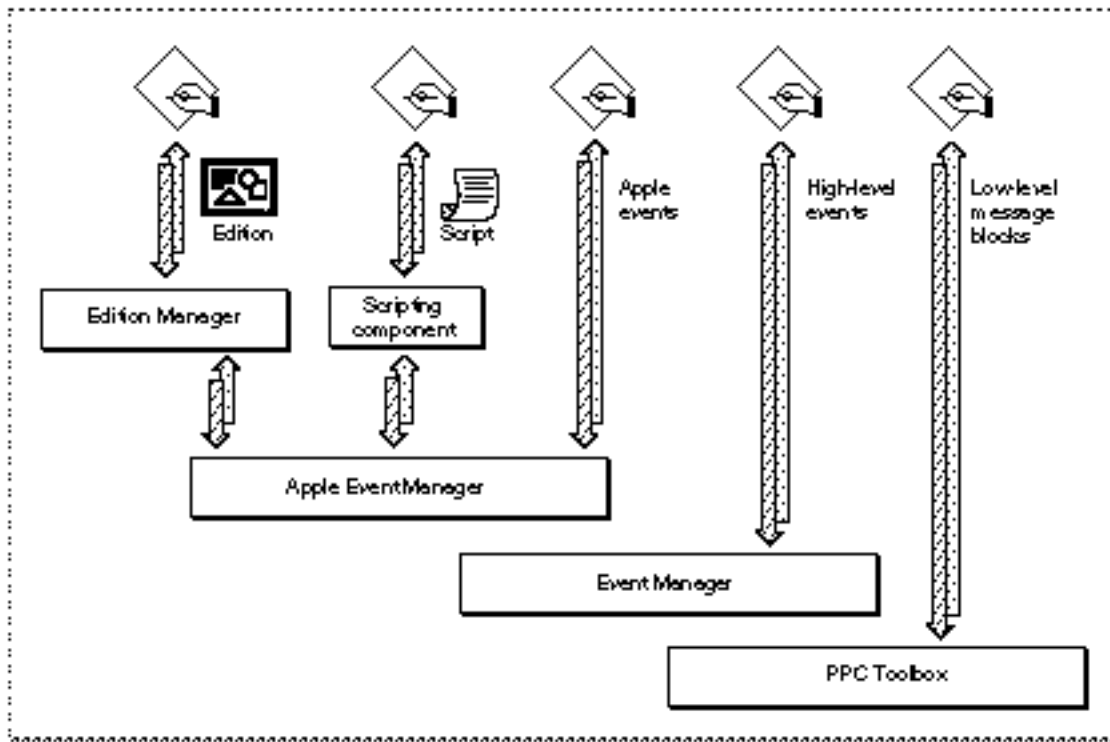
The five forms of IAC shown in Figure 1-1 can be summarized as follows:

n *Sharing dynamic data.* **The Edition Manager allows users to copy data from one application's document to another application's document, updating information automatically when the data in the original document changes. The verbs** *publish* **and** *subscribe* **describe this form of dynamic data sharing, and the noun** *edition* **describes a copy of the data to be shared. Applications that support dynamic data sharing must implement the Create Publisher and Subscribe To menu commands. The Edition Manager provides the interface that allows applications to share editions.**

   You can let users publish and subscribe on a local volume or across a network. In general, users should be able to publish or subscribe to anything that they can copy or paste. "Sharing Data Among Applications," which begins on page 1-6, describes how you can use the publish and subscribe features in your application.

n *Scripting.* **The OSA includes the Apple Event Manager, the Apple events defined by the** *Apple Event Registry: Standard Suites,* **and the routines supported by scripting components, which applications can use via the Component Manager to execute scripts. Script-editing applications such as Script Editor (not shown in Figure 1-1) allow users to manipulate and execute scripts.**

   Each scripting language has a corresponding scripting component that can execute scripts written in that language. Scripting components typically implement a text-based scripting language based on Apple events. For example, the AppleScript component implements AppleScript, the standard user scripting language defined by

Apple Computer, Inc. When the AppleScript component executes a script, it performs the actions described in the script, including sending Apple events to applications when necessary.

"Supporting AppleScript and Other Scripting Languages," which begins on page 1-13, describes how the OSA makes it possible for your application to

n provide human-language equivalents to Apple event codes so that scripting components can send your application the appropriate Apple events during script execution

n allow users to record their actions in the form of a script

n manipulate and execute scripts

n *Sending and responding to Apple events.* Your application can send Apple events directly to other applications to request services or information or to provide information. To support AppleScript and most other scripting languages based on the OSA, your application must be able to respond to Apple events. "Sending and Responding to Apple Events," which begins on page 1-9, describes how applications can send and respond to Apple events with the aid of the Apple Event Manager.

n *Sending and responding to other high-level events.* The Event Manager allows applications to support high-level events other than Apple events. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about high-level events.

n *Exchanging message blocks.* The PPC Toolbox allows applications to exchange blocks of data with each other by reading and writing low-level message blocks. This method of communication is most useful for applications that are closely integrated, specifically designed to work together, or dependent on each other for information. It can also be used in code that is not event-based. See "Exchanging Message Blocks" on page 1-22 for a summary of the capabilities provided by the PPC Toolbox.

All forms of IAC are based on the premise that applications cooperate with each other. Both the application sending a high-level event or low-level message block and the application receiving it must agree on the protocol for communication. You can ensure effective high-level communication between your application and other Macintosh applications by supporting the standard Apple events defined in the *Apple Event Registry: Standard Suites.*

Sharing Data Among Applications

All Macintosh applications can use the Scrap Manager to share static data by allowing the user to copy and paste data between documents. Dynamic data sharing, or automated copy and paste operations between applications, extends this capability to dynamically changing data. The Edition Manager lets applications share dynamic data at the user's request. You incorporate publish and subscribe capabilities in your application much as you incorporate copy and paste capabilities.

A user can publish data by selecting a portion of text, graphics, or other data in a document and choosing Create Publisher from the Edit menu. In response, your application saves the selected information in a separate file. This stored information is referred to as an **edition.** The user can subscribe to an edition by choosing Subscribe To from the Edit menu; when the user selects a file that contains an edition, your application includes the information from the edition in the current document. The information in an edition can be shared by many documents.

Figure 1-2 shows the principal relationships among the Edition Manager, the publishing application, the subscribing application, and the file that contains the edition. In addition to the relationships illustrated in the figure, the Edition Manager uses the Apple Event Manager to communicate with applications that are sharing dynamic data.

Figure 1-2   Sharing data with the aid of the Edition Manager
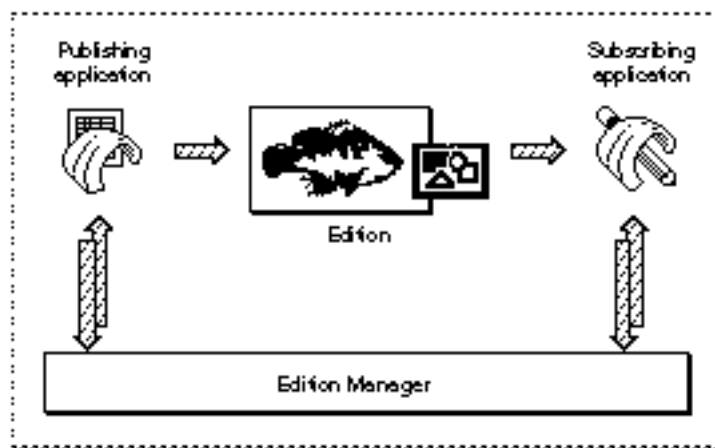
A **publisher** is a portion of a document that is made available to other documents through an edition. A **subscriber** is a portion of a document that reads the information from an edition.

Figure 1-3 shows a document containing a publisher, a file containing an edition, and a document containing a subscriber. The bottom fish in the Fishes of the World document is a publisher. The information from this publisher is made available to other documents through the Illustration edition. The Aquarium poster document contains a subscriber that gets its information from the Illustration edition. Note that when a user selects a publisher or subscriber within a document, your application should display a border surrounding the publisher or subscriber.

In general, when a user modifies the contents of a publisher and saves the document, your application should write the new data to the edition. The Edition Manager then uses the Apple Event Manager to inform all open applications with subscribers to the edition that it has been updated. These applications can then automatically update the subscribers in the documents.

Figure 1-3   A publisher, an edition, and a subscriber
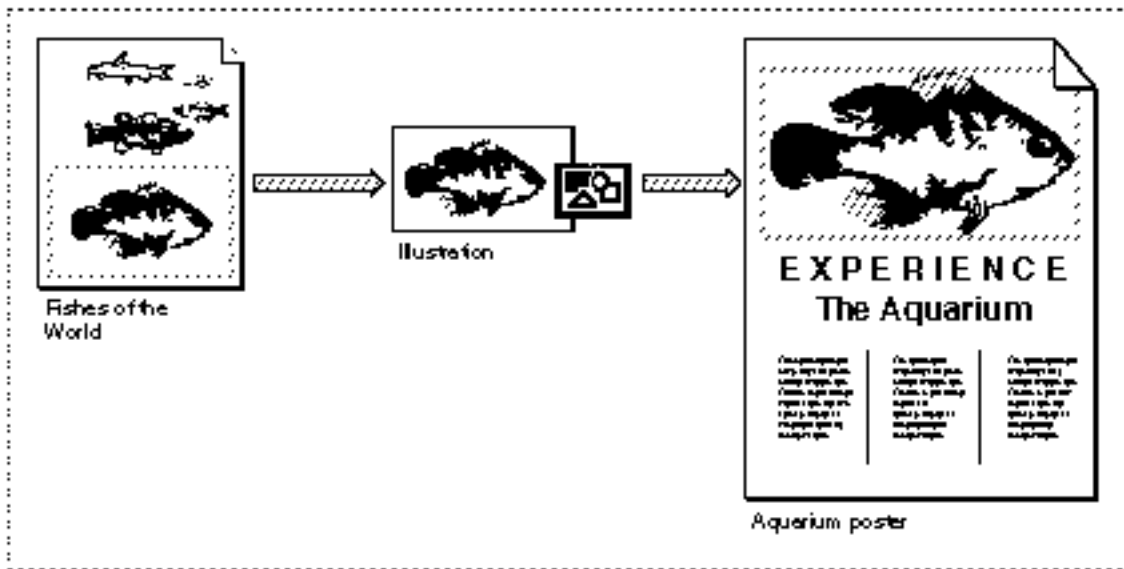
For example, suppose the user changes the color of the fish in the Fishes of the World document shown in Figure 1-3, then saves the document. This automatically changes the Illustration edition, and the subscribing application can update the Aquarium poster document if that's what the user wants to do.

Figure 1-4 shows how a user might create a poster from information contained in other documents.

Figure 1-4   Sharing dynamic data with other applications
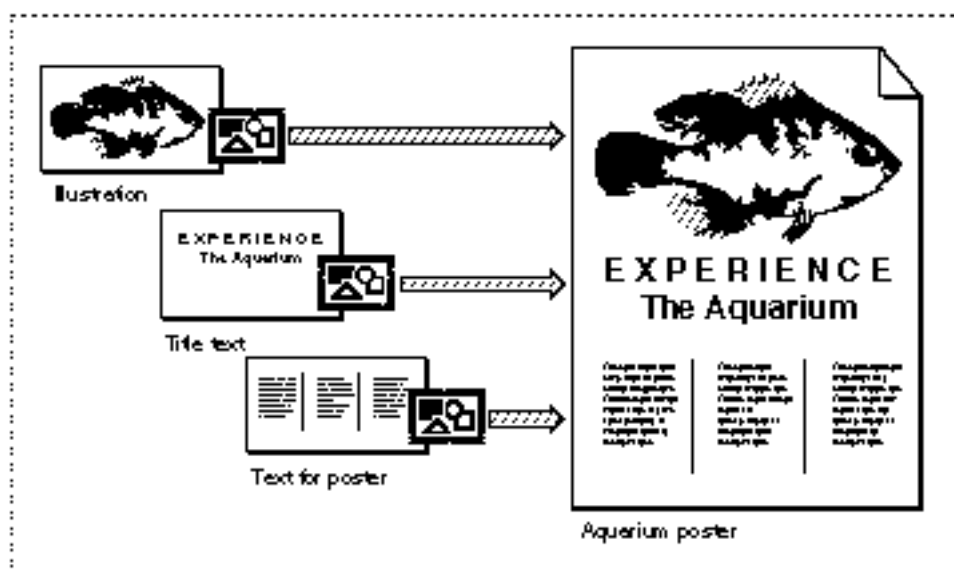
Your application should save the new information in the edition whenever the user edits the publisher and saves the document that contains the publisher—unless the user has indicated that the information should be saved in the edition on request only. When the user saves new information in an edition, the Edition Manager replaces the previous contents.

When an edition is updated, the Edition Manager informs your application. Your application should then update any subscribers (unless the user has indicated that updates should be incorporated on request only).

For example, suppose a user opens a word-processing document called My Stocks that accesses information from an edition called Stock Report. The Stock Report edition might be updated twice a day by an online database. As the information in the edition changes, the My Stocks document can receive automatic updates with the latest information.

You can implement publish and subscribe capabilities in your application by using the routines provided by the Edition Manager and supporting the related Apple events. The chapter "Edition Manager" in this book provides sample code that shows how to add these features to your application. The chapter "Responding to Apple Events" in this book describes how to support the related Apple events.

Sending and Responding to Apple Events

An Apple event is a high-level event that conforms to the Apple Event Interprocess Messaging Protocol. The Apple Event Manager uses the Event Manager to send Apple events between applications on the same computer or between applications on remote computers.

Applications typically use Apple events to request services and information from other applications or to provide services and information in response to such requests. For example, any application can use the Get Data Apple event to request that your application locate and return a particular set of data, such as a table. If your application supports the Get Data event, it should be able to recognize the event and respond by locating the requested data and returning a copy of the data to the application that requested it.

Communication between two applications that support Apple events is initiated by a **client application,** which sends an Apple event to request a service or information. For example, a client application might request services such as printing specific files, checking the spelling of a list of words, or performing a numeric calculation; or it might request information, such as one customer's address or a list of names and addresses of all customers living in Ohio. The application providing the service or the requested information is called a **server application.** The client and server applications can reside on the same local computer or on remote computers connected to a network.

Figure 1-5 shows the relationships among a client application, the Apple Event Manager, and a server application. The client application uses Apple Event Manager routines to create and send the Apple event, and the server application uses Apple Event Manager routines to interpret the Apple event and respond appropriately. If the client application so requests, the server application adds information to a reply Apple event that the Apple Event Manager returns to the client application.

Figure 1-5   Sending and responding to Apple events with the aid of the Apple Event Manager
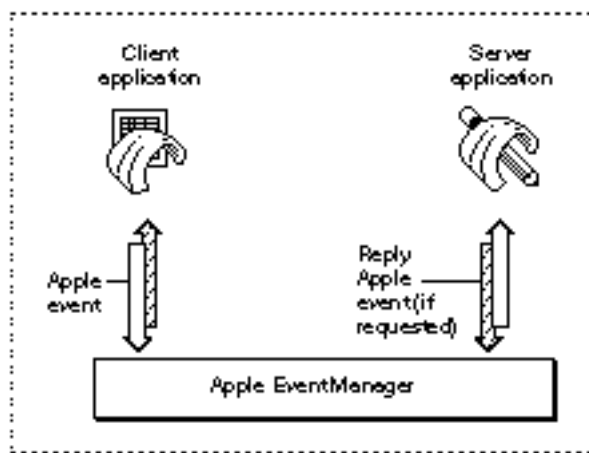
If an Apple event is one of the standard events defined in the *Apple Event Registry: Standard Suites,* the client application can construct the event and the server application can interpret it according to the standard definition for that event. To ensure that your application can respond to Apple events sent by other applications, you should support the standard Apple events that are appropriate for your application.

Standard Apple Events

The current edition of *Apple Event Registry: Standard Suites* defines the standard **suites** of Apple events, which are groups of related events that are usually implemented together. The Apple Event Registrar maintains the *Apple Event Registry: Standard Suites* and other information about the ongoing development of Apple event suites.

The standard suites include the following:

n *The Required suite* consists of the four Apple events that the Finder sends to applications. These events are Open Application, Open Documents, Print Documents, and Quit Application. The Finder uses the required events as part of the mechanisms in System 7 and later versions for launching and terminating applications. To support System 7, your application must support the required Apple events as described in the chapter "Responding to Apple Events" in this book.

n *The Core suite* consists of the basic Apple events, including Get Data, Set Data, Move, Delete, and Save, that nearly all applications use to communicate. You should support the Apple events in the Core suite that make sense for your application.

n A *functional-area suite* consists of a group of Apple events that support a related functional area. Functional-area suites include the Text suite and the Database suite. You can decide which functional-area suites to support according to which features your application provides. For example, most word-processing applications should support the Text suite, and most database applications should support the Database suite.

You do not need to implement all Apple events at once. You should begin by supporting the required Apple events, then add support for the events sent by the Edition Manager, the core events, and the functional-area events as appropriate for your application.

If necessary, you can extend the definitions of the standard Apple events to suit specific capabilities of your application. You can also define your own custom Apple events. However, only those applications that choose to support your custom Apple events explicitly will be able to make use of them. If all applications communicated solely by means of custom Apple events, every application would have to support all other applications' custom events. Instead of creating custom Apple events, try to use the standard Apple events and extend their definitions as necessary.

Apple events describe actions to be performed by the applications that receive them. In addition to a vocabulary of actions, or "verbs," effective communication between applications requires a method of referring to windows, data (such as words or graphic elements), files, folders, volumes, zones, and other items on which actions can be performed. The Apple Event Manager provides a method for specifying structured names, or "noun phrases," that applications can use to describe the objects on which Apple events act.

The *Apple Event Registry: Standard Suites* includes definitions for **Apple event object classes,** which are simply names for objects that can be acted upon by each kind of Apple event. Applications use these definitions and Apple Event Manager routines to create complex descriptions of almost any discrete item in another application or its documents. For example, an application could use Apple Event Manager routines and standard object class definitions to construct a Get Data event that requests "the most recent invoice to John Chapman in the Invoices database on the Archives server in the Accounting zone" and send the event to the appropriate application across the network.

An **Apple event object** is any item supported by an application, such as a word, paragraph, shape, or document, that can be described in an Apple event. In the example just given, the specified invoice, the Invoices database, the Archives server, and the Accounting zone are nested Apple event objects. Nearly any item that a user can differentiate and manipulate on a Macintosh computer can be described as an Apple event object of a specified object class nested within other Apple event objects. When handling an Apple event that includes such a description, an application must locate the specified Apple event object and perform the requested action on it.

Most of the standard Apple events defined in the *Apple Event Registry: Standard Suites* require your application to recognize specific Apple event object classes. Support for the standard Apple events, including Apple event object classes, allows your application to respond to requests for services or information from any other application or process.

Handling Apple Events

Figure 1-6 shows a common Apple event from the Core suite, the Set Data event. The SurfDB application is the client; it sends a Set Data event to the SurfCharter application. This event requests that SurfCharter use some new sales figures generated by SurfDB to update the data for the chart named "Summary of Sales" in the document named "Sales Chart." The Apple event contains information that identifies an action—setting data—and a description of the Apple event object on which to perform the action—"the chart named Summary of Sales in the document named Sales Report." The Apple event also includes the new data for the chart.

Figure 1-6   A Set Data event
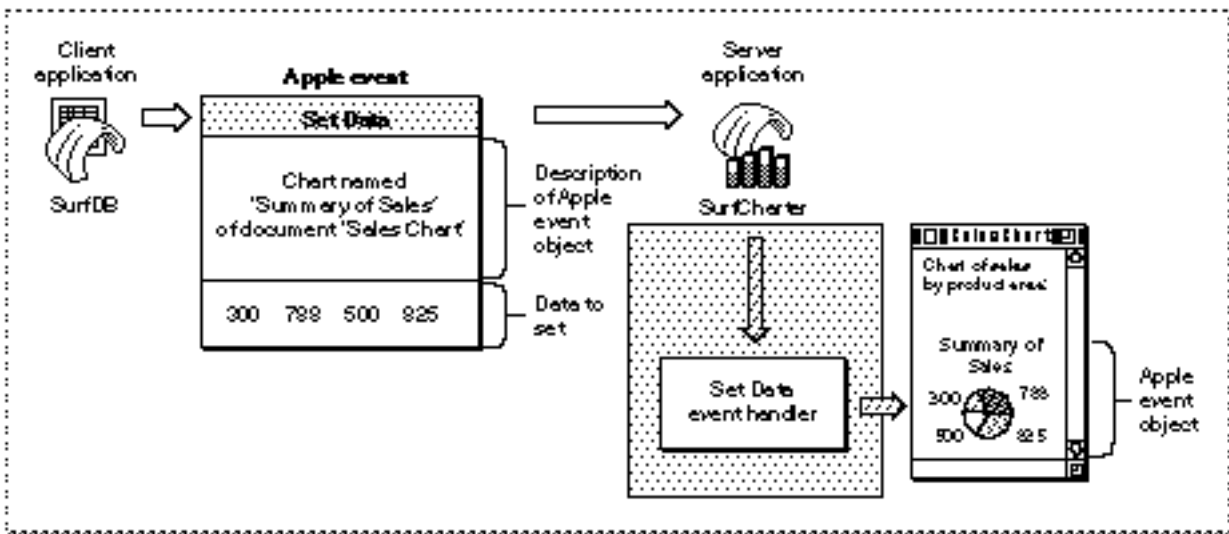
To respond appropriately, the SurfCharter application in Figure 1-6 can use the Apple Event Manager to determine what kind of Apple event has been sent and pass the event to the appropriate Apple event handler. An **Apple event handler** is an application-defined function that extracts pertinent data from an Apple event, performs the requested action, and returns a result. In this case, the Set Data event handler must locate an Apple event object—that is, the specified chart in the specified document—and change the data displayed in the chart as requested.

The Apple Event Manager provides routines that a server application can use in its Apple event handlers to take apart an Apple event and examine its contents. The SurfCharter application in Figure 1-6 can interpret the contents of the Set Data Apple event according to the definition of that event in the *Apple Event Registry: Standard Suites.* The Set Data event handler uses both Apple Event Manager routines and the SurfCharter application's own routines to locate the chart and make the requested change.

The Apple Event Manager also provides routines that a client application can use to construct and send an Apple event. However, the most important requirement for applications that support IAC is the ability to respond to Apple events, because this ability is essential for an application that users can control through scripts. The next section describes how you can use Apple events to support scripting in your application.

The chapter "Introduction to Apple Events" in this book provides an overview of Apple events and describes how you can use the Apple Event Manager to implement Apple events in your application. The chapters "Responding to Apple Events," "Creating and Sending Apple Events," "Resolving and Creating Object Specifier Records," and "Recording Apple Events" provide detailed information about the Apple Event Manager.

Supporting AppleScript and Other Scripting Languages

A **script** is any collection of data that, when executed by the appropriate program, causes a corresponding action or series of actions. For example, some database, telecommunications, and page-layout applications allow users to automate repetitive or conditional tasks by means of scripts written in proprietary scripting languages. The HyperTalk® scripting language allows users to control the behavior of HyperCard® stacks. Macro programs can automate tasks at the level of mouse clicks and keystrokes.

The Open Scripting Architecture (OSA) provides a standard mechanism that allows users to control multiple applications with scripts written in a variety of scripting languages. Each scripting language has a corresponding scripting component. When a scripting component executes a script, it performs the actions described in the script, including sending Apple events to applications if necessary.

The OSA comprises the following parts:

n The Apple Event Manager allows applications to respond to Apple events sent by scripting components (see the previous section, "Sending and Responding to Apple Events").

n The *Apple Event Registry: Standard Suites* defines the standard vocabulary of Apple events.

n The standard scripting component data structures, routines, and resources allow applications to interact with any scripting component.

n The AppleScript component implements the AppleScript scripting language.

The **AppleScript component,** which implements the **AppleScript scripting language**, is the implementation of the OSA provided by Apple Computer, Inc. Users can view a script written in the AppleScript scripting language in several different **dialects,** or versions of the AppleScript language that resemble specific human languages or programming languages.

Figure 1-7 shows the relationships among some of these parts. The client application in Figure 1-7 is Script Editor, an application provided by Apple Computer, Inc., that allows users to record, edit, and execute scripts. The client application could also be any other application that uses the standard scripting component routines to execute scripts. Script Editor uses the Component Manager to open a connection with the scripting component that created the script to be executed.

Figure 1-7   How a scripting component executes a script
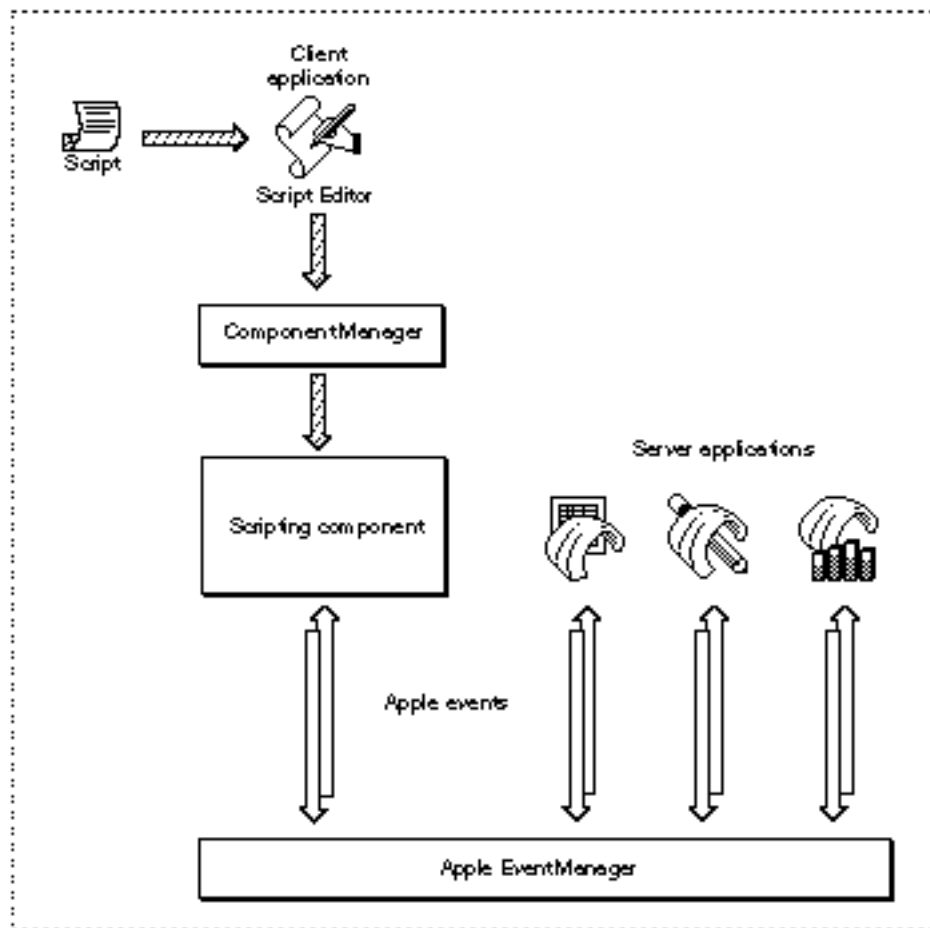
Like sound resources, scripts can be stored in applications and documents as well as in distinct script files that can be manipulated from the Finder. Script Editor allows users to execute scripts stored in **script files.** Users can also execute special script files called **script applications** simply by opening them from the Finder.

During script execution, scripting components perform actions described in the script, using the Apple Event Manager to send Apple events when necessary. The server applications shown in Figure 1-7 use the Apple Event Manager to examine the contents of the Apple events they receive and to respond appropriately. A server application always responds to the same Apple event in the same way, regardless of whether the event is sent by a scripting component or directly by a client application.

You can take advantage of the OSA in three ways:

n You can make your application **scriptable,** or capable of responding to Apple events sent to it by a scripting component. An application is scriptable if it

  n Responds to the appropriate standard Apple events. See the previous section, "Sending and Responding to Apple Events."

  n Provides an Apple event terminology extension (`'aete'`) resource that describes which Apple events your application supports and the corresponding human-language terminology for use in scripts. The `'aete'` resource allows scripting components to interpret scripts correctly and send the appropriate Apple events to your application during script execution.

  By executing scripts, users of scriptable applications can perform almost any task that they would otherwise perform by choosing menu commands, typing, and so on. Users can also execute scripts to perform many tasks that might otherwise be difficult to accomplish, especially repetitive or conditional tasks that involve multiple applications.

n You can make your application **recordable**—that is, capable of sending Apple events to itself in response to user actions such as choosing a menu command or changing the contents of a document. After a user has turned on recording for a particular scripting component, the scripting component receives copies of all subsequent Apple events and records them in the form of a script.

n You can have your application manipulate and execute scripts with the aid of a scripting component. To do so, your application must

  n use the Component Manager to open a connection with the appropriate component

  n use the standard scripting component routines to record, edit, compile, save, load, or execute scripts when necessary

  Users of applications that execute scripts can modify the applications' behavior by editing the scripts. For example, a user of an invoice program might be able to write a script that checks and if necessary updates customer information in a separate database application each time the user posts an invoice.

The sections that follow describe these three kinds of scripting capabilities in more detail. The chapter "Introduction to Scripting" in this book provides an overview of the way scripting components work and how you can implement support for scripting in your application.

Scriptable Applications

If your application can respond to standard Apple events sent by other applications, it can also respond to the same Apple events sent by a scripting component. Before executing a script that controls your application, a scripting component must associate the human-language terms used in the script with specific Apple event codes supported by your application. Scriptable applications provide this information in an Apple event terminology extension (`'aete'`) resource.

Because scripting components can obtain information from `'aete'` resources about the nature of different applications' support for Apple events, a single script can describe complex tasks performed cooperatively by several specialized applications. For example, a user can execute an AppleScript script to locate all records in a database with specific characteristics, update a series of charts based on those records, import the charts into a page-layout document, and send the document to a remote computer on the network via electronic mail.

When a user executes such a script, the AppleScript component attempts to perform the actions the script describes, including sending Apple events to various applications when necessary. To map human-language terms used in the script to the corresponding Apple events supported by each application, the AppleScript component looks up the terms in the applications' `'aete'` resources. Each human-language term specified by an application's `'aete'` resource has a corresponding Apple event code. After the AppleScript component has identified the Apple event codes for the terms used in a script, it can create and send the Apple events that perform the actions described in the script.

To respond appropriately to the Apple events sent to it by the AppleScript component, the database application in this example must be able to locate records with specific characteristics so that it can identify and return the requested data. The other applications involved must support Apple events that perform the other actions described in the script.

One line in such a script might be a statement like this:

```
copy Totals to chart "Summary of Sales" of document "Sales Chart"
```

In this statement, the word `Totals` is a variable that has been set earlier in the same script to the value of the new data generated by a database application. The statement causes the AppleScript component to send a Set Data event updating the chart named "Summary of Sales." Figure 1-8 shows how the AppleScript component would execute this statement. (Figure 1-6 on page 1-12 shows a database application that sends a similar Set Data event directly.)

To interpret the terms in this script statement correctly, the AppleScript component must be able to look them up in the SurfCharter application's `'aete'` resource, which maps those terms to the corresponding codes for Apple events, object classes, and so on used by the Apple Event Manager. The AppleScript component can then create and send the Set Data event to SurfCharter.

When it receives the Set Data event, the SurfCharter application uses the Apple Event Manager to determine what kind of Apple event has been sent and to pass the event to SurfCharter's handler for that event, which in turn locates the chart and changes its data as requested.

The chapter "Introduction to Scripting" in this book describes how the `aete` resource works. The chapter "Apple Event Terminology Resources" describes how to define terminology for use by the AppleScript component and how to create an `aete` resource.

Figure 1-8   A Set Data event sent during script execution
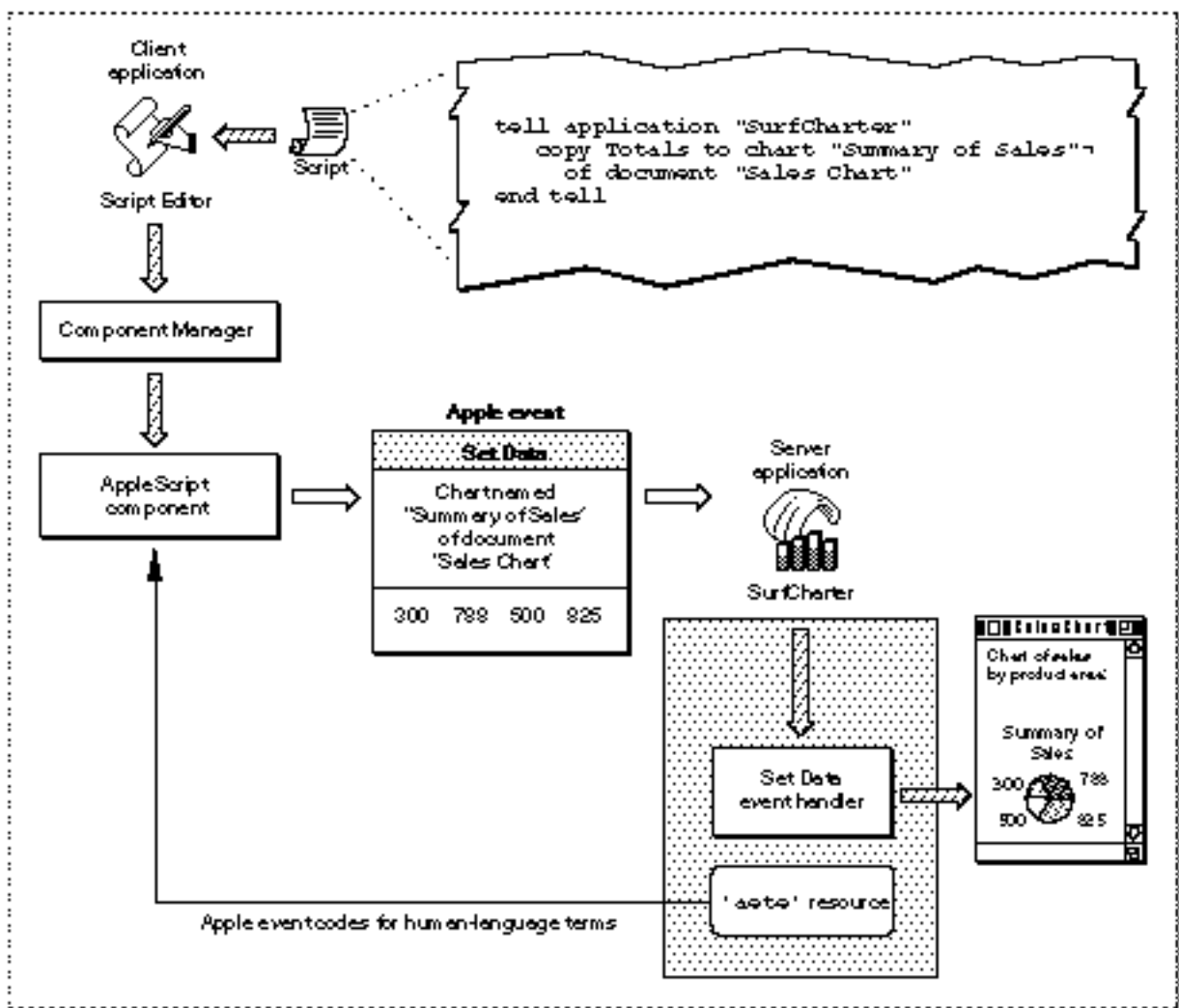
Recordable Applications

If you decide to make your application scriptable, you can also make it recordable, allowing users to record their actions in your application in the form of a script. Even users with little or no knowledge of a particular scripting language can record their actions in recordable applications in the form of a script. More knowledgeable users can record scripts and then edit or combine them as desired.

Applications generally have two parts: the code that implements the application's user interface and the code that actually performs the work of the application when the user manipulates the interface. To make your application fully recordable, you should separate these two parts of your application, using Apple events to connect user actions with the work your application performs.

Any significant user action within a recordable application should generate Apple events that a scripting component can record as statements in a script. For example, when a user chooses New from the File menu, a recordable application sends itself a Create Element event, and the application's handler for that event creates the new document. Implementing Apple events in this manner is called **factoring** your application. A factored application acts as both the client and the server application for the Apple events it sends to itself.

In general, a recordable application should generate Apple events for any user action that could be reversed by the Undo command. A recordable application can usually handle a greater variety of Apple events than it can record, since it must record the same action the same way every time even though Apple events might be able to trigger that action in several different ways.

A **recordable event** is any Apple event that any recordable application sends to itself while recording is turned on for the local computer (with the exception of events that the application indicates it does not want to be recorded). After a user turns on recording from the Script Editor application, the Apple Event Manager sends copies of all recordable events to Script Editor. A scripting component previously selected by the user handles each copied event for Script Editor by translating the event into the scripting component's scripting language and recording the translation as part of a Script Editor script. When a scripting component executes a recorded script, it sends the corresponding Apple events to the applications in which they were recorded.

Figure 1-9 illustrates how Apple event recording works. The user performs a significant action (such as choosing New from the File menu), and the SurfCharter application sends itself an Apple event to perform the task associated with that action. If recording is turned on, the Apple Event Manager automatically sends a copy of each recordable Apple event to the application (for example, Script Editor) that initiated recording. The scripting component handles the copy of each recordable event by translating it and recording it as part of a script. To translate each Apple event correctly, the scripting component must first check what equivalent human-language terminology the SurfCharter application uses for that Apple event. The scripting component then records the equivalent statement in the script.

The chapter "Recording Apple Events" in this book describes the Apple Event Manager's recording mechanism in more detail and explains how to use Apple events to factor your application.

Figure 1-9   Recording user actions in a factored application
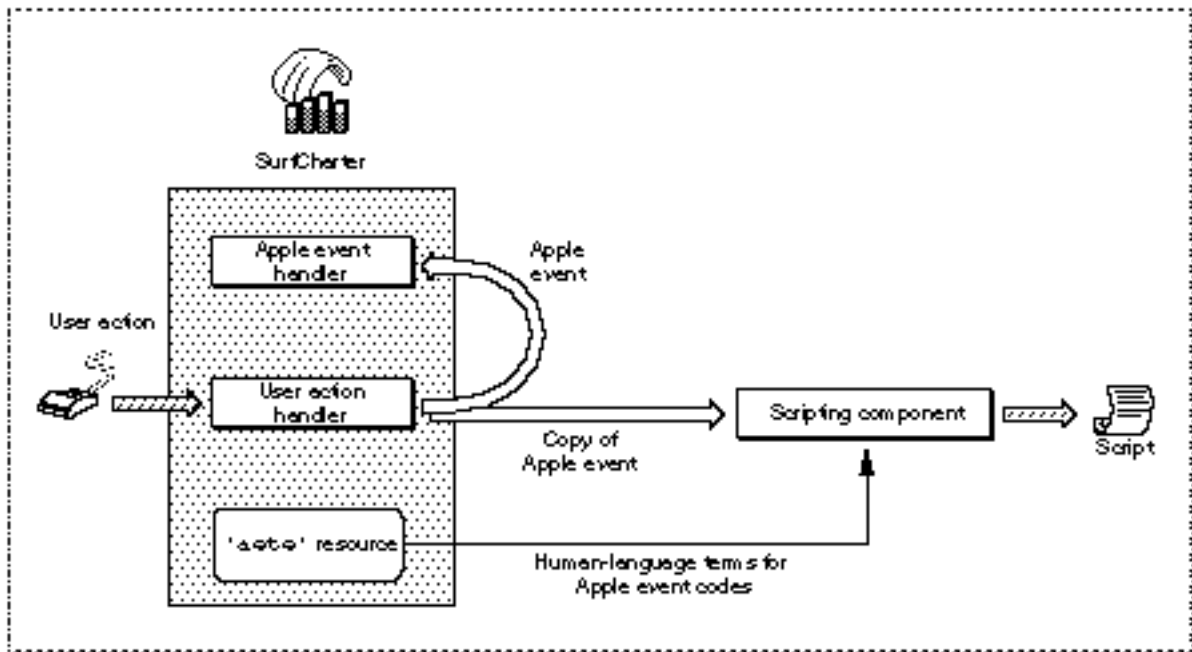
Applications That Manipulate and Execute Scripts

Like sound resources, scripts can be stored either as separate files with their own icons in the Finder or within an application or its documents. Your application can store and execute scripts regardless of whether it is scriptable or recordable. If your application is scriptable, however, it can execute scripts that control its own behavior, thus acting as both the client application and the server application for the corresponding Apple events.

Your application can establish a connection with any scripting component that is registered with the Component Manager on the same computer. Each scripting component can manipulate and execute scripts written in the corresponding scripting language (or, as in the case of AppleScript, one of the scripting language's dialects) when your application calls the standard scripting component routines.

You can use the standard scripting component routines to

n   get a handle to a script so you can save the script in a preferences file, in the data fork of a document, or as a separate script file

n   manipulate scripts associated with any part of your application or its documents, including both Apple event objects and other objects defined by the application

n   let users record and edit scripts

n   compile and execute scripts

Figure 1-10 shows how an application might execute a script that controls its own behavior. The appropriate user action handler executes the script in response to a user action, which can be almost anything: choosing a menu command, clicking a button, tabbing from one table cell to another, and so on. The script might consist of a single statement that describes some default action, such as saving or printing, or a series of statements that describe a series of tasks, such as setting default preferences or styles. Figure 1-10 shows a script that corresponds to a single Apple event, but the script could just as easily correspond to a whole series of Apple events. If your application allows users to modify such a script, they can modify the behavior of your application to suit their needs.

Figure 1-10   Controlling an application's own behavior by executing a script
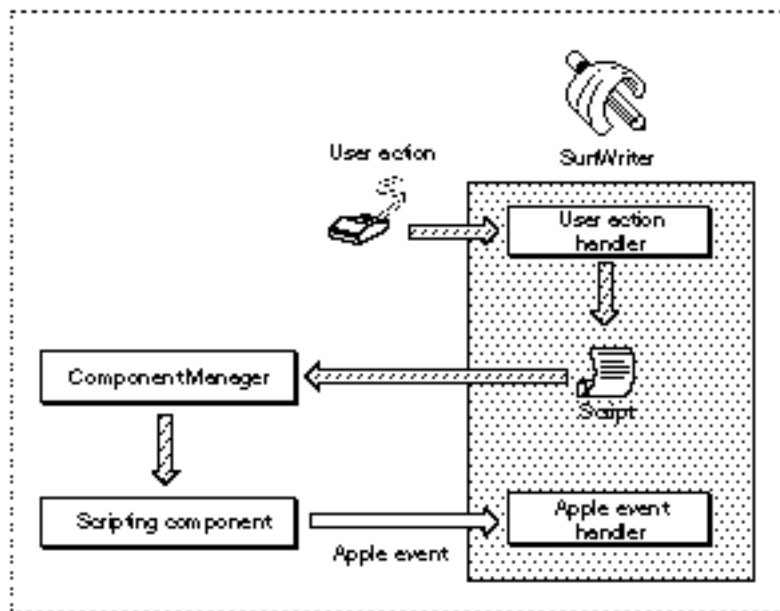
Your application can associate a script with any Apple event object or application-defined object and execute the script when that object is manipulated in some way. The script can describe actions to be taken by your application, as in Figure 1-10, or actions to be taken by several applications. For example, a user of a word-processing application might attach a script to a specific word so that the application executes the script whenever that word is double-clicked. Such a script could trigger Apple events that look up and display related information from a separate document, run a QuickTime movie, perform a calculation, play a voice annotation, and so on.

Figure 1-11 shows one way that a script can be used to control two or more applications. When a user chooses the Post Invoice command in the accounting application, the user action handler for that menu command executes a default script for posting an invoice. That script might describe actions such as saving the invoice, updating the sales journal, and so on. The scripting component sends Apple events to the accounting application to perform these actions.

Figure 1-11   Posting an invoice and updating a database by executing a script
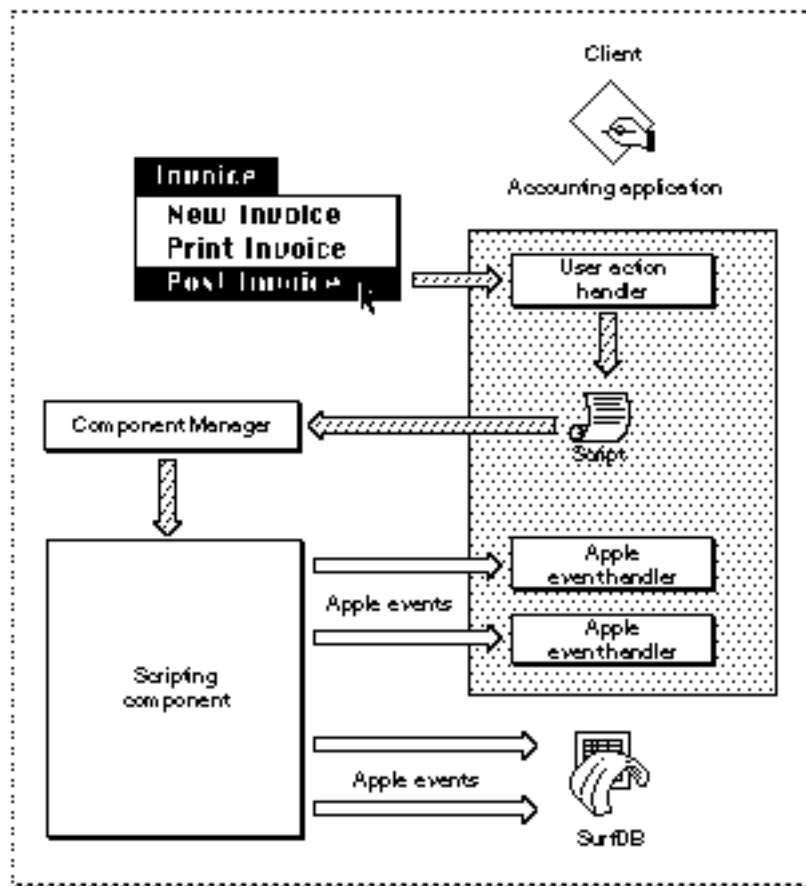
The accounting application also allows users to open the default invoice-posting script in Script Editor and modify it so that additional actions are performed when it is executed. For example, as shown in Figure 1-11, the script could instruct the SurfDB application to update a database of customer information in addition to performing the default posting actions. In this case, the scripting component sends Apple events to both the accounting application and SurfDB to carry out all the actions described by the script.

There is no limit to the actions such a script can describe. In addition to sending the Apple events shown in Figure 1-11, the invoice-posting script could be used to trigger Apple events that cause other applications to perform a credit check, send the invoice to the customer by electronic mail, forward inventory information to a remote server on the network, and so on.

The chapter "Scripting Components" in this book describes how your application can use the standard scripting component routines to manipulate and execute its own scripts and allow users to modify those scripts.

Exchanging Message Blocks

You should be able to meet most of your application's IAC needs by using the Apple Event Manager or the Event Manager. However, if you need low-level control or services not provided by the Apple Event Manager or the Event Manager, you can use the PPC Toolbox. The PPC Toolbox lets you send large amounts of data to other applications located on the same computer or across a network. The PPC Toolbox can also be used by pieces of code that are not event-driven. The PPC Toolbox is usually called by the Operating System; device drivers, desk accessories, or other code modules can also use it.

You cannot use the PPC Toolbox to send data between applications unless both your application and the application you're communicating with are open at the same time. To initiate communication, one program opens a port and requests a session with another program. The target application must also open a port and accept the request. Once a session is established, the two programs can read and write low-level message blocks.

The PPC Toolbox also provides a standard user interface that allows a user working in one application to select another application with which to exchange data, whether the communication is achieved by means of Apple events, other high-level events, or message blocks.

The chapter "Program-to-Program Communications Toolbox" in this book describes how programs can exchange low-level message blocks.

Edition Manager

Contents

This chapter describes how you can use the Edition Manager to allow your users to share and automatically update data from numerous documents and applications.

The Edition Manager is available only in System 7 or later. It can be used by many different applications located on a single disk or throughout a network of Macintosh computers. To test for the existence of the Edition Manager, use the `Gestalt` function, described in *Inside Macintosh: Operating System Utilities.*

Read the information in this chapter if you want your application's documents to share and automatically update data, or if you want to share and automatically update data with documents created by other applications that support the Edition Manager.

For example, a user might want to capture sales figures and totals from within a spreadsheet and then include this information in a word-processing document that summarizes sales for a given month. The Edition Manager establishes a connection between these two documents. When a user modifies the spreadsheet, the information in the word-processing document can be automatically updated to contain the latest changes. To accomplish this, both the spreadsheet application and the word-processing application must support the features of the Edition Manager.

To use this chapter, you should be familiar with sending and receiving high-level events, described in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.* Your application must also support Apple events to receive Apple events from the Edition Manager. See the following chapters in this book for detailed information on Apple events.

The Edition Manager provides you with the ability to

n   capture data within a document and integrate it into another document

n   modify information in a document and automatically update any document that shares its data

n   share information between applications on the same computer or across a network of Macintosh computers

Building the capabilities of the Edition Manager into your program is similar to building cut-and-paste features into your program. Text, graphics, spreadsheet cells, database reports—any data that you can select, you can make accessible to other applications that support the Edition Manager. The next section provides an overview of the main elements of the Edition Manager. Following sections discuss how to implement these features in your application.

This chapter also describes an advanced feature that allows applications to share data directly from a file.

Introduction to Publishers, Subscribers, and Editions

A **section** is a portion of a document that shares its contents with other documents. The Edition Manager supports two types of sections: publishers and subscribers. A **publisher** is a section within a document that makes its data available to other documents or applications. A **subscriber** is a section within a document that obtains its data from other documents or applications.

Your application writes a copy of the data in each publisher to a separate file called an **edition container.** The actual data that is written to the edition container is referred to as the **edition.** Your application obtains the data for each subscriber by reading data from the edition container. Note that throughout this chapter, the term *edition* refers to the edition container and the data it contains.

You **publish** data when you want to make it available to other documents and applications. When data is published, it is stored in an edition container. You **subscribe** to data that a publisher makes available by reading an edition from its container.

Note

Section and *edition container* are programmatic terms. You should not use them in your application or your documentation. Use *publishers, subscribers,* and *editions.* You should also refrain from using other terms such as *publication* or *subscription* to describe the dynamic sharing of information provided by the Edition Manager. Use the terms *publish* and *subscribe* to describe the Edition Manager features. u

Each edition has an icon that is visible from the Finder. Figure 2-1 shows the default edition icon.

Figure 2-1   The default edition icon

The name that the user specifies for the edition is located next to the edition icon. For information on providing icons for the editions created by your application, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials.* Figure 2-2 illustrates a document containing a single publisher, its corresponding edition, and a subscriber to the edition in another document.

Figure 2-2   A publisher, an edition, and a subscriber
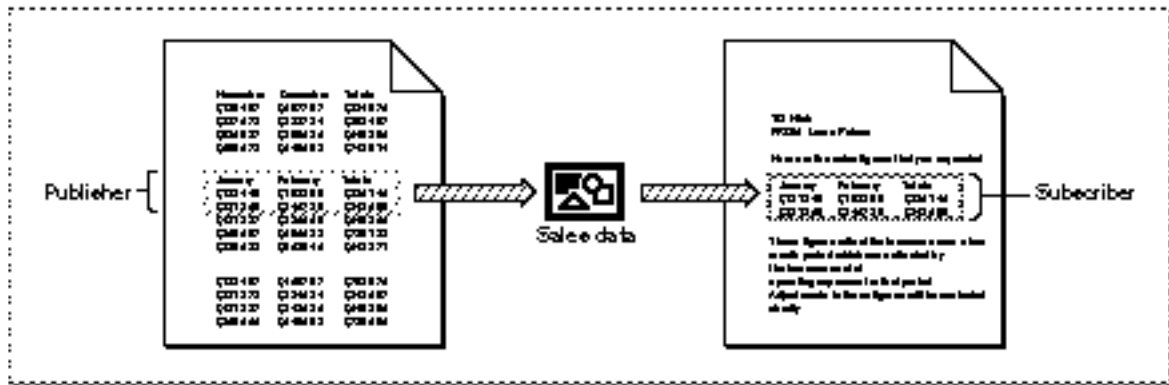
Note that the publisher and subscriber borders illustrated in Figure 2-2 may appear slightly different from the borders you see on the screen. Figure 2-6 on page 2-9 shows the publisher and subscriber borders as they appear onscreen.

Data always flows in one direction, from publisher to edition to subscriber. Documents that contain publishers and subscribers do not have to be open at the same time to share data. Whenever the user saves a document that contains a publisher, the edition changes to reflect the current data from the publisher. All subscribers update their contents from the edition. Any number of subscribers can subscribe to a single edition.

To create a publisher within a document, a user selects an area of the document to share and chooses Create Publisher from the Edit menu (see Figure 2-7 on page 2-10). Figure 2-3 shows the dialog box that your application should display when the user chooses Create Publisher.

Figure 2-3   The publisher dialog box
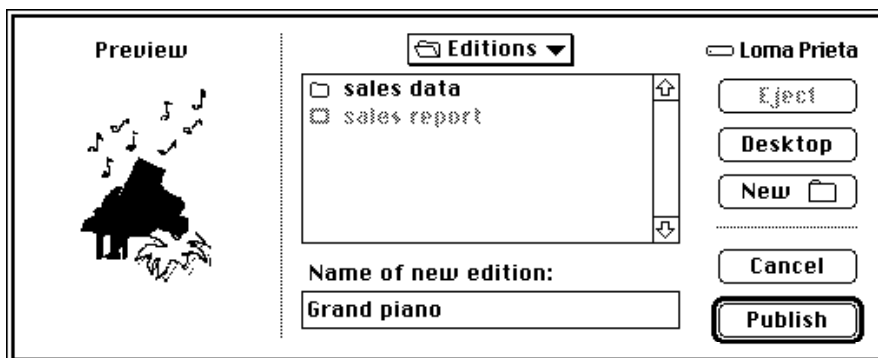
Your application provides a thumbnail sketch of the edition data, which the Edition Manager displays in the preview area of the publisher dialog box. Your preview of the edition in this dialog box should provide a visual cue about the type of information that the user has selected to publish.

A preview area also appears in the subscriber dialog box (see Figure 2-4). This preview, too, should provide a visual cue about the type of information the edition contains. For example, it should allow users to distinguish between text information and spreadsheet arrays.

The publisher dialog box uses the extended interface of the standard file dialog box that accompanies System 7. The user navigates through the contents of the disk using the mouse or keyboard.

A user can modify a publisher within a document just like any other portion of a document. As a default, each time a user saves a document containing a publisher, your application should automatically write the publisher's data to the edition. You also need to provide the user with the choice of sending new publisher data to an edition manually (that is, only at the user's specific request). You should provide these options by using the publisher options dialog box described later in "Using Publisher and Subscriber Options" beginning on page 2-43.

For example, one user may choose to update an edition automatically each time a document is saved. This update mode is useful for a user who creates a publisher within a spreadsheet application that records stock information. Each time the user updates the stock information and saves the spreadsheet, a new edition automatically becomes available to subscribers.

Another user may choose to update an edition only upon request. This update mode might be useful for a user who creates a publisher within a word-processing application for a quarterly sales report. The user incrementally updates the sales report throughout the entire quarter but does not want this information to be available to subscribers until the end of the quarter. Only at the end of each quarter does the user specifically request to update the edition and make it available to any subscribers.

To create a subscriber within a document, the user places the insertion point and chooses Subscribe To from the Edit menu. Figure 2-4 shows the dialog box that your application should display when the user chooses Subscribe To.

Figure 2-4   The subscriber dialog box

The subscriber dialog box also uses the extended interface of the standard file dialog box introduced with System 7. Initially, the dialog box should highlight the name of the last edition published or subscribed to. This allows a user to create a publisher and immediately subscribe to its edition.

A subscriber receives its data from a single edition. By default, your application should automatically update a document containing a subscriber whenever a new edition is available. You also need to provide the user with the choice of receiving the latest edition manually (that is, only when the user specifically requests it). You can provide these options by using the subscriber options dialog box described later in "Using Publisher and Subscriber Options" beginning on page 2-43.

For example, one user may choose to receive new editions automatically as they become available. This update mode is useful for a user who subscribes to information from an edition that consists of daily sales figures. This user automatically acquires each version of the sales information as it becomes available.

Another user may choose to receive a new edition only upon request. This update mode is useful for a user who creates a subscriber to an edition that consists of graphics data (such as a company logo). The user may require only periodic versions of the logo and not need frequent updates. In this case, your application should update the subscriber with a new edition only when the user specifically requests it.

A user can select, cut, copy, or paste an entire subscriber. Although the contents of the subscriber as a whole can be modified, a user cannot edit portions of a subscriber. For example, a user can underline or italicize the entire subscriber text but cannot delete a sentence or rotate a single graphic object. This restriction protects the user from losing changes to a subscriber when a new edition arrives. Remember that, as a default, new editions should automatically update a subscriber. Any changes that a user made to the subscriber text would have to be reapplied by the user when the new edition arrives. See "Modifying a Subscriber" on page 2-59 for further information.

A single document can contain any number or combination of publishers and subscribers. Figure 2-5 shows an example of a document that contains two publishers and one subscriber (and their corresponding editions). Remember that data always flows in one direction, from publisher to edition to subscriber. The "Concert flyer" document contains a publisher that is subscribed to by the "Benefit concert" document. The "Concert flyer" document also subscribes to a portion of the "Pianos & palm trees" document. In addition, the "Concert flyer" document as a whole is subscribed to by the "Sample flyer" document.

Figure 2-5 A document and its corresponding editions
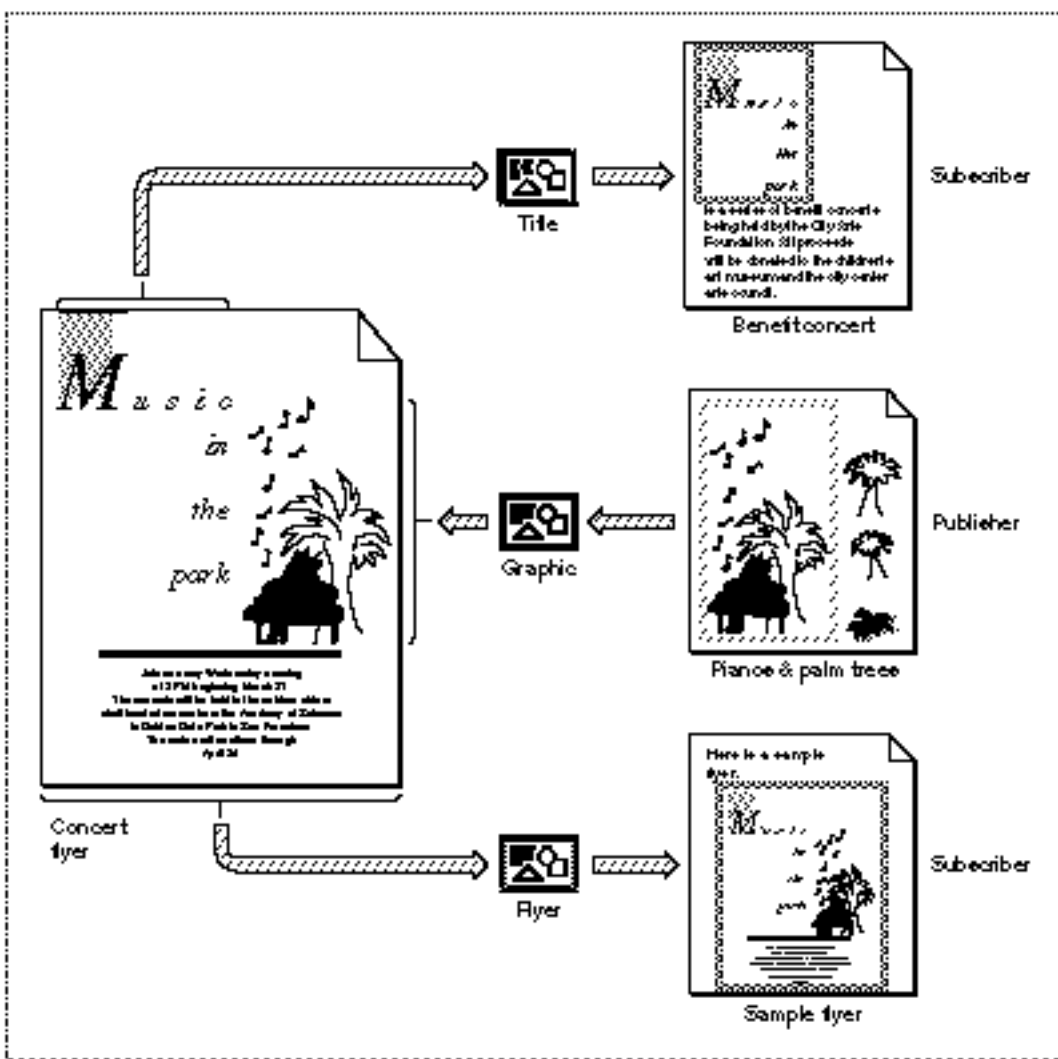
You should distinguish each selected publisher and subscriber within a document with a border. Display a publisher border as three pixels wide with 50 percent gray lines, and display a subscriber border as three pixels wide with 75 percent gray lines. A rectangle of one white pixel should separate the data from the border itself. Borders should be drawn *outside* the contents of publishers and subscribers so that data is not obscured. See Figure 2-6 for an illustration of the borders as they appear onscreen. See "Displaying Publisher and Subscriber Borders" on page 2-50 for detailed information on how to implement borders for specific applications.

Figure 2-6 shows a document containing a publisher and a document containing a subscriber, with borders displayed for each.

Borders for publishers and subscribers should behave like the borders of 'PICT' graphics within a word-processing document. Your application should display a border whenever the user clicks within the content area of a publisher or a subscriber. Your application should hide the border whenever the user clicks outside the content area. See "Displaying Publisher and Subscriber Borders" on page 2-50 for detailed information on how to implement borders for specific applications.

Figure 2-6   Publisher and subscriber borders
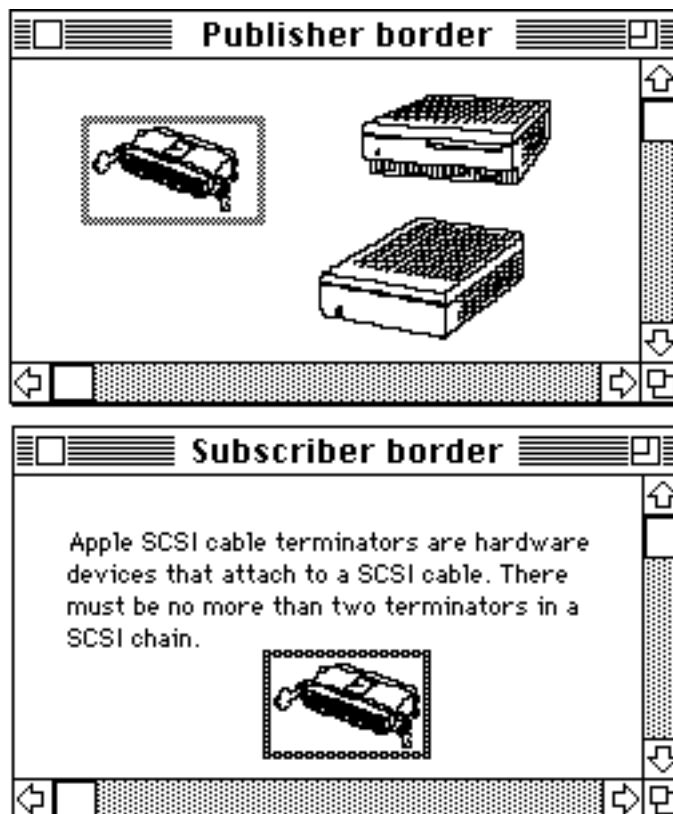
You also need to support the standard Edition Manager menu commands in the
Edit menu. These menu items include

n Create Publisher…

n Subscribe To…

n Publisher/Subscriber Options…

n Show/Hide Borders (optional)

n Stop All Editions (optional)

Use a divider to separate the Edition Manager menu commands from the standard
Edit menu commands Cut, Copy, and Paste. Figure 2-7 shows the standard Edition
Manager menu commands.

Figure 2-7   Edition Manager commands in the Edit menu

```
┌─────────────────────┐
│ Edit                │
├─────────────────────┤
│ Undo          ⌘Z    │
├─────────────────────┤
│ Cut           ⌘H    │
│ Copy          ⌘C    │
│ Paste         ⌘U    │
│ Select All    ⌘A    │
├─────────────────────┤
│ Create Publisher... │
│ Subscribe To...     │
│ Publisher Options...│
├─────────────────────┤
│ Show Clipboard      │
└─────────────────────┘
```

The Subscriber Options menu command should toggle with the Publisher Options menu
command. When a user selects a subscriber and then accesses the menu bar, your
application should adjust its menus so that the Subscriber Options menu command
appears in the Edit menu. When a user selects a publisher and then accesses the menu
bar, your application should adjust its menus so that the Publisher Options menu
command appears in the Edit menu. In addition, you may support a Show Borders
menu command that toggles with Hide Borders to display or hide all publishers and
subscriber borders within documents. You may also support a Stop All Editions menu
command to provide a method for temporarily suspending all update activity in a
document. When the user chooses this command, you should place a checkmark next to
it. You should also stop all publishers from sending data to editions and all subscribers
from receiving new editions. When the user chooses this command again, remove the
checkmark and update any subscribers that are set up to receive new editions
automatically.

If you find that you need all of the available space in the Edit menu for your application's commands, you may create a hierarchical menu for the Edition Manager menu commands. If you choose to implement this structure, you should allow users to access the Edition Manager menu commands through a Publishing menu command in the Edit menu. Because this menu structure is not as accessible to users, you should implement it only if you have no other alternative.

Figure 2-8 shows the Edition Manager menu commands in a hierarchical menu structure.

Figure 2-8   Edition Manager commands under the Publishing menu command
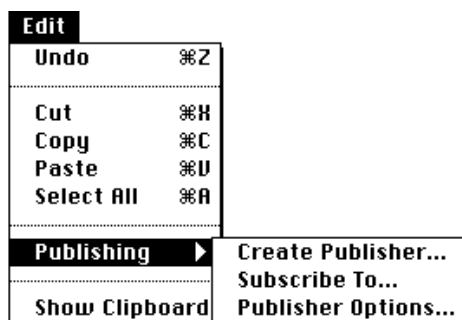
For each publisher or subscriber within an open document, you must have a section record and an alias record. The section record contains a time stamp that records the version of the data that resides in the section. The section record also identifies the section as either a publisher or subscriber, and it establishes a unique identity for each publisher or subscriber. The section record does *not* contain the data within the section. The alias record is a reference to the edition container from the document that contains the corresponding publisher or subscriber section.

There are special options associated with publishers and subscribers within documents. Your application can use the publisher and subscriber options dialog boxes provided by the Edition Manager to make these choices available to the user. For example, a user can select Open Publisher within the subscriber options dialog box to access the document containing the publisher. Your application can also allow a user to cancel subscribers or publishers within documents, specify when to update an edition from a publisher, or specify when to update a subscriber with a new edition. These options are described in "Using Publisher and Subscriber Options" beginning on page 2-43.

About the Edition Manager

The next section discusses how to save, open, read, and write a document that shares data. In addition, it describes how to

n   make data accessible to other applications

n   integrate data into numerous documents

n   set update options

n   implement borders

n   modify shared data

n   customize dialog boxes

n   subscribe to data in non-edition files

Using the Edition Manager

This section describes how your application can

n   receive Apple events from the Edition Manager

n   set up a section record and alias record for open documents containing sections

n   save a document that contains sections

n   open a document that contains sections

n   read and write sections

n   create a publisher within a document, create its edition container, and write data to it

n   create a subscriber within a document and read its data from an edition

To begin, you must determine whether the Edition Manager is available on the system by using the `Gestalt` function with the `gestaltEditionMgrAttr` (`'edtn'`) selector. If the `response` parameter returns 1 in the bit defined by the `gestaltEditionMgrPresent` constant (bit 0), the Edition Manager is present.

If the Edition Manager is present, load it into memory using the `InitEditionPack` function. This function determines whether the machine has enough space in the system heap for the Edition Manager to operate.

```
err := InitEditionPack;
```

If the `InitEditionPack` function returns `noErr`, you have enough space to load the package. If you do not have enough space, the application can either terminate itself or continue with the Edition Manager functionality disabled.

Receiving Apple Events From the Edition Manager

Applications that use the Edition Manager must support Apple events. This requires that your application support the required Open Documents event and Apple events sent by the Edition Manager. See the chapter "Introduction to Apple Events" in this book for general information on Apple events.

Apple events sent by the Edition Manager arrive as high-level events. The `EventRecord` data type defines the event record.

```
TYPE EventRecord =
    RECORD
        what:       Integer;     {kHighLevelEvent}
        message:    LongInt;     {'sect'}
        when:       LongInt;
        where:      Point;       {'read', 'writ', 'cncl', 'scrl'}
        modifiers:  Integer;
    END;
```

The Edition Manager can send these Apple events with the event class and event ID as shown here:

n   **Section Read events** (`'sect'` `'read'`)

n   **Section Write events** (`'sect'` `'writ'`)

n   **Section Cancel events** (`'sect'` `'cncl'`)

n   **Section Scroll events** (`'sect'` `'scrl'`)

Each time your application creates a publisher or a subscriber, the Edition Manager registers its section. When an edition is updated, the Edition Manager scans its list to locate registered subscribers. For each registered subscriber that is set up to receive updated editions automatically, your application receives a Section Read event.

If the Edition Manager discovers that an edition file is missing while registering a publisher, it creates a new edition file and sends the publisher a Section Write event.

When you receive a Section Cancel event, you need to cancel the specified section. Note that the current Edition Manager does not send you Section Cancel events, but you do need to provide a handler for future expansion.

If the user selects a subscriber within a document and then selects Open Publisher in the subscriber options dialog box, the publishing application receives the Open Documents event and opens the document containing the publisher. The publishing application also receives a Section Scroll event. Scroll to the location of the publisher, display this section on the user's screen, and turn on its border.

See "Opening and Closing a Document Containing Sections" beginning on page 2-22 for detailed information on registering and unregistering a section and writing data to an edition. See "Using Publisher and Subscriber Options" beginning on page 2-43 for information on publisher and subscriber options.

After receiving an Apple event sent by the Edition Manager, use the Apple Event Manager to extract the section handle. In addition, you must also call the IsRegisteredSection function to determine whether the section is registered. It is possible (because of a race condition) to receive an event for a section that you recently disposed of or unregistered. One way to ensure that an event corresponds to a valid section is to call the IsRegisteredSection function after you receive an event.

```
err := IsRegisteredSection (sectionH);
```

Listing 2-1 illustrates how to use the Apple Event Manager and install an event handler to handle Section Read events. You can write similar code for Section Write events, Section Scroll events, and Section Cancel events.

Listing 2-1   Accepting Section Read events and verifying if a section is registered

```
{the following goes in your initialization code}
myErr := AEInstallEventHandler(sectionEventMsgClass {'sect'},
                               sectionReadMsgID {'read'},
                               @MyHandleSectionReadEvent, 0,
                               FALSE);

{this is the routine the Apple Event Manager calls when a }
{ Section Read event arrives}

FUNCTION MyHandleSectionReadEvent(theAppleEvent,
                                  reply: AppleEvent;
                                  refCon: LongInt): OSErr;
VAR
   myErr:       OSErr;
   sectionH:    SectionHandle;
BEGIN
   {get section handle out of Apple event message buffer}
   myErr := MyGetSectionHandleFromEvent(theAppleEvent, sectionH);
   IF myErr = noErr THEN
   BEGIN
      IF IsRegisteredSection(sectionH) = noErr THEN
         {if section is registered, read the new data}
         MyHandleSectionReadEvent := DoSectionRead(sectionH);
   END
   ELSE
      MyHandleSectionReadEvent := myErr;
END; {MyHandleSectionReadEvent}
```

```
{this routine reads in subscriber data and updates its display}
FUNCTION DoSectionRead(subscriber: SectionHandle): OSErr;
BEGIN
   {your code here}
END;  {DoSectionRead}

{this is part of your Apple event-handling code}
FUNCTION MyGetSectionHandleFromEvent(theAppleEvent: AppleEvent;
                                     VAR sectionH: SectionHandle)
                                     : OSErr;
VAR
   ignoreType:    DescType;
   ignoreSize:    Size;
BEGIN
   {parse section handle out of message buffer}
   MyGetSectionHandleFromEvent
      := AEGetParamPtr( theAppleEvent,    {event to parse}
                        keyDirectObject,  {look for direct }
                                          { object}
                        typeSectionH,     {want a SectionHandle}
                        ignoreType,       {ignore type it could }
                                          { get}
                        @sectionH,        {put SectionHandle }
                                          { here}
                        SizeOf(sectionH), {size of storage for }
                                          { SectionHandle}
                        ignoreSize);      {ignore storage it }
                                          { used}
END; {MyGetSectionHandleFromEvent}
```

In addition to the Section Read, Section Write, Section Cancel, and Section Scroll events, your application can also respond to the Create Publisher event. For more information on this event, as well as additional information on how to handle Apple events, see the chapter "Responding to Apple Events" in this book.

Creating the Section Record and Alias Record

Your application is responsible for creating a section record and an alias record for each publisher and subscriber section within an open document.

The section record identifies each section as a publisher or subscriber and provides identification for each section. The section record does not contain the data within the section; it describes the attributes of the section. Your application must provide its own method for associating the data within a section with its section record. Your application is also responsible for saving the data in the section.

The `alias` field of the section record contains a handle to its alias record. The alias record is a reference to the edition container from the document that contains the publisher or subscriber section. You should be familiar with the Alias Manager's conventions for creating alias records and identifying files, folders, and volumes to locate files that have been moved, copied, or restored from backup. For information on the Alias Manager, see *Inside Macintosh: Files.*

When a user saves a document, your application should store all section records and alias records in the document's resource fork. Corresponding section records and alias records should have the same resource ID.

Figure 2-9 shows a document containing a publisher and subscriber, and the corresponding section records and alias records.

Figure 2-9   A document with a publisher and subscriber and its resource fork
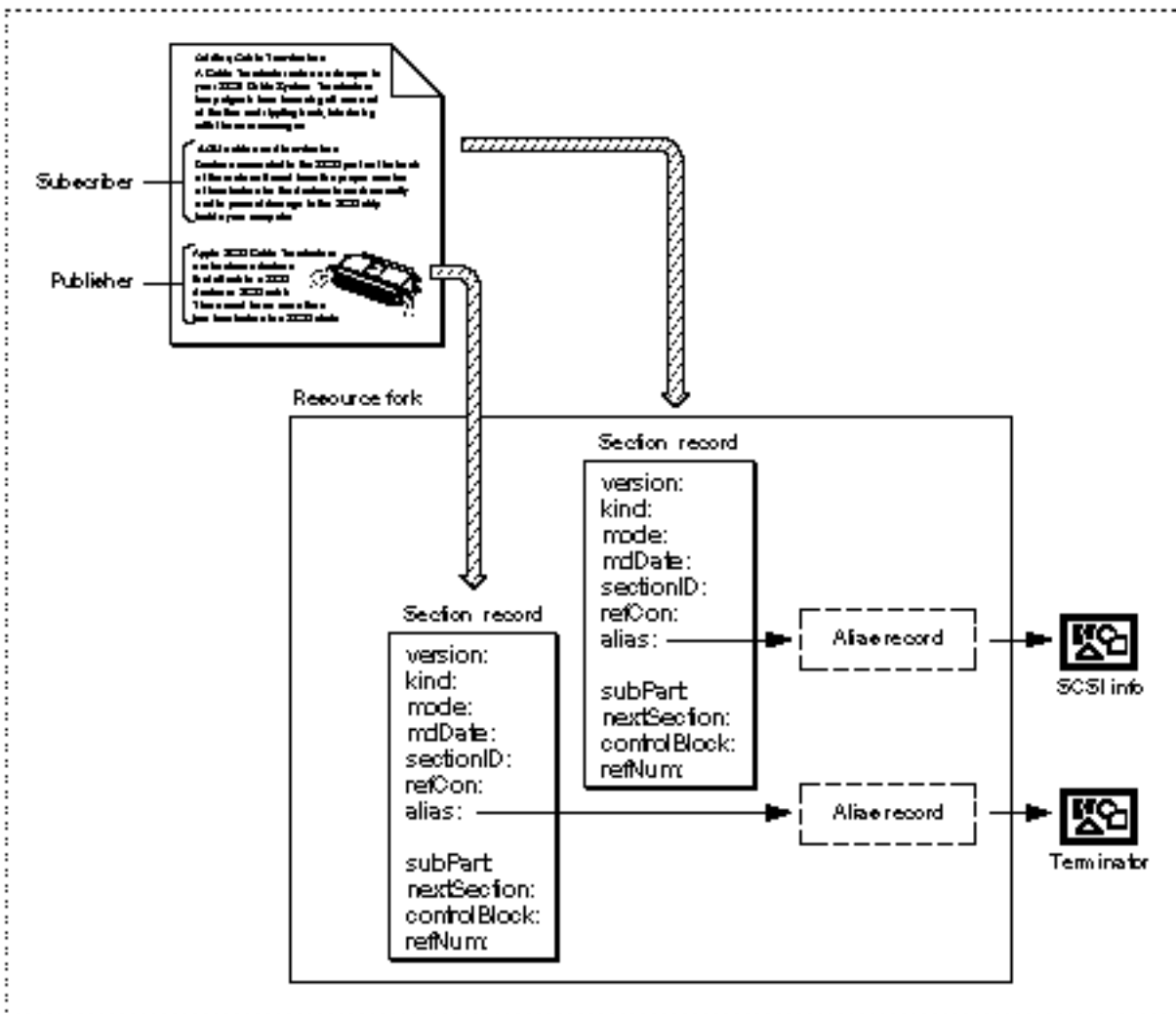
A section record contains information to identify the data contained within a section as a publisher or a subscriber, a time stamp to record the last modification of the section, and unique identification for each section. The `SectionRecord` data type defines the section record.

```
TYPE SectionRecord =
   RECORD
      version:       SignedByte;    {always 1 in 7.0}
      kind:          SectionType;   {publisher or subscriber}
      mode:          UpdateMode;    {automatic or manual}
      mdDate:        TimeStamp;     {last change in document}
      sectionID:     LongInt;       {application-specific, }
                                    { unique per document}
      refCon:        LongInt;       {application-specific}
      alias:         AliasHandle;   {handle to alias record}

      {The following fields are private and are set up by the }
      { RegisterSection function described later within this }
      { chapter. Do not modify the private fields.}

      subPart:       LongInt;       {private}
      nextSection:   SectionHandle; {private, do not use as a }
                                    { linked list}
      controlBlock:  Handle;        {may be used for comparison }
                                    { only}
      refNum:        EditionRefNum; {private}
END;
```

Field descriptions

`version`	Indicates the version of the section record, currently $01.
`kind`	Defines the section type as either publisher or subscriber with the `stPublisher` or `stSubscriber` constant.
`mode`	Indicates if editions are updated automatically or manually.
`mdDate`	Indicates which version (modification date) of the section's contents is contained within the publisher or subscriber. The `mdDate` field is set to 0 when you create a new subscriber section and to the current time when you create a new publisher. Be sure to update this field each time publisher data is modified. The section's modification date is compared to the edition's modification date to determine whether the section and the edition contain the same data. The section modification date is displayed in the publisher and subscriber options dialog boxes. See "Closing an Edition" on page 2-28 for detailed information.

sectionID	Provides a unique number for each section within a document. A simple way to implement this is to create a counter for each document that is saved to disk with the document. The counter should start at 1. The section ID is currently used as a tie breaker in the `GoToPublisherSection` function when there are multiple publishers to the same edition in a single document. The section ID should not be 0 or –1. See "Duplicating Publishers and Subscribers" on page 2-58 for information on multiple publishers.
refCon	Reference constant available for application-specific use.
alias	Contains a handle to the alias record for a particular section within a document.

Whenever the user creates a publisher or subscriber, call the `NewSection` function to create the section record and the alias record.

```
err := NewSection(container, sectionDocument, kind, sectionID,
                  initialMode, sectionH);
```

The `NewSection` function creates a new section record (either publisher or subscriber), indicates whether editions are updated automatically or manually, sets the modification date, and creates an alias record from the document containing the section to the edition container.

You can set the `sectionDocument` parameter to `NIL` if the current document has never been saved. Use the `AssociateSection` function to update the alias record of a registered section when the user names or renames a document by choosing Save As from the File menu. If you are creating a subscriber with the `initialMode` parameter set to receive new editions automatically, your application receives a Section Read event each time a new edition becomes available for this subscriber.

If an error is encountered, the `NewSection` function returns `NIL` in the `sectionH` parameter. Otherwise, `NewSection` returns a handle to the allocated section record in the `sectionH` parameter.

Set the `initialMode` parameter to the update mode for each subscriber and publisher created. You can specify the update mode using these constants:

```
CONST    sumAutomatic    = 0;       {subscriber receives new }
                                     { editions automatically}
         sumManual       = 1;       {subscriber receives new }
                                     { editions manually}
         pumOnSave       = 0;       {publisher sends new }
                                     { editions on save}
         pumManual       = 1;       {publisher does not send }
                                     { new editions until user }
                                     { request}
```

See "Using Publisher and Subscriber Options" beginning on page 2-43 for detailed information on update modes for publishers and subscribers. See Listing 2-4 beginning on page 2-33 for code that uses the `NewSection` function to create a publisher. See Listing 2-6 on page 2-40 for code that uses `NewSection` to create a subscriber.

Saving a Document Containing Sections

When saving a document that contains sections, you should write out each section record as a resource of type `'sect'` and write out each alias record as a resource of type `'alis'` with the same ID as the section record. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for detailed information on resources.

If a user closes a document that contains newly created publishers without attempting to save its contents, you should display an alert box similar to the one shown in Figure 2-10.

Figure 2-10   The new publisher alert box
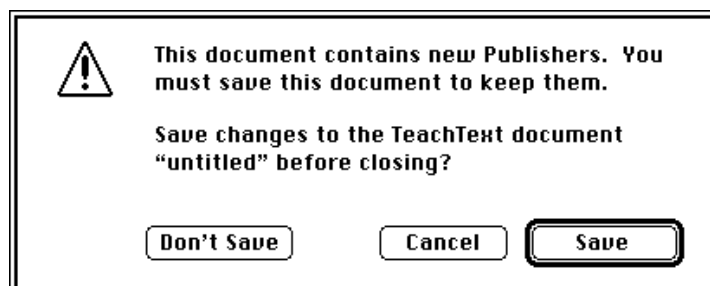
If you keep the section records and alias records for each publisher and subscriber as resources, you can use the `ChangedResource` or `WriteResource` function. If you detach the section records and alias records from each section, you need to clone the handles and use the `AddResource` function. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for detailed information on the `ChangedResource`, `WriteResource`, and `AddResource` functions.

Use the `PBExchangeFiles` function to ensure that the file ID remains the same each time you save a document that contains sections. Saving a file typically involves creating a new file (with a temporary name), writing data to it, closing it, and then deleting the original file that you are replacing. You rename the temporary file with the original filename, which leads to a new file ID. The `PBExchangeFiles` function swaps the contents of the two files (even if they are open) by getting both catalog entries and swapping the allocation pointers. If the files are open, the file control block (FCB) is updated so that the reference numbers still access the same contents (under a new name). See *Inside Macintosh: Files* for detailed information on the `PBExchangeFiles` function.

Listing 2-2 illustrates how to save a file that contains sections. If the contents of a publisher have changed since the last save, the application-defined procedure `MySaveDocument` writes the publisher's data to its edition. It then writes out to the saved document the section records and alias records of all publishers and subscribers. `MySaveDocument` calls another application-defined routine, `MyGetSectionAliasPair`, to return a handle and resource ID to a section. As described earlier, you should write out the eligible section records and alias records as resources to allow for future compatibility. There are several different techniques for saving or adding resources; this listing illustrates one technique. The section handles are still valid after using the `AddResource` function because this listing illustrates just saving, not closing, the file.

Before you write out sections, you need to see if any publisher sections share the same control block. Publishers that share the same control block share the same edition.

If a user creates an identical copy of a file by choosing Save As from the File menu and does not make any changes to this new file, you simply use the `AssociateSection` function to indicate to the Edition Manager which document a section is located in.

Listing 2-2   Saving a document containing sections

```pascal
PROCEDURE MySaveDocument(thisDocument: MyDocumentInfoPtr;
                         numberOfSections: Integer);
VAR
   aSectionH:        SectionHandle;
   copiedSectionH:   Handle;
   copiedAliasH:     Handle;
   resID:            Integer;
   thisone:          Integer;
   myErr:            OSErr;
BEGIN
   FOR thisone := 1 TO numberOfSections DO
   BEGIN
      aSectionH := MyGetSectionAliasPair(thisDocument, thisone,
                                         resID);
      IF (aSectionH^^.kind = stPublisher) &
         (aSectionH^^.mode = pumOnSave) &
         (MyCheckForDataChanged(aSectionH)) THEN
         DoWriteEdition(aSectionH);
   END; {end of for}
   {set the curResFile to the resource fork of thisDocument}
   UseResFile(thisDocument^.resForkRefNum);
   {write all section and alias records to the document}
   FOR thisone := 1 TO numberOfSections DO
   BEGIN
      {given an index, get the next section handle and resID }
      { from your internal list of sections for this file}
      aSectionH := MyGetSectionAliasPair(thisDocument, thisone,
                                         resID);
      {check for duplication of control block values}
      MyCheckForDupes(thisDocument, numberOfSections);
      {save section record to disk}
      copiedSectionH := Handle(aSectionH);
      myErr := HandToHand(copiedSectionH);
      AddResource(copiedSectionH, rSectionType, resID, '');
      {save alias record to disk}
      copiedAliasH := Handle(aSectionH^^.alias);
      myErr := HandToHand(copiedAliasH);
      AddResource(copiedSectionH, rAliasType, resID, '');
   END; {end of for}
   {write rest of document to disk}
END;
```

Opening and Closing a Document Containing Sections

When opening a document that contains sections, your application should use the `GetResource` function to get the section record and the alias record for each publisher and subscriber. Set the `alias` field of the section record to be the handle to the alias. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for detailed information on the `GetResource` function.

You also need to register each section using the `RegisterSection` function. The `RegisterSection` function informs the Edition Manager that a section exists.

```
err := RegisterSection(sectionDocument, sectionH,
                        aliasWasUpdated);
```

The `RegisterSection` function adds the section record to the Edition Manager's list of registered sections. This function assumes that the `alias` field of each section record is a handle to the alias record. The alias record is a reference to the edition container from the section's document. If the `RegisterSection` function successfully locates the edition container for a particular section, the section is registered through a shared control block. The control block is a private field in the section record.

If the `RegisterSection` function cannot find the edition container for a particular subscriber, `RegisterSection` returns the `containerNotFoundWrn` result code. If the `RegisterSection` function cannot find the edition container for a particular publisher, `RegisterSection` creates an empty edition container for the publisher in the last place the edition was located. The Edition Manager sends your application a Section Write event for that section.

When a user attempts to open a document that contains multiple publishers to the same edition, you should warn the user by displaying an alert box (see "Duplicating Publishers and Subscribers" on page 2-58).

When a user opens a document that contains a subscriber (with an update mode set to automatic), receives a new edition, and then closes the document without making any changes to the file, you should update the document and simply allow the user to close it. You do not need to prompt the user to save changes to the file.

When closing a document that contains sections, you must unregister each section (using the `UnRegisterSection` function) and dispose of each corresponding section record and alias record.

```
err := UnRegisterSection(sectionH);
```

The `UnRegisterSection` function removes the section record from the list of registered sections and unlinks itself from the shared control block.

Listing 2-3 illustrates how to open an existing file that contains sections. As described earlier, you should retrieve the section and alias resources, connect the pair through the `alias` field of the section record, and register the section with the Edition Manager. There are many different techniques for retrieving resources; this listing shows one technique. If an alias was out of date and was updated by the Alias Manager during the resolve, the Edition Manager sets the `aliasWasUpdated` parameter of the `RegisterSection` function to `TRUE`. This means that you should save the document. Additionally, your application must maintain its own list of registered sections for each open document that contains sections. You use this list to write out new editions for updated publishers within a document.

Listing 2-3   Opening a document containing sections

```
PROCEDURE MyOpenExistingDocument(thisDocument: MyDocumentInfoPtr);
VAR
    sectionH:           SectionHandle;
    aliasH:             AliasHandle;
    aliasWasUpdated:    Boolean;
    registerErr:        OSErr;
    resID:              Integer;
    theResType:         ResType;
    thisone:            Integer;
    numberOfSections:   Integer;
    aName:              Str255;
BEGIN
    UseResFile(thisDocument^.resForkRefNum);
    {find out the number of section resources}
    numberOfSections := Count1Resources(rSectionType);
    FOR thisone := 1 TO numberOfSections DO
    BEGIN
        sectionH := SectionHandle(Get1IndResource(rSectionType,
                                                  thisone));
        IF sectionH = NIL THEN  {something could be wrong with }
            MySectionErr;       { the file, handle appropriately}
        {get resource ID of the section & use same ID for alias}
        GetResInfo(Handle(sectionH), resID, theResType, aName);
        {detaching is not necessary, but it is convenient}
        DetachResource(Handle(sectionH));
        {get the alias}
        aliasH := AliasHandle(Get1Resource(rAliasType, resID));
        IF aliasH = NIL THEN    {something could be wrong with }
            MyAliasErr;         { the file, handle appropriately}
        DetachResource(Handle(aliasH));
```

```
    {connect section and alias together}
    sectionH^^.alias := aliasH;
    {register the section}
    registerErr := RegisterSection(thisDocument^.fileSpec,
                                    sectionH, aliasWasUpdated);
    {The RegisterSection function may return an error if }
    { a section is not registered. This is not a fatal error. }
    { Continue looping to register remaining sections.}
    {add this section/alias pair to your internal bookkeeping}
    MyAddSectionAliasPair(thisDocument, sectionH, resID);
    IF aliasWasUpdated THEN
        {If alias has changed, make a note of this. }
        { It's important to know this when you save.}
        MyAliasHasChanged(sectionH);
  END; {end of FOR}
END;
```

Reading and Writing a Section

Your application writes publisher data to an edition. New publisher data replaces the previous contents of the edition, making the previous edition information irretrievable. Your application reads data from an edition for each subscriber within a document.

The following sections describe how to

n use different formats to write to or read from an edition

n open an edition to initiate writing or reading

n set a format mark

n write to or read from an edition

n close an edition after successfully writing or reading data

Formats in an Edition

You can write data to an edition in several different formats. These formats are the same as scrap format types. Scrap format types are indicated by a four-character tag.

Typically, when a user copies data, you identify the scrap format types and then write the data to the scrap. With the Edition Manager, when a user decides to publish data, you identify the format types and then write the data to an edition. You can write multiple formats of the same data.

For an edition, you should write your preferred formats first. In general, to write data to an edition, your application should use either `'TEXT'` format or `'PICT'` format. This allows your application to share data with most other applications. To subscribe to an edition, your application should be able to read both `'TEXT'` and `'PICT'` files. In addition, your application can write any other private formats that you want to support.

Scrap format types are described in the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox.*

A few special formats are defined as constants.

```
CONST kPublisherDocAliasFormat    = 'alis';{alias record from the }
                                          { edition to publisher}
      kPreviewFormat              = 'prvw';{'PICT' thumbnail }
                                          { sketch}
      kFormatListFormat           = 'fmts';{lists all available }
                                          { formats}
```

The `kPublisherDocAliasFormat` (`'alis'`) format is written by the Edition Manager. It is an alias record from the edition to the publisher's document. Appended to the end of the alias is the section ID of the publisher, which the Edition Manager uses to distinguish between multiple publishers to a single edition. You should discourage users from making multiple copies of the same publisher. See "Duplicating Publishers and Subscribers" on page 2-58 for detailed information.

In addition to writing a publisher's data to an edition in the `'TEXT'` format or `'PICT'` format, your application can also write data to an edition in the `kPreviewFormat` (`'prvw'`) format. If you provide a `'prvw'` format in an edition, the Edition Manager uses it to display a preview of the edition data in the preview area of the subscriber dialog box. The `'prvw'` format has the same format as a `'PICT'` file. To draw a preview in the `'prvw'` format, the Edition Manager calls `DrawPicture` with a rectangle of 120 by 120 pixels. (See *Inside Macintosh: Imaging* for more information about `DrawPicture`.) Your application should provide data in a `'prvw'` format so that the data displays well in a rectangle of this size. Your application can also use this preview to display subscriber data within a document (to save space).

If your application does not provide a preview in the `'prvw'` format for an edition, the Edition Manager attempts to provide a preview by using the edition's `'PICT'` format. To draw a preview in the `'PICT'` format, the Edition Manager examines the picture's bounding rectangle and calls `DrawPicture` with a rectangle that scales the picture proportionally and centers it in a 120-by-12-pixel area.

The `kFormatListFormat` (`'fmts'`) format is a virtual format that is read but never written. It is a list of all the formats and their lengths. Applications can use this format in place of the `EditionHasFormat` function (described in "Choosing Which Edition Format to Read" on page 2-41), which provides a procedural interface to determine which formats are available.

If your application can read two or more of the available formats, use `'fmts'` to determine the priority of these formats for a particular edition. The order of `'fmts'` reflects the order in which the formats were written.

The `FormatsAvailable` data type defines a record for the `'fmts'` format.

```
TYPE FormatsAvailable = ARRAY[0..0] OF
    RECORD
        theType:    FormatType;     {format type for an edition}
        theLength:  LongInt;        {length of edition format }
                                    { type}
    END;
```

For example, an edition container may have a format type `'TEXT'` of length 100, and a format type `'styl'` of length 32. A subscriber to this edition can open it and then read the format type `'fmts'` to list all available formats. In this example, it returns 16 bytes: `'TEXT'` $00000064 `'styl'` $00000020.

Opening an Edition

For a publisher, use the `OpenNewEdition` function to initiate the writing of data to an edition. (Note that the edition container must already exist before you initiate writing; see "Creating the Edition Container" beginning on page 2-32.)

```
err := OpenNewEdition(publisherSectionH, fdCreator,
                      publisherSectionDocument, refNum);
```

The `publisherSectionH` parameter is the publisher section that you are writing to the edition. The `fdCreator` parameter is the Finder creator type of the new edition. (The edition container file already has a creator type; you can specify the same creator type or establish a new creator type for the edition.)

The `publisherSectionDocument` parameter specifies the document that contains the publisher. This parameter is used to create an alias from the edition to the publisher's document. If you pass `NIL` for `publisherSectionDocument`, an alias is not made in the edition file. The `refNum` parameter returns the reference number for the edition.

For a subscriber, use the `OpenEdition` function to initiate the reading of data from an edition.

```
err := OpenEdition(subscriberSectionH, refNum);
```

The `subscriberSectionH` parameter is a handle to the section record for a given section. The `refNum` parameter returns the reference number for the edition.

The user may rename or move the edition in the Finder. Before writing to or reading data from an edition, the Edition Manager verifies the name of the edition. This process is referred to as *synching* or synchronization. Synching ensures that the Edition Manager's existing edition names correspond to the Finder's existing edition names by updating the control block.

Format Marks

Each format has its own mark. The mark indicates the next position of a read or write operation. Initially, a mark automatically defaults to 0. After reading or writing data, the format mark is set past the last position written to or read from. The mark is similar to the File Manager's current read or write position marker for a data fork. Any time that an edition is open (after calling the `OpenEdition` or the `OpenNewEdition` function), any of the marks for each format can be queried or set.

To set the current mark for a section format to a new location, use the `SetEditionFormatMark` function.

```
err := SetEditionFormatMark(whichEdition, whichFormat,
                           setMarkTo);
```

To get the current mark for a format in an edition file, use the `GetEditionFormatMark` function.

```
err := GetEditionFormatMark(whichEdition, whichFormat,
                           currentMark);
```

Reading and Writing Edition Data

The Edition Manager allows you to read or write data a few bytes at a time (as with a data fork of a Macintosh file) instead of in one block (as with the Scrap Manager). You can read sequentially by setting the mark to 0 and repeatedly calling read, or you can jump to a specific offset by setting the mark there. The Edition Manager also adds the capability to stream multiple formats by keeping a separate mark for each format. This allows you to write a few bytes of one format and then write a few bytes of another format, and so forth.

Once you have opened the edition container for a particular publisher, you can begin writing data to the edition. Use the `WriteEdition` function to write publisher data to an edition.

```
err := WriteEdition(whichEdition, whichFormat, buffPtr, buffLen);
```

The `WriteEdition` function writes the specified format (beginning at the current mark for that format type) from the buffer pointed to by the `buffPtr` parameter up to `buffLen` bytes.

After you open the edition container for a subscriber and determine which formats to read, use the `ReadEdition` function to read edition data.

```
err := ReadEdition(whichEdition, whichFormat, buffPtr, buffLen);
```

The `ReadEdition` function reads the data with the specified format (`whichFormat`) from the edition into the buffer. The `ReadEdition` function begins reading at the current mark for that format and continues to read up to `buffLen` bytes. The actual number of bytes read is returned in the `buffLen` parameter. Once the `buffLen` parameter returns a value smaller than the value you have specified, there is no additional data to read, and the `ReadEdition` function returns a `noErr` result code.

Note

The Translation Manager (if it is available) attempts implicit translation under certain circumstances. For instance, it does so when your application attempts to read from an edition a format type that is not in the edition. In this case, the Translation Manager attempts to translate the data into the requested format. For more information, see the chapter "Translation Manager" in *Inside Macintosh: More Macintosh Toolbox.* u

Closing an Edition

When you are done writing to or reading data from an edition, call the `CloseEdition` function.

```
err := CloseEdition(whichEdition, successful);
```

Each time a user edits a publisher within a document, you must update the modification date in the section record (even if the data is not yet written). When the update mode is set to Manually, the user can compare the modification dates for a publisher and its edition in the publisher options dialog box. One modification date indicates when the publisher last wrote data to the edition, and the other modification date indicates when the publisher section was last edited.

If the `successful` parameter for a publisher is `TRUE`, the `CloseEdition` function makes the newly written data available to subscribers and sets the modification date in the `mdDate` field of the edition to correspond to the modification date of the publisher's section record. If the two dates differ, the Edition Manager sends a Section Read event to all current subscribers.

If the `successful` parameter for a subscriber is `TRUE`, the `CloseEdition` function sets the modification date of the subscriber's section record to correspond to the modification date of the edition.

If you cannot successfully read from or write data to an edition, set the `successful` parameter to `FALSE`. For a publisher, data is not written to the edition, but it should still be saved with the document that contains the section. When the document is next saved, data can then be written to the edition. See "Closing an Edition After Reading or Writing" on page 2-88 for additional information on the `CloseEdition` function.

Creating a Publisher

You need to support a Create Publisher menu command in the Edit menu. When a user selects a portion of a document and chooses Create Publisher from this menu, you should display the publisher dialog box on the user's screen. The Create Publisher menu command should remain dimmed until the user selects a portion of a document.

Use the `NewPublisherDialog` function to display the publisher dialog box on the user's screen. This function is similar to the `CustomPutFile` procedure described in the chapter "Standard File Package" in *Inside Macintosh: Files*.

```
err := NewPublisherDialog(reply);
```

The dialog box contains space for a preview (a thumbnail sketch) of the edition and a space for the user to type the name of the edition in which to write the publisher data. Figure 2-11 illustrates a sample publisher dialog box.

Figure 2-11   A sample publisher dialog box
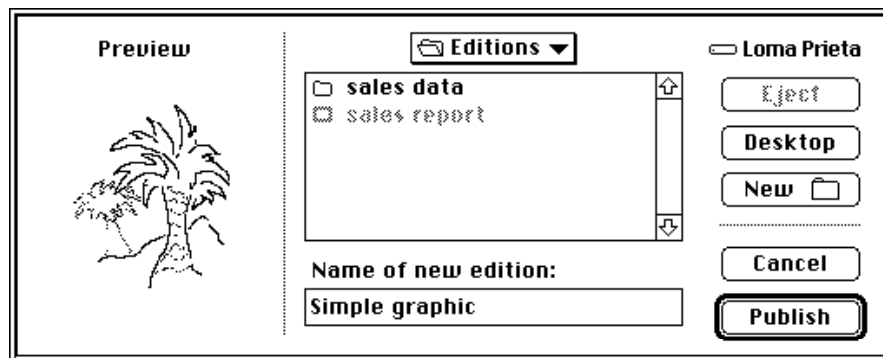
The `NewPublisherDialog` function displays the preview (provided by your application), displays a text box with the default name of the edition (provided by your application), and handles all user input until the user clicks Publish or Cancel.

You pass a new publisher reply record as a parameter to the `NewPublisherDialog` function.

```
TYPE NewPublisherReply =
   RECORD
      canceled:   Boolean;            {user clicked Cancel}
      replacing:  Boolean;            {user chose existing }
                                      { filename for an edition}
      usePart:    Boolean;            {always FALSE in version 7.0}
      preview:    Handle;             {handle to 'prvw', 'PICT', }
                                      { 'TEXT', or 'snd ' data}
      previewFormat:                  {type of preview}
                  FormatType;
      container:  EditionContainerSpec;{initially, default name }
                                       { and location of edition; }
                                       { on return, edition name & }
                                       { location chosen by the }
                                       { user to publish data to}
   END;
```

You fill in the `usePart`, `preview`, `previewFormat`, and `container` fields of the new publisher reply record.

Always set the `usePart` field to `FALSE`. The `preview` field should contain either `NIL` or the data to display in the preview. The `previewFormat` field should contain `'PICT'`, `'TEXT'`, `'snd '`, or `'prvw'`.

Set the `container` field to be the default name and folder for the edition. The default name should reflect the data contained in the publisher. For example, if a user publishes a bar chart of sales information entitled "sales data," then the default name for the edition could also be "sales data." Otherwise, you should use the document name followed by a hyphen (-) and a number to establish uniqueness. For example, your default name could be "January Totals - 3."

If the document has not been saved, the default name should be "untitled edition *<n>*" where *n* is a number to establish uniqueness. The default folder should be the same as the edition for the last publisher created in the same document. If this is the first publisher in the document, the default folder should be the same folder that the document is in.

The `canceled` field of the new publisher reply record indicates whether the user clicked Cancel. The `replacing` field indicates whether the user chose to replace an existing edition file. If `replacing` returns `FALSE`, call the `CreateEditionContainerFile` function to create an edition file.

The `container` field is of data type `EditionContainerSpec`.

```
TYPE EditionContainerSpec =
   RECORD
      theFile:       FSSpec;       {record that identifies the }
                                   { file to contain edition data}
      theFileScript: ScriptCode; {script code of filename}
      thePart:       LongInt;     {which part of file, }
                                   { always kPartsNotUsed}
      thePartName:   Str31;       {not used in version 7.0}
      thePartScript: ScriptCode; {not used in version 7.0}
   END;
```

The field `theFile` is a file system specification record, a data structure of type `FSSpec`. You identify the edition using a volume reference number, directory ID, and filename. When specifying an edition, follow the standard conventions described in *Inside Macintosh: Files.*

After filling in the fields of the new publisher reply record, pass it as a parameter to the `NewPublisherDialog` function, which displays the publisher dialog box.

```
err := NewPublisherDialog(reply);
```

After displaying the publisher dialog box, use the `CreateEditionContainerFile` function to create the edition container, and then use the `NewSection` function to create the section record and the alias record. See the next section, "Creating the Edition Container," and "Creating the Section Record and Alias Record" on page 2-15 for detailed information.

The following code segment illustrates how your application might respond to the user choosing the Create Publisher menu item. In this case, the code sets up the preview for the edition, sets the default name for the edition container, and calls an application-defined function (`DoNewPublisher`, shown in Listing 2-4 on page 2-33) to display the publisher dialog box on the user's screen. An application might call the `DoNewPublisher` function in response to the user's choosing Create Publisher from the Edit menu or in response to handling the Create Publisher event. The chapter "Responding to Apple Events" in this book gives an example of a handler for the Create Publisher event.

```
VAR
   thisDocument:      MyDocumentInfoPtr;
   promptForDialog:   Boolean;
   preview:           Handle;
   previewFormat:     FormatType;
   defaultLocation:   EditionContainerSpec;
   myErr:             OSErr;
```

```
BEGIN
    {Get a preview to show the user. The MyGetPreviewForSelection }
    { function returns a handle to the preview.}
    preview := MyGetPreviewForSelection(thisDocument);
    previewFormat := 'TEXT';
    defaultLocation := MyGetDefaultEditionSpec(thisDocument);
    promptForDialog := TRUE;
    myErr := DoNewPublisher(thisDocument, promptForDialog, preview,
                            previewFormat, defaultLocation);
END;
```

Creating the Edition Container

Use the `CreateEditionContainerFile` function to create an edition container to
hold the publisher data.

```
err := CreateEditionContainerFile(editionFile, fdCreator,
                                  editionFileNameScript);
```

This function creates an edition container. The edition container is empty (that is, it does
not contain any formats) at this time.

To associate an icon with the edition container, create the appropriate entries for the icon
in your application's bundle. See the chapter "Finder Interface" in *Inside Macintosh:
Macintosh Toolbox Essentials* for additional information. Depending on the contents of the
edition, the file type will be `'edtp'` (for graphics), `'edtt'` (for text), or `'edts'`
(for sound).

After creating the edition container, use the `NewSection` function to create the section
record and alias record for the section.

Listing 2-4 illustrates how to create a publisher. The `DoNewPublisher` function shown
in the listing is a function provided by an application. Note that an application might call
the `DoNewPublisher` function in response to the user's choosing the Create Publisher
command or in response to the Create Publisher event. The chapter "Responding to
Apple Events" in this book gives an example of a handler for the Create Publisher event.

The parameters to the `DoNewPublisher` function include a pointer to information
about the document, a Boolean value that indicates if the function should display the
new publisher dialog box, the preview for the edition, the preview format, and an
edition container.

The function displays the publisher dialog box if requested, letting the user accept or
change the name of the edition and the location where the edition should reside. Use the
`CreateEditionContainerFile` function to create the edition with the given name
and location. Use the `NewSection` function to create a new section for the publisher.

After the section is created, you must write out the edition data. Be sure to add the newly created section to your list of sections for this document. There are several different techniques for creating publishers and unique IDs; this listing displays one technique.

After creating the edition container and creating a new section record, the `DoNewPublisher` function calls another application-defined routine, `DoWriteEdition`, to open the edition and write data to it.

Listing 2-4   Creating a publisher

```
FUNCTION DoNewPublisher(thisDocument: MyDocumentInfoPtr;
                        promptForDialog: Boolean;
                        preview: Handle;
                        previewFormat: FormatType;
                        editionSpec: EditionContainerSpec)
                        : OSErr;
VAR
   getLastErr, dialogErr:  OSErr;
   createErr, sectionErr:  OSErr;
   resID:                  Integer;
   thisSectionH:           SectionHandle;
   reply:                  NewPublisherReply;
BEGIN
   {set up info for new publisher reply record}
   reply.replacing := FALSE;
   reply.usePart := FALSE;
   reply.preview := preview;
   reply.previewFormat := previewFormat;
   reply.container := editionSpec;
   IF promptForDialog THEN
   BEGIN {user interaction is allowed}
      {display dialog box and let user select}
      dialogErr := NewPublisherDialog(reply);
      {dispose of preview data handle}
      DisposeHandle(reply.preview);
      IF dialogErr <> noErr THEN MyErrHandler(dialogErr);
      IF reply.canceled THEN
      BEGIN    {do nothing if user canceled}
         DoNewPublisher := userCanceledErr;
         EXIT(DoNewPublisher);
      END;
   END;  {of promptForDialog}
```

```
IF NOT reply.replacing THEN
BEGIN
   {if user isn't replacing an existing file, create a new one}
   createErr :=
      CreateEditionContainerFile(reply.container.theFile,
                                 kAppSignature,
                                 reply.container.theFileScript);
   IF createErr <> noErr THEN
   BEGIN
      DoNewPublisher := errAEPermissionDenied;
      EXIT(DoNewPublisher);
   END;
END; {of not replacing}
{Advance counter to make a new unique sectionID for this }
{ document. It is not required that you equate section IDs }
{ with resources.}
thisDocument^.nextSectionID := thisDocument^.nextSectionID + 1;
{create a publisher section}
sectionErr := NewSection(reply.container,
                         thisDocument^.fileSpecPtr,
                         stPublisher,
                         thisDocument^.nextSectionID,
                         pumOnSave, thisSectionH);
IF (sectionErr <> noErr) & (sectionErr <> multiplePublisherWrn)
      & (sectionErr <> notThePublisherWrn) THEN
   MyErrHandler(sectionErr);
resID := thisDocument^.nextSectionID;
{add this section/alias pair to app's internal bookkeeping}
MyAddSectionAliasPair(thisDocument, thisSectionH, resID);
{write out first edition}
DoWriteEdition(thisSectionH);
{Remember that the section and alias records need to be }
{ saved as resources when the user saves the document.}
{set the function result appropriately}
DoNewPublisher := MyGetLastError;
END;
```

Opening an Edition Container to Write Data

Several routines are required to write (publish) data from a publisher to an edition container. (For information on creating an edition container, see the previous section.) Before writing data to an edition, you must use the `OpenNewEdition` function. This function should be used only for a publisher within a document. Use this function to initiate the writing of data to an edition.

```
err := OpenNewEdition(publisherSectionH, fdCreator,
                      publisherSectionDocument, refNum);
```

A user may try to save a document containing a publisher that is unable to write its data to an edition—because another publisher (that shares the same edition) is writing, another subscriber (that shares the same edition) is reading, or a publisher located on another computer is registered to the section. In such a case, you may decide to refrain from writing to the edition so that the user does not have to wait. You should also refrain from displaying an error to the user. The contents of the publisher are saved to disk with the document. The next time that the user saves the document, you can write the publisher data to the edition. You should display an alert box to discourage users from making multiple copies of the same publisher and pasting them in the same or other documents (see "Duplicating Publishers and Subscribers" on page 2-58).

If a user clicks Send Edition Now within the publisher options dialog box (to write publisher data to an edition manually), and the publisher is unable to write its data to its edition (for any of the reasons outlined above), you should display an error message.

After you are finished writing data to an edition, use the `CloseEdition` function to close the edition.

Listing 2-5 illustrates how to write data to an edition. For an existing edition container, you must open the edition, write each format using the `WriteEdition` function, and close the edition using the `CloseEdition` function. This listing shows how to write text only. If the edition is written successfully, subscribers receive Section Read events.

Listing 2-5   Writing data to an edition

```pascal
PROCEDURE DoWriteEdition(thePublisher: SectionHandle);
VAR
   eRefNum:       EditionRefNum;
   openErr:       OSErr;
   writeErr:      OSErr;
   closeErr:      OSErr;
   thisDocument:  MyDocumentInfoPtr;
   textHandle:    Handle;
BEGIN
   {find out which document this section belongs to}
   thisDocument := MyFindDocument(thePublisher);
   {open edition for writing}
   openErr := OpenNewEdition(thePublisher, kAppSignature,
                             thisDocument^.fileSpecPtr, eRefNum);
   IF openErr <> noErr THEN
      MyErrHandler(openErr);{handle error and exit}
   {get the text data to write}
   textHandle := MyGetTextInSection(thePublisher, thisDocument);
   {write out text data}
   HLock(textHandle);
   writeErr := WriteEdition(eRefNum, 'TEXT', textHandle^,
                            GetHandleSize(textHandle));
   HUnLock(textHandle);
   IF writeErr <> noErr THEN
   BEGIN
      {There were problems writing; simply close the edition. }
      { When successful = FALSE, the edition data <> section }
      { data. Note: this isn't fatal or bad; it just means }
      { that the data wasn't written and no Section Read events }
      { will be generated.}
      closeErr := CloseEdition(eRefNum, FALSE);
   END
   ELSE
   BEGIN
      {The write was successful; now close the edition. }
      { When successful = TRUE, the edition data = section data.}
      { This edition is now available to any subscribers. }
      { Section Read events will be sent to current subscribers.}
      closeErr := CloseEdition(eRefNum, TRUE);
   END;
END;
```

Creating a Subscriber

You need to create a Subscribe To menu command in the Edit menu. When a user chooses Subscribe To from this menu, your application should display the subscriber dialog box on the user's screen.

Use the `NewSubscriberDialog` function to display the subscriber dialog box on the user's screen. This function is similar to the `CustomGetFile` procedure described in the chapter "Standard File Package" in *Inside Macintosh: Files*.

To create a subscriber, you must get information from the user, such as the name of the edition being subscribed to. The dialog box displays a listing of all available editions and allows the user to see a preview (thumbnail sketch) of the edition selected. Figure 2-12 shows a sample subscriber dialog box.

Figure 2-12   A sample subscriber dialog box
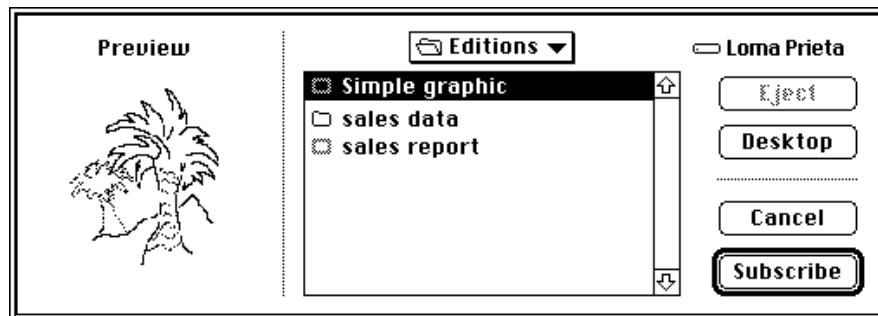
The subscriber dialog box allows the user to choose an edition to subscribe to. The `NewSubscriberDialog` function handles all user interaction until a user clicks Subscribe or Cancel. When a user selects an edition container, the Edition Manager accesses the preview for the edition container (if it is available) and displays it.

You pass a new subscriber reply record as a parameter to the `NewSubscriberDialog`
function.

```
TYPE  NewSubscriberReply =
      RECORD
          canceled:   Boolean;      {user clicked Cancel}
          formatsMask:SignedByte; {formats required}
          container:  EditionContainerSpec;{initially, default }
                                    { name & location of edition }
                                    { to subscribe to; on return, }
                                    { edition name & location }
                                    { chosen by the user}
      END;
```

The `canceled` field returns a Boolean value of `TRUE` if the user clicked Cancel. To
indicate which edition format types (text, graphics, or sound) your application can read,
you set the `formatsMask` field to one or more of these constants:

```
CONST kPICTformatMask   = 1;            {can subscribe to 'PICT'}
      kTEXTformatMask   = 2;            {can subscribe to 'TEXT'}
      ksndFormatMask    = 4;            {can subscribe to 'snd '}
```

To support a combination of formats, add the constants together. For example, a
`formatsMask` of 3 displays both graphics and text edition format types in the
subscriber dialog box.

The `container` field is of data type `EditionContainerSpec`. You must initialize the
`container` field with the default edition volume reference number, directory ID,
filename, and part. To do so, use the `GetLastEditionContainerUsed` function to
obtain the name of the last edition displayed in the dialog box.

```
err := GetLastEditionContainerUsed(container);
```

This function returns the last edition container for which a new section was created
using the `NewSection` function. If there is no last edition, or if the edition was deleted,
`GetLastEditionContainerUsed` still returns the correct volume reference number
and directory ID to use, but leaves the filename blank and returns the `fnfErr`
result code.

The `container` field is of data type `EditionContainerSpec`.

```
TYPE EditionContainerSpec =
   RECORD
      theFile:          FSSpec;          {file containing edition }
                                         { data}
      theFileScript:    ScriptCode;      {script code of filename}
      thePart:          LongInt;         {which part of file, }
                                         { always kPartsNotUsed}
      thePartName:      Str31;           {reserved}
      thePartScript:    ScriptCode;      {reserved}
   END;
```

The field `theFile` is of type `FSSpec`. See *Inside Macintosh: Files* for further information on file system specification records.

After filling in the fields of the new subscriber reply record, pass it as a parameter to the `NewSubscriberDialog` function, which displays the subscriber dialog box.

```
err := NewSubscriberDialog(reply);
```

After displaying the subscriber dialog box, call the `NewSection` function to create the section record and the alias record. See "Creating the Section Record and Alias Record" beginning on page 2-15 for detailed information.

If the subscriber is set up to receive new editions automatically (not manually), the Edition Manager sends your application a Section Read event. Whenever your application receives a Section Read event, it should read the contents of the edition into the subscriber.

Listing 2-6 illustrates how to create a subscriber. As described earlier, you must set up and display the subscriber dialog box to allow the user to subscribe to any of the available editions. After your application creates a subscriber, your application receives a Section Read event to read in the data being subscribed to. Be sure to add the newly created section to your list of sections for this file. There are many different techniques for creating subscribers and unique IDs; this listing displays one technique.

Listing 2-6   Creating a subscriber

```
PROCEDURE DoNewSubscriber(thisDocument: MyDocumentInfoPtr);
VAR
   getLastErr:    OSErr;
   dialogErr:     OSErr;
   sectionErr:    OSErr;
   resID:         Integer;
   thisSectionH:  SectionHandle;
   reply:         NewSubscriberReply;
BEGIN
   {put default edition name into reply record}
   getLastErr := GetLastEditionContainerUsed(reply.container);
   {can subscribe to pictures or text}
   reply.formatsMask := kPICTformatMask + kTEXTformatMask;
   {display dialog box & let user select edition to subscribe to}
   dialogErr := NewSubscriberDialog(reply);
   IF dialogErr <> noErr THEN
      MyErrHandler(dialogErr);   {handle error and exit}
   IF reply.canceled THEN
      EXIT(DoNewSubscriber);     {do nothing if user canceled}
   {Advance counter to make a new unique sectionID for this }
   { document. It is not necessary to equate section IDs with }
   { resources.}
   thisDocument^.nextSectionID := thisDocument^.nextSectionID + 1;
   {create a subscriber section}
   sectionErr := NewSection(reply.container,
                            thisDocument^.fileSpecPtr,
                            stSubscriber,
                            thisDocument^.nextSectionID,
                            sumAutomatic, thisSectionH);
   IF sectionErr <> noErr THEN
      MyErrHandler(sectionErr);{handle error and exit}
   resID := thisDocument^.nextSectionID;
   {add this section/alias pair to app's internal bookkeeping}
   MyAddSectionAliasPair(thisDocument, thisSectionH, resID);
   {Remember that you will receive a Section Read event to read }
   { in the edition that you just subscribed to because the }
   { initial mode is set to sumAutomatic.}
   {Remember that the section and alias records need to be saved }
   { as resources when the user saves the document.}
END;
```

Opening an Edition Container to Read Data

Before reading data from an edition, you must use the `OpenEdition` function. Your application should only use this function for a subscriber. Use this function to initiate the reading of data from an edition.

```
err := OpenEdition(subscriberSectionH, refNum);
```

As a precaution, you should retain the old data until the user can no longer undo. This allows you to undo changes if the user requests it.

Your application can supply a procedure such as `DoReadEdition` to read in data from the edition to a subscriber. When your application opens a document containing a subscriber that is set up to receive new editions automatically, the Edition Manager sends you a Section Read event if the edition has been updated. The Section Read event supplies the handle to the section that requires updating. Listing 2-7, shown in the next section, provides an example of reading data from an edition.

Choosing Which Edition Format to Read

After your application opens the edition container for a subscriber, it can look in the edition for formats that it understands. To accomplish this, use the `EditionHasFormat` function.

```
err := EditionHasFormat(whichEdition, whichFormat, formatSize);
```

The `EditionHasFormat` function returns the `noTypeErr` result code if a requested format is not available. If the requested format is available, this function returns the `noErr` result code, and the `formatSize` parameter contains the size of the data in the specified format or `kFormatLengthUnknown` (–1), which signifies that the size is unknown.

Note

The Translation Manager (if it is available) attempts implicit translation under certain circumstances. For instance, it does so when your application attempts to read from an edition a format type that is not in the edition. In this case, the Translation Manager attempts to translate the data into the requested format. For more information, see the chapter "Translation Manager" in *Inside Macintosh: More Macintosh Toolbox.* u

After your application opens the edition container and determines which formats it wants to read, call the `ReadEdition` function to read in the edition data. See "Reading and Writing Edition Data" on page 2-27 for detailed information.

After you have completed writing the edition data into the subscriber section, call the `CloseEdition` function to close the edition. See "Closing an Edition" on page 2-28 for detailed information.

Listing 2-7 illustrates how to read data from an edition. As described earlier, you must open the edition, determine which formats to read, use the ReadEdition function to read in data, and then use the CloseEdition function to close the edition. This listing shows how to read only text.

Listing 2-7   Reading in edition data

```
PROCEDURE DoReadEdition(theSubscriber: SectionHandle);
VAR
    eRefNum:          EditionRefNum;
    openErr:          OSErr;
    readErr:          OSErr;
    closeErr:         OSErr;
    thisDocument:     MyDocumentInfoPtr;
    textHandle:       Handle;
    formatLen:        Size;
BEGIN
    {find out which document this section belongs to}
    thisDocument := MyFindDocument(theSubscriber);
    {open the edition for reading}
    openErr := OpenEdition(theSubscriber, eRefNum);
    IF openErr <> noErr THEN
        MyErrHandler(openErr);   {handle error and exit}
    {look for 'TEXT' format}
    IF EditionHasFormat(eRefNum, 'TEXT', formatLen) = noErr THEN
    BEGIN
        {get the handle of location to read to}
        textHandle := MyGetTextInSection(theSubscriber,
                                        thisDocument);
        SetHandleSize(textHandle, formatLen);
        HLock(textHandle);
        readErr := ReadEdition(eRefNum, 'TEXT', textHandle^,
                              formatLen);
        MyUpdateSubscriberText(theSubscriber, textHandle, readErr);
        HUnLock(textHandle);
        IF readErr = noErr THEN
        BEGIN
            {The read was successful; now close the edition. When }
            { successful = TRUE, the section data = edition data.}
            closeErr := CloseEdition(eRefNum, TRUE);
            EXIT(DoReadEdition);
        END;
    END;  {of EditionHasFormat}
```

```
   {'TEXT' format wasn't found or read error; just close }
   { the edition. FALSE tells the Edition Manager that your }
   { application did not get the latest edition.}
   closeErr := CloseEdition(eRefNum, FALSE);
END;
```

Using Publisher and Subscriber Options

You can allow users to set several special options associated with publishers and subscribers. To set these preferences, users change settings in two dialog boxes provided by the Edition Manager: publisher options and subscriber options. To make these dialog boxes available to the user, provide a command in the Edit menu that toggles between Publisher Options (when the user has selected a publisher within a document) and Subscriber Options (when a user has selected a subscriber within a document).

When a user chooses one of these menu commands, you need to display the appropriate dialog box. Use the `SectionOptionsDialog` function to display the publisher options or subscriber options dialog box on the user's screen.

```
err := SectionOptionsDialog(reply);
```

Each dialog box contains information regarding the section and its edition. Figure 2-13 shows the publisher options dialog box with the update mode set to On Save.

Figure 2-13   The publisher options dialog box with update mode set to On Save
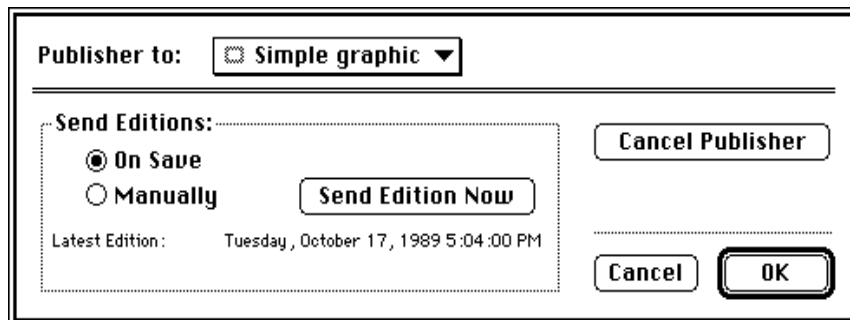
Figure 2-14 shows the publisher options dialog box with the update mode set to Manually.

Figure 2-14   The publisher options dialog box with update mode set to Manually
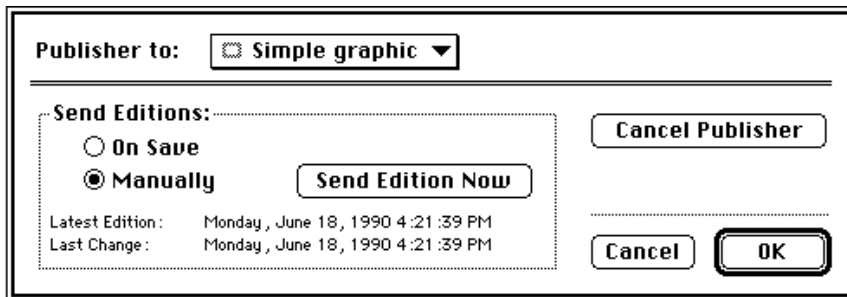
As a shortcut for the user, you should display the publisher options dialog box when the user double-clicks a publisher section in a document.

Figure 2-15 shows the subscriber options dialog box with the update mode set to Automatically.

Figure 2-15   The subscriber options dialog box with update mode set to Automatically
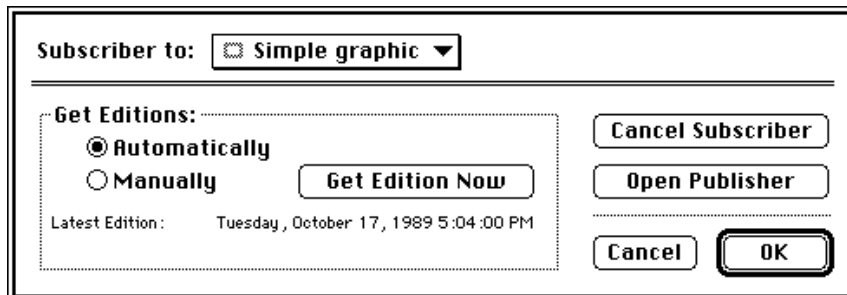
Figure 2-16 shows the subscriber options dialog box with the update mode set to Manually.

Figure 2-16   The subscriber options dialog box with update mode set to Manually
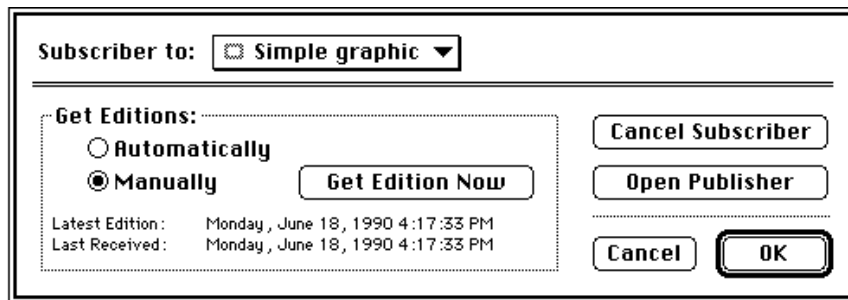
As a shortcut for the user, you should display the subscriber options dialog box when the user double-clicks a subscriber section in a document.

You pass a section options reply record as a parameter to the SectionOptionsDialog function.

```
TYPE SectionOptionsReply =
   RECORD
      canceled:   Boolean;          {user clicked Cancel}
      changed:    Boolean;          {changed section record}
      sectionH:   SectionHandle;    {handle to the specified }
                                    { section record}
      action:     ResType;          {action codes}
   END;
```

Set the sectionH parameter to the handle to the section record for the section the user selected.

Upon return of the SectionOptionsDialog function, the canceled and changed fields are set. If the canceled field is set to TRUE, the user clicked Cancel. Otherwise, this field is set to FALSE. If the changed field is set to TRUE, the section record is changed. For example, the user may have changed the update mode.

The SectionOptionsDialog function returns in the action parameter the code for one of five user actions. The function dismisses the publisher and subscriber options dialog boxes after the user clicks a button.

n Action code is 'read' for a click of the Get Edition Now button.

n Action code is 'writ' for a click of the Send Edition Now button.

n Action code is 'goto' for a click of the Open Publisher button.

n Action code is 'cncl' for a click of the Cancel Publisher or Cancel Subscriber button.

n Action code is '   ' ($20202020) for a click of the OK button.

Listing 2-8 shows an example of how your application can respond to the action codes received from the section options reply record. You can use several different techniques for this purpose; this listing shows one technique.

Listing 2-8   Responding to action codes

```
PROCEDURE DoOptionsDialog(theSection: SectionHandle);
VAR
   reply:            SectionOptionsReply;
   theEditionInfo:   EditionInfoRecord;
   action:           ResType;
   sodErr, geiErr:   OSErr;
   gpiErr, gpsErr:   OSErr;

BEGIN
   reply.sectionH := theSection;
   {display options dialog box}
   sodErr := SectionOptionsDialog(reply);
   {determine what the user did and handle appropriately}
   IF reply.canceled THEN  {user selected the Cancel button}
      EXIT(DoOptionsDialog);
   IF reply.changed THEN
      {the section record has changed; make note of this}
      MySectionHasChanged(theSection);
      {if you customize, you may want to do some }
      { post-processing now}
   {get the action code}
   action := reply.action;
   IF (action = 'read') THEN
   BEGIN    {user selected Get Edition Now button}
      DoReadEdition(theSection);
      EXIT(DoOptionsDialog);
   END;
```

```
   IF (action = 'writ') THEN
   BEGIN    {user selected Send Edition Now button}
      DoWriteEdition(theSection);
      EXIT(DoOptionsDialog);
   END;
   IF (action = 'goto') THEN
   BEGIN    {user selected Open Publisher button}
      geiErr := GetEditionInfo(theSection, theEditionInfo);
      IF geiErr <> noErr THEN
         MyErrHandler(geiErr);{handle error and exit}
      gpsErr := GotoPublisherSection(theEditionInfo.container);
      IF gpsErr <> noErr THEN
         MyErrHandler(gpsErr);{handle error and exit}
      EXIT(DoOptionsDialog);
   END;
   IF (action = 'cncl') THEN
   BEGIN {User selected Cancel Publisher or Cancel Subscriber }
      { button. Call the UnRegisterSection function and dispose }
      { of the section record and alias record.}
      EXIT(DoOptionsDialog);
   END;
END;
```

The following sections describe the features of the publisher and subscriber options
dialog boxes.

Publishing a New Edition While Saving or Manually

By default, your application should write publisher data to an edition each time the user
saves the document and the contents of the publisher differ from the latest edition. In the
publisher options dialog box, the user can choose to write new data to an edition each
time the document is saved (by clicking On Save) or only upon the user's specific
request (by clicking Manually).

When the update mode is set to manual, a user must click the Send Edition Now button
in the publisher options dialog box to write publisher data to an edition. When a user
clicks this button, the section options reply record contains the action code 'writ'. In
this case, you should write out the new edition. Writing to an edition manually is useful
when a user tends to save a document numerous times while revising it.

Each time the user saves the document, check the update mode of the publisher section.
If the publisher section sends its data to an edition when the document is saved, check
whether the publisher data has changed since it was last written to the edition. If so,
write the publisher's data to the new edition.

In addition, you may also support a Stop All Editions menu command to provide a method for temporarily suspending all update activity. See "Introduction to Publishers, Subscribers, and Editions" beginning on page 2-4 for additional information.

Subscribing to an Edition Automatically or Manually

By default, your application should subscribe to an edition each time new edition data becomes available. In the subscriber options dialog box, the user can choose to read new data from an edition as the data is available (by clicking Automatically) or only upon the user's specific request (by clicking Manually).

When the update mode is set to manual, the user must click the Get Edition Now button in the subscriber options dialog box to receive new editions. When a user clicks this button, the section options reply record contains the action code `'read'`. In this case, you should read in the new edition. See "Opening an Edition Container to Read Data" beginning on page 2-41 for detailed information.

When the update mode is set to automatic, your application receives a Section Read event each time a new edition becomes available. In response, you should read the new edition data beginning with the `OpenEdition` function.

Your application does not receive Section Read events for subscribers that receive new editions manually.

You may also support a Stop All Editions menu command to provide a method for temporarily suspending all update activity. See "Introduction to Publishers, Subscribers, and Editions" beginning on page 2-4 for additional information.

Canceling Sections Within Documents

The option of canceling publishers and subscribers is available to the user through the Cancel Publisher and Cancel Subscriber buttons in the corresponding options dialog boxes. When the user clicks one of these buttons, the action code of the section options reply record is `'cncl'`. See "Relocating an Edition" on page 2-60 for additional information on canceling a section.

When a user cancels a section (either a publisher or subscriber) and then saves the document, or when a user closes an untitled document (which contains newly created sections) without saving it, you must unregister each corresponding section record and alias record using the `UnRegisterSection` function. In addition, you should also delete the section record and alias record using the `DisposeHandle` procedure. See *Inside Macintosh: Memory* for additional information on the `DisposeHandle` procedure.

When a user cancels a publisher section and then saves the document, or when a user closes an untitled document (which contains newly created publishers) without saving it, you must also delete any corresponding edition containers (in addition to deleting section records and alias records).

Do not delete an edition container file, section record, or alias record until the user saves the document; the user may decide to undo changes before saving the document.

To locate the appropriate edition container to be deleted (before you use the `UnRegisterSection` function), use the `GetEditionInfo` function.

```
err := GetEditionInfo(sectionH, editionInfo);
```

The `editionInfo` parameter is a record of data type `EditionInfoRecord`.

```
TYPE EditionInfoRecord =
   RECORD
      crDate:    TimeStamp;           {date edition container }
                                      { was created}
      mdDate:    TimeStamp;           {date of last change}
      fdCreator: OSType;              {file creator}
      fdType:    OSType;              {file type}
      container: EditionContainerSpec;{the edition}
   END;
```

The `GetEditionInfo` function returns the edition container as part of the edition information.

The `crDate` field contains the creation date of the edition. The `mdDate` field contains the modification date of the edition.

The `fdType` and the `fdCreator` fields are the type and creator of the edition file. The `container` field includes a volume reference number, directory ID, filename, script, and part number for the edition.

To remove the edition container, use the `DeleteEditionContainerFile` function.

```
err := DeleteEditionContainerFile(editionFile);
```

Locating a Publisher Through a Subscriber

The user can locate a publisher from a subscriber within a document by clicking the Open Publisher button in the subscriber options dialog box. As a shortcut, Apple suggests that you also allow the user to locate a publisher by selecting a subscriber in a document and pressing Option–double-click.

When the action code of the `SectionOptionsReply` record is `'goto'`, use the `GoToPublisherSection` function.

```
err := GoToPublisherSection(container);
```

The `GoToPublisherSection` function locates the correct document by resolving the alias in the edition, and it launches the document's application if necessary (the Edition Manager sends an Open Documents event). The Edition Manager then sends the publishing application a Section Scroll event. If the document containing the requested publisher is located on the same computer as its subscriber, the document opens and scrolls to the location of the publisher. If the document containing the requested publisher is located on a shared volume (using file sharing), the document opens and scrolls to the location of the publisher only if the user has privileges to open the document from the Finder.

You need to provide the `GoToPublisherSection` function with the edition container. To accomplish this, use the `GetEditionInfo` function. See the previous section, "Canceling Sections Within Documents," for information on the `GetEditionInfo` function.

Renaming a Document Containing Sections

If a user renames a document that contains sections by choosing Save As from the File menu, or if a user pastes a portion of a document that contains a section into another document, use the `AssociateSection` function.

Use the `AssociateSection` function to update the alias record of a registered section.

```
err := AssociateSection (sectionH, newSectionDocument);
```

The `AssociateSection` function internally calls the `UpdateAlias` function. It is also possible to update the alias record using the Alias Manager (see the chapter "Alias Manager" in *Inside Macintosh: Files* for additional information).

Displaying Publisher and Subscriber Borders

Each publisher and subscriber within a document should have a border that appears when a user selects the contents of these sections. You should display a publisher border as three pixels wide with 50 percent gray lines and a subscriber border as three pixels wide with 75 percent gray lines. Separate the contents of the section from the border itself with one pixel of white space. To create your borders, you should use patterns, not colors. Depending on the user's monitor type, colors may not be distinguishable.

In general, borders for publishers and subscribers should behave like the borders of `'PICT'` graphics in a word-processing document. A border should appear when the user clicks the content area of a publisher or a subscriber and disappear when the user clicks outside the content area of a section. You can also make all publisher and subscriber borders appear or disappear by implementing an optional Show/Hide Borders menu command.

Figure 2-17 displays the Edition Manager Show/Hide Borders menu command in the Edit menu.

Figure 2-17   Edit menu with Show/Hide Borders menu command

```
┌─────────────────────────┐
│ Edit                    │
│ Undo              ⌘Z    │
│ ........................│
│ Cut               ⌘H    │
│ Copy              ⌘C    │
│ Paste             ⌘U    │
│ Select All        ⌘A    │
│ ........................│
│ Create Publisher...     │
│ Subscribe To...         │
│ Subscriber Options...   │
│ Show Borders            │
│ ........................│
│ Show Clipboard          │
└─────────────────────────┘
```

Depending on your application, you may choose to include resize handles or similar components in your borders. See "Object-Oriented Graphics Borders" on page 2-56 for an example of resize handles.

Whenever a user selects a portion of a publisher or sets the insertion point within a publisher, you should display the border as 50 percent gray. A user can copy the contents of a publisher or subscriber without copying the section itself by selecting the data, copying, and then pasting the data in a new location. A user can cut and paste a selection that contains an *entire* publisher or subscriber, but you should discourage users from making multiple copies of a publisher. See "Duplicating Publishers and Subscribers" on page 2-58 for detailed information.

When the user modifies a publisher, your application should grow or shrink its border to accommodate the new dimension of the section.

You should display only one publisher border within a document at a time. If a cursor is inserted within a publisher that is contained within a larger publisher, you should display only the smaller, internal publisher border. If it is absolutely necessary to display all section borders within a document at the same time, you can create a Show/Hide Borders menu item.

You do not need to provide support for publishers contained within other publishers. If you do not, you should dim the Create Publisher menu command (to indicate that it is not selectable) when a user attempts to create a publisher within an existing publisher.

Figure 2-18 shows the recommended border behavior for publishers. The top window shows a publisher with its borders displayed. The middle window shows how the borders look when a user selects some of the contents of a section. The bottom window shows how the borders look when a user selects data within a document that includes a publisher section.

Figure 2-18   Publisher borders
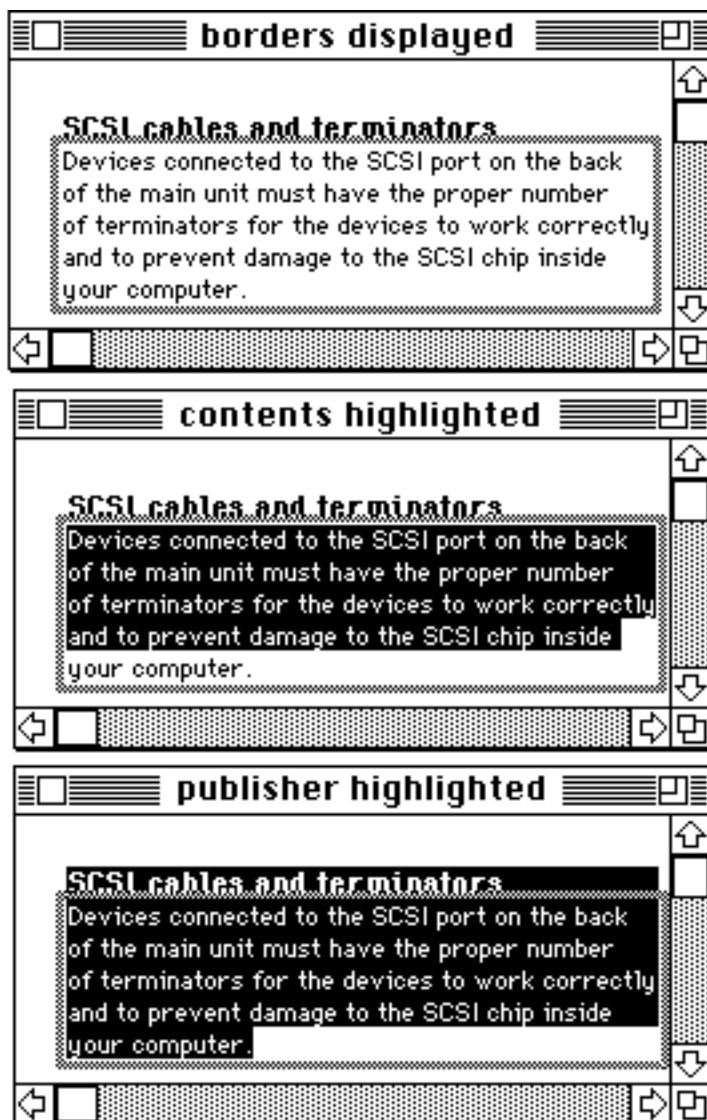
Figure 2-19 shows the recommended border behavior for subscribers. The top window shows a subscriber with its borders displayed. The middle window shows how the borders look when a user selects the contents of a section. The bottom window shows how the borders look when a user selects data within a document that includes a subscriber section.

Figure 2-19   Subscriber borders
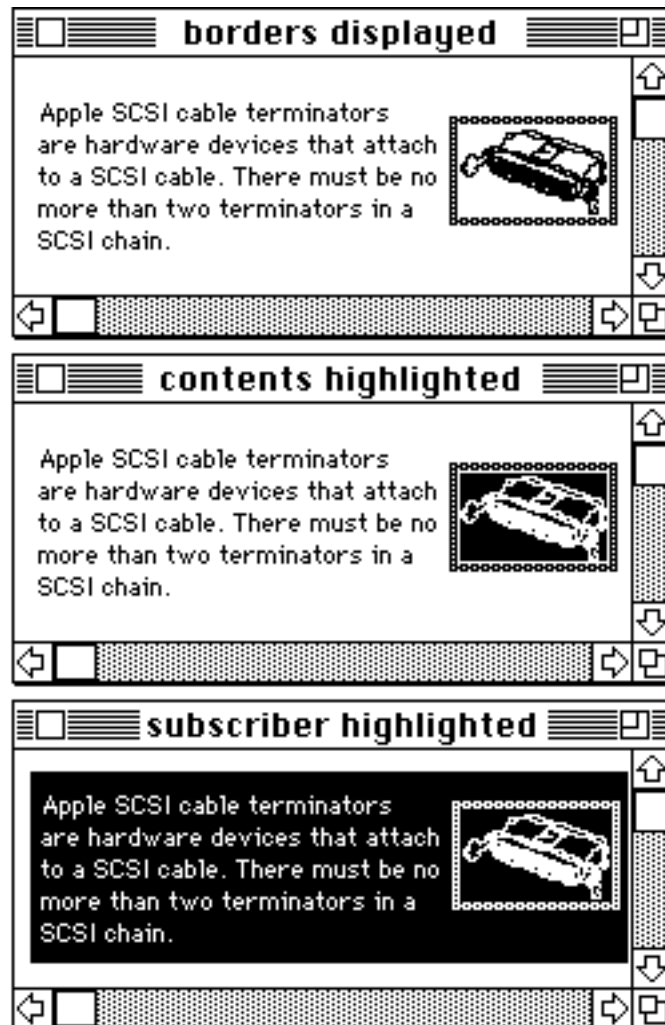
If a user tries to select only a portion of a subscriber, you should highlight the entire contents of the subscriber. A user cannot edit the data in a subscriber. See "Modifying a Subscriber" on page 2-59 for detailed information.

If a user cancels a section using the publisher or subscriber options dialog box, your application should leave the contents of the section within the document, but you should be sure to remove the borders from this data, as it is no longer considered a section.

Generally, the appearance and function of publisher and subscriber borders should be the same across different applications. The following sections entitled "Text Borders," "Spreadsheet Borders," "Object-Oriented Graphics Borders," and "Bitmapped Graphics Borders" describe specialized features for publisher and subscriber borders in word-processing, spreadsheet, or graphics applications.

Text Borders

In word-processing documents, a publisher may contain other publishers. However, one publisher should not *overlap* another publisher. You should display only one publisher border at a time. If an insertion point is placed within a publisher that is encompassed by another larger publisher, you should display only the smaller internal publisher border.

In exceptional cases, it may be necessary to display more than one publisher or subscriber border at a time. For example, a publisher may consist of a paragraph that includes a marker for a footnote. The data contained within the footnote should also be considered part of the publisher. When a user selects the paragraph, you should simultaneously display a border around the footnote.

The border of a publisher that contains text should be located between characters within the text. The insertion point, when placed on such a boundary, should gravitate toward the publisher. That is, a click in front (to the left) of a publisher border should place the cursor inside the publisher, so that subsequent typing goes inside the publisher. Clicking at the end (to the right) of a publisher border should also place the cursor inside the publisher.

Whenever two separate borders are adjacent, the boundary click should go in between them. This is also true for a border that is next to other nontextual aspects of a document, such as `'PICT'` graphics or page breaks.

When a user removes information from a publisher that contains text data, you should resize the border so that it becomes smaller. When a user adds information to the publisher, you should enlarge the border to accommodate the new text. The insertion point should remain within the publisher.

If a user highlights the entire contents of a publisher and then chooses Cut from the Edit menu, you should not delete the publisher border within the document. The user may intend to delete the existing publisher data and replace it with new data, or the user may want to move the entire publisher and its data to a new location. Figure 2-20 shows this state.

Figure 2-20   A publisher with contents removed

The first quarter summary of our regional sales shows the effectiveness of our new training program. It is clear that we need to capture the remaining sales potential.

You should leave the cursor inside the small publisher border for further typing. If the user inserts the cursor in a new location (instead of typing data inside the existing border), you need to remove the empty publisher border from the document to allow the user to move the publisher. This effectively deletes the publisher from the document. If the user pastes the publisher that is currently held in the scrap, you should re-create its border. If the user cuts or copies other data from the document before pasting the publisher from the scrap, the publisher should be removed from the scrap.

Spreadsheet Borders

Borders around spreadsheet data or other data in arrays should look and behave very much like text borders. Figure 2-21 shows a typical border within a spreadsheet document.

Figure 2-21   A publisher border within a spreadsheet document

	A	B	C	D	E
1		January	February	March	
2	Cogs	14890	17849	274945	
3	Sprockets	16494	184384	304890	
4	Widgets	3780	5839	7900	
5					

Note that the border goes below the column headers (A, B, C, D) and to the right of the row labels (1, 2, 3, 4)—it should not overlap these cell boundaries. The border at the bottom and the border on the right side can be placed within the adjacent cells (outside of the cells that constitute the publisher).

Unlike borders in word-processing applications, borders in spreadsheet documents (or other documents with array data) can overlap. That is, a user can select a row of cells to be a publisher and an overlapping column of those cells to be another publisher. You should never display more than one publisher border at a time. When a user selects a spreadsheet cell that is part of more than one publisher, you should display only the border of the publisher that was last edited. (This can be accomplished by comparing the modification dates of the publishers.)

If it is absolutely necessary to display all section borders within a document at the same time, you can create a Show/Hide Borders command in the Edit menu to toggle all borders on and off.

When data is added to or deleted from a publisher that consists of a spreadsheet cell or other array, you should resize its border to accommodate the addition or deletion of data. A publisher should behave like a named range in a spreadsheet. For example, if a user cuts a row within a publisher that consists of a named range in a spreadsheet, you should shrink the publisher data and its border correspondingly.

When a user cuts a publisher and its entire contents within a spreadsheet document, the entire section should be held in the scrap. Do not leave an empty publisher border in a spreadsheet (as recommended for text borders). If a user attempts to paste a copy of an existing publisher, you should warn the user by displaying an alert box (see "Duplicating Publishers and Subscribers" on page 2-58).

Object-Oriented Graphics Borders

In an object-oriented drawing application, the publisher border should fit just around the selected objects.

You can provide resize handles that appear with all drawing objects to allow the user to resize the border of a publisher. Figure 2-22 shows a publisher border with resize handles.

Figure 2-22   A publisher border with resize handles
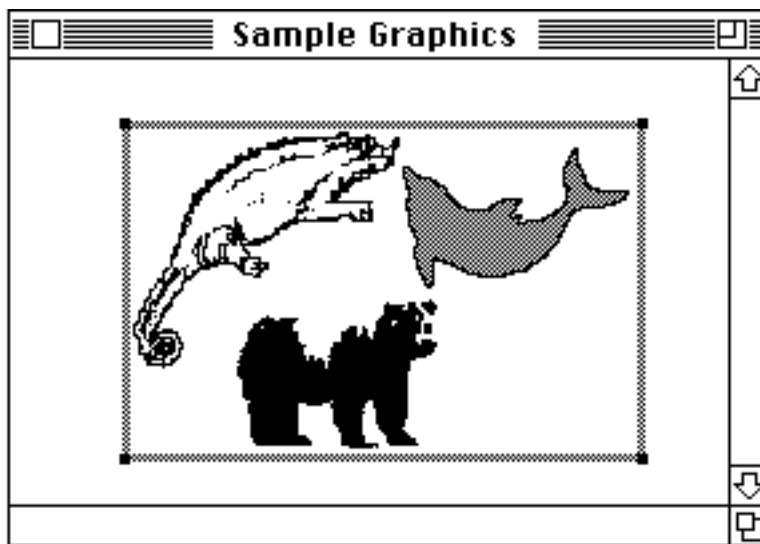
An application can make publisher borders appear to float over the area the user publishes. The border acts like a clipping rectangle—anything within the border becomes the publisher. Figure 2-23 shows a publisher that contains clipped graphics and its subscriber in another application.

A user can create publishers and subscribers that overlap each other. Thus, borders may overlap and it may no longer be possible to turn on a particular border when the user clicks within a publisher. Drawing applications should provide a menu command, Show Borders, that toggles to Hide Borders. This command should allow users to turn all publisher and subscriber borders on or off.

Figure 2-23   A publisher and subscriber with clipped graphics
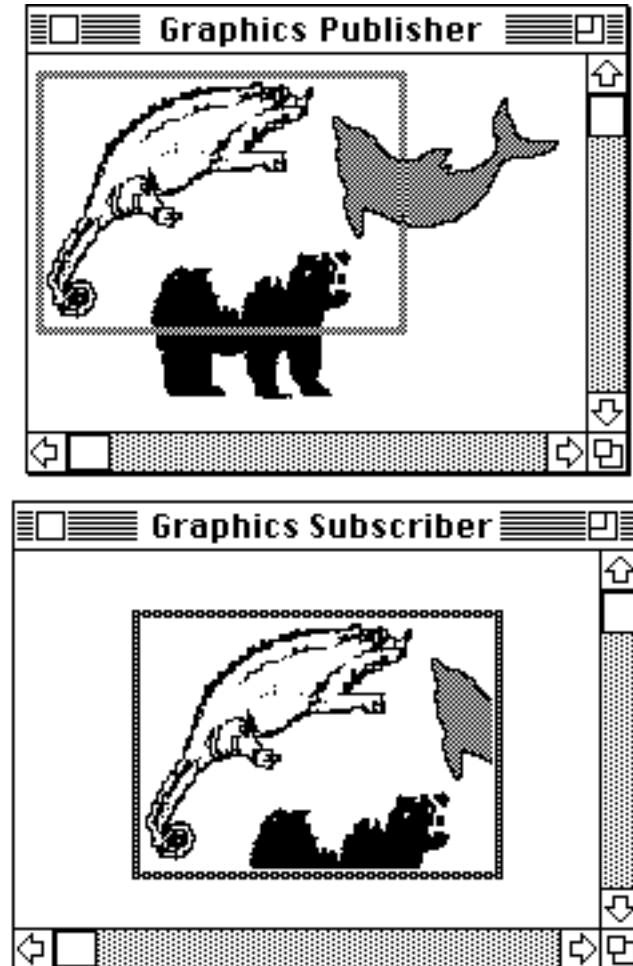
Bitmapped Graphics Borders

Creating a border around bitmapped graphics in applications is similar to doing so in object-oriented drawing applications. The border appears around the selected area. The user can create overlapping publishers and subscribers in bitmapped graphics applications. You need to provide a Show/Hide Borders command to allow users to turn all borders on and off.

Duplicating Publishers and Subscribers

Whenever a user clicks a publisher or subscriber border, you should change the contents of the section to a selected state. You should discourage users from making multiple copies of a publisher and pasting them in the same or other documents, because the contents of the edition would be difficult or impossible to predict. Multiple copies of the same publisher also contain the same control block value. See "Creating and Registering a Section" on page 2-74 for detailed information on control blocks.

When a user attempts to create a copy of a publisher that already exists, you should display an alert box such as the one shown in Figure 2-24.

Figure 2-24   Creating multiple publishers alert box
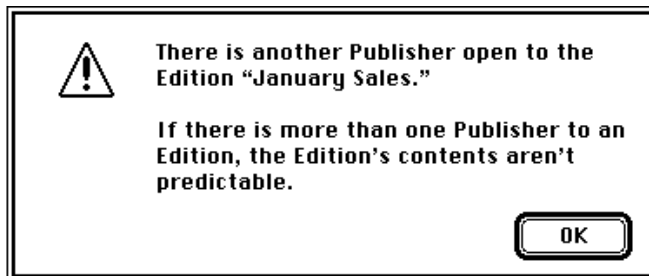
When a user attempts to save a document that contains multiple copies of the same publisher, display an alert box such as the one shown in Figure 2-25.

Figure 2-25   Saving multiple publishers alert box
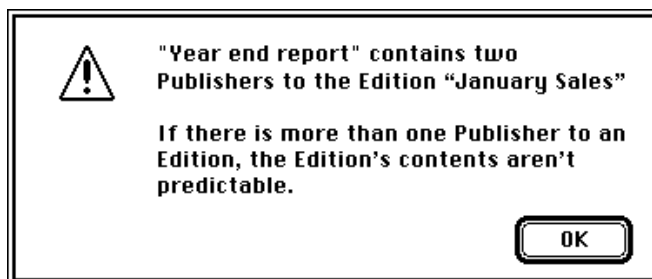
If a user decides to ignore your alert box, your application should still save the document, but you should continue to display this error message *every time* the user saves this document.

A user can modify the contents of any duplicate publisher, but the contents of the edition will be whichever publisher was the last to write.

When a user chooses to copy and paste or duplicate a section, use the `HandToHand` function (described in *Inside Macintosh: Memory*) to duplicate the section record and alias record. Set the `alias` field of the cloned section record to the handle of the cloned alias record and generate a unique section identification number for it. In addition, you should also place the section data, section record, and alias record in the scrap.

Use the `RegisterSection` function (described in "Opening and Closing a Document Containing Sections" on page 2-22) to register the cloned section's section record.

A user can select the *contents* of a publisher without selecting the border and copy just the data to a new location. In this case, the user has simply copied data (and not the publisher). Do not create a border for this data in the new location.

Modifying a Subscriber

When the user selects data or clicks the data area of a subscriber, you should highlight the entire contents of the subscriber using inverse video. Although you shouldn't allow a user to edit the information in a subscriber, you can allow a user to make global adornments to subscribers. In other words, users can change the font, size, or other characteristics of the *entire* subscriber. For example, a user might select a subscriber within a document and change all text from plain to bold. However, you should discourage users from modifying the individual elements contained within a subscriber—for example, by editing a sentence or rotating an individual graphic object.

Remember that each time a new edition arrives for a subscriber, any modifications that the user has introduced are overwritten. Global changes to a subscriber are much easier for your application to regenerate.

Note

Although adornments should be global and never partial, you may still need to give users the ability to select portions of a subscriber, for instance, when performing spell checking and search-and-replace operations. u

If you do allow a user to edit a subscriber section, provide an enable/disable editing option within the subscriber options dialog box using the `SectionOptionsExpDialog` function, described in "Customizing Dialog Boxes" beginning on the next page. When you allow a user to edit a subscriber, you should change the subscriber from a selected state to editable data.

Because a user can modify a publisher just like any other portion of a document, its subscriber may change in size as well as content. For example, a user may modify a publisher by adding two additional columns to a spreadsheet.

Relocating an Edition

In the Finder, users cannot move an edition across volumes. To relocate an edition, the user must first select its publisher and cancel the section (remember to remove the border). The user needs to republish and then select a new volume location for the edition. As a convenience for the user, you should retain the selection of all the publisher data after the user cancels the section to make it easy to republish the section.

Customizing Dialog Boxes

The expandable dialog box functions allow you to add items to the bottom of the dialog boxes, apply alternate mapping of events to item hits, apply alternate meanings to the item hits, and choose the location of the dialog boxes. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* and the chapter "Standard File Package" in *Inside Macintosh: Files* for additional information.

The expandable versions of these dialog boxes require five additional parameters. Use the `NewPublisherExpDialog` function to expand the publisher dialog box.

```
err := NewPublisherExpDialog (reply, where, expansionDITLresID,
                             dlgHook, filterProc, yourDataPtr);
```

Use the `NewSubscriberExpDialog` function to expand the subscriber dialog box.

```
err := NewSubscriberExpDialog (reply, where, expansionDITLresID,
                              dlgHook, filterProc, yourDataPtr);
```

Use the `SectionOptionsExpDialog` function to expand the publisher options and the subscriber options dialog boxes.

```
err := SectionOptionsExpDialog (reply, where, expansionDITLresID,
                               dlgHook, filterProc, yourDataPtr);
```

The `reply` parameter is a pointer to a `NewPublisherReply`, `NewSubscriberReply`, or `SectionOptionsReply` record, respectively.

You can automatically center the dialog box by passing (–1, –1) in the `where` parameter.

The `expansionDITLresID` parameter should contain 0 or a valid item list (`'DITL'`) resource ID. This integer is the resource ID of an item list whose items are appended to the end of the standard item list. The dialog items keep their relative positions, but they are moved as a group to the bottom of the dialog box. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for additional information on item lists.

The `filterProc` parameter should be a pointer to an expandable modal-dialog filter function or `NIL`. An expandable modal-dialog filter function is similar to a modal-dialog filter function or event filter function except that an expandable modal-dialog filter function accepts two extra parameters. The `ModalDialog` procedure calls the expandable modal-dialog filter function you provide in this parameter.

Providing a filter function enables you to map real events (such as a mouse-down event) to an item hit (such as clicking the Cancel button). For instance, you may want to map a keyboard equivalent to an item hit. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on the `ModalDialog` procedure.

The `dlgHook` parameter should be a pointer to an expandable dialog hook function or `NIL`. An expandable dialog hook function is similar to a dialog hook function except that an expandable dialog hook function accepts an additional parameter. The `NewSubscriberExpDialog`, `NewPublisherExpDialog`, and `SectionOptionsExpDialog` functions call your expandable dialog hook function after each call to the `ModalDialog` procedure. The dialog hook function should take the appropriate action, such as filling in a checkbox. The `itemOffset` parameter to the procedure is the number of items in the item list before the expansion dialog items. You need to subtract the item offset from the item hit to get the relative item number in the expansion dialog item list. The expandable dialog hook function should return as its function result the absolute item number.

When the Edition Manager displays subsidiary dialog boxes in front of another dialog box on the user's screen, your dialog hook and event filter functions should check the `refCon` field in the `WindowRecord` data type (from the `window` field in the `DialogRecord`) to determine which window is currently in the foreground. The main dialog box for the `NewPublisherExpDialog` and the `NewSubscriberExpDialog` functions contains the following constant:

```
CONST    sfMainDialogRefCon   = 'stdf';   {new publisher and }
                                          { new subscriber}
```

The main dialog box for the `SectionOptionsExpDialog` function contains the following constant:

```
CONST    emOptionsDialogRefCon   = 'optn';   {options dialog}
```

See "Summary of the Edition Manager" beginning on page 2-106 for additional constants.

The `yourDataPtr` parameter is reserved for your use. It is passed back to your dialog hook and event filter function. This parameter does not have to be of type `Ptr`—it can be any 32-bit quantity that you want. In Pascal, you can pass `yourDataPtr` in register A6 , and declare your dialog hook and modal-dialog filter as local functions without the last parameter. The stack frame is set up properly for these functions to access their parent local variables. See the chapter "Standard File Package" in *Inside Macintosh: Files* for detailed information.

For the `NewPublisherExpDialog` and `NewSubscriberExpDialog` functions, all the pseudo-items for the Standard File Package—such as `sfHookFirstCall`(–1), `sfHookNullEvent`(100), `sfHookRebuildList`(101), and `sfHookLastCall`(–2)—can be used, as well as `emHookRedrawPreview`(150).

For the `SectionOptionsExpDialog` function, the only valid pseudo-items are
`sfHookFirstCall`(–1), `sfHookNullEvent`(100), `sfHookLastCall`(–2),
`emHookRedrawPreview`(150), `emHookCancelSection`(160),
`emHookGoToPublisher`(161), `emHookGetEditionNow`(162),
`emHookSendEditionNow`(162), `emHookManualUpdateMode`(163), and
`emHookAutoUpdateMode`(164). See the chapter "Standard File Package" in
Inside Macintosh: Files for information on pseudo-items.

Subscribing to Non-Edition Files

Using the Edition Manager, a subscriber can read data directly from another document,
such as an entire `'PICT'` file, instead of subscribing to an edition. This feature is for
advanced applications that can set up bottleneck procedures for reading. Figure 2-26
shows a document that is subscribing directly to a `'PICT'` file.

Figure 2-26   Subscribing directly to a `'PICT'` file
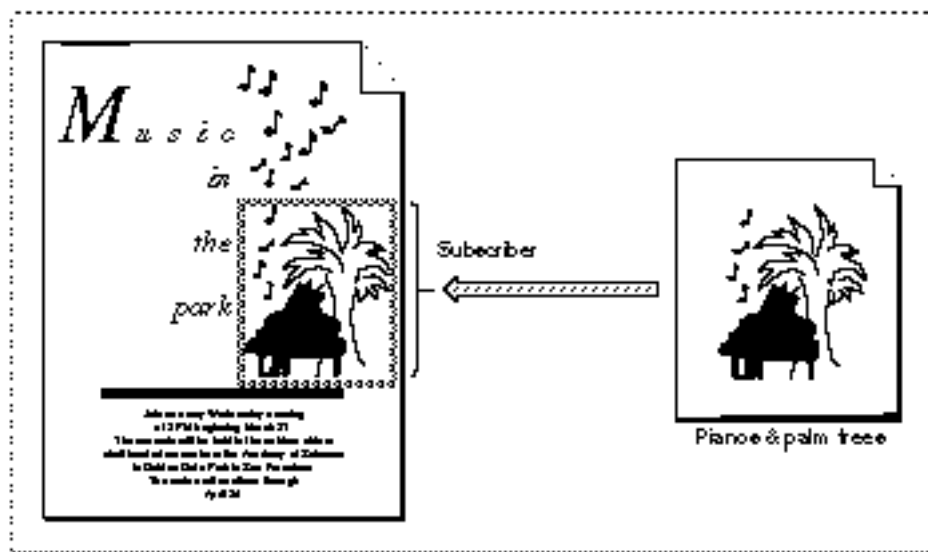
For each application, the Edition Manager keeps a pointer to a bottleneck function.
The Edition Manager never opens or closes an edition container directly. Instead, the
Edition Manager calls the current edition opener. The `InitEditionPack` function
(described on page 2-74) sets up the current system opener function.

To override the standard opener function, create an opener function that contains the following parameters:

```
FUNCTION MyOpener (selector: EditionOpenerVerb;
                   VAR PB: EditionOpenerParamBlock): OSErr;
```

Your opener needs to know which formats the file contains and how the data is supposed to be read or written.

The opener function is passed an edition opener verb in the `selector` parameter, which identifies the action the opener function should perform. The opener can allocate a handle or pointer to contain information such as file reference numbers. This value is passed to the I/O routines in the `ioRefNum` field of the edition opener parameter block.

The `eoOpen` and `eoOpenNew` edition opener verbs (described in "Calling an Edition Opener" on page 2-64) return a pointer to a function to do the actual reading and writing.

The following sections describe

n how to get the current edition opener

n how to set your own edition opener

n how to call an edition opener

n the edition opener parameters

Getting the Current Edition Opener

When you want to get the current edition opener, use the `GetEditionOpenerProc` function.

```
err := GetEditionOpenerProc(opener);
```

The `opener` parameter returns a pointer to the current edition opener. A different current opener is kept for each application. One application's opener is never called by another application.

Setting an Edition Opener

You can provide your own edition opener. To do so, use the `SetEditionOpenerProc` function.

```
err := SetEditionOpenerProc(@MyOpener);
```

The `@MyOpener` parameter is a pointer to the edition opener function that you are providing. If you set the current opener to be a routine in your own code, be sure to call the `GetEditionOpenerProc` function first so that you can save the previous opener. If your opener is passed a selector that it does not understand, use the previous opener provided by the Edition Manager to handle it. See the next section for a list of selectors.

Calling an Edition Opener

You use the `CallEditionOpenerProc` function to call an edition opener. Since the Edition Manager is a package that may move, a real pointer cannot be safely returned for the standard opener and I/O routines. The system opener and the I/O routines are returned as a value that is not a valid address to a procedure. The `CallEditionOpenerProc` and `CallFormatIOProc` functions check for these values and call the system openers.

You should never assume that a value for a system opener is a fixed constant.

```
err := CallEditionOpenerProc (selector, PB, routine);
```

Set the `selector` parameter to one of the edition opener verbs. The edition opener verbs include

n   `eoCanSubscribe`

n   `eoOpen`

n   `eoClose`

n   `eoOpenNew`

n   `eoCloseNew`

The `PB` parameter of the `CallEditionOpenerProc` function is an edition opener parameter block.

```
TYPE EditionOpenerParamBlock =
    RECORD
        info:        EditionInfoRecord;   {edition container to }
                                          { be subscribed to}
        sectionH:    SectionHandle;       {publisher or }
                                          { subscriber }
                                          { requesting open}
        document:    FSSpecPtr;           {document passed}
        fdCreator:   OSType;              {Finder creator type}
        ioRefNum:    LongInt;             {reference number}
        ioProc:      FormatIOProcPtr;     {routine to read }
                                          { formats}
        success:     Boolean;             {reading or writing }
                                          { was successful}
        formatsMask: SignedByte;          {formats required to }
                                          { subscribe}
    END;
```

The `routine` parameter of the `CallEditionOpenerProc` function is a pointer to an edition opener function.

The following list shows which fields of the edition opener parameter block are used by the edition opener verbs:

Opener verb	Field	Description	Called by
eoCanSubscribe	info	Edition container to subscribe to.	NewSubscriberDialog function for a subscriber
	formatsMask	Formats required to subscribe.	
	Return value	A noErr code indicates that an edition container can be subscribed to. A noTypeErr code indicates that an edition container cannot be subscribed to.	
eoOpen	info	Edition container to open for reading.	OpenEdition and GetStandardFormats functions for a subscriber
	sectionH	Subscriber section requesting open or NIL.	
	ioRefNum	Reference number for use by I/O routine. Not the same as EditionRefNum.	
	ioProc	I/O routine to call to read formats.	
	Return value	A noErr code or appropriate error code.	
eoClose	info	Edition container to be closed for reading.	CloseEdition and GetStandardFormats functions for a subscriber
	sectionH	Subscriber section requesting close or NIL.	
	ioRefNum	Value returned by eoOpen.	
	ioProc	Value returned by eoOpen.	
	success	Success value passed to the CloseEdition function.	
	Return value	A noErr code or appropriate error code.	

Opener verb	Field	Description	Called by (continued)
eoOpenNew	info	Edition container to open for writing.	OpenNewEdition function for a publisher
	sectionH	Publisher section requesting open or NIL.	
	document	Document pointer passed into the OpenNewEdition function.	
	fdCreator	The fdCreator passed into the OpenNewEdition function.	
	ioRefNum	Reference number for use by I/O routine. Not the same as EditionRefNum.	
	ioProc	I/O routine to call to write formats.	
	Return value	A noErr code or appropriate error code.	
eoCloseNew	info	Edition container to be closed after writing.	CloseEdition function for a publisher
	sectionH	Publisher section requesting close or NIL.	
	ioRefNum	Value returned by eoOpenNew.	
	ioProc	Value returned by eoOpenNew.	
	success	Success value passed to the CloseEdition function.	
	Return value	A noErr code or appropriate error code.	

As Listing 2-9 demonstrates, you install your own edition opener function by first saving the current opener and then installing your own opener. The listing also shows an edition opener, the MyEditionOpener function. When it receives the eoCanSubscribe opener verb, the MyEditionOpener function calls another application-defined routine, MyCanSubscribe. The Edition Manager sends your edition opener this verb to help it build the list of files displayed by the NewSubscriber function. The MyCanSubscribe function returns noErr if it can subscribe to the file; otherwise, it calls the original edition opener to handle the request.

Listing 2-9   Using your own edition opener function

```
VAR
   gOriginalOpener: EditionOpenerProcPtr;{global variable}

PROCEDURE MyInstallMyOpener;
BEGIN
   FailOSErr(GetEditionOpenerProc(gOriginalOpener));
   FailOSErr(SetEditionOpenerProc(@MyEditionOpener));
END; {MyInstallMyOpener}

FUNCTION MyEditionOpener (selector: EditionOpenerVerb;
                         VAR PB: EditionOpenerParamBlock)
                          : OSErr;
BEGIN
   WITH PB DO
   BEGIN
      CASE selector OF
         eoCanSubscribe:
            MyEditionOpener := MyCanSubscribe(PB);
         eoOpen:
            MyEditionOpener := MyEditionOpen(PB);
         eoClose:
            MyEditionOpener := MyEditionClose(PB);
         OTHERWISE
            {call the original edition opener}
            MyEditionOpener
               := CallEditionOpenerProc(selector, PB,
                                     gOriginalOpener);
      END; {of CASE}
   END; {of WITH}
END; {MyEditionOpener}

FUNCTION MyCanSubscribe (VAR PB: EditionOpenerParamBlock): OSErr;
BEGIN
   {check file type to see if it is a file you can emulate as an }
   { edition}
   IF PB.info.fdType = {for example}'PICT' THEN
      MyCanSubscribe := noErr
   ELSE {otherwise, let the saved edition opener decide}
        MyCanSubscribe := CallEditionOpenerProc(eoCanSubscribe,
                                     PB, gOriginalOpener);
END; {MyCanSubscribe}
```

Opening and Closing Editions

Each time the Edition Manager opens or closes an edition container, it calls the current edition opener procedure and passes it an opener verb and a parameter block.

Your opener must be careful when closing documents since a document may already have been opened by another application. Be sure to use the Open/Deny modes whenever possible. Do not close a document if it was already open when your application opened it.

Listing Files That Can Be Subscribed To

The `NewSubscriberDialog` function calls the edition opener function and passes the `eoCanSubscribe` opener verb in the `selector` parameter to build the list of files that can be subscribed to. The preview in the subscriber dialog box is generated by calling the `GetStandardFormats` function (described in "Edition Container Formats" on page 2-101), which calls the format I/O procedure with the verbs `eoOpen`, `ioHasFormat`, `ioRead`, and then `eoClose`. See "Calling a Format I/O Function" on this page for detailed information on format I/O verbs.

Reading From and Writing to Files

The I/O procedure is a routine that actually reads and writes the data. It too has an interface of a selector and a parameter block.

To override the standard reading and writing functions, create an I/O function. Note that you also need to provide your own opener function to call your I/O function. See "Calling an Edition Opener" on page 2-64.

```
FUNCTION MyIO (selector: FormatIOVerb;
               VAR PB: FormatIOParamBlock): OSErr;
```

Calling a Format I/O Function

To indicate to the Edition Manager which format I/O function to use, use the `CallFormatIOProc` function.

```
err := CallFormatIOProc (selector, PB, routine);
```

Set the `selector` parameter to one of the format I/O verbs. The format I/O verbs include

n `ioHasFormat`

n `ioReadFormat`

n `ioNewFormat`

n `ioWriteFormat`

The `PB` parameter of the `CallFormatIOProc` function contains a format I/O parameter block.

```
TYPE FormatIOParamBlock =
   RECORD
      ioRefNum:       LongInt;      {reference number}
      format:         FormatType;   {edition format type}
      formatIndex:    LongInt;      {opener-specific enumeration }
                                    { of formats}
      offset:         LongInt;      {offset into format}
      buffPtr:        Ptr;          {data starts here}
      buffLen:        LongInt;      {length of data}
   END;
```

The `routine` parameter of the `CallFormatIOProc` function is a pointer to a format I/O function.

The following list shows which fields of `FormatIOParamBlock` are used by the format I/O verbs:

Format I/O verb	Parameter	Description	Called by
ioHasFormat	ioRefNum	I/O reference number returned by opener.	EditionHasFormat, GetStandardFormats, and ReadEdition functions
	format	Check for this format.	
	formatIndex	An optional enumeration of the supplied format.	
	buffLen	If found, return the length size or –1 if size is unknown.	
	Return value	A noErr or noTypeErr code.	

Format I/O verb	Parameter	Description	Called by (continued)
ioReadFormat	ioRefNum	I/O reference number returned by opener.	ReadEdition and GetStandardFormats functions
	format	Get this format.	
	formatIndex	Value returned by ioHasFormat.	
	offset	Read format beginning from this offset.	
	buffPtr	Put data beginning here.	
	buffLen	Specify buffer length to read, and return actual amount received.	
	Return value	A noErr code, or appropriate error code.	
ioNewFormat	ioRefNum	I/O reference number returned by opener.	SetEditionFormatMark and WriteEdition functions
	format	Create this format.	
	formatIndex	An optional enumeration of the supplied format.	
	Return value	A noErr code, or appropriate error code.	
ioWriteFormat	ioRefNum	I/O reference number returned by opener.	WriteEdition function
	format	Get this format.	
	formatIndex	Value returned by ioNewFormat.	
	offset	Write format beginning from this offset.	
	buffPtr	Get data beginning here.	
	buffLen	Specify buffer length to write.	
	Return value	A noErr code or appropriate error code.	

The marks for each format are kept by the Edition Manager. The format I/O function only needs to be able to read or write, beginning at any offset. If you know that your application always reads an entire format sequentially, you can ignore the offset.

Edition Manager Reference

This section describes the data structures and routines that are specific to the Edition Manager. The "Data Structures" section describes the edition container record and the section record. The "Edition Manager Routines" section describes the routines your application can use to implement publish and subscribe features in your application.

Data Structures

This section describes the edition container record and the section record. See page 2-91 for a description of the new subscriber reply record, page 2-93 for a description of the new publisher reply record, page 2-95 for a description of the section options record, and page 2-99 for a description of the edition info record. For information on the edition opener parameter block and format I/O parameter block, see page 2-103 and page 2-104, respectively.

The Edition Container Record

An edition container record identifies a specific edition file. Many Edition Manager routines require an edition container record as a parameter. The `EditionContainerSpec` data type defines an edition container record.

```
TYPE EditionContainerSpec =
    RECORD
        theFile:        FSSpec;        {file containing edition }
                                       { data}
        theFileScript:  ScriptCode;    {script code of filename}
        thePart:        LongInt;       {which part of file, }
                                       { always kPartsNotUsed}
        thePartName:    Str31;         {reserved}
        thePartScript:  ScriptCode;    {reserved}
    END;
```

Field descriptions

theFile   A file specificiation record that identifies the name and location of the edition file. Specify the file using the standard conventions for file specification records as described in the chapter "Introduction to File Management" in *Inside Macintosh: Files.*

theFileScript   A script code that identifies the script in which the name of the document is to be displayed in the Finder. A script code of `smSystemScript` represents the default system script.

thePart   A value that must always be set to `kPartsNotUsed` in System 7.

thePartName   Reserved.

thePartScript   Reserved.

The Section Record

A section record identifies a specific publisher or subscriber section. It contains information to identify the section as a publisher or a subscriber, a time stamp to record the last modification of the section, and unique identification for each section. Many Edition Manager routines require a handle to a section record as a parameter. The `SectionRecord` data type defines a section record.

```
TYPE SectionRecord =
   RECORD
      version:      SignedByte;     {always 1 in 7.0}
      kind:         SectionType;    {publisher or subscriber}
      mode:         UpdateMode;     {automatic or manual}
      mdDate:       TimeStamp;      {last change in document}
      sectionID:    LongInt;        {application-specific, }
                                    { unique per document}
      refCon:       LongInt;        {application-specific}
      alias:        AliasHandle;    {handle to alias record}

      {The following fields are private and are set up by the }
      { RegisterSection function. Do not modify the private }
      { fields.}
      subPart:      LongInt;        {private}
      nextSection:  SectionHandle;  {private, do not use as a }
                                    { linked list}
      controlBlock: Handle;         {may be used for comparison }
                                    { only}
      refNum:       EditionRefNum;  {private}
END;
```

Field descriptions

version   Indicates the version of the section record, currently $01.

kind   Defines the section type as either publisher or subscriber with the `stPublisher` or `stSubscriber` constant.

mode   Indicates if editions are updated automatically or manually.

mdDate	Indicates which version (modification date) of the section's contents is contained within the publisher or subscriber. The mdDate field is set to 0 when you create a new subscriber section and to the current time when you create a new publisher. Be sure to update this field each time publisher data is modified. The section's modification date is compared to the edition's modification date to determine whether the section and the edition contain the same data. The section modification date is displayed in the publisher and subscriber options dialog boxes. See "Closing an Edition" on page 2-28 for detailed information.
sectionID	Provides a unique number for each section within a document. A simple way to implement this is to create a counter for each document that is saved to disk with the document. The counter should start at 1. The section ID is currently used as a tie breaker in the GoToPublisherSection function when there are multiple publishers to the same edition in a single document. The section ID should not be 0 or –1. See "Duplicating Publishers and Subscribers" on page 2-58 for information on multiple publishers.
refCon	Reference constant available for application-specific use.
alias	Contains a handle to the alias record for a particular section within a document.

Whenever the user creates a publisher or subscriber, call the NewSection function (described on page 2-75) to create a section record and alias record.

Edition Manager Routines

This section describes the routines you use to

n   initialize the Edition Manager

n   create and register a section

n   create and delete an edition container

n   set and locate a format mark

n   read in edition data

n   write out edition data

n   close an edition after reading or writing

n   display dialog boxes

n   locate a publisher and edition from a subscriber

n   read and write non-edition files

Result codes appear at the end of each function where applicable. In addition to the specific result codes listed, you may receive errors generated by the Alias Manager, File Manager, and Memory Manager.

Initializing the Edition Manager

You use the `InitEditionPack` function to initialize the Edition Manager. Note that you should call this function only once.

InitEditionPack

Before calling the `InitEditionPack` function, be sure to determine whether the Edition Manager is available on your system by using the `Gestalt` function with the `gestaltEditionMgrAttr` (`'edtn'`) selector.

```
FUNCTION InitEditionPack: OSErr;
```

DESCRIPTION

The `InitEditionPack` function returns an error if the package could not be loaded into the system heap and properly initialized.

RESULT CODES

noErr	0	No error
memFullErr	–108	Could not load package

Creating and Registering a Section

You use the `NewSection` function to create a new section (either publisher or subscriber) and alias record (which is a reference to the edition container from the document containing the publisher or subscriber section).

The `NewSection` function registers a section much as the `RegisterSection` function informs the Edition Manager about a section (except that the `NewSection` function does not resolve an alias to find the edition container).

When a section needs to be disposed of because the document containing the section is being closed or because the user has canceled the section, you need to call the `UnRegisterSection` function before disposing of the section.

Using the `IsRegisteredSection` function, your application must verify that each event received is for a registered section. This is necessary because your application may have just called `UnRegisterSection` while the event was already being held in the event queue.

If a user saves a document that contains sections under another name (using Save As) or pastes a portion of a document that contains a section into another document, use the `AssociateSection` function to update the section's alias record.

NewSection

Use the `NewSection` function to create a new section record and alias record for a new publisher or subscriber.

```
FUNCTION NewSection (container: EditionContainerSpec;
                     sectionDocument: FSSpecPtr;
                     kind: SectionType; sectionID: LongInt;
                     initialMode: UpdateMode;
                     VAR sectionH: SectionHandle): OSErr;
```

container   The edition you want to publish or subscribe to.

sectionDocument
             The volume reference number, directory ID, and filename of the document that contains a section. The `sectionDocument` parameter can be `NIL` if your current document has never been saved. If so, when the user finally saves the document, remember to call the `AssociateSection` function for each section to update its alias record.

kind   The type of section (publisher or subscriber) being created.

sectionID   A unique number for a section within a document. The `NewSection` function initializes the `sectionID` field of the new section record with the specified value. Do not use 0 or –1 for an ID number; these numbers are reserved. If your application copies a section, you need to specify a unique number for the copied section.

initialMode
             The update mode for the section. For publishers this is either the `pumOnSave` or `pumManual` constant, and for subscribers it is either `sumAutomatic` or `sumManual`. A subscriber created with `sumAutomatic` mode automatically receives a Section Read event. To prevent this initial Section Read event, you should set the `initialMode` parameter to `sumManual` and then, when `NewSection` returns, set the `mode` field of the section record to `sumAutomatic`.

sectionH   The `NewSection` function returns a handle to the allocated section record in this parameter. If an error occurs, `NewSection` returns `NIL` in this parameter.

DESCRIPTION

The `NewSection` function allocates two handles in the current zone: one handle for the section record and another handle for the alias record. Note that you are responsible for unregistering handles created by the Edition Manager.

Your application receives the `multiplePublisherWrn` result code if there is another registered publisher to the same edition. Your application receives the `notThePublisherWrn` result code if another publisher (to the same edition) was the last section to write to the edition. The `multiplePublisherWrn` result code takes priority over the `notThePublisherWrn` result code.

RESULT CODES

noErr	0	No error
editionMgrInitErr	−450	Manager not initialized
badSectionErr	−451	Not a valid section type
badSubPartErr	−454	Bad edition container spec
multiplePublisherWrn	−460	Already is a publisher
notThePublisherWrn	−463	Not the publisher

SEE ALSO

For information on the edition container record, see page 2-71. For information on the
section record, see "The Section Record" beginning on page 2-72. For information on file
specification records, see *Inside Macintosh: Files*. See Listing 2-4 on page 2-33 for an
example that uses NewSection to create a publisher and Listing 2-6 on page 2-40 for
an example that creates a subscriber using NewSection.

RegisterSection

When opening a document that contains sections, register each section using the
RegisterSection function.

```
FUNCTION RegisterSection (sectionDocument: FSSpec;
                          sectionH: SectionHandle;
                          VAR aliasWasUpdated: Boolean): OSErr;
```

sectionDocument
    The volume reference number, directory ID, and filename of the
    document that contains a section.

sectionH   A handle to the section record for a given section.

aliasWasUpdated
    A Boolean value that returns TRUE if the alias for the edition container
    subscribed to was out of date and was updated. This may occur if the
    edition file was moved to a new location or was renamed.

DESCRIPTION

The RegisterSection function adds the section record to the Edition Manager's list of
registered sections and tries to allocate a control block. After calling the
RegisterSection function, the controlBlock field of the section record contains
either NIL or a valid control block.

For a subscriber, the controlBlock field contains NIL if the RegisterSection
function could not locate the edition container being subscribed to. The
RegisterSection function then returns either the containerNotFoundWrn or the
userCanceledErr result code. For a publisher, if the RegisterSection function
could not locate its corresponding edition container, the Edition Manager creates an

edition container in the last place the edition was located and creates a control block for it. If the RegisterSection function could not locate a publisher's corresponding edition container or its volume, the controlBlock field contains NIL. You should never re-register a section that is already registered.

Note that you can compare control blocks for individual sections. If two sections contain the same control block value, these sections publish or subscribe to the same edition (unless the control block is NIL). The Edition Manager keeps track of how many sections are referencing a control block to know when it can be deallocated. The control block maintains a count of how many sections are referencing it. Each time you use the UnRegisterSection function, the control block subtracts 1 from the number of sections. When the number of sections reaches 0, the control block is deallocated.

Your application receives the multiplePublisherWrn result code if there is another registered publisher to the same edition. Your application receives the notThePublisherWrn result code if another publisher (to the same edition) was the last section to write to the edition. The multiplePublisherWrn result code takes priority over the notThePublisherWrn result code.

RESULT CODES

noErr	0	No error
userCanceledErr	–128	User clicked Cancel in dialog box
editionMgrInitErr	–450	Manager not initialized
badSectionErr	–451	Not valid section type
multiplePublisherWrn	–460	Already is a publisher
containerNotFoundWrn	–461	Alias was not resolved
notThePublisherWrn	–463	Not the publisher

SEE ALSO

For information on the section record, see "The Section Record" beginning on page 2-72. For information on file specification records, see *Inside Macintosh: Files*. For additional information and an example of the use of RegisterSection, see "Opening and Closing a Document Containing Sections" beginning on page 2-22.

UnRegisterSection

When a section needs to be disposed of because the document containing the section is being closed or because the user has canceled the section, you need to call the UnRegisterSection function before disposing of the section.

```
FUNCTION UnRegisterSection (sectionH: SectionHandle): OSErr;
```

sectionH   A handle to the section record for a given section.

DESCRIPTION

The `UnRegisterSection` function removes the section from the Edition Manager's list of registered sections. You can then dispose of the section record and alias record with standard Memory Manager and Resource Manager calls. Once unregistered, a section does not receive any events and cannot read or write any data. Depending on your Clipboard strategy, you may want to unregister sections that have been cut into the Clipboard.

RESULT CODES

noErr	0	No error
fBsyErr	−47	Section doing I/O
editionMgrInitErr	−450	Manager not initialized
notRegisteredSectionErr	−452	Not registered

IsRegisteredSection

Upon receiving a section event, your application must call the `IsRegisteredSection` function to verify that the event received is for a registered section. You must call `IsRegisteredSection` before handling a section event because your application may have just called `UnRegisterSection` while the event was already being held in the event queue.

```
FUNCTION IsRegisteredSection (sectionH: SectionHandle): OSErr;
```

sectionH   A handle to the section record for a given section.

DESCRIPTION

The `IsRegisteredSection` function returns a result code (not a Boolean value) indicating whether the section is registered. A `noErr` result code indicates that a section is registered.

RESULT CODES

noErr	0	No error
notRegisteredSectionErr	−452	Not registered

SEE ALSO

For an example of the use of `IsRegisteredSection`, see Listing 2-1 on page 2-14.

AssociateSection

If a user saves a document that contains sections under another name (using Save As) or pastes a portion of a document that contains a section into another document, use the `AssociateSection` function to update the section's alias record.

```
FUNCTION AssociateSection (sectionH: SectionHandle;
                              newSectionDocument: FSSpecPtr): OSErr;
```

sectionH   A handle to the section record for a given section.

newSectionDocument
              The volume reference number, directory ID, and filename of the new document.

DESCRIPTION

The `AssociateSection` function calls `UpdateAlias` on the section's alias record.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter

SEE ALSO

For information on the `UpdateAlias` function, see the chapter "Alias Manager" in *Inside Macintosh: Files.*

Creating and Deleting an Edition Container

Each time a user creates a new publisher section within a document to an edition that does not already exist, you use the `CreateEditionContainerFile` function to create an empty edition container.

To remove the edition container, use the `DeleteEditionContainerFile` function.

CreateEditionContainerFile

You use the `CreateEditionContainerFile` function to create an empty edition container.

```
FUNCTION CreateEditionContainerFile
                    (editionFile: FSSpec; fdCreator: OSType;
                     editionFileNameScript: ScriptCode): OSErr;
```

editionFile
: The volume reference number, directory ID, and filename for the edition container being created.

fdCreator
: The creator type for the edition.

editionFileNameScript
: The script of the filename. (You can get this value from the theFileScript field of an edition container specification record.)

DESCRIPTION

The CreateEditionContainerFile function creates an empty edition container file (it does not contain any formats). This function sets the file type of the edition to 'edtu'. As soon as you write data to the edition, the Edition Manager updates the type (to 'edtp' for graphics, 'edtt' for text, or 'edts' for sound). If your application writes both 'TEXT' and 'PICT' formats to the edition, the Edition Manager sets the file type to the type that was written first. If your application has a bundle, you should designate an icon for the appropriate edition types that you can write.

RESULT CODES

noErr	0	No error
dskFulErr	–34	Disk is full
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
dirNFErr	–120	Directory not found
editionMgrInitErr	–450	Manager not initialized

SEE ALSO

For information on file specification records, see *Inside Macintosh: Files*. For an example of the use of CreateEditionContainerFile, see Listing 2-4 on page 2-33.

DeleteEditionContainerFile

If a user cancels a publisher section within a document or closes a document containing a newly created publisher without saving, you need to remove the edition container.

To locate the appropriate edition container to be deleted, use the GetEditionInfo function. You use the UnRegisterSection function (only after using the GetEditionInfo function) to unregister the section record and alias record of the publisher being canceled.

To remove the edition container, use the `DeleteEditionContainerFile` function.

```
FUNCTION DeleteEditionContainerFile (editionFile: FSSpec): OSErr;
```

editionFile
        The volume reference number, directory ID, and filename for the edition
        container being deleted.

DESCRIPTION

If the user cancels a publisher, do not call the `DeleteEditionContainerFile`
function until the user saves the document. This allows the user to undo changes and
revert to the last saved version of the document.

The `DeleteEditionContainerFile` function deletes the edition container only if
there is no registered publisher. You need to unregister a publisher before you can delete
its corresponding edition container.

You should use the `DeleteEditionContainerFile` function even if there are
subscribers to the edition. When a subscriber section tries to read in data, it receives
an error if the edition container has been deleted.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
dirNFErr	–120	Directory not found
editionMgrInitErr	–450	Manager not initialized

SEE ALSO

See page 2-98 for detailed information on the `GetEditionInfo` function. See page 2-77
for information on the `UnRegisterSection` function. For information on file
specification records, see *Inside Macintosh: Files*.

Setting and Getting a Format Mark

Use the `SetEditionFormatMark` function to set the current mark for a section
format and the `GetEditionFormatMark` function to get the current mark for a
particular format.

SetEditionFormatMark

A format mark indicates the next position of a read or write operation. Initially, a mark defaults to 0. After reading or writing data, the format mark is set past the last position written to or read from. To set the current mark for a given format, use the `SetEditionFormatMark` function.

```
FUNCTION SetEditionFormatMark (whichEdition: EditionRefNum;
                                   whichFormat: FormatType;
                                   setMarkTo: LongInt): OSErr;
```

whichEdition
: The reference number for the edition.

whichFormat
: The format type for the edition.

setMarkTo   The offset for the next read or write for this format.

DESCRIPTION

The `SetEditionFormatMark` function sets the current mark for the specified format type according to the value of the `setMarkTo` parameter.

RESULT CODES

noErr	0	No error
rfNumErr	−51	Bad edition reference number
noTypeErr	−102	Unknown format (subscriber only)
editionMgrInitErr	−450	Manager not initialized

GetEditionFormatMark

Use the `GetEditionFormatMark` function to get the current mark for a particular format.

```
FUNCTION GetEditionFormatMark (whichEdition: EditionRefNum;
                                   whichFormat: FormatType;
                                   VAR currentMark: LongInt): OSErr;
```

whichEdition
: The reference number for the edition.

whichFormat
: The format type whose mark you want to get.

currentMark
: The `GetEditionFormatMark` function returns the mark for the specified format in this parameter.

DESCRIPTION

If the edition does not support the format specified in the `whichFormat` parameter, you receive a `noTypeErr` result code.

RESULT CODES

noErr	0	No error
rfNumErr	–51	Bad edition reference number
noTypeErr	–102	Unknown format
editionMgrInitErr	–450	Manager not initialized

Reading in Edition Data

To initiate the reading of data from an edition (for a subscriber), use the `OpenEdition` function.

Use the `EditionHasFormat` function to learn in which formats the edition data is available.

Use the `ReadEdition` function to read data from an edition. This function reads from the current mark for the specified format.

OpenEdition

To initiate the reading of data from an edition (for a subscriber), use the `OpenEdition` function.

```
FUNCTION OpenEdition (subscriberSectionH: SectionHandle;
                        VAR refNum: EditionRefNum): OSErr;
```

subscriberSectionH
: A handle to the section record for a given section.

refNum
: The `OpenEdition` function returns the reference number for the edition in this parameter.

DESCRIPTION

The `OpenEdition` function opens an edition for reading and returns a reference number that your application can use to refer to this edition in other Edition Manager routines. Multiple subscribers can each call the `OpenEdition` function simultaneously (each call returns a different reference number) and read data from a single edition. If a publisher (located on a different machine) is writing to an edition when you use the `OpenEdition` function, you receive an `flLckedErr` result code.

RESULT CODES

noErr	0	No error
fnfErr	–43	File not found
flLckedErr	–45	Publisher writing to an edition
permErr	–54	Not a subscriber
editionMgrInitErr	–450	Manager not initialized

SEE ALSO

EditionHasFormat

Use the EditionHasFormat function to learn in which formats the edition data is available.

```
FUNCTION EditionHasFormat (whichEdition: EditionRefNum;
                           whichFormat: FormatType;
                           VAR formatSize: Size): OSErr;
```

whichEdition
:   The reference number for the edition.

whichFormat
:   The format type that you are requesting. For the whichFormat parameter, you should decide which formats to read in the same way that you do when reading data from the scrap. You can also get a list of all the available formats and their respective lengths by reading the kFormatListFormat ('fmts') format.

formatSize
:   The EditionHasFormat function returns the format length in this parameter.

DESCRIPTION

If the requested format is available, the EditionHasFormat function returns noErr, and the formatSize parameter returns the size of the data in the specified format or kFormatLengthUnknown (–1), which signifies that the size is unknown. You should therefore continue to read the format until there is no more data.

Note

The Translation Manager (if it is available) attempts implicit translation under certain circumstances. For instance, it does so when your application attempts to read from an edition a format type that is not in the edition. In this case, the Translation Manager attempts to translate the data into the requested format. For more information, see the chapter "Translation Manager" in *Inside Macintosh: More Macintosh Toolbox.* u

RESULT CODES

noErr	**0**	No error
rfNumErr	**−51**	Bad edition reference number
noTypeErr	**−102**	Format not available
editionMgrInitErr	**−450**	Manager not initialized

SEE ALSO

For an example of the use of EditionHasFormat, see Listing 2-7 beginning on page 2-42. For information about the Translation Manager and Scrap Manager, see *Inside Macintosh: More Macintosh Toolbox.*

ReadEdition

Use the ReadEdition function to read data from an edition. This function reads from the current mark for the specified format.

```
FUNCTION ReadEdition (whichEdition: EditionRefNum;
                      whichFormat: FormatType; buffPtr: UNIV Ptr;
                      VAR buffLen: Size): OSErr;
```

whichEdition
: The reference number for the edition.

whichFormat
: The format type that you want to read.

buffPtr
: A pointer to the buffer into which you want to read the data.

buffLen
: The number of bytes that you want to read into the buffer. The ReadEdition function returns the actual number of bytes read in the buffLen parameter.

DESCRIPTION

The `ReadEdition` function reads data from the edition into the specified buffer. `ReadEdition` returns in the `buffLen` parameter the total number of bytes read into the buffer. If the `buffLen` parameter returns a value smaller than the value you have specified, there is no additional data to read, and the `ReadEdition` function returns a `noErr` result code. If you use the `ReadEdition` function after all data is read in, the `ReadEdition` function returns an `eofErr` result code.

You can read data from an edition while a publisher on the same machine is writing data to the same edition. The data that you are reading is the old edition (not the data that the publisher is writing). If the publisher finishes writing data before you are through reading the old edition data, the `ReadEdition` function returns an `abortErr` result code. If the `ReadEdition` function returns an `abortErr` result code, you should stop trying to read data and use the `CloseEdition` function with the `successful` parameter set to `FALSE`.

Note

The Translation Manager (if it is available) attempts implicit translation under certain circumstances. For instance, it does so when your application attempts to read from an edition a format type that is not in the edition. In this case, the Translation Manager attempts to translate the data into the requested format. For more information, see the chapter "Translation Manager" in *Inside Macintosh: More Macintosh Toolbox.* u

RESULT CODES

`noErr`	0	No error
`abortErr`	–27	Publisher has written a new edition
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`eofErr`	–39	No more data of that format
`rfNumErr`	–51	Bad edition reference number
`noTypeErr`	–102	Format not available
`editionMgrInitErr`	–450	Manager not initialized

SEE ALSO

For an example of the use of `ReadEdition`, see Listing 2-7 beginning on page 2-42.

Writing out Edition Data

To initiate the writing of data from a publisher to its edition container, use the `OpenNewEdition` function. (To create an edition container, use the `CreateEditionContainerFile` function, as described on page 2-79.)

Use the `WriteEdition` function to write data to an edition.

OpenNewEdition

To initiate the writing of data from a publisher to its edition container, use the
OpenNewEdition function.

```
FUNCTION OpenNewEdition (publisherSectionH: SectionHandle;
                         fdCreator: OSType;
                         publisherSectionDocument: FSSpecPtr;
                         VAR refNum: EditionRefNum): OSErr;
```

publisherSectionH
: The publisher section that is writing to the edition.

fdCreator
: The Finder creator type of the new edition icon.

publisherSectionDocument
: The document that contains the publisher. This parameter is used to
create an alias from the edition to the publisher's document. If you pass
NIL for publisherSectionDocument, an alias is not made in the
edition file.

refNum
: The OpenNewEdition function returns the reference number
for the edition in this parameter. You specify this reference
number as a parameter for subsequent calls to WriteEdition,
SetEditionFormatMark, and CloseEdition to specify which
publisher is writing its data to an edition. If the edition cannot be opened
for writing because there is another publisher writing to it, or because the
file system does not allow writing, an error is returned and
OpenNewEdition sets refNum to NIL.

DESCRIPTION

The OpenNewEdition function opens an edition for writing. The function returns an
flLckdErr result code if there is a subscriber on another machine reading data from the
same edition. The OpenNewEdition function returns a permErr result code if there is a
registered publisher to that edition on another machine.

The Edition Manager allows two registered publishers that are located on the *same*
machine to write to the same edition. Note that multiple publishers cannot write to the
same edition simultaneously—only one publisher can write to an edition at a given time.

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
flLckdErr	–45	Edition in use by another section
permErr	–54	Registered publisher on another machine
wrPermErr	–61	Not a publisher
editionMgrInitErr	–450	Manager not initialized

WriteEdition

Use the `WriteEdition` function to write data to an edition. This function begins
writing at the current mark for the specified format.

```
FUNCTION WriteEdition (whichEdition: EditionRefNum;
                       whichFormat: FormatType;
                       buffPtr: UNIV Ptr; buffLen: Size): OSErr;
```

whichEdition
          The reference number for the edition.
whichFormat
          The format type that you want to write.
buffPtr   A pointer to the buffer containing the data to write to the edition.
buffLen   The number of bytes that you want to write to the edition.

DESCRIPTION

The `WriteEdition` function writes the specified number of bytes to the edition. If
the data cannot be entirely written to the edition, the `WriteEdition` function returns
an error.

RESULT CODES

noErr	0	No error
dskFulErr	−34	Disk is full
ioErr	−36	I/O error
rfNumErr	−51	Bad edition reference number
editionMgrInitErr	−450	Manager not initialized

Closing an Edition After Reading or Writing

After finishing reading from or writing to an edition, use the `CloseEdition` function to
close the edition.

CloseEdition

Use the `CloseEdition` function to close an edition after you finish reading from or writing to it.

```
FUNCTION CloseEdition (whichEdition: EditionRefNum;
                       successful: Boolean): OSErr;
```

whichEdition
The reference number for the edition.

successful
A value that indicates whether your application was successful (`TRUE`) or unsuccessful (`FALSE`) in reading from or writing data to the edition.

DESCRIPTION

When a subscriber successfully finishes reading data from the edition, the `CloseEdition` function takes the modification date of the edition file that you have read and puts it in the `mdDate` field of the subscriber's section record. This indicates that the data contained in the edition and the subscriber section within the document are the same.

When a subscriber is unsuccessful in reading data from an edition (because there is not enough memory, or you didn't find a format that you can read), set the `successful` parameter to `FALSE`. The `CloseEdition` function then closes the edition, but does not set the `mdDate` field. This implies that the subscriber is not updated with the latest edition.

When a publisher successfully finishes writing data to an edition, the `CloseEdition` function makes the data that the publisher has written to the edition available to any subscribers and sets the corresponding edition file's modification date (`ioFlMdDat`) to the `mdDate` field of the publisher's section record. The Edition Manager then sends a Section Read event to all current subscribers set to automatic update mode. At this point, the file type of the edition file is set based on the first known format that the publisher wrote.

When a publisher is unsuccessful in writing data to an edition, the `CloseEdition` function discards what the publisher has written to the edition. The data contained in the edition prior to writing remains unchanged, and Section Read events are not sent to subscribers.

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
fnOpnErr	–38	File not open
rfNumErr	–51	Bad edition reference number
editionMgrInitErr	–450	Manager not initialized

Displaying Dialog Boxes

The Edition Manager supports three dialog boxes: publisher, subscriber, and options
dialog boxes. Your application can display simple dialog boxes that appear centered on
the user's screen, or you can customize your dialog boxes.

Use the `GetLastEditionContainerUsed` function to get the default edition to
display.

Use the `NewSubscriberDialog` function to display the subscriber dialog box on the
user's screen and use the `NewPublisherDialog` function to display the publisher
dialog box on the user's screen. Unlike the Standard File Package routines, the
`NewPublisherDialog` and the `NewSubscriberDialog` functions allow you to
specify the initial volume reference number and directory ID so that there can be one
default location for editions for all applications.

You use the `SectionOptionsDialog` function to display the publisher options and
subscriber options dialog boxes on the user's screen.

The `NewSubscriberExpDialog`, `NewPublisherExpDialog`, and
`SectionOptionsExpDialog` functions are the same as the simple dialog functions but
have five additional parameters.

GetLastEditionContainerUsed

Use the `GetLastEditionContainerUsed` function to get the default edition to
display. This function allows a user to easily subscribe to the data recently published.

```
FUNCTION GetLastEditionContainerUsed
                    (VAR container: EditionContainerSpec): OSErr;
```

container   If the `GetLastEditionContainerUsed` function locates the last
             edition for which a section was created, the `container` parameter
             contains its volume reference number, directory ID, filename, and part,
             and returns a `noErr` result code. (The last edition created is associated
             with the last time that your application or another application located on
             the same machine used the `NewSection` function.)

DESCRIPTION

If the last edition used is missing, the `GetLastEditionContainerUsed` function
returns an `fnfErr` result code, but still returns the correct volume reference number and
directory ID that you should use for the `NewSubscriberDialog` function.

Pass the information from the GetLastEditionContainerUsed function to the
NewSubscriberDialog function.

RESULT CODES

noErr	0	No error
fnfErr	–43	Edition container not found
editionMgrInitErr	–450	Manager not initialized

SEE ALSO

For an example of the use of GetLastEditionContainerUsed, see Listing 2-6
beginning on page 2-40. For a description of the edition container record, see page 2-71.
The NewSubscriberDialog function is described next.

NewSubscriberDialog

When a user chooses the Subscribe To menu command, your application should call the
NewSubscriberDialog function to allow the user to choose an edition to subscribe to.

```
FUNCTION NewSubscriberDialog
                    (VAR reply: NewSubscriberReply): OSErr;
```

reply   The new subscriber reply record. You specify a location to use as the
            default edition container in the container field of this record. You also
            specify in the formatsMask field which edition format types
            NewSubscriberDialog should display. The NewSubscriberDialog
            function returns information concerning the user's choice in the
            canceled and container fields of this record.

```
TYPE NewSubscriberReply =
   RECORD
      canceled:     Boolean;                  {user canceled }
                                              { dialog box}
      formatsMask:  SignedByte;               {formats required}
      container:    EditionContainerSpec;{edition selected}
   END;
```

Field descriptions

canceled   The NewSubscriberDialog function returns in this field a value
                 that indicates whether the user canceled the dialog box. The
                 function returns TRUE in the canceled field if the user canceled
                 the dialog box. Otherwise, the function returns FALSE in this field
                 and returns in the container field the edition container for the
                 new subscriber.

formatsMask	The formatsMask field indicates which edition format type (text, graphics, and sound) to display within the subscriber dialog box. You can set the formatsMask field to the following constants: kTEXTformatMask (1), kPICTformatMask (2), or ksndFormatMask (4). To support a combination of formats, add the constants together. For example, a formatsMask of 3 displays both graphics and text edition format types in the subscriber dialog box.
container	The edition container of the last edition published or subscribed to. You provide in this parameter the location and filename to use as the default edition to subscribe to. If the user clicks the Subscribe button, NewSubscriberDialog returns FALSE in the canceled field and returns the selected edition container for the new subscriber in the container field.

DESCRIPTION

The NewSubscriberDialog function displays the subscriber dialog box on the user's screen. The NewSubscriberDialog function (which is based on the CustomGetFile procedure described in the chapter "Standard File Package" in *Inside Macintosh: Files*) switches to the volume reference number and directory ID and selects the filename of the edition container that you specified in the container field of the reply parameter. Use the GetLastEditionContainerUsed function to get the edition container of the last edition that was either published or subscribed to, then set the container field to this edition container. This allows the user to publish and then easily subscribe.

Note that if an edition does not contain either 'PICT', 'TEXT', or 'snd ' data, the NewSubscriberDialog function does not list the edition file in the new subscriber dialog box (unless you install an opener that can recognize the edition's data in response to the eoCanSubscribe verb).

RESULT CODES

noErr	0	No error
editionMgrInitErr	−450	Manager not initialized or could not load package
badSubPartErr	−454	Bad edition container spec

SEE ALSO

For an illustration of the new subscriber dialog box, see Figure 2-12 on page 2-37. For an example of the use of NewSubscriberDialog, see Listing 2-6 beginning on page 2-40. For a description of the edition container record, see page 2-71. For information on edition openers, see "Subscribing to Non-Edition Files" beginning on page 2-62.

NewPublisherDialog

When a user selects a portion of a document and then chooses the Create Publisher menu command, your application should call the `NewPublisherDialog` function to allow the user to choose a name and location of the edition to which your application writes the publisher data. Your application specifies a location and name to use as the default edition and provides a preview of the publisher data to the `NewPublisherDialog` function.

```
FUNCTION NewPublisherDialog
                    (VAR reply: NewPublisherReply): OSErr;
```

reply   A new publisher reply record. You specify a location to use as the default edition container in the `container` field of this record. You also specify information in the `usePart`, `preview`, and `previewFormat` fields. The `NewPublisherDialog` function returns information concerning the user's choice in the `canceled`, `replacing`, and `container` fields of this record.

```
TYPE NewPublisherReply =
    RECORD
        canceled:       Boolean;        {user canceled dialog box}
        replacing:      Boolean;        {user chose existing }
                                        { filename for an edition}
        usePart:        Boolean;        {always false in version 7.0}
        preview:        Handle;         {handle to 'prvw', 'PICT', }
                                        { 'TEXT', or 'snd ' data}
        previewFormat:  FormatType;     {type of preview}
        container:      EditionContainerSpec;
                                        {edition chosen}
    END;
```

Field descriptions

canceled   The `NewPublisherDialog` function returns in this field a value that indicates whether the user canceled the dialog box. The function returns `TRUE` in the `canceled` field if the user canceled the dialog box. The function returns `FALSE` in this field if the user clicked the Publish button and returns in the `container` field the edition container for the new publisher.

replacing   The `NewPublisherDialog` function returns `TRUE` in the `replacing` field if the user chose an existing filename from the list of available editions and confirmed this replacement. If the value of the `replacing` field is `TRUE`, do not call the `CreateEditionContainerFile` function. If the value of this field and the `canceled` field is `FALSE`, you can call `CreateEditionContainerFile` to create a new edition container.

usePart   A value that must be set to `FALSE` before calling the
                            `NewPublisherDialog` function.

preview   A handle to `'prvw'`, `'PICT'`, `'TEXT'`, or `'snd '` data. The
                            `NewPublisherDialog` function displays this data in the preview
                            area of the dialog box.

previewFormat   A value that indicates which type of data the handle in the
                            `preview` field references.

container   An edition container record that specifies the volume reference
                            number, directory ID, and filename to use as the default edition to
                            publish the data to. The `NewPublisherDialog` function returns in
                            this field the edition container that the user selected.

DESCRIPTION

The `NewPublisherDialog` function displays the new publisher dialog box on
the user's screen. The `NewPublisherDialog` function (which is based on the
`CustomPutFile` procedure described in the chapter "Standard File Package" in
Inside Macintosh: Files) switches to the volume reference number and directory ID
specified by the edition container, sets the editable text item to the filename specified by
the edition container, and displays a preview of the publisher data in the new publisher
dialog box. The `NewPublisherDialog` function handles all user interaction until the
user clicks the Cancel or Publish button.

You should deallocate the handle referenced by the `preview` field to free up memory.

RESULT CODES

noErr	0	No error
editionMgrInitErr	−450	Manager not initialized or could not load package
badSubPartErr	−454	Bad edition container spec

SEE ALSO

For an illustration of the new publisher dialog box, see Figure 2-11 on page 2-29. For an
example of the use of `NewPublisherDialog`, see Listing 2-4 beginning on page 2-33.
For a description of the edition container record, see page 2-71.

SectionOptionsDialog

Use the `SectionOptionsDialog` function to display the publisher options and
subscriber options dialog boxes on the user's screen.

```
FUNCTION SectionOptionsDialog
                    (VAR reply: SectionOptionsReply): OSErr;
```

reply   The `reply` parameter contains a section options reply record. You specify
                 a handle to the publisher's or subscriber's section record in the
                 `sectionH` field of this record. The `SectionOptionsDialog` function
                 returns information concerning the user's actions in the `canceled`,
                 `changed`, and `action` fields.

```
TYPE SectionOptionsReply =
   RECORD
      canceled:   Boolean;              {user canceled dialog box}
      changed:    Boolean;              {changed the section record}
      sectionH:   SectionHandle;        {handle to the specified }
                                        { section record}
      action:     ResType;             {action codes}
   END;
```

Field descriptions

canceled   The `SectionOptionsDialog` function returns in this field a value
                 that indicates whether the user canceled the dialog box. The
                 function returns `TRUE` in the `canceled` field if the user canceled
                 the dialog box. Otherwise, the function returns `FALSE` in this field.

changed   The `SectionOptionsDialog` function returns `TRUE` in this field if
                 the user changed the section record. For example, the update mode
                 may have changed. Otherwise, the function returns `FALSE` in this
                 field.

sectionH   A handle to the section record for the section the user selected.

action   The `SectionOptionsDialog` function returns in this field the
                 code for one of five user actions: action code `'read'` for user
                 selection of the Get Edition Now button, action code `'writ'` for
                 user selection of the Send Edition Now button, action code `'goto'`
                 for user selection of the Open Publisher button, action code
                 `'cncl'` for user selection of the Cancel Publisher or Cancel
                 Subscriber button, or action code `'    '` ($20202020) for user
                 selection of the OK button.

DESCRIPTION

The `SectionOptionsDialog` function displays the appropriate options dialog box for
the specified section record. The function displays information about the subscriber or
publisher, such as its latest edition and current update mode setting, and allows the user
to perform various actions. The `SectionOptionsDialog` function handles all user
interaction until the user selects a button. The function returns the user's action in the
`action` field of the `reply` parameter; your application should then perform the
corresponding action.

RESULT CODES

noErr	0	No error
memFullErr	–108	Memory full

NewSubscriberExpDialog, NewPublisherExpDialog, SectionOptionsExpDialog

The NewSubscriberExpDialog, NewPublisherExpDialog, and SectionOptionsExpDialog functions are the same as the simple dialog functions but have five additional parameters. These additional parameters allow you to add items to the bottom of the dialog boxes, apply alternate mapping of events to item hits, apply alternate meanings to the item hits, and choose the location of the dialog boxes.

```
FUNCTION NewSubscriberExpDialog
                (VAR reply: NewSubscriberReply; where: Point;
                 expansionDITLresID: Integer;
                 dlgHook: ExpDlgHookProcPtr;
                 filterProc: ExpModalFilterProcPtr;
                 yourDataPtr: UNIV Ptr): OSErr;


FUNCTION NewPublisherExpDialog
                (VAR reply: NewPublisherReply; where: Point;
                 expansionDITLresID: Integer;
                 dlgHook: ExpDlgHookProcPtr;
                 filterProc: ExpModalFilterProcPtr;
                 yourDataPtr: UNIV Ptr): OSErr;


FUNCTION SectionOptionsExpDialog
                (VAR reply: SectionOptionsReply; where: Point;
                 expansionDITLresID: Integer;
                 dlgHook: ExpDlgHookProcPtr;
                 filterProc: ExpModalFilterProcPtr;
                 yourDataPtr: UNIV Ptr): OSErr;
```

reply   A new subscriber reply, new publisher reply, or section options
              reply record. You specify information in the fields of this record
              just as you do in the the corresponding fields of records used by
              NewSubscriberDialog, NewPublisherDialog, and
              SectionOptionsDialog.

where   A point that specifies a location on the screen where the function displays
              the dialog box. You can automatically center the dialog box by passing
              (–1, –1) in the where parameter.

expansionDITLresID

A value of 0 or a valid item list (`'DITL'`) resource ID. This integer is the ID of a dialog item list whose items are appended to the end of the standard dialog item list. The dialog items keep their relative positions, but they are moved as a group to the bottom of the dialog box.

dlgHook

A pointer to an expandable dialog hook function or `NIL`. An expandable dialog hook function is similar to a dialog hook function except that an expandable dialog hook function accepts an additional parameter. The `NewSubscriberExpDialog`, `NewPublisherExpDialog`, and `SectionOptionsExpDialog` functions call your expandable dialog hook function after each call to the `ModalDialog` procedure. The expandable dialog hook function should take the appropriate action, such as filling in a checkbox. The `itemOffset` parameter to the expandable dialog hook function is the number of items in the item list before your expansion dialog items. You need to subtract the item offset from the item hit to get the relative item number in the expansion item list. The expandable dialog hook function should return as its function result the absolute item number.

filterProc

A pointer to an expandable modal-dialog filter function or `NIL`. An expandable modal-dialog filter function is similar to a modal-dialog filter function or event filter function except that an expandable modal-dialog filter function accepts two extra parameters. The `ModalDialog` procedure calls the expandable modal-dialog filter function you provide in this parameter. An expandable modal-dialog filter function allows you to map real events (such as a mouse-down event) to an item hit (such as clicking a Cancel button). For instance, you may want to map a keyboard equivalent to an item hit.

yourDataPtr

Reserved for your use. It is passed back to your hook and event filter function. This parameter does not have to be of type `Ptr`—it can be any 32-bit quantity that you want. In Pascal, you can pass `yourDataPtr` in register A6, and declare your dialog hook and event filter as local functions without the last parameter. The stack frame is set up properly for these functions to access their parent local variables.

DESCRIPTION

The `NewPublisherExpDialog`, `NewSubscriberExpDialog`, and `SectionOptionsExpDialog` functions display the appropriate dialog box, handle user interaction, and call any functions you have provided in the `dlgHook` and `filterProc` parameters.

For the `NewPublisherExpDialog` and `NewSubscriberExpDialog` functions, all the pseudo-items for the Standard File Package such as `hookFirstCall`(–1), `hookNullEvent`(100), `hookRebuildList`(101), and `hookLastCall`(–2) can be used, as well as `hookRedrawPreview`(150).

For the `SectionOptionsExpDialog` function, the only valid pseudo-items are `hookFirstCall`(–1), `hookNullEvent`(100), `hookLastCall`(–2), `emHookRedrawPreview`(150), `emHookCancelSection`(160), `emHookGoToPublisher`(161), `emHookGetEditionNow`(162), `emHookSendEditionNow`(162), `emHookManualUpdateMode`(163), and `emHookAutoUpdateMode`(164).

If you provide an expandable dialog hook function, it must contain the following parameters:

```
FUNCTION MyExpDlgHook (itemOffset: Integer; itemHit: Integer;
                       theDialog: DialogPtr;
                       yourDataPtr: Ptr): Integer;
```

If you provide an expandable modal-dialog filter function, it must contain the following parameters.

```
FUNCTION MyExpModalFilter (theDialog: DialogPtr;
                           VAR theEvent: EventRecord;
                           itemOffset: Integer;
                           VAR itemHit: Integer;
                           yourDataPtr: Ptr): Boolean;
```

SEE ALSO

See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for additional information on item lists. See the chapter "Standard File Package" in *Inside Macintosh: Files* for information on dialog hook and modal-dialog filter functions.

Locating a Publisher and Edition From a Subscriber

The `GetEditionInfo` function returns information about a section's edition such as its location, last modification date, creator, and type.

Once you locate a section's edition, you can use the `GoToPublisherSection` function to find the document containing the publisher.

GetEditionInfo

Use the `GetEditionInfo` function to obtain information about a section's edition, such as its location, last modification date, creator, and type.

```
FUNCTION GetEditionInfo
                    (sectionH: SectionHandle;
                     VAR editionInfo: EditionInfoRecord): OSErr;
```

sectionH   A handle to the section record for a given section.

editionInfo
              An edition information record. The GetEditionInfo function returns
              the public information contained in the section's control block.

DESCRIPTION

The Edition Manager ensures that the existing edition name corresponds to the Finder's
existing edition name. If the controlBlock field of the section record is set to NIL or
the edition cannot be located, the GetEditionInfo function returns an fnfErr
result code.

The GetEditionInfo function returns information about the section's edition in a data
structure of type EditionInfoRecord.

```
TYPE EditionInfoRecord =
   RECORD
      crDate:    TimeStamp;              {date edition container }
                                         { was created}
      mdDate:    TimeStamp;              {date of last change}
      fdCreator: OSType;                 {file creator}
      fdType:    OSType;                 {file type}
      container: EditionContainerSpec;{the edition}
   END;
```

Field descriptions

crDate   The creation date of the edition.

mdDate   The modification date of the edition.

fdCreator   The creator of the edition file.

fdType   The file type of the edition file.

container   An edition container record, which specifies the volume reference
              number, directory ID, filename, script, and part number for the
              edition.

RESULT CODES

noErr	0	No error
fnfErr	–43	Not registered or file moved
editionMgrInitErr	–450	Manager not initialized

SEE ALSO

For an example of the use of GetEditionInfo, see Listing 2-8 beginning on page 2-46.
For another use of this function, see "Canceling Sections Within Documents" beginning
on page 2-48. For a description of the edition container record, see page 2-71.

GoToPublisherSection

When the user wants to locate the publisher for a particular subscriber (by clicking Open Publisher in the subscriber options dialog box), the SectionOptionsDialog function returns the action code 'goto' in the action field of the section options reply record. When you receive this action code, you should open the document containing the publisher.

First, use the GetEditionInfo function to find the edition container. Then use the GoToPublisherSection function to open the document containing the publisher.

```
FUNCTION GoToPublisherSection
                          (container: EditionContainerSpec): OSErr;
```

container   An edition container record, which specifies volume reference number, directory ID, and filename of the subscriber's edition. You obtain the edition container by calling the GetEditionInfo function.

DESCRIPTION

The GoToPublisherSection function resolves the alias in the edition to find the document containing its publisher. In general, this function internally uses the GetStandardFormats function to get the alias to the publisher document and then resolves the alias. It next sends the Finder an Apple event to open the document (which launches its application if necessary) and, after the publisher is registered, sends a Section Scroll event to the publisher.

As an optimization, if there is a registered publisher, the GoToPublisherSection function simply sends a Section Scroll event to the publisher.

If the edition does not contain an alias and there are no registered publishers, then the GoToPublisherSection function sends an Open Documents event to open the edition to the creating application.

If the edition container is not an edition file (as is the case when you are using bottlenecks to subscribe to non-edition files), the GoToPublisherSection function sends the Finder an Apple event to open that file.

RESULT CODES

noErr	0	No error
fnfErr	–43	File not found
editionMgrInitErr	–450	Manager not initialized
badSubPartErr	–454	Invalid edition container

Edition Container Formats

The Edition Manager calls the `GetStandardFormats` function to get the alias used in the `GoToPublisherSection` function and to get the preview shown in the subscriber dialog box. You probably do not need to call this function directly.

GetStandardFormats

You probably do not need to call the `GetStandardFormats` function directly because the Edition Manager calls this function.

```
FUNCTION GetStandardFormats
                (container: EditionContainerSpec;
                 VAR previewFormat: FormatType;
                 preview, publisherAlias, formats: Handle): OSErr;
```

container   An edition container record that specifies the edition volume reference number, directory ID, filename, and part.

previewFormat
           The `GetStandardFormats` function returns in this parameter a handle to the first format of the requested format type that it finds in the edition.

preview   A format type. The `GetStandardFormats` function looks for a format of the type specified in this parameter and returns in this parameter the format type of the first format that it finds. The function tries to find one of four formats: `'prvw'`, `'PICT'`, `'TEXT'`, or `'snd '`.

publisherAlias
           The `publisherAlias` parameter reads the format `kPublisherDocAliasFormat` (`'alis'`).

formats   The `formats` parameter reads the virtual format `kFormatListFormat` (`'fmts'`).

DESCRIPTION

You should pass in valid handles for the formats that you want and `NIL` for the formats that you don't want. The handles are resized to the size of the data.

If one of the requested formats cannot be found, `GetStandardFormats` returns a `noTypeErr` result code.

RESULT CODES

noErr	0	No error
noTypeErr	–102	Edition container not found
editionMgrInitErr	–450	Manager not initialized

Reading and Writing Non-Edition Files

The Edition Manager never opens or closes an edition container directly—it calls the current edition opener. See "Subscribing to Non-Edition Files" beginning on page 2-62 for additional information.

To override the standard opener function, create an opener function that contains the following parameters:

```
FUNCTION MyOpener (selector: EditionOpenerVerb;
                    VAR PB: EditionOpenerParamBlock): OSErr;
```

When this function is called by the Edition Manager, the `selector` parameter is set to one of the edition opener verbs (`eoOpen`, `eoClose`, `eoOpenNew`, `eoCloseNew`, `eoCanSubscribe`). The `PB` parameter contains an edition opener parameter block record.

Use the `GetEditionOpenerProc` function to locate the current edition opener and use the `SetEditionOpenerProc` function to provide your own edition opener.

Use the `CallEditionOpenerProc` function to call an edition opener and use the `CallFormatIOProc` function to call a format I/O function.

GetEditionOpenerProc

Use the `GetEditionOpenerProc` function to locate the current edition opener.

```
FUNCTION GetEditionOpenerProc
                    (VAR opener: EditionOpenerProcPtr): OSErr;
```

opener   The `GetEditionOpenerProc` function returns a pointer to the current edition opener function in this parameter.

SetEditionOpenerProc

Use the `SetEditionOpenerProc` function to provide your own edition opener.

```
FUNCTION SetEditionOpenerProc
                    (opener: EditionOpenerProcPtr): OSErr;
```

opener   A pointer to the edition opener function that you are providing.

CallEditionOpenerProc

Use the `CallEditionOpenerProc` function to call an edition opener.

```
FUNCTION CallEditionOpenerProc
                    (selector: EditionOpenerVerb;
                     VAR PB: EditionOpenerParamBlock;
                     routine: EditionOpenerProcPtr): OSErr;
```

selector   An edition opener verb. When the `CallEditionOpenerProc` function
              is called by the Edition Manager, the `selector` parameter is set to one of
              the edition opener verbs (`eoOpen`, `eoClose`, `eoOpenNew`, `eoCloseNew`,
              `eoCanSubscribe`).

PB   An edition opener parameter block.

routine   A pointer to an edition opener function.

DESCRIPTION

The Edition Manager calls an edition opener function whenever it needs to open or close
an edition. The Edition Manager passes an edition opener parameter block as one of the
parameters to an edition opener function. The edition opener parameter block is defined
by this structure:

```
TYPE EditionOpenerParamBlock =
   RECORD
      info:          EditionInfoRecord;    {edition container to }
                                           { be subscribed to}
      sectionH:      SectionHandle;        {publisher or }
                                           { subscriber }
                                           { requesting open}
      document:      FSSpecPtr;            {document passed}
      fdCreator:     OSType;               {Finder creator type}
      ioRefNum:      LongInt;              {reference number}
      ioProc:        FormatIOProcPtr;      {routine to read }
                                           { formats}
      success:       Boolean;              {reading or writing }
                                           { was successful}
      formatsMask:   SignedByte;           {formats required to }
                                           { subscribe}
   END;
```

To override the standard reading and writing functions, you should create an I/O function that contains the following parameters.

```
FUNCTION MyIO (selector: FormatIOVerb;
                VAR PB: FormatIOParamBlock): OSErr;
```

Set the `selector` parameter to one of the format I/O verbs (`ioHasFormat`, `ioReadFormat`, `ioNewFormat`, `ioWriteFormat`). The `PB` parameter contains a format I/O parameter block record.

SEE ALSO

See "Calling an Edition Opener" beginning on page 2-64 for additional information.

CallFormatIOProc

Use the `CallFormatIOProc` function to call a format I/O function.

```
FUNCTION CallFormatIOProc (selector: FormatIOVerb;
                            VAR PB:FormatIOParamBlock;
                            routine: FormatIOProcPtr): OSErr;
```

selector   A format I/O verb (`ioHasFormat`, `ioReadFormat`, `ioNewFormat`, `ioWriteFormat`).

PB   A format I/O parameter block record.

routine   A pointer to a format I/O function.

DESCRIPTION

The Edition Manager calls a format I/O function whenever it needs to read from or write to an edition. The Edition Manager passes a format I/O parameter block as one of the parameters to a format I/O procedure. The format I/O parameter block is defined by this structure:

```
TYPE FormatIOParamBlock =
    RECORD
        ioRefNum:       LongInt;        {reference number}
        format:         FormatType;     {edition format type}
        formatIndex:    LongInt;        {opener-specific enumeration }
                                        { of formats}
        offset:         LongInt;        {offset into format}
        buffPtr:        Ptr;            {data starts here}
        buffLen:        LongInt;        {length of data}
    END;
```

See "Calling a Format I/O Function" beginning on page 2-68 for additional information.

Application-Defined Routines

Your application can provide an edition opener function, format I/O function, expandable dialog hook function, and expandable modal-dialog filter function. For the routine declarations of the edition opener and format I/O functions, see "Reading and Writing Non-Edition Files" beginning on page 2-102. For the routine declarations of the expandable dialog hook and expandable modal-dialog filter functions, see the description of `NewSubscriberExpDialog`, `NewPublisherExpDialog`, and `SectionOptionsExpDialog` beginning on page 2-96.

Summary of the Edition Manager

Pascal Summary

Constants

```
CONST
  {resource types}
  rSectionType                 = 'sect';    {resource type for a }
                                            { section}
  {section types}
  stSubscriber                 = $01;       {subscriber section type}
  stPublisher                  = $0A;       {publisher section type}
  {update modes}
  sumAutomatic                 = 0;         {subscriber receives new }
                                            { editions automatically}
  sumManual                    = 1;         {subscriber receives new }
                                            { editions manually}
  pumOnSave                    = 0;         {publisher sends new }
                                            { editions on save}
  pumManual                    = 1;         {publisher does not send }
                                            { new editions until user }
                                            { request}
  {edition container subpart number}
  kPartsNotUsed                = 0;         {edition is the whole file}
  kPartNumberUnknown           = -1;        {not used in version 7.0}

  {preview size}
  kPreviewWidth                = 120;       {preview width}
  kPreviewHeight               = 120;       {preview height}

  {special formats}
  kPublisherDocAliasFormat     = 'alis';    {alias record from the }
                                            { edition to publisher}
  kPreviewFormat               = 'prvw';    {'PICT' thumbnail sketch}
  kFormatListFormat            = 'fmts';    {list of all available }
                                            { formats and their sizes}
```

```
{bits for formatMask}
kPICTformatMask              = 1;        {graphics format}
kTEXTformatMask              = 2;        {text format}
ksndFormatMask               = 4;        {sound format}


{Finder types for edition files}
kPICTEditionFileType         = 'edtp';   {contains 'PICT', }
kTEXTEditionFileType         = 'edtt';   { 'TEXT', and }
ksndEditionFileType          = 'edts';   { 'snd ' file types}
kUnknownEditionFileType      = 'edtu';   {unknown file type}
{miscellaneous}
kFormatLengthUnknown         = -1;       {length of format unknown}


{message IDs for Apple events sent by the Edition Manager}
sectionEventMsgClass         = 'sect';   {Apple events sent by the }
                                         { Edition Manager}
sectionReadMsgID             = 'read';   {Section Read events}
sectionWriteMsgID            = 'writ';   {Section Write events}
sectionScrollMsgID           = 'scrl';   {Section Scroll events}
sectionCancelMsgID           = 'cncl';   {Section Cancel events}


{refCon field when displaying stacked dialog boxes}
sfMainDialogRefCon           = 'stdf';   {new publisher and }
                                         { new subscriber}
sfNewFolderDialogRefCon      = 'nfdr';   {new folder}
sfReplaceDialogRefCon        = 'rplc';   {replace dialog}
sfStatWarnDialogRefCon       = 'stat';   {warning dialog}
sfErrorDialogRefCon          = 'err ';   {error dialog}
emOptionsDialogRefCon        = 'optn';   {options dialog}
emCancelSectionDialogRefCon  = 'cncl';   {cancel section}
emGotoPubErrDialogRefCon     = 'gerr';   {locate publisher}


{pseudo-item hits for dialogHooks}
emHookRedrawPreview          = 150;      {for NewPublisher or }
                                         { NewSubscriber dialogs}
emHookCancelSection          = 160;      {for SectionOptions dialog}
emHookGoToPublisher          = 161;      {for SectionOptions dialog}
emHookGetEditionNow          = 162;      {for SectionOptions dialog}
emHookSendEditionNow         = 162;      {for SectionOptions dialog}
emHookManualUpdateMode       = 163;      {for SectionOptions dialog}
emHookAutoUpdateMode         = 164;      {for SectionOptions dialog}
```

Data Types

```
TYPE TimeStamp                   = LongInt;      {seconds since 1904}
     EditionRefNum               = Handle;       {for use in Edition I/O}
     UpdateMode                  = Integer;      {sumAutomatic, }
                                                 { sumManual, }
                                                 { pumOnSave, pumManual}
     SectionType                 = SignedByte;   {stSubscriber or }
                                                 { stPublisher}
     FormatType                  = PACKED ARRAY[1..4] OF CHAR;
                                                 {similar to ResType used }
                                                 { by the Scrap Manager}

     SectionHandle               = ^SectionPtr;
     SectionPtr                  = ^SectionRecord;
     SectionRecord               =
     RECORD
        version:                 SignedByte;     {always 1 in version 7.0}
        kind:                    SectionType;    {publisher or subscriber}
        mode:                    UpdateMode;     {automatic or manual}
        mdDate:                  TimeStamp;      {last change to section}
        sectionID:               LongInt;        {application-specific, }
                                                 { unique per document}
        refCon:                  LongInt;        {application-specific}
        alias:                   AliasHandle;    {handle to alias record}

        {The following fields are private and are set up by the }
        { RegisterSection function.}

        subPart:                 LongInt;        {private}
        nextSection:             SectionHandle;  {private}
        controlBlock:            Handle;         {private}
        refNum:                  EditionRefNum;  {private}
     END;

     EditionContainerSpecPtr =^EditionContainerSpec;
     EditionContainerSpec =
     RECORD
        theFile:                 FSSpec;         {file containing edition }
                                                 { data}
        theFileScript:           ScriptCode;     {script code of filename}
        thePart:                 LongInt;        {which part of file, }
                                                 { always kPartsNotUsed}
```

```
   thePartName:                 Str31;          {reserved}
   thePartScript:               ScriptCode;     {reserved}
END;

FormatsAvailable = ARRAY[0..0] OF
RECORD
   theType:                     FormatType;     {format type for an }
                                                { edition}
   theLength:                   LongInt;        {length of edition format }
                                                { type}
END;

EditionInfoRecord =
RECORD
   crDate:                      TimeStamp;      {date edition container }
                                                { was created}
   mdDate:                      TimeStamp;      {date of last change}
   fdCreator:                   OSType;         {file creator}
   fdType:                      OSType;         {file type}
   container:                   EditionContainerSpec;
                                                {the edition}
END;

NewPublisherReply =
RECORD
   canceled:        Boolean;                    {user canceled dialog box}
   replacing:       Boolean;                    {user chose existing }
                                                { filename for an edition}
   usePart:         Boolean;                    {always FALSE in version 7.0}
   preview:         Handle;                     {handle to 'prvw', 'PICT',}
                                                { 'TEXT', or 'snd ' data}
   previewFormat:   FormatType;                 {type of preview}
   container:       EditionContainerSpec;
                                                {edition chosen}
END;

NewSubscriberReply =
RECORD
   canceled:        Boolean;                    {user canceled dialog box}
   formatsMask:     SignedByte;                 {formats required}
   container:       EditionContainerSpec;
                                                {edition selected}
END;
```

```
SectionOptionsReply =
RECORD
   canceled:        Boolean;              {user canceled dialog box}
   changed:         Boolean;              {changed the section }
                                          { record}
   sectionH:        SectionHandle;        {handle to the specified }
                                          { section record}
   action:          ResType;              {action codes}
END;


EditionOpenerVerb= (eoOpen, eoClose, eoOpenNew, eoCloseNew,
                       eoCanSubscribe);
EditionOpenerParamBlock =
RECORD
   info:            EditionInfoRecord;    {edition container to }
                                          { be subscribed to}
   sectionH:        SectionHandle;        {publisher or subscriber }
                                          { requesting open}
   document:        FSSpecPtr;            {document passed}
   fdCreator:       OSType;               {Finder creator type}
   ioRefNum:        LongInt;              {reference number}
   ioProc:          FormatIOProcPtr;      {routine to read formats}
   success:         Boolean;              {reading or writing was }
                                          { successful}
   formatsMask:     SignedByte;           {formats required to }
                                          { subscribe}
END;


FormatIOVerb = (ioHasFormat, ioReadFormat, ioNewFormat, ioWriteFormat);
FormatIOParamBlock =
RECORD
   ioRefNum:        LongInt;              {reference number}
   format:          FormatType;           {edition format type}
   formatIndex:     LongInt;              {opener-specific enumeration }
                                          { of formats}
   offset:          LongInt;              {offset into format}
   buffPtr:         Ptr;                  {data starts here}
   buffLen:         LongInt;              {length of data}
END;
```

Edition Manager Routines

Initializing the Edition Manager

```
FUNCTION InitEditionPack    : OSErr;
```

Creating and Registering a Section

```
FUNCTION NewSection             (container: EditionContainerSpec;
                                 sectionDocument: FSSpecPtr; kind: SectionType;
                                 sectionID: LongInt; initialMode: UpdateMode;
                                 VAR sectionH: SectionHandle): OSErr;
FUNCTION RegisterSection        (sectionDocument: FSSpec;
                                 sectionH: SectionHandle;
                                 VAR aliasWasUpdated: Boolean)
                                 : OSErr;
FUNCTION UnRegisterSection   (sectionH: SectionHandle): OSErr;
FUNCTION IsRegisteredSection
                                (sectionH: SectionHandle): OSErr;
FUNCTION AssociateSection     (sectionH: SectionHandle;
                                 newSectionDocument: FSSpecPtr): OSErr;
```

Creating and Deleting an Edition Container

```
FUNCTION CreateEditionContainerFile
                                (editionFile: FSSpec; fdCreator: OSType;
                                 editionFileNameScript: ScriptCode): OSErr;
FUNCTION DeleteEditionContainerFile
                                (editionFile: FSSpec): OSErr;
```

Setting and Getting a Format Mark

```
FUNCTION SetEditionFormatMark
                                (whichEdition: EditionRefNum;
                                 whichFormat: FormatType;
                                 setMarkTo: LongInt): OSErr;
FUNCTION GetEditionFormatMark
                                (whichEdition: EditionRefNum;
                                 whichFormat: FormatType;
                                 VAR currentMark: LongInt): OSErr;
```

Reading in Edition Data

```
FUNCTION OpenEdition        (subscriberSectionH: SectionHandle;
                             VAR refNum: EditionRefNum): OSErr;
FUNCTION EditionHasFormat   (whichEdition: EditionRefNum;
                             whichFormat: FormatType;
                             VAR formatSize: Size): OSErr;
FUNCTION ReadEdition        (whichEdition: EditionRefNum;
                             whichFormat: FormatType; buffPtr: UNIV Ptr;
                             VAR buffLen: Size): OSErr;
```

Writing out Edition Data

```
FUNCTION OpenNewEdition     (publisherSectionH: SectionHandle;
                             fdCreator: OSType;
                             publisherSectionDocument: FSSpecPtr;
                             VAR refNum: EditionRefNum): OSErr;
FUNCTION WriteEdition       (whichEdition: EditionRefNum;
                             whichFormat: FormatType; buffPtr: UNIV Ptr;
                             buffLen: Size): OSErr;
```

Closing an Edition After Reading or Writing

```
FUNCTION CloseEdition       (whichEdition: EditionRefNum;
                             successful: Boolean): OSErr;
```

Displaying Dialog Boxes

```
FUNCTION GetLastEditionContainerUsed
                            (VAR container: EditionContainerSpec): OSErr;
FUNCTION NewSubscriberDialog
                            (VAR reply: NewSubscriberReply): OSErr;
FUNCTION NewPublisherDialog (VAR reply: NewPublisherReply): OSErr;
FUNCTION SectionOptionsDialog
                            (VAR reply: SectionOptionsReply): OSErr;
FUNCTION NewSubscriberExpDialog
                            (VAR reply: NewSubscriberReply; where: Point;
                             expansionDITLresID: Integer;
                             dlgHook: ExpDlgHookProcPtr;
                             filterProc: ExpModalFilterProcPtr;
                             yourDataPtr: UNIV Ptr): OSErr;
```

```
FUNCTION NewPublisherExpDialog
                            (VAR reply: NewPublisherReply; where: Point;
                             expansionDITLresID: Integer;
                             dlgHook: ExpDlgHookProcPtr;
                             filterProc: ExpModalFilterProcPtr;
                             yourDataPtr: UNIV Ptr): OSErr;
FUNCTION SectionOptionsExpDialog
                            (VAR reply: SectionOptionsReply; where: Point;
                             expansionDITLresID: Integer;
                             dlgHook: ExpDlgHookProcPtr;
                             filterProc: ExpModalFilterProcPtr;
                             yourDataPtr: UNIV Ptr): OSErr;
```

Locating a Publisher and Edition From a Subscriber

```
FUNCTION GetEditionInfo      (sectionH: SectionHandle;
                             VAR editionInfo: EditionInfoRecord): OSErr;
FUNCTION GoToPublisherSection
                            (container: EditionContainerSpec): OSErr;
```

Edition Container Formats

```
FUNCTION GetStandardFormats (container: EditionContainerSpec;
                             VAR previewFormat: FormatType;
                             preview, publisherAlias,
                             formats: Handle): OSErr;
```

Reading and Writing Non-Edition files

```
FUNCTION GetEditionOpenerProc
                            (VAR opener: EditionOpenerProcPtr): OSErr;
FUNCTION SetEditionOpenerProc
                            (opener: EditionOpenerProcPtr): OSErr;
FUNCTION CallEditionOpenerProc
                            (selector: EditionOpenerVerb;
                             VAR PB: EditionOpenerParamBlock;
                             routine: EditionOpenerProcPtr): OSErr;
FUNCTION CallFormatIOProc    (selector: FormatIOVerb;
                             VAR PB: FormatIOParamBlock;
                             routine: FormatIOProcPtr): OSErr;
```

Application-Defined Routines

```
FUNCTION MyExpDlgHook        (itemOffset: Integer; itemHit: Integer;
                             theDialog: DialogPtr;
                             yourDataPtr: Ptr): Integer;
```

```
FUNCTION MyExpModalFilter    (theDialog: DialogPtr;
                              VAR theEvent: EventRecord;
                              itemOffset: Integer; VAR itemHit: Integer;
                              yourDataPtr: Ptr): Boolean;
FUNCTION MyOpener            (selector: EditionOpenerVerb;
                              VAR PB: EditionOpenerParamBlock): OSErr;
FUNCTION MyIO                (selector: FormatIOVerb;
                              VAR PB: FormatIOParamBlock): OSErr;
```

C Summary

Constants

```
CONST
enum {
   /*resource types*/
   #define rSectionType           'sect'      /*resource type for a */
                                              /* section*/


   /*section types*/
   stSubscriber             = 0x01,     /*subscriber section type*/
   stPublisher              = 0x0A,     /*publisher section type*/


   /*update modes*/
   sumAutomatic             = 0,        /*subscriber receives new */
                                        /* editions automatically*/
   sumManual                = 1,        /*subscriber receives new */
                                        /* editions manually*/
   pumOnSave                = 0,        /*publisher sends new */
                                        /* editions on save*/
   pumManual                = 1,        /*publisher does not send */
                                        /* new editions until user */
                                        /* request*/


   /*edition container subpart number*/
   kPartsNotUsed            = 0,        /*edition is the whole file*/
   kPartNumberUnknown       = -1,       /*not used in version 7.0*/


   /*preview size*/
   kPreviewWidth            = 120,      /*preview width*/
   kPreviewHeight           = 120,      /*preview height*/
```

```
/*special formats*/
#define kPublisherDocAliasFormat 'alis'   /*alias record from the */
                                          /* edition to publisher*/
#define kPreviewFormat           'prvw'   /*'PICT' thumbnail sketch*/
#define kFormatListFormat        'fmts'   /*list of all available */
                                          /* formats and their sizes*/


/*bits for formatMask*/
kPICTformatMask             = 1,          /*graphics format*/
kTEXTformatMask             = 2,          /*text format*/
ksndFormatMask              = 4,          /*sound format*/


/*Finder types for edition files*/
#define kPICTEditionFileType    'edtp'    /*contains 'PICT', */
#define kTEXTEditionFileType    'edtt'    /* 'TEXT', and */
#define ksndEditionFileType     'edts'    /* 'snd ' file types*/
#define kUnknownEditionFileType 'edtu'    /*unknown file type*/


/*pseudo-item hits for dialogHooks*/
emHookRedrawPreview         = 150,        /*for NewPublisher or */
                                          /* NewSubscriber dialogs*/
emHookCancelSection         = 160,        /*for SectionOptions dialog*/
emHookGoToPublisher         = 161,        /*for SectionOptions dialog*/
emHookGetEditionNow         = 162,        /*for SectionOptions dialog*/
emHookSendEditionNow        = 162,        /*for SectionOptions dialog*/
emHookManualUpdateMode      = 163,        /*for SectionOptions dialog*/
emHookAutoUpdateMode        = 164         /*for SectionOptions dialog*/
};
   /*edition opener verbs*/
enum {eoOpen, eoClose, eoOpenNew, eoCloseNew, eoCanSubscribe};

enum {
   /*refCon field when displaying stacked dialog boxes*/
   #define emOptionsDialogRefCon       'optn'    /*options dialog*/
   #define emCancelSectionDialogRefCon 'cncl'    /*cancel section*/
   #define emGotoPubErrDialogRefCon    'gerr'    /*locate publisher*/

   kFormatLengthUnknown        = -1         /*length of format unknown*/
};
   /*refCon field when displaying stacked dialog boxes*/
   #define sfMainDialogRefCon    'stdf'     {new publisher and }
                                            { new subscriber}
   #define sfNewFolderDialogRefCon'nfdr'    {new folder}
```

```
   #define sfReplaceDialogRefCon  'rplc'      {replace dialog}
   #define sfStatWarnDialogRefCon 'stat'      {warning dialog}
   #define sfErrorDialogRefCon    'err '      {error dialog}

   /*message IDs for Apple events sent by the Edition Manager*/
   #define sectionEventMsgClass   'sect'      /*Apple events sent by the */
                                              /* Edition Manager*/
   #define sectionReadMsgID       'read'      /*Section Read events*/
   #define sectionWriteMsgID      'writ'      /*Section Write events*/
   #define sectionScrollMsgID     'scrl'      /*Section Scroll events*/
   #define sectionCancelMsgID     'cncl'      /*Section Cancel events*/
```

Data Types

```
typedef unsigned long TimeStamp;              /*seconds since 1904*/
typedef Handle EditionRefNum;`                /*used in Edition I/O*/
typedef short UpdateMode;                     /*update mode: sumAutomatic, */
                                              /* sumManual, */
                                              /* pumOnSave, pumManual*/
typedef char SectionType;                     /*one byte, stSubscriber */
                                              /* or stPublisher*/
typedef unsigned long FormatType;             /*similar to ResType*/

struct SectionRecord {
   SignedByte version;                        /*always 1x01 in version 7.0*/
   SectionType kind;                          /*stPublisher or */
                                              /* stSubscriber*/
   UpdateMode mode;                           /*automatic or manual*/
   TimeStamp mdDate;                          /*last change to section*/
   long sectionID;                            /*application-specific, */
                                              /* unique per document*/
   long refCon;                               /*application-specific*/
   AliasHandle alias;                         /*handle to alias record*/
   long subPart;                              /*private*/
   struct SectionRecord **nextSection;        /*private*/
   Handle controlBlock;                       /*private*/
   EditionRefNum refNum;                      /*private*/
};

typedef struct SectionRecord SectionRecord;
typedef SectionRecord *SectionPtr, **SectionHandle;
```

```
struct EditionContainerSpec {
   FSSpec theFile;                              /*file containing */
                                                /* edition data*/
   ScriptCode theFileScript;                    /*script code of filename*/
   long thePart;                                /*which part of file, */
                                                /* always kPartsNotUsed*/
   Str31 thePartName;                           /*reserved*/
   ScriptCode thePartScript;                    /*reserved*/
};

typedef struct EditionContainerSpec EditionContainerSpec;
typedef EditionContainerSpec *EditionContainerSpecPtr;

struct EditionInfoRecord {
   TimeStamp crDate;                    /*date edition container */
                                        /* was created*/
   TimeStamp mdDate;                    /*date of last change*/
   OSType fdCreator;                    /*file creator*/
   OSType fdType;                       /*file type*/
   EditionContainerSpec container;      /*the edition*/
};

typedef struct EditionInfoRecord EditionInfoRecord;

struct NewPublisherReply {
   Boolean canceled;                    /*user canceled dialog box*/
   Boolean replacing;                   /*user chose existing */
                                        /* filename for an edition*/
   Boolean usePart;                     /*always FALSE in version */
                                        /* 7.0*/
   Handle preview;                      /*handle to 'prvw', 'PICT',*/
                                        /* 'TEXT', or 'snd ' data*/
   FormatType previewFormat;            /*type of preview*/
   EditionContainerSpec container;      /*edition chosen*/
};
typedef struct NewPublisherReply NewPublisherReply;

struct NewSubscriberReply {
   Boolean canceled;                    /*user canceled dialog box*/
   unsigned char formatsMask;           /*formats required*/
   EditionContainerSpec container;      /*edition selected*/
};
typedef struct NewSubscriberReply NewSubscriberReply;
```

```
struct SectionOptionsReply {
   Boolean canceled;                         /*user canceled dialog box*/
   Boolean changed;                          /*changed the section */
                                             /* record*/
   SectionHandle sectionH;                   /*handle to the specified */
                                             /* section record*/
   ResType action;                           /*action codes*/
};


typedef struct SectionOptionsReply SectionOptionsReply;


typedef pascal Boolean (*ExpModalFilterProcPtr) (DialogPtr theDialog,
                              EventRecord *theEvent, short itemOffset,
                              short *itemHit, Ptr yourDataPtr);


typedef pascal short (*ExpDlgHookProcPtr) (short itemOffset, short itemHit,
                                    DialogPtr theDialog, Ptr yourDataPtr);


typedef unsigned char EditionOpenerVerb;


struct EditionOpenerParamBlock {
   EditionInfoRecord info;                   /*edition container to */
                                             /* be subscribed to*/
   SectionHandle sectionH;                   /*publisher or subscriber */
                                             /* requesting open*/
   FSSpecPtr document;                       /*document passed*/
   OSType fdCreator;                         /*Finder creator type*/
   long ioRefNum;                            /*reference number*/
   FormatIOProcPtr ioProc;                   /*routine to read formats*/
   Boolean success;                          /*reading or writing was */
                                             /* successful*/
   unsigned char formatsMask;                /*formats required to */
                                             /* subscribe*/
};
typedef struct EditionOpenerParamBlock EditionOpenerParamBlock;


typedef pascal short (*EditionOpenerProcPtr) (EditionOpenerVerb selector,
                                     FormatIOParamBlock *PB);


enum {ioHasFormat, ioReadFormat, ioNewFormat, ioWriteFormat};
typedef unsigned char FormatIOVerb;
```

```
struct FormatIOParamBlock {
   long ioRefNum;                           /*reference number*/
   FormatType format;                       /*edition format type*/
   long formatIndex;                        /* opener-specific */
                                            /* enumeration */
                                            /* of formats*/
   unsigned long offset;                    /*offset into format*/
   Ptr buffPtr;                             /*data starts here*/
   unsigned long buffLen;                   /*length of data*/
};


typedef struct FormatIOParamBlock FormatIOParamBlock;


typedef pascal short (*FormatIOProcPtr) (FormatIOVerb selector,
                                  FormatIOParamBlock *PB);
```

Edition Manager Routines

Initializing the Edition Manager

```
pascal OSErr InitEditionPack(void)
```

Creating and Registering a Section

```
pascal OSErr NewSection       (const EditionContainerSpec *container,
                               const FSSpec *sectionDocument,
                               SectionType kind, long sectionID,
                               UpdateMode initialMode,
                               SectionHandle *sectionH);
pascal OSErr RegisterSection
                              (const FSSpec *sectionDocument,
                               SectionHandle sectionH,
                               Boolean *aliasWasUpdated);
pascal OSErr UnRegisterSection
                              (SectionHandle sectionH);
pascal OSErr IsRegisteredSection
                              (SectionHandle sectionH);
pascal OSErr AssociateSection
                              (SectionHandle sectionH,
                               const FSSpec *newSectionDocument);
```

Creating and Deleting an Edition Container

```
pascal OSErr CreateEditionContainerFile
                        (const FSSpec *editionFile, OSType fdCreator,
                         ScriptCode editionFileNameScript);
pascal OSErr DeleteEditionContainerFile
                        (const FSSpec *editionFile);
```

Setting and Getting a Format Mark

```
pascal OSErr SetEditionFormatMark
                        (EditionRefNum whichEdition,
                         FormatType whichFormat,
                         unsigned long setMarkTo);
pascal OSErr GetEditionFormatMark
                        (EditionRefNum whichEdition,
                         FormatType whichFormat,
                         unsigned long *currentMark);
```

Reading in Edition Data

```
pascal OSErr OpenEdition    (SectionHandle subscriberSectionH,
                             EditionRefNum *refNum);
pascal OSErr EditionHasFormat
                            (EditionRefNum whichEdition,
                             FormatType whichFormat,
                             Size *formatSize);
pascal OSErr ReadEdition    (EditionRefNum whichEdition,
                             FormatType whichFormat, void *buffPtr,
                             Size *buffLen);
```

Writing out Edition Data

```
pascal OSErr OpenNewEdition (SectionHandle publisherSectionH,
                             OSType fdCreator,
                             const FSSpec *publisherSectionDocument,
                             EditionRefNum *refNum);
pascal OSErr WriteEdition   (EditionRefNum whichEdition,
                             FormatType whichFormat, const void *buffPtr,
                             Size *buffLen);
```

Closing an Edition After Reading or Writing

```
pascal OSErr CloseEdition   (EditionRefNum whichEdition,
                             Boolean successful);
```

Displaying Dialog Boxes

```
pascal OSErr GetLastEditionContainerUsed
                            (EditionContainerSpec *container);
pascal OSErr NewSubscriberDialog
                            (NewSubscriberReply *reply);
pascal OSErr NewPublisherDialog
                            (NewPublisherReply *reply);
pascal OSErr SectionOptionsDialog
                            (SectionOptionsReply *reply);
pascal OSErr NewSubscriberExpDialog
                            (NewSubscriberReply *reply, Point where,
                             short expansionDITLresID,
                             ExpDlgHookProcPtr dlgHook,
                             ExpModalFilterProcPtr filterProc,
                             void *yourDataPtr);
pascal OSErr NewPublisherExpDialog
                            (NewPublisherReply *reply, Point where,
                             short expansionDITLresID,
                             ExpDlgHookProcPtr dlgHook,
                             ExpModalFilterProcPtr filterProc,
                             void *yourDataPtr);
pascal OSErr SectionOptionsExpDialog
                            (SectionOptionsReply *reply, Point where,
                             short expansionDITLresID,
                             ExpDlgHookProcPtr dlgHook,
                             ExpModalFilterProcPtr filterProc,
                             void *yourDataPtr);
```

Locating a Publisher and Edition From a Subscriber

```
pascal OSErr GetEditionInfo (const SectionHandle sectionH,
                             EditionInfoRecord *editionInfo);
pascal OSErr GoToPublisherSection
                            (const EditionContainerSpec *container);
```

Edition Container Formats

```
pascal OSErr GetStandardFormats
                            (const EditionContainerSpec *container,
                             FormatType *previewFormat,
                             Handle preview, Handle publisherAlias,
                             Handle formats);
```

Reading and Writing Non-Edition files

```
pascal OSErr GetEditionOpenerProc
                    (EditionOpenerProcPtr *opener);
pascal OSErr SetEditionOpenerProc
                    (EditionOpenerProcPtr opener);
pascal OSErr CallEditionOpenerProc
                    (EditionOpenerVerb selector,
                     EditionOpenerParamBlock *PB,
                     EditionOpenerProcPtr routine);
pascal OSErr CallFormatIOProc
                    (FormatIOVerb selector,
                     FormatIOParamBlock *PB,
                     FormatIOProcPtr routine);
```

Application-Defined Routines

```
pascal OSErr MyExpDlgHook    (short itemOffset, short itemHit,
                             DialogPtr theDialog,
                             Ptr yourDataPtr);
pascal OSErr MyExpModalFilter
                            (DialogPtr theDialog,
                             EventRecord *theEvent,
                             short itemOffset, short *itemHit,
                             Ptr yourDataPtr);
pascal OSErr MyOpener        (EditionOpenerVerb selector,
                             EditionOpenerParamBlock *PB);
pascal OSErr MyIO            (FormatIOVerb selector,
                             FormatIOParamBlock *PB);
```

Result Codes

noErr	0	No error
abortErr	–27	Publisher has written a new edition
dskFulErr	–34	Disk is full
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnOpnErr	–38	File not open
eofErr	–39	No additional data in the format
fnfErr	–43	Edition container not found
flLckedErr	–45	Publisher writing to an edition
fBsyErr	–47	Section doing I/O
paramErr	–50	Invalid parameter
rfNumErr	–51	Bad edition reference number
permErr	–54	Not a subscriber

wrPermErr	**−61**	**Not a publisher**
noTypeErr	**−102**	**Format not available**
memFullErr	**−108**	**Memory full**
dirNFErr	**−120**	**Directory not found**
userCanceledErr	**−128**	**User clicked Cancel in dialog box**
editionMgrInitErr	**−450**	**Manager not initialized or could not load package**
badSectionErr	**−451**	**Not a valid section type**
notRegisteredSectionErr	**−452**	**Not registered**
badSubPartErr	**−454**	**Bad edition container spec or invalid edition container**
multiplePublisherWrn	**−460**	**Already is a publisher**
containerNotFoundWrn	**−461**	**Alias was not resolved**
notThePublisherWrn	**−463**	**Not the publisher**

Introduction to Apple Events

Contents

This chapter introduces Apple events and the Apple Event Manager. Later chapters describe how your application can use the Apple Event Manager to respond to and send Apple events, locate Apple event objects, and record Apple events.

The interapplication communication (IAC) architecture for Macintosh computers consists of five parts: the Edition Manager, the Open Scripting Architecture (OSA), the Apple Event Manager, the Event Manager, and the Program-to-Program Communications (PPC) Toolbox. The chapter "Introduction to Interapplication Communication" in this book provides an overview of the relationships among these parts.

The *Apple Event Registry: Standard Suites* defines both the actions performed by the standard Apple events, or "verbs," and the standard Apple event object classes, which can be used to create "noun phrases" describing objects on which Apple events act. If your application uses the Apple Event Manager to respond to some of these standard Apple events, you can make it scriptable—that is, capable of responding to scripts written in a scripting language such as AppleScript. In addition, your application can use the Apple Event Manager to create and send Apple events and to allow user actions in your application to be recorded as Apple events.

Before you use this chapter or any of the other chapters about the Apple Event Manager, you should be familiar with the chapters "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* and "Process Manager" in *Inside Macintosh: Processes.*

This chapter begins by describing Apple events and some of the data structures they contain. The rest of the chapter introduces the use of the Apple Event Manager to

n   respond to Apple events

n   send Apple events to request services or information

n   work with object specifier records

n   classify Apple event objects

n   locate Apple event objects

Finally, this chapter summarizes the tasks you can perform with the Apple Event Manager and explains where to locate information you need to perform those tasks.

About Apple Events

An **Apple event** is a high-level event that conforms to the Apple Event Interprocess Messaging Protocol. The Apple Event Manager uses the services of the Event Manager to send Apple events between applications on the same computer, between applications on remote computers, or from an application to itself.

Applications typically use Apple events to request services and information from other applications or to provide services and information in response to such requests. Communication between two applications that support Apple events is initiated by a **client application,** which sends an Apple event to request a service or information. The application providing the service or the requested information is called a **server application.** The client and server applications can reside on the same local computer or on remote computers connected to a network. An application can also send Apple events to itself, thus acting as both client and server.

Figure 3-1 shows a common Apple event, the Open Documents event. The Finder is the client; it requests that the SurfWriter application open the documents named "Dec. Invoice" and "Nov. Invoice." The SurfWriter application responds to the Finder's request by opening windows for the specified documents.

Figure 3-1   An Open Documents event
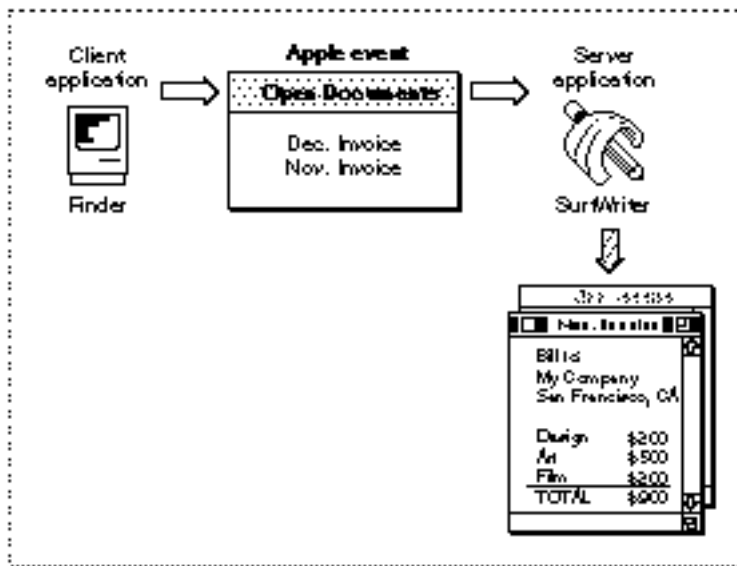
The Finder is considered the client application for the Open Documents event shown in Figure 3-1 because the Finder initiates the request for a service. The Finder can also be considered the source application for the same Open Documents event. A **source application** for an Apple event is one that sends an Apple event to another application or to itself. Similarly, the SurfWriter application can be described as either the server application or the target application for the Open Documents event shown in Figure 3-1. A **target application** for an Apple event is the one addressed to receive the Apple event. The terms *client application* and *source application* are not always synonymous, nor are the terms *server application* and *target application.* Typically, an Apple event client application sends an Apple event requesting a service to an Apple event server application;

in this case, the server application is the target application for the Apple event. A server application may return information to the client in a reply Apple event—in which case, the client application is the target application for the reply.

To perform the requested service—that is, to open the specified documents—the SurfWriter application shown in Figure 3-1 first uses the Apple Event Manager to identify the event (the Open Documents event) and to dispatch the event to SurfWriter's handler for that Apple event. An **Apple event handler** is an application-defined function that extracts pertinent data from an Apple event, performs the requested action, and (usually) returns a result. In this case, SurfWriter's Open Documents event handler examines the Apple event to determine which documents to open (Dec. Invoice and Nov. Invoice), then opens them as requested.

To identify Apple events and respond appropriately, every application can rely on a vocabulary of standard Apple events that developers and Apple have established for all applications to use. These events are defined in the *Apple Event Registry: Standard Suites.* The standard **suites,** or groups of related Apple events that are usually implemented together, include the Required suite, the Core suite, and functional-area suites such as the Text suite and the Database suite. To function as a server application, your application should be able to respond to all the Apple events in the Required suite and any of those in the Core and functional-area suites that it is likely to receive. For example, most word-processing applications should be capable of responding to the Apple events in the Text suite, and most database applications should be capable of responding to those in the Database suite.

If necessary, you can extend the definitions of the standard Apple events to match specific capabilities of your application. You can also define your own custom Apple events; however, before defining custom events, you should check with the Apple Event Registrar to find out whether you can adapt existing Apple event definitions or definitions still under development to the needs of your application.

By supporting the standard Apple events in your application, you ensure that your application can communicate effectively with other applications that also support them. Instead of supporting many different custom events for a limited number of applications, you can support a relatively small number of standard Apple events that can be used by any number of applications.

You can begin supporting Apple events by making your application a reliable server application: first for the required Apple events, then for the core and functional-area Apple events as appropriate. Once your application can respond to the appropriate standard Apple events, you can make it **scriptable,** or capable of responding to instructions written in a system-wide scripting language such as AppleScript. If necessary, your application can also send Apple events to itself or to other applications.

"About the Apple Event Manager," which begins on page 3-48, provides more information about the steps you need to take to support Apple events in your application.

The next section, "Apple Events and Apple Event Objects," describes how Apple events can describe data and other items within an application or its documents. Subsequent sections describe the basic organization of Apple events and the data structures from which they are constructed.

Apple Events and Apple Event Objects

The Open Documents event shown in Figure 3-1, like the other three required events, specifies an action and the applications or documents to which that action applies. The *Apple Event Registry: Standard Suites* provides a vocabulary of actions for use by all applications. In addition to a vocabulary of actions, effective communication between applications requires a method of referring to windows, data (such as words or graphic elements), files, folders, volumes, zones, and other discrete items on which actions can be performed. The Apple Event Manager includes routines that allow any application to construct or interpret "noun phrases" that describe the objects on which Apple events act.

Most of the Apple event definitions in the *Apple Event Registry: Standard Suites* include definitions of **Apple event object classes,** which are simply names for objects that can be acted upon by each kind of Apple event. An **Apple event object** is any distinct item supported by an application that can be described within an Apple event. Apple event objects can be anything that an application can locate on the basis of such a description, including items that a user can differentiate and manipulate while using an application, such as words, paragraphs, shapes, windows, or style formats.

The definition for each Apple event object class in the *Apple Event Registry: Standard Suites* determines only how that kind of Apple event object should be described within an Apple event, not how it should be represented internally by an individual application. You do not have to write your application in an object-oriented programming language to support Apple event objects. Instead, you need to organize your application so that it can interpret a request for specific Apple event objects, locate the objects, and perform the requested action on them.

Figure 3-2 shows a common Apple event, the Get Data event from the Core suite. In this example, the SurfCharter application is the client application; it requests data contained in a specific table in a SurfWriter document. To obtain the data it wants, the SurfCharter application must include a description of the data in the Get Data event it sends to SurfWriter. This description identifies the requested data as an Apple event object called a table. The table is named "Summary of Sales" and is located in a document named "Sales Report."

The SurfWriter application's Get Data handler extracts information about the request, locates the specified table, and returns a result. The Apple Event Manager provides a reply Apple event to which the SurfWriter application adds the requested information in the form requested by the Get Data event. The Apple Event Manager sends the reply event back to the SurfCharter application, which can use the requested data in whatever way is appropriate—in this case, displaying it as a pie chart.

Figure 3-2   A Get Data event
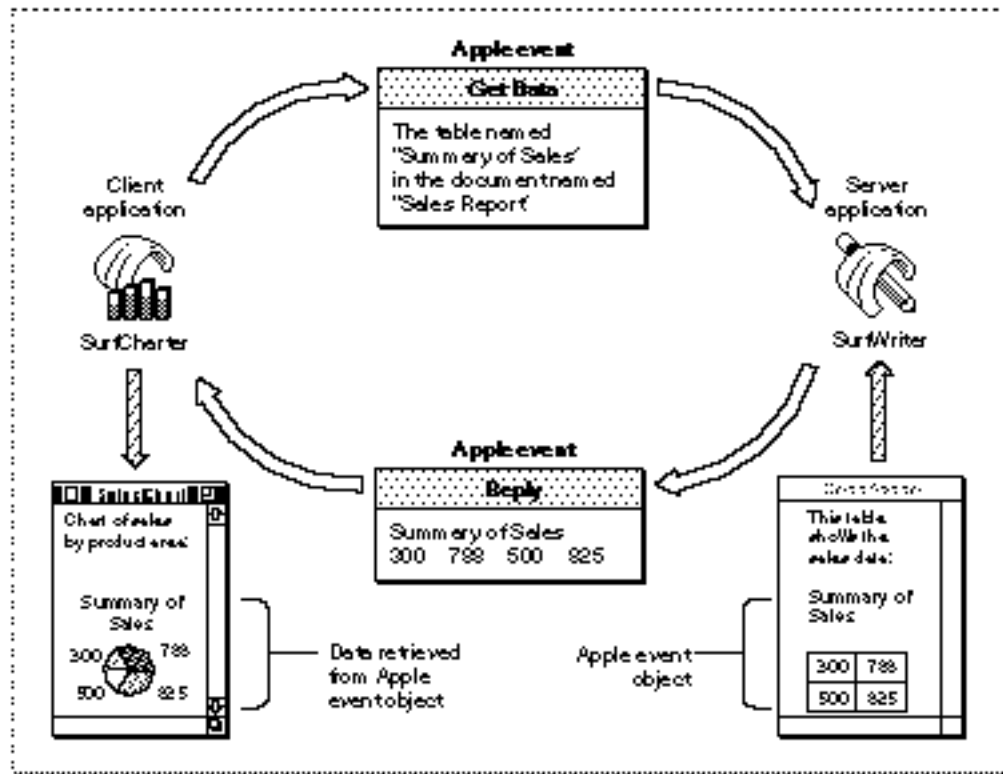
Apple Event Attributes and Parameters

When an application creates and sends an Apple event, the Apple Event Manager uses
arguments passed to Apple Event Manager routines to construct the data structures that
make up the Apple event. An Apple event consists of attributes (which identify the
Apple event and denote its task) and, often, parameters (which contain information to be
used by the target application).

An **Apple event attribute** is a record that identifies the event class, event ID, target
application, or some other characteristic of an Apple event. Taken together, the attributes
of an Apple event denote the task to be performed on any data specified in the Apple
event's parameters. A client application can use Apple Event Manager routines to add
attributes to an Apple event. After receiving an Apple event, a server application can use
Apple Event Manager routines to extract and examine its attributes.

An **Apple event parameter** is a record containing data that the target application uses. Unlike Apple event attributes (which contain information that can be used by both the Apple Event Manager and the target application), Apple event parameters contain data used only by the target application. For example, the Apple Event Manager uses the event class and event ID attributes to identify the server application's handler for a specific Apple event, and the server application must have a handler to process the event identified by those attributes. By comparison, the list of documents contained in a parameter to an Open Documents event is used only by the server application. As with attributes, a client application can use Apple Event Manager routines to add parameters to an Apple event, and a server application can use Apple Event Manager routines to extract and examine the parameters of an Apple event it has received.

Note that Apple event parameters are different from the parameters of Apple Event Manager functions. Apple event parameters are records used by the Apple Event Manager; function parameters are arguments you pass to the function or that the function returns to you. You can specify both Apple event parameters and Apple event attributes in parameters to Apple Event Manager functions. For example, the `AEGetParamPtr` function uses a buffer to return the data contained in an Apple event parameter. You can specify the Apple event parameter whose data you want in one of the parameters of the `AEGetParamPtr` function.

Apple Event Attributes

Apple events are identified by their event class and event ID attributes. The **event class** is the attribute that identifies a group of related Apple events. The event class appears in the `message` field of the event record for an Apple event. For example, the four required Apple events have the value `'aevt'` in the `message` fields of their event records. The value `'aevt'` can also be represented by the `kCoreEventClass` constant. Several event classes are shown here:

Event class	Value	Description
kCoreEventClass	'aevt'	A required Apple event
kAECoreSuite	'core'	A core Apple event
kAEFinderEvents	'FNDR'	An event that the Finder accepts
kSectionEventMsgClass	'sect'	An event sent by the Edition Manager

The **event ID** is the attribute that identifies the particular Apple event within its event class. In conjunction with the event class, the event ID uniquely identifies the Apple event and communicates what action the Apple event should perform. (The event IDs appear in the `where` field of the event record for an Apple event. For more information about event records, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.) For example, the event ID of an Open Documents event has the value `'odoc'` (which can also be represented by the `kAEOpenDocuments` constant). The `kCoreEventClass` constant in combination with the `kAEOpenDocuments` constant identifies the Open Documents event to the Apple Event Manager.

Here are the event IDs for the four required Apple events:

Event ID	Value	Description
kAEOpenApplication	'oapp'	Perform tasks required when a user opens your application without opening or printing any documents
kAEOpenDocuments	'odoc'	Open documents
kAEPrintDocuments	'pdoc'	Print documents
kAEQuitApplication	'quit'	Quit your application

In addition to the event class and event ID attributes, every Apple event must include an attribute that specifies the target application's address. Remember that the target application is the one addressed to receive the Apple event. Your application can send an Apple event to itself or to another application (on the same computer or on a remote computer connected to the network).

Every Apple event must include event class, event ID, and target address attributes. Some Apple events can include other attributes; see "Keyword-Specified Descriptor Records," which begins on page 3-15, for a complete list.

Apple Event Parameters

As with attributes, there are various kinds of Apple event parameters. A **direct parameter** usually specifies the data to be acted upon by the target application. For example, the direct parameter of the Print Documents event contains a list of documents. Some Apple events also take **additional parameters,** which the target application uses in addition to the data specified in the direct parameter. Thus, an Apple event for arithmetic operations might include additional parameters that specify operands in an equation.

The *Apple Event Registry: Standard Suites* describes all parameters as either required or optional. A **required parameter** is one that must be present for the target application to carry out the task denoted by the Apple event. An **optional parameter** is a supplemental Apple Event parameter that also can be used to specify data to the target application. Optional parameters need not be included in an Apple event; default values for optional parameters are part of the event definition. The target application that handles the event must supply default values if the optional parameters are omitted.

Direct parameters are usually defined as required parameters in the *Apple Event Registry: Standard Suites;* additional parameters may be defined as either required or optional. However, the Apple Event Manager does not enforce the definitions of required and optional events. Instead, the source application specifies, when it sends the event, which Apple event parameters the target can treat as if they were optional.

For more information about optional parameters, see "Specifying Optional Parameters for an Apple Event," which begins on page 5-7.

Interpreting Apple Event Attributes and Parameters

Figure 3-3 shows the major Apple event attributes and direct parameter for the
Open Documents event introduced in Figure 3-1.

Figure 3-3   Major attributes and direct parameter of an Open Documents event
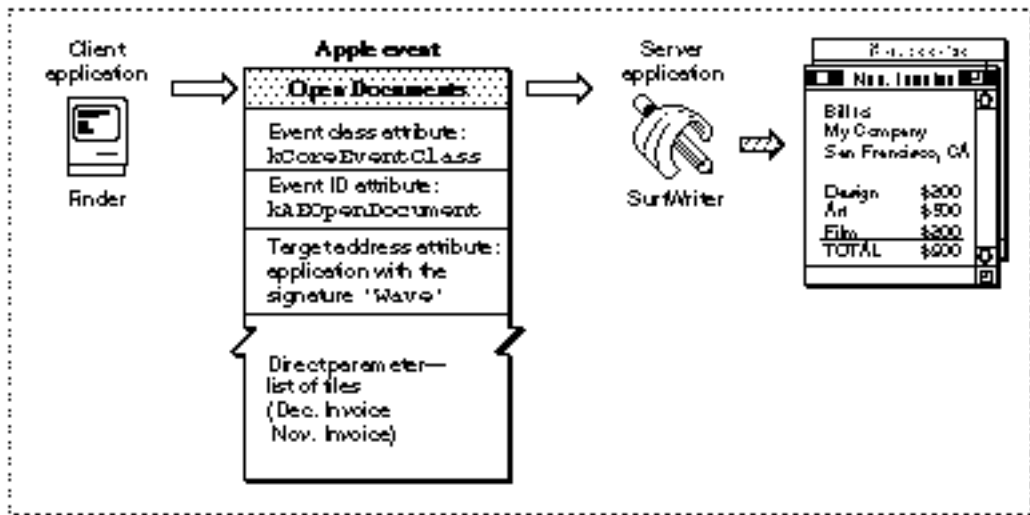
When the SurfWriter application receives any high-level event, it calls the
`AEProcessAppleEvent` function to process the event. For an Apple event such as the
Open Documents event shown in Figure 3-3, the `AEProcessAppleEvent` function uses
the event class and event ID attributes to dispatch the event to the SurfWriter
application's Open Documents handler. In response, the Open Documents handler opens
the documents specified in the direct parameter.

The definition of a given Apple event in the *Apple Event Registry: Standard Suites* suggests
how the source application can organize the data in the Apple event's parameters
and how the target application interprets that data. The data in an Apple event
parameter may use standard or private data types and may include a description of an
Apple event object. Each Apple event handler provided by an application should be
written with the format of the expected data in mind.

Apple events can use standard data types, such as strings of text, long integers,
Boolean values, and alias records, for the corresponding data in Apple event parameters.
For example, the Get Data event can contain an optional parameter specifying the form
in which the requested data should be returned. This optional parameter always consists
of a list of four-character codes denoting desired descriptor types in order of preference.
Apple events can also use special data types defined by the Apple Event Manager.

Apple event parameters often contain descriptions of Apple event objects. These
descriptions make use of a standard classification scheme summarized in "The
Classification of Apple Event Objects," which begins on page 3-39.

For example, every Get Data event includes a required parameter that describes the Apple event object containing the data requested by the client application. Thus, one application can send a Get Data event to another application, requesting, for instance, one paragraph of a document, the first and last paragraphs of a document, all pictures in the document, all paragraphs containing the word "sales," or pages 10 through 12 of the document.

Figure 3-4 shows the Apple event attributes and direct parameter for the Get Data event introduced in Figure 3-2. The direct parameter for the Get Data event sent by the SurfCharter application describes the requested Apple event object as a table called "Summary of Sales" in the document "Sales Report." Both the table and the document are Apple event objects. The description of an Apple event object always includes a description of its location. In most cases, Apple event objects are located inside other Apple event objects.

Figure 3-4   Major attributes and direct parameter of a Get Data event
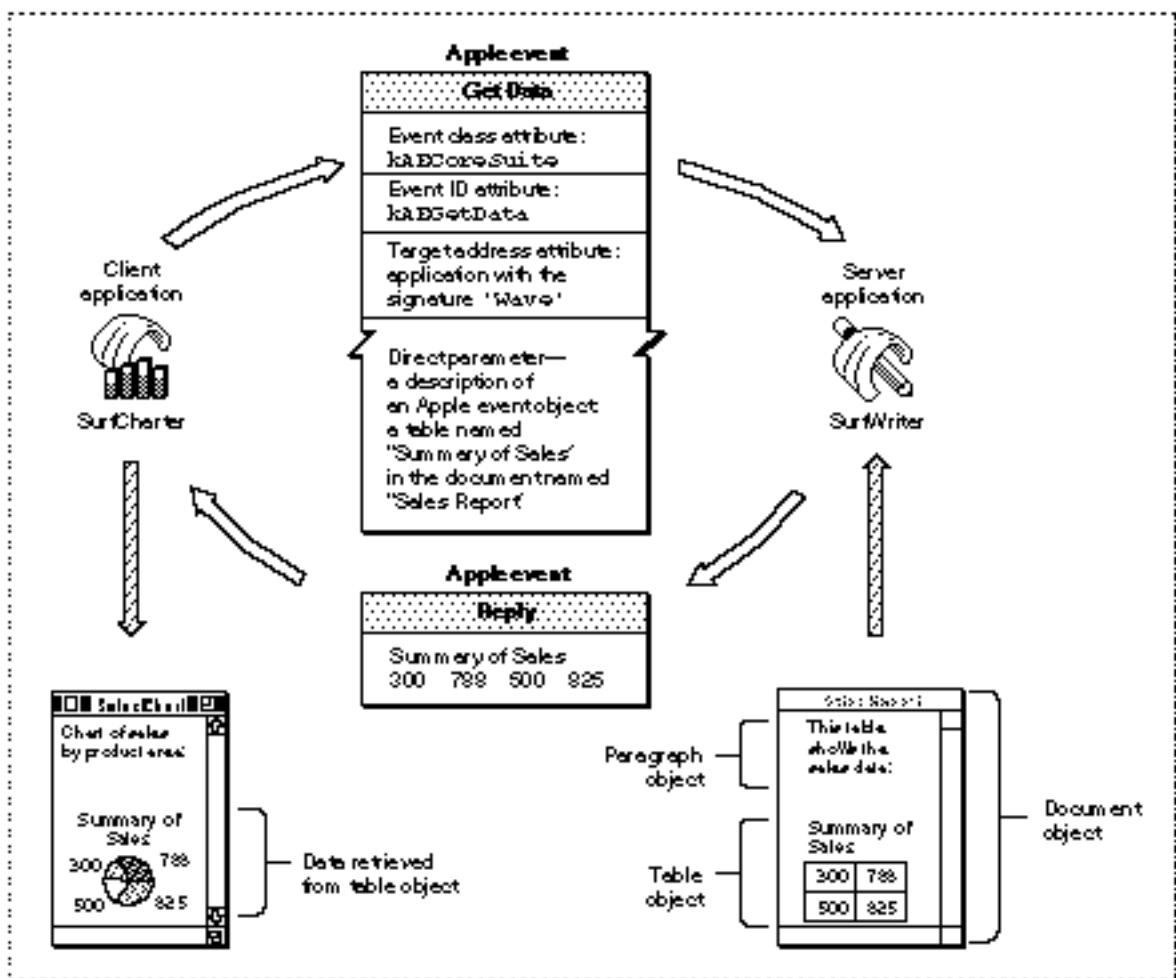
To process the information in the Get Data event, the SurfWriter application calls the
`AEProcessAppleEvent` function. The `AEProcessAppleEvent` function uses the
event class and event ID attributes to dispatch the event to the SurfWriter application's
handler for the Get Data Apple event. The SurfWriter application responds to the Get
Data event by **resolving** the description of the Apple event object—that is, by using the
`AEResolve` function, other Apple Event Manager routines, and its own
application-defined functions to locate the table named "Summary of Sales." After
locating the table, SurfWriter adds a copy of the table's data to the reply event, which the
Apple Event Manager then sends to the SurfCharter application. The SurfCharter
application then displays the data in its active window.

The next section, "Data Structures Within Apple Events," describes the data structures
the Apple Event Manager uses for Apple event attributes and parameters.

Data Structures Within Apple Events

The Apple Event Manager constructs its own internal data structures to contain the
information in an Apple event. Neither the sender nor the receiver of an Apple event
should manipulate data directly after it has been added to an Apple event; each should
rely on Apple Event Manager functions to do so.

This section describes the most important data structures used by the Apple event
Manager to construct Apple events. The first structure described is the descriptor record,
a data structure of type `AEDesc`. Applications may access the data in an individual
descriptor record directly if it is not part of another Apple Event Manager data structure.

In some cases it is convenient for the Apple Event Manager to describe descriptor
records by data types that indicate their contents; thus, it also defines data structures
such as type `AEAddressDesc`, `AEDescList`, and `AERecord`, which are descriptor
records used to hold addresses, lists of other descriptor records, and Apple event
parameters, respectively. These and most of the other data structures described in this
section are formally defined as data structures of type `AEDesc`; they differ only in the
purposes for which they are used.

Descriptor Records

Descriptor records are the building blocks used by the Apple Event Manager to construct
Apple event attributes and parameters. A **descriptor record** is a data structure of type
`AEDesc`; it consists of a handle to data and a descriptor type that identifies the type of
the data to which the handle refers.

```
TYPE AEDesc =
RECORD                                  {descriptor record}
   descriptorType:  DescType;  {type of data}
   dataHandle:      Handle;    {handle to data}
END;
```

If a descriptor record exists separately from other Apple Event Manager data structures, it is possible to retrieve the data associated with its handle by dereferencing the handle twice. After a descriptor record has been added to any other Apple Event Manager data structure, you must use Apple Event Manager routines to extract data from the descriptor record.

The **descriptor type** is a structure of type `DescType`, which in turn is of data type `ResType`—that is, a four-character code. Constants are usually used in place of these four-character codes when referring to descriptor types. Descriptor types represent various data types. Here are some of the major descriptor type constants, their values, and the kinds of data they identify.

Descriptor type	Value	Description of data
`typeBoolean`	`'bool'`	1-byte Boolean value
`typeChar`	`'TEXT'`	Unterminated string
`typeLongInteger`	`'long'`	32-bit integer
`typeShortInteger`	`'shor'`	16-bit integer
`typeMagnitude`	`'magn'`	Unsigned 32-bit integer
`typeAEList`	`'list'`	List of descriptor records
`typeAERecord`	`'reco'`	List of keyword-specified descriptor records
`typeAppleEvent`	`'aevt'`	Apple event record
`typeEnumerated`	`'enum'`	Enumerated data
`typeType`	`'type'`	Four-character code
`typeFSS`	`'fss '`	File system specification
`typeKeyword`	`'keyw'`	Apple event keyword
`typeNull`	`'null'`	Nonexistent data (handle whose value is `NIL`)

For a complete list of the basic descriptor types used by the Apple Event Manager, see Table 4-2 on page 4-57.

Figure 3-5 illustrates the logical arrangement of a descriptor record with a descriptor type of `typeChar`, which specifies that the data handle refers to an unterminated string (in this case, the text "Summary of Sales").

Figure 3-5   A descriptor record whose data handle refers to an unterminated string
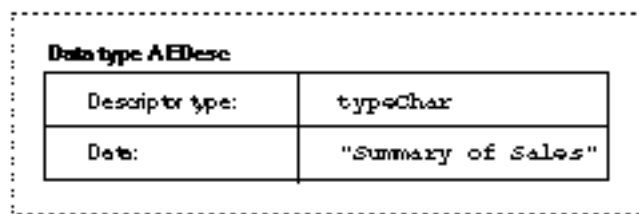
Figure 3-6 illustrates the logical arrangement of a descriptor record with a descriptor type of `typeType`, which specifies that the data handle refers to a four-character code (in this case the constant `kCoreEventClass`, whose value is `'aevt'`). This descriptor record can be used in an Apple event attribute that identifies the event class for any Apple event in the Core suite.

Figure 3-6   A descriptor record whose data handle refers to event class data
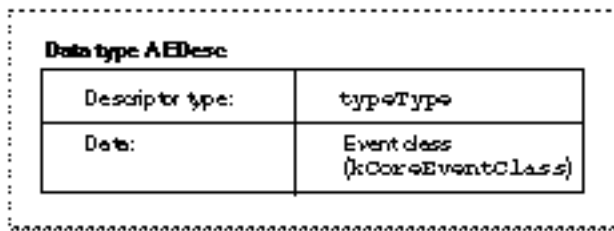
Every Apple event includes an attribute specifying the address of the target application. A descriptor record that contains an application's address is called an **address descriptor record.**

```
TYPE  AEAddressDesc = AEDesc;          {address descriptor record}
```

The address in an address descriptor record can be specified as one of these four basic types (or as any other descriptor type you define that can be coerced to one of these types):

Descriptor type	Value	Description
typeApplSignature	'sign'	Application signature
typeSessionID	'ssid'	Session reference number
typeTargetID	'targ'	Target ID record
typeProcessSerialNumber	'psn '	Process serial number

Like several of the other data structures defined by the Apple Event Manager for use in Apple event attributes and Apple event parameters, an address descriptor record is identical to a descriptor record of data type `AEDesc`; the only difference is that the data for an address descriptor record must always consist of an application's address.

Keyword-Specified Descriptor Records

After the Apple Event Manager has assembled the necessary descriptor records as the attributes and parameters of an Apple event, your application cannot examine the contents of the Apple event directly. Instead, your application must use Apple Event Manager routines to request each attribute and parameter by keyword. **Keywords** are arbitrary names used by the Apple Event Manager to keep track of various descriptor records. The AEKeyword data type is defined as a four-character code.

```
TYPE  AEKeyword = PACKED ARRAY[1..4] OF Char; {keyword for a }
                                             { descriptor record}
```

Constants are typically used for keywords. Here is a list of the keyword constants for Apple event attributes:

Attribute keyword	Value	Description
keyAddressAttr	'addr'	Address of target or client application
keyEventClassAttr	'evcl'	Event class of Apple event
keyEventIDAttr	'evid'	Event ID of Apple event
keyEventSourceAttr	'esrc'	Nature of the source application
keyInteractLevelAttr	'inte'	Settings for allowing the Apple Event Manager to bring a server application to the foreground, if necessary, to interact with the user
keyMissedKeywordAttr	'miss'	Keyword for first required parameter remaining in an Apple event
keyOptionalKeywordAttr	'optk'	List of keywords for parameters of the Apple event that should be treated as optional by the target application
keyOriginalAddressAttr	'from'	Address of original source of Apple event if the event has been forwarded (available only in version 1.01 or later versions of the Apple Event Manager)
keyReturnIDAttr	'rtid'	Return ID for reply Apple event
keyTimeoutAttr	'timo'	Length of time, in ticks, that the client will wait for a reply or a result from the server
keyTransactionIDAttr	'tran'	Transaction ID identifying a series of Apple events

Here is a list of the keyword constants for commonly used Apple event parameters:

Parameter keyword	Value	Description
keyDirectObject	'----'	Direct parameter
keyErrorNumber	'errn'	Error number parameter
keyErrorString	'errs'	Error string parameter

The *Apple Event Registry: Standard Suites* defines additional keyword constants for Apple event parameters that can be used with specific Apple events.

The Apple Event Manager associates keywords with specific descriptor records by means of a **keyword-specified descriptor record,** a data structure of type `AEKeyDesc` that consists of a keyword and a descriptor record.

```
TYPE  AEKeyDesc = {keyword-specified descriptor record}
RECORD
    descKey:      AEKeyword;     {keyword}
    descContent:  AEDesc;        {descriptor record}
END;
```

Figure 3-7 illustrates a keyword-specified descriptor record with the keyword `keyEventClassAttr`—the keyword that identifies an event class attribute. The figure shows the logical arrangement of the event class attribute for the Open Documents event shown in Figure 3-3 on page 3-10. The descriptor record in Figure 3-7 is identical to the one in Figure 3-6; its descriptor type is `typeType`, and the data to which its handle refers identifies the event class as `kCoreEventClass`.

Figure 3-7   A keyword-specified descriptor record for the event class attribute of an Open Documents event
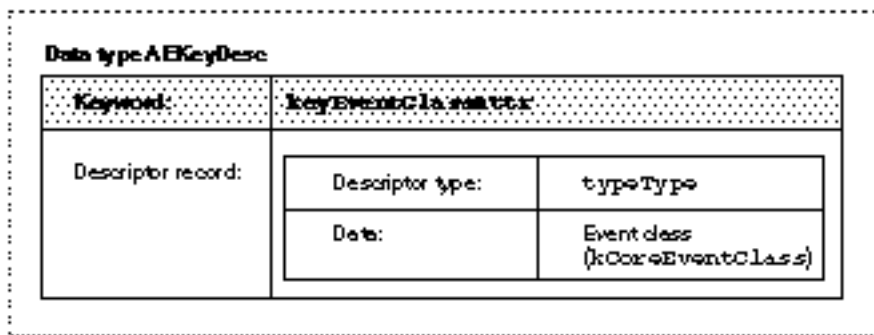
Descriptor Lists

When extracting data from an Apple event, you use Apple Event Manager functions to copy data to a buffer specified by a pointer, or to return a descriptor record whose data handle refers to a copy of the data, or to return lists of descriptor records (called descriptor lists).

As previously noted, the descriptor record (of data type `AEDesc`) is the fundamental structure in Apple events, and it consists of a descriptor type and a handle to data. A **descriptor list** is a data structure of type `AEDescList` defined by the data type `AEDesc`—that is, a descriptor list is a descriptor record whose data handle refers to a list of other descriptor records (unless it is an empty list).

```
TYPE  AEDescList = AEDesc;          {list of descriptor records}
```

Like several other Apple Event Manager data structures, a descriptor list is identical to a descriptor record of data type `AEDesc`; the only difference is that the data in a descriptor list must always consist of a list of other descriptor records.

Figure 3-8 illustrates the logical arrangement of the descriptor list that specifies the direct parameter of the Open Documents event shown in Figure 3-3 on page 3-10. This descriptor list consists of a list of descriptor records that contain alias records to filenames. (See the chapter "Alias Manager" in *Inside Macintosh: Files* for a detailed description of alias records.)

Figure 3-8   A descriptor list for a list of aliases
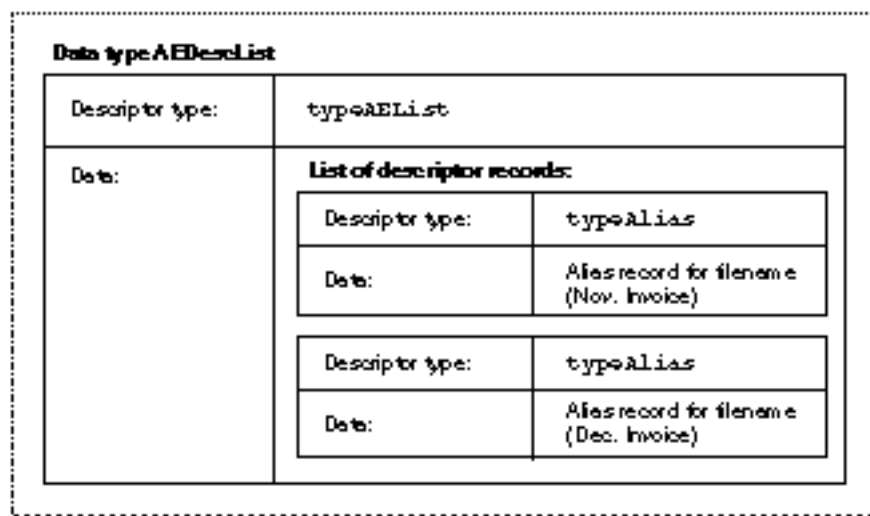
The descriptor list in Figure 3-8 provides the data for a keyword-specified descriptor record. Keyword-specified descriptor records for Apple event parameters can in turn be combined in an **AE record**, which is a descriptor list of data type `AERecord`.

```
TYPE  AERecord = AEDescList;          {list of keyword-specified }
                                      { descriptor records}
```

The handle for a descriptor list of data type `AERecord` refers to a list of keyword-specified descriptor records that can be used to construct Apple event parameters. The Apple Event Manager provides routines that allow your application to create AE records and extract data from them when creating or responding to Apple events.

An AE record has the descriptor type `typeAERecord` and can be coerced to several other descriptor types. An **Apple event record**, which is different from an AE record, is another special descriptor list of data type `AppleEvent` and descriptor type `typeAppleEvent`.

```
TYPE  AppleEvent = AERecord;            {list of attributes and }
                                        { parameters necessary for }
                                        { an Apple event}
```

An Apple event record describes a full-fledged Apple event. Like the data for an AE record, the data for an Apple event record consists of a list of keyword-specified descriptor records. Unlike an AE record, the data for an Apple event record is divided into two parts, one for attributes and one for parameters. This division within the Apple event record allows the Apple Event Manager to distinguish between an Apple event's attributes and its parameters.

Descriptor lists, AE records, and Apple event records are all descriptor records whose handles refer to a nested list of other descriptor records. The data associated with each data type may be organized differently and is used by the Apple Event Manager for different purposes. In each case, however, the data is identified by a handle in a descriptor record. This means that you can pass an Apple event record to any Apple Event Manager function that expects an AE record. Similarly, you can pass Apple event records and AE records, as well as descriptor lists and descriptor records, to any Apple Event Manager functions that expect records of data type `AEDesc`.

When you use the `AECreateAppleEvent` function, the Apple Event Manager creates an Apple event record containing the attributes for an Apple event's event class, event ID, target address, return ID, and transaction ID. You then use Apple Event Manager functions such as `AEPutParamDesc` and `AEPutAttributeDesc` to add or modify attributes and to add any necessary parameters to the Apple event.

Figure 3-9 shows an example of a complete Apple event—a data structure of type `AppleEvent` containing a list of keyword-specified descriptor records that name the attributes and parameters of an Open Documents event. The figure includes the event class attribute shown in Figure 3-7 and the descriptor list shown in Figure 3-8, which forms the direct parameter—the keyword-specified descriptor record with the keyword `keyDirectObject`. The entire figure corresponds to the Open Documents event shown in Figure 3-3 on page 3-10.

The next two sections, "Responding to Apple Events" and "Creating and Sending Apple Events," provide a quick overview of the steps your application must take to respond to and send Apple events.

Figure 3-9   Data structures within an Open Documents event
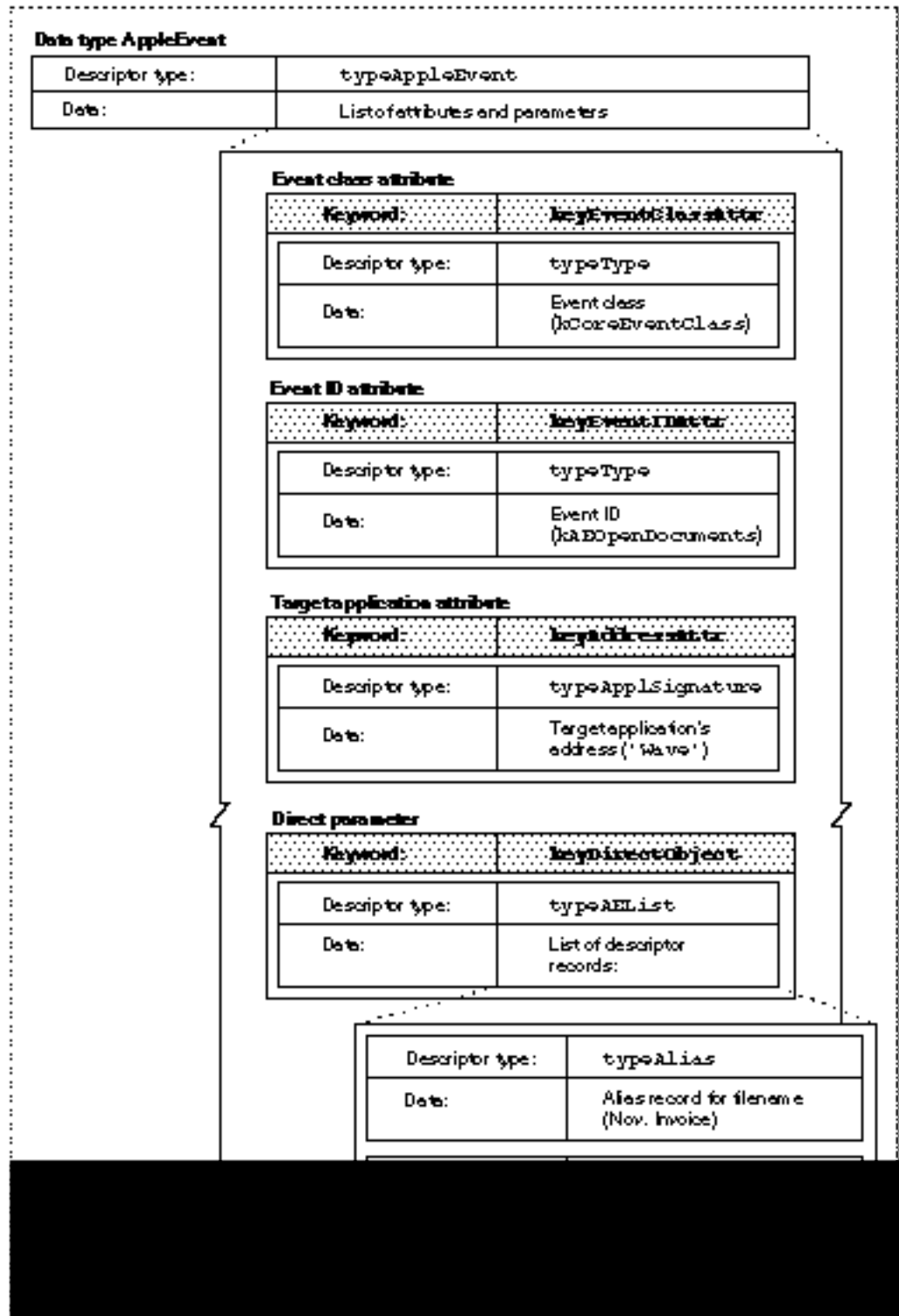
Responding to Apple Events

A client application typically uses the Apple Event Manager to create and send an Apple event requesting a service or information. A server application responds by using the Apple Event Manager to process the Apple event, extract data from the attributes and parameters of the Apple event, and if necessary add requested data to the reply event returned by the Apple Event Manager to the client application. The server usually provides its own Apple event handler for performing the action requested by the client's Apple event.

As its first step in supporting Apple events, your application should support the required Apple events sent by the Finder. If you plan to implement publish and subscribe capabilities, your application must also respond to the Apple events sent by the Edition Manager. Your application should also be able to respond to the standard Apple events that other applications are likely to send to it or that it can send to itself. This section provides a quick overview of the tasks your application must perform in responding to Apple events.

To respond to Apple events, your application must

n   set the appropriate flags in its `'SIZE'` resource

n   test for high-level events in its event loop

n   provide Apple event handlers for the Apple events it supports

n   use the `AEInstallEventHandler` function to install its Apple event handlers

n   use the `AEProcessAppleEvent` function to process Apple events

Before your application can respond to Apple events sent from remote computers, the user of your application must allow network users to link to your application. To do this, the user selects your application in the Finder, chooses Sharing from the File menu, and then clicks the Allow Remote Program Linking checkbox. If the user has not yet started program linking, the Sharing command offers to display the Sharing Setup control panel so that the user can start program linking. The user must also authorize remote users for program linking by using the Users & Groups control panel. Program linking and setting up authenticated sessions are described in the chapter "Program-to-Program Communications Toolbox" in this book.

Accepting and Processing Apple Events

To accept or send Apple events (or any other high-level events), you must set the appropriate flags in your application's `'SIZE'` resource and include code to handle high-level events in your application's main event loop.

Two flags in the `'SIZE'` resource determine whether an application receives high-level events:

n The `isHighLevelEventAware` flag must be set for your application to receive any high-level events.

n The `localAndRemoteHLEvents` flag must be set for your application to receive high-level events sent from another computer on the network.

An Apple event (like all high-level events) is identified by a message class of `kHighLevelEvent` in the `what` field of the event record. You test the `what` field of the event record to determine whether it contains the value represented by the `kHighLevelEvent` constant; if your application defines any high-level events other than Apple events, you should also test the `message` field of the event record to determine whether the high-level event is something other than an Apple event. If the high-level event is not one that you've defined for your application, assume that it is an Apple event. (You are encouraged to use Apple events instead of defining your own high-level events whenever possible.)

After determining that an event is an Apple event, use the `AEProcessAppleEvent` function to let the Apple Event Manager identify the event. Figure 3-10 shows how the SurfWriter application accepts and begins to process an Apple event sent by the Finder.

Figure 3-10   Accepting and processing an Open Documents event
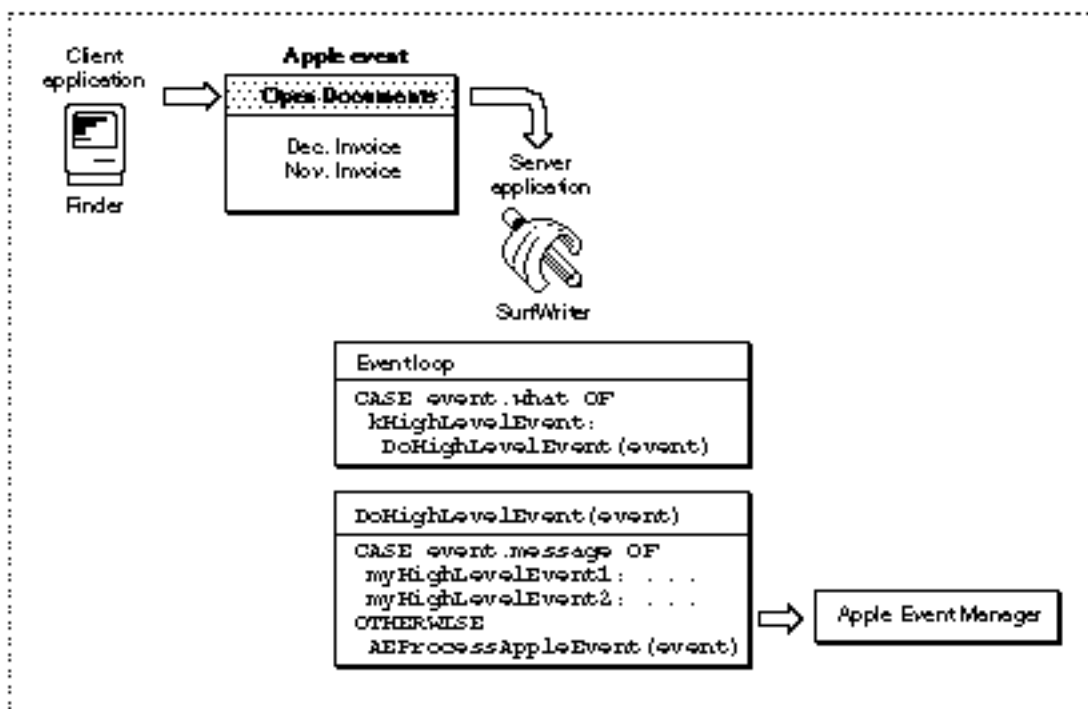
The AEProcessAppleEvent function first identifies the Apple event by examining the data in the event class and event ID attributes. The function then uses that data to call the Apple event handler that your application provides for that event. The Apple event handler extracts the pertinent data from the Apple event, performs the requested action, and returns a result. For example, if the event has an event class of kCoreEventClass and an event ID of kAEOpenDocuments, the AEProcessAppleEvent function calls your application's handler for the Open Documents event.

Before your application attempts to accept or process any Apple events, it must use the AEInstallEventHandler function to install Apple event handlers. This function installs handlers in an **Apple event dispatch table** for your application; the Apple Event Manager uses this table to map Apple events to handlers in your application. When your application calls the AEProcessAppleEvent function to process an Apple event, the Apple Event Manager checks the Apple event dispatch table and, if your application has installed a handler for that Apple event, calls that handler. Figure 3-11 shows how the flow of control passes from your application to the Apple Event Manager and back to your application.

Figure 3-11　　The Apple Event Manager calling the handler for an Open Documents event
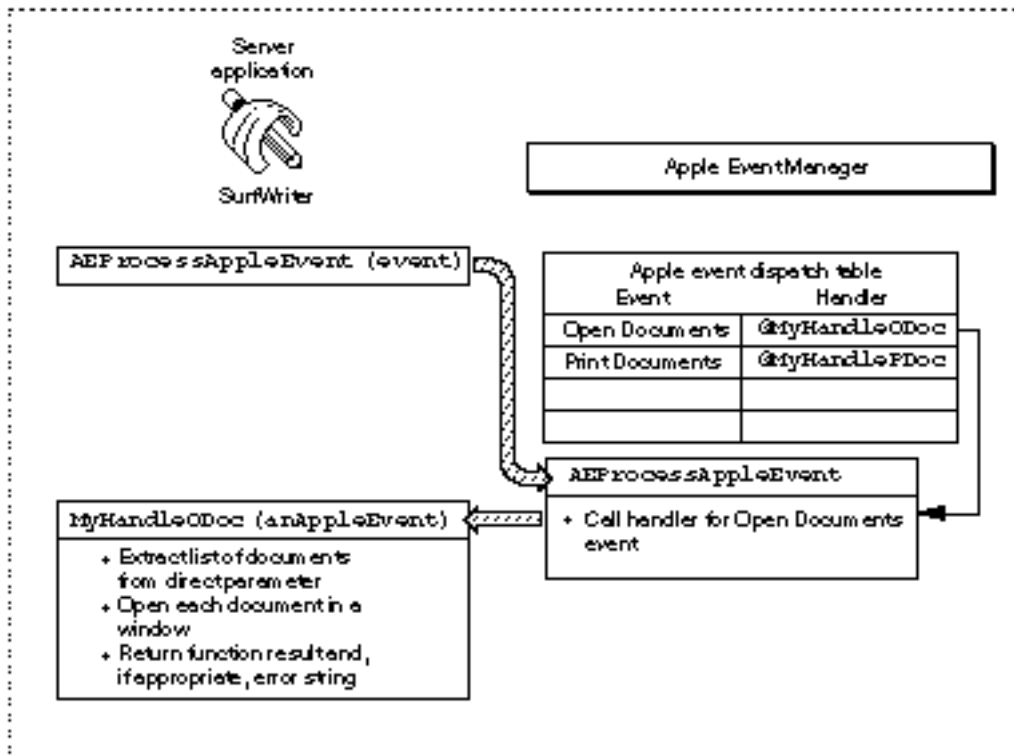
About Apple Event Handlers

Your Apple event handlers must generally perform the following tasks:

n   extract the parameters and attributes from the Apple event

n   check that all the required parameters have been extracted

n   locate any Apple event objects specified by object specifier records in the Apple event
    parameters

n   if your application needs to interact with the user, use the `AEInteractWithUser`
    function to bring it to the foreground

n   perform the action requested by the Apple event

n   dispose of any copies of descriptor records that have been created

n   add information to the reply Apple event if requested

This section describes how your application's Apple event handlers can use
the Apple Event Manager to accomplish some of these tasks. The chapter
"Responding to Apple Events" in this book provides detailed information
about handling Apple events and interacting with the user.

Extracting and Checking Data

You must use Apple Event Manager functions to extract the data from Apple events.
You can also use Apple Event Manager functions to extract data from descriptor records,
descriptor lists, and AE records. Most of these routines are available in two forms: they
either return a copy of the data in a buffer or return a copy of the descriptor record for
the data, including a copy of the data.

The following list shows the main functions you can use to access the data of an
Apple event:

Function	Description
`AEGetParamPtr`	Uses a buffer to return a copy of the data contained in an Apple event parameter. Usually used to extract data of fixed length or known maximum length; for example, to extract the result code from the `keyErrorNumber` parameter of a reply Apple event.
`AEGetParamDesc`	Returns a copy of the descriptor record or descriptor list for an Apple event parameter. Usually used to extract data of variable length; for example, to extract the descriptor list for a list of alias records specified in the direct parameter of the Open Documents event.
`AEGetAttributePtr`	Uses a buffer to return a copy of the data contained in an Apple event attribute. Used to extract data of fixed length or known maximum length; for example, to determine the source of an Apple event by extracting the data from the `keyEventSourceAttr` attribute.

continued

Function	Description (continued)
AEGetAttributeDesc	Returns a copy of the descriptor record for an attribute. Used to extract data of variable length; for example, to make a copy of a descriptor record containing the address of an application.
AECountItems	Returns the number of descriptor records in a descriptor list. Used, for example, to determine the number of alias records for documents specified in the direct parameter of the Open Documents event.
AEGetNthPtr	Uses a buffer to return a copy of the data for a descriptor record contained in a descriptor list. Used to extract data of fixed length or known maximum length; for example, to extract the name and location of a document from the descriptor list specified in the direct parameter of the Open Documents event.
AEGetNthDesc	Returns a copy of a descriptor record from a descriptor list. Used to extract data of variable length; for example, to get the descriptor record containing an alias record from the list specified in the direct parameter of the Open Documents event.

You can specify the descriptor type of the resulting data for these functions; if this type is different from the descriptor type of the attribute or parameter, the Apple Event Manager attempts to coerce it to the specified type. In the direct parameter of the Open Documents event, for example, each descriptor record in the descriptor list is an alias record; each alias record specifies a document to be opened. As explained in the chapter "Introduction to File Management" of *Inside Macintosh: Files,* all your application usually needs is the file system specification (FSSpec) record of the document. When you extract the descriptor record from the descriptor list, you can request that the Apple Event Manager return the data to your application as a file system specification record instead of an alias record.

After extracting all known Apple event parameters, your handler should check that it retrieved all the parameters that the source application considered to be required. To do so, determine whether the keyMissedKeywordAttr attribute exists. If so, your handler has not retrieved all the required parameters, and it should return an error.

Although the *Apple Event Registry: Standard Suites* defines Apple event parameters as either required or optional, the Apple Event Manager does not enforce the definitions of required and optional events. Instead, the source application specifies, when it sends the event, which Apple event parameters the target can treat as if they were optional. For more information about optional parameters, see "Specifying Optional Parameters for an Apple Event," which begins on page 5-7.

If any of the Apple event parameters include object specifier records, your handler should use the AEResolve function, other Apple Event Manager routines, and your own application-defined functions to locate the corresponding Apple event objects. For more information about locating Apple event objects, see "Working With Object Specifier Records," which begins on page 3-32.

Interacting With the User

In some cases, the server may need to interact with the user when it handles an Apple event. For example, your handler for the Print Documents event may need to display a print options dialog box and get settings from the user before printing. By specifying flags to the `AESetInteractionAllowed` function, you can set preferences to allow user interaction with your application (a) only when your application is sending the Apple event to itself, (b) only when the client application is on the same computer as your application, or (c) for any event sent by any client application on any computer. In addition, your handler should always use the `AEInteractWithUser` function before displaying a dialog box or alert box or otherwise interacting with the user. The `AEInteractWithUser` function determines whether user interaction can occur and takes appropriate action depending on the circumstances.

Both the client and server specify their preferences for user interaction: the client specifies whether the server should be allowed to interact with the user, and the server specifies when it allows user interaction while processing an Apple event. The Apple Event Manager does not allow a server application to interact with the user in response to a client application's Apple event unless at least two conditions are met: First, the client application must set flags in the `sendMode` parameter of the `AESend` function indicating that user interaction is allowed. Second, the server application must either set no user interaction preferences, in which case `AEInteractWithUser` assumes that only interaction with a client on the local computer is allowed; or it must set flags to the `AESetInteractionAllowed` function indicating that user interaction is allowed.

If these two conditions are met and if `AEInteractWithUser` determines that both the client and server applications allow user interaction under the current circumstances, `AEInteractWithUser` brings your application to the foreground if it isn't already in the foreground. Your application can then display its dialog box or alert box or otherwise interact with the user. The `AEInteractWithUser` function brings your server application to the front either directly or after the user responds to a notification request.

For detailed information about how to specify flags to the `AESetInteractionAllowed` function and how the Apple Event Manager determines whether user interaction is allowed, see the section "Interacting With the User," which begins on page 4-45.

Performing the Requested Action and Returning a Result

When your application responds to an Apple event, it should perform the standard action requested by that event. For example, your application should respond to the Open Documents event by opening the specified documents in titled windows just as if the user had selected each document from the Finder and then chosen Open from the File menu.

Many Apple events can ask your application to return data. For instance, if your application is a spelling checker, the client application might expect your application to return data in the form of a list of misspelled words. Figure 3-14 on page 3-38 shows a similar example: a Get Data event that asks the server application to locate a specific Apple event object and return the data associated with it.

If the client application requests a reply, the Apple Event Manager prepares a reply Apple event by passing a default reply Apple event to your handler. If the client application does not request a reply, the Apple Event Manager passes a **null descriptor record**—that is, a descriptor record of type `typeNull` whose data handle has the value `NIL`—to your handler instead of a default reply Apple event. The default reply Apple event has no parameters when it is passed to your handler, but your handler can add parameters to it. If your application is a spelling checker, for example, you can return a list of misspelled words in a parameter. However, your handler should check whether the reply Apple event exists before attempting to add any attributes or parameters to it. Any attempt to add an Apple event attribute or parameter to a null descriptor record generates an error.

When you extract a descriptor record using the `AEGetParamDesc`, `AEGetAttributeDesc`, `AEGetNthDesc`, or `AEGetKeyDesc` function, the Apple Event Manager creates a copy of the descriptor record for you to use. When your handler is finished using a copy of a descriptor record, you should dispose of it—and thereby deallocate the memory used by its data—by calling the `AEDisposeDesc` function.

Note

Outputs from functions such as `AEGetKeyPtr` and other routines whose names end in `-Ptr` use a buffer rather than a descriptor record to return data. Because these functions don't require the use of `AEDisposeDesc`, it is preferable to use them for any data that is not identified by a handle. u

Your Apple event handler should always set its function result either to `noErr` if it successfully handles the Apple event or to a nonzero result code if an error occurs. If your handler returns a nonzero result code, the Apple Event Manager adds a `keyErrorNumber` parameter to the reply Apple event (unless you have already added a `keyErrorNumber` parameter). This parameter contains the result code that your handler returns. The client should check whether the `keyErrorNumber` parameter exists to determine whether your handler performed the requested action. In addition to returning a result code, your handler can also return an error string in the `keyErrorString` parameter of the reply Apple event. The client can use this string in an error message to the user.

If the client application requested a reply, the Apple Event Manager returns the reply Apple event, which is identified by the event class `kCoreEventClass` and by the event ID `kAEAnswer`. When the client has finished using the reply Apple event, it should dispose of both the reply event and the original event—and thereby deallocate the memory they use—by calling the `AEDisposeDesc` function. The Apple Event Manager takes care of disposing both the Apple event and the reply Apple event after a server

application's handler returns to `AEProcessAppleEvent`, but a server application is responsible for disposing of any Apple event data structures it creates while extracting data from the Apple event.

Figure 3-12 shows the entire process of responding to an Apple event.

Figure 3-12   Responding to an Open Documents event
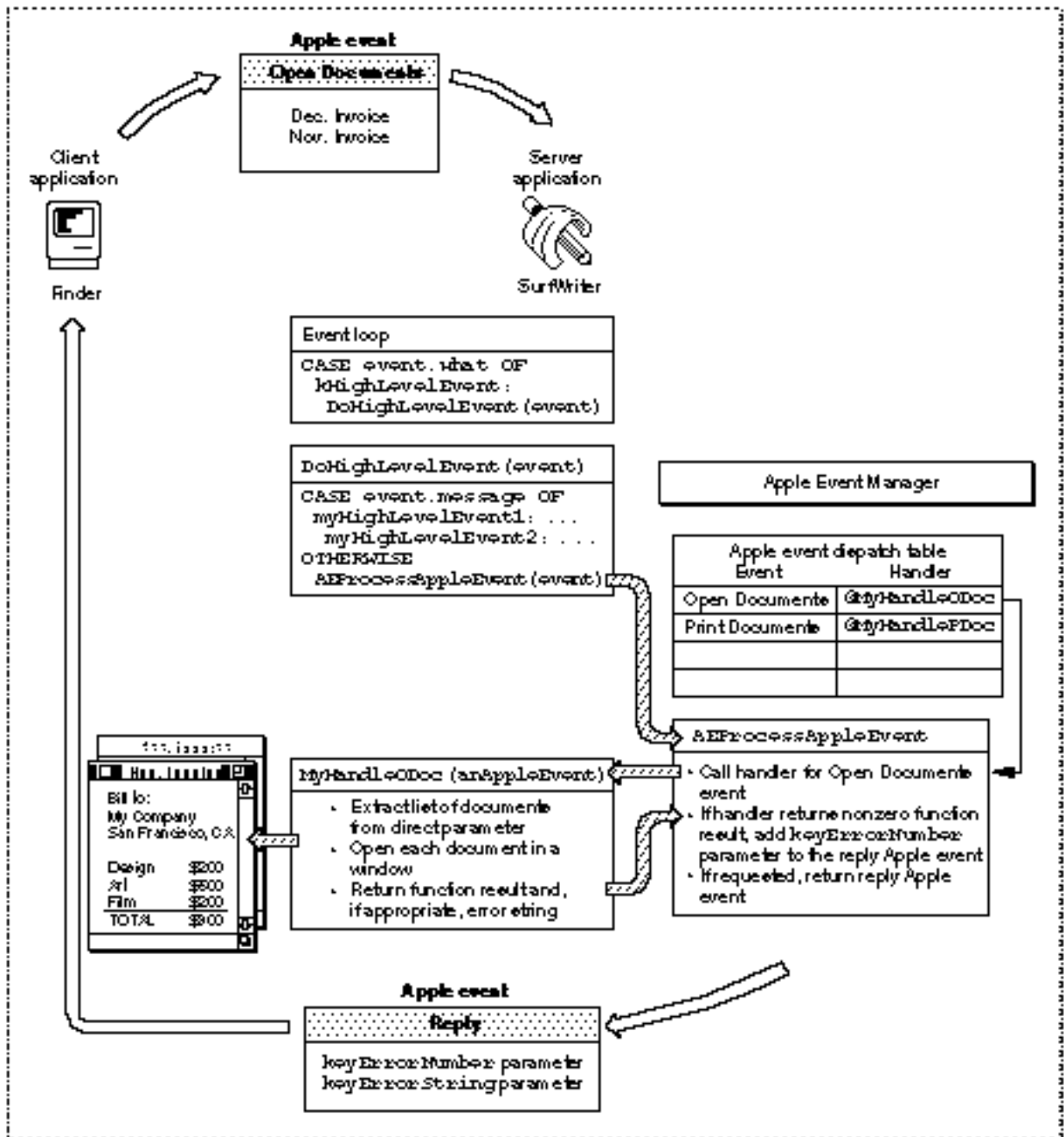
When your handler returns a result code to the Apple Event Manager, you have finished your response to the client application's Apple event.

Creating and Sending Apple Events

Your application can use Apple events to request services or information from other applications, send information to other applications, or trigger actions within your application. For example, you can use the core Apple event Get Data to request specific data from another application's documents. Similarly, you can use other Apple events to request services—for example, asking a spell-checking application to check the text in a document created by your application. Consult the *Apple Event Registry: Standard Suites* for the format and function of the standard Apple events that you want your application to send.

To communicate with another application by sending an Apple event, your application must

n   set the appropriate flags in its `'SIZE'` resource

n   create an Apple event record by calling the `AECreateAppleEvent` function

n   use Apple Event Manager functions to add parameters and any additional attributes to the Apple event

n   call the `AESend` function to send the Apple event

n   dispose of any copies of descriptor records that you have created

n   handle the reply Apple event (if necessary)

The sections that follow describe how your application can use the Apple Event Manager to accomplish these tasks. The chapter "Creating and Sending Apple Events" in this book provides detailed information about creating and sending Apple events.

To act as a server for your application, the target application must support high-level events and must be running. The server can be your own application, another application running on the user's computer, or an application running on another user's computer connected to the network.

Your application should also allow the user to choose among the various applications available as servers. The `PPCBrowser` function allows users to select target applications on their own computers or on computers connected to the network. The `PPCBrowser` function presents a standard user interface for choosing a target application, much as the Standard File Package provides a standard user interface for opening and saving files. See the chapter "Program-to-Program Communications Toolbox" in this book for details on using the `PPCBrowser` function.

If the server application is on a remote computer on a network, the user of that computer must allow program linking to the server application. The user of the server application does this by selecting the application icon in the Finder, choosing Sharing from the File menu, then clicking the Allow Remote Program Linking checkbox. If the user has not yet started program linking, the Sharing command offers to display the Sharing Setup control panel so that the user can start program linking. The user must also authorize remote users for program linking by using the Users & Groups control panel. Program linking and setting up authenticated sessions are described in the chapter "Program-to-Program Communications Toolbox" in this book.

Creating an Apple Event Record

Use the `AECreateAppleEvent` function to create an Apple event record. Using the arguments you pass to the `AECreateAppleEvent` function, the Apple Event Manager constructs the data structures describing the event class, the event ID, and the target address attributes of an Apple event. The event class and event ID, of course, identify the particular event you wish to send. The target address identifies the intended recipient of the Apple event.

You can specify two other attributes with the `AECreateAppleEvent` function: the reply ID and the transaction ID. For the reply ID attribute, you usually specify the `kAutoGenerateReturnID` constant to the `AECreateAppleEvent` function. This constant ensures that the Apple Event Manager generates a unique return ID for the reply Apple event returned from the server. For the transaction ID attribute, you usually specify the `kAnyTransactionID` constant, which indicates that this Apple event is not one of a series of interdependent Apple events.

Adding Apple Event Attributes and Parameters

The Apple event record created with the `AECreateAppleEvent` function serves as a foundation for the Apple event you want to send. Descriptor records and descriptor lists are the building blocks from which the complete Apple event record is constructed. To create descriptor records and descriptor lists and add items to a descriptor list, use the following functions:

Function	Description
AECreateDesc	Takes a descriptor type and a pointer to data and converts them into a descriptor record
AECreateList	Creates an empty descriptor list or AE record.
AEPutPtr	Takes a descriptor type and a pointer to data and adds the data to a descriptor list as a descriptor record; used, for example, to add to a descriptor list a number used as the parameter of an Apple event requesting a calculation.
AEPutDesc	Adds a descriptor record to a descriptor list; used, for example, to add to a descriptor list an alias record used as the direct parameter of an Apple event requesting file manipulation.

To add the remaining attributes and parameters necessary for your Apple event to the
Apple event record, you can use these additional Apple Event Manager functions:

Function	Description
AEPutParamPtr	Takes a keyword, descriptor type, and pointer to data and adds the data to an Apple event record as a parameter with the specified keyword (replacing any existing parameter with the same keyword); used, for example, to put numbers into the parameters of an Apple event that asks the server to perform a calculation.
AEPutParamDesc	Takes a keyword and a descriptor record and adds the descriptor record to an Apple event record as a parameter with the specified keyword (replacing any existing parameter with the same keyword); used, for example, to place a descriptor list containing alias records into the direct parameter of an Apple event that requests a server to manipulate files.
AEPutAttributePtr	Takes a keyword, descriptor type, and pointer to data and adds the descriptor record to an Apple event record as an attribute with the specified keyword (replacing any existing attribute with the same keyword); used, for example, to change the transaction ID of an Apple event record that is waiting to be sent.
AEPutAttributeDesc	Takes a keyword and a descriptor record and adds the descriptor record to an Apple event record as an attribute with the specified keyword (replacing any existing attribute with the same keyword); used, for example, to replace the descriptor record used for the target address attribute in an Apple event record waiting to be sent.

Apple event parameters for core events and functional-area events can include
descriptions of Apple event objects in special descriptor records called object specifier
records. For an overview of object specifier records, see "Working With Object Specifier
Records," which begins on page 3-32.

Sending an Apple Event and Handling the Reply

After you add all the attributes and parameters required for the Apple event, use the
AESend function to send the Apple event. The Apple Event Manager uses the
Event Manager to transmit the Apple event to the server application.

The AESend function requires that you specify whether your application should wait for
a reply from the server. If you specify that you want a reply, the Apple Event Manager
prepares a reply Apple event for your application by passing a default reply Apple event
to the server. The Apple Event Manager returns any nonzero result code from the
server's handler in the keyErrorNumber parameter of the reply Apple event.
The server can return an error string in the keyErrorString parameter of the reply
Apple event. The server can also use the reply Apple event to return any data you
requested—for example, the results of a numeric calculation or a list of misspelled words.

You specify how your application should wait for a reply by using one of these flags in the `sendMode` parameter of the `AESend` function:

Flag	Description
kAENoReply	Your application does not want a reply Apple event.
kAEQueueReply	Your application wants a reply Apple event; the reply appears in your event queue as soon as the server has the opportunity to process and respond to your Apple event.
kAEWaitReply	Your application wants a reply Apple event and is willing to give up the processor while waiting for the reply; for example, if the server application is on the same computer as your application, your application yields the processor to allow the server to respond to your Apple event.

If you specify the `kAEWaitReply` flag, you should provide an idle function. This function should process any non-high-level events that occur while your application is waiting for a reply. You supply a pointer to your idle function as a parameter to the `AESend` function. So that your application can process other Apple events while it is waiting for a reply, you can also specify an optional filter function to the `AESend` function.

If you specify the `kAENoReply` flag, the reply Apple event prepared by the Apple Event Manager for the server application consists of a null descriptor record.

If your Apple event may require the user to interact with the server application (for example, to specify print or file options), you can communicate your user interaction preferences to the server by specifying additional flags in the `sendMode` parameter of the `AESend` function. These flags specify the conditions, if any, under which the server application can interact with the user and, if interaction is allowed, whether the server should come directly to the foreground or post a notification request.

The server application specifies its own preferences for user interaction by specifying flags to the `AESetInteractionAllowed` function, as described in the previous section. The interaction of the client and server applications' preferences is explained in detail in "Interacting With the User," which begins on page 4-45.

After you send an Apple event, your application is responsible for disposing of the Apple event record—and thereby deallocating the memory its data uses—by calling the `AEDisposeDesc` function. If you create one descriptor record and add it to another, the Apple Event Manager adds a copy of the newly created one to the existing one and also makes a copy of the associated data. For example, you might use the `AECreateDesc` function to create a descriptor record that you wish to add to an Apple event. When you use the `AEPutParamDesc` function, it adds a copy of your newly created descriptor record, including its data, as a parameter to an existing Apple event. When you no longer need the original descriptor record, you should call `AEDisposeDesc` to dispose of it.

Your application should dispose of all the descriptor records that are created for the purposes of adding parameters and attributes to an Apple event. You normally dispose of your Apple event and its reply after you receive a result from the `AESend` function. You should dispose of these even if `AESend` returns an error result.

If you specify the `kAEWaitReply` flag, the reply Apple event is returned in a parameter you pass to the `AESend` function. If you specify the `kAEQueueReply` flag to the `AESend` function, the reply Apple event is returned in the event queue. In this case, the reply is identified by the event class `kCoreEventClass` and the event ID `kAEAnswer`. Your application processes reply events in its event queue in the same way that server applications process Apple events.

Your application should check for the existence of the `keyErrorNumber` parameter of the reply Apple event to ensure that the server performed the requested action. The server can also return, in the `keyErrorString` parameter, any error messages you need to display to the user.

Whenever a server application provides an error string, it should also provide an error number. However, you can't count on all server applications to do so. The absence of the `keyErrorNumber` parameter doesn't necessarily mean that there won't an error string provided in the `keyErrorString` parameter. A client application should therefore check for both the `keyErrorNumber` and `keyErrorString` parameters before assuming that no error has occurred. If a string has been provided without an error number, an error has occurred.

After extracting the information it needs from the reply event, your handler should dispose of the reply by calling the `AEDisposeDesc` function. Similarly, when your handler no longer needs descriptor records it has extracted from the reply, it should call `AEDisposeDesc` to dispose of them.

The next section provides an overview of the way a source application identifies Apple event objects supported by a target application. If you are starting by supporting only the Required suite and the Apple events sent by the Edition Manager, you can skip the next section and go directly to "About the Apple Event Manager," which begins on page 3-48.

Working With Object Specifier Records

Most of the standard Apple events allow the source application to refer, in an Apple event parameter, to Apple event objects within the target application or its documents. The Apple Event Manager allows applications to construct and interpret such references by means of a standard classification system for Apple event objects. This system, described in detail in the *Apple Event Registry: Standard Suites,* is summarized in "The Classification of Apple Event Objects," which begins on page 3-39. A description in an Apple event parameter that uses this classification system takes the form of an object specifier record.

An **object specifier record** is a descriptor record of descriptor type
`typeObjectSpecifier` that describes the location of one or more Apple event objects:
for example, the table "Summary of Sales" in the document "Sales Report," or the third
row in that table, or the last row of the column "Totals." With the aid of
application-defined functions, the Apple Event Manager can conduct a step-by-step
search according to such instructions in an object specifier record, locating first the
document, then the table, then other objects, and so on until the requested object has
been identified. Object specifier records can specify many combinations of identifying
characteristics that cannot be specified using one of the simple data types.

This section introduces object specifier records and the organization of their data. You
need to read this section (a) if you plan to support the Core suite or any of the standard
functional-area suites and (b) if you want to make your application scriptable—that is,
capable of responding to scripts written in a scripting language.

IMPORTANT

An object specifier record identifies one or more Apple event objects
among many; it contains a description of each object, not the object
itself. An Apple event object described by an object specifier record
exists only in the server application's document or in the server
application itself. s

A client application cannot retrieve an Apple event object from a server application
unless the server application can accurately locate it. Thus, to locate characters of a
specific color, a server application must be able to identify a single character's color; to
locate a character in a cell, a server application must be able to locate both the table and
the cell.

A client application can create object specifier records for use as Apple event
parameters. Scripting components can also create object specifier records as Apple
event parameters for the Apple events they generate in the course of executing a script.
A server application that receives an Apple event containing an object specifier record
should resolve the object specifier record—that is, locate the requested Apple event
objects.

To respond to core and functional-area Apple events received by your application, you
must first define a hierarchy of Apple event objects for your application that you want
other applications or scripting languages to be able to describe. The Apple event objects
for your application should be based as closely as possible on the standard object classes
described by the *Apple Event Registry: Standard Suites.* After you have decided which of
the standard Apple event objects make sense for your application, you can write
functions that locate objects on the basis of information in an object specifier record. If
you want your application to send specific Apple events to other applications, you must
also write functions that can create object specifier records and add them to Apple
events. Your application does not need to create object specifier records in order to be
scriptable. However, to write functions that can help the Apple Event Manager resolve
object specifier records, you need to know how they are constructed.

"Finding Apple Event Objects," which begins on page 3-46, provides an overview of the way the Apple Event Manager works with your application-defined functions to locate the Apple event objects described in an object specifier record. The chapter "Resolving and Creating Object Specifier Records" in this book describes in detail how to support object specifier records as a server or client application.

Data Structures Within an Object Specifier Record

The organization of the data for an object specifier record is nearly identical to that of the data for an AE record. An object specifier record is a structure of data type `AEDesc` whose data handle usually refers to four keyword-specified descriptor records describing one or more Apple event objects. An AE record is a structure of data type `AERecord` whose data handle refers to one or more Apple event parameters.

The four keyword-specified descriptor records for an object specifier record provide information about the requested Apple event object or objects.

Keyword	Description of data
`keyAEDesiredClass`	Four-character code indicating the object class ID
`keyAEContainer`	A description of the container for the requested object, usually in the form of another object specifier record
`keyAEKeyForm`	Four-character code for the key form, which indicates how to interpret the key data
`keyAEKeyData`	Key data, used to distinguish the desired Apple event object from other objects of the same object class in the same container

For example, the data for an object specifier record identifying a table named "Summary of Sales" in a document named "Sales Report" consists of four keyword-specified descriptor records that provide the following information:

n the object class ID for a table

n another object specifier record identifying the document "Sales Report" as the container for the table

n a key form constant indicating that the key data contains a name

n key data that consists of the string "Summary of Sales"

The **object class ID** specifies the Apple event object class to which the object belongs. An Apple event object class is a category for Apple event objects that share specific characteristics (see "Apple Events and Apple Event Objects" on page 3-6). The characteristics of each object class are listed in the *Apple Event Registry: Standard Suites.* For example, the Core suite defines object classes for documents, paragraphs, words, windows, and floating windows. The first keyword-specified descriptor record in an object specifier record uses a four-character code or a constant to specify the object class ID. The object class for words, for example, can be identified by either the object class ID `'cwor'` or the constant `cWord`.

Note

The object class ID identifies the object class of an Apple event object described in an object specifier record, whereas the event class and event ID identify an Apple event. u

The **container** for an Apple event object is usually another Apple event object. For example, the container for a document might be a window, and the container for characters, delimited items, or a word might be another word, a paragraph, or a document. The container is identified by the second keyword-specified descriptor record in an object specifier record; usually this is another object specifier record. The container can also be specified by a null descriptor record, which indicates a default container or a container already known to the Apple Event Manager.

The descriptor record in an object specifier record that identifies an Apple event object's container can in turn use another object specifier record to identify the container's container, and so on until the Apple event object is fully specified. For example, an object specifier record identifying a paragraph might specify the paragraph's container with another object specifier record that identifies a page. That object specifier record might in turn specify the page's container with another object specifier record identifying a document. The ability to nest one object specifier record within another in this way makes it possible to identify elements such as "the first row in the table named 'Summary of Sales' in the document named 'Sales Report.'"

The **key form** and **key data** distinguish the desired Apple event object from other Apple event objects of the same object class. The key form describes the form the key data takes. The third keyword-specified descriptor record in an object specifier record usually specifies the key form with one of seven standard constants:

Key form	Value	Corresponding key data
formPropertyID	'prop'	Property ID for an element's property
formName	'name'	Element's name
formUniqueID	'ID '	A value that uniquely identifies an object within its container or across an application
formAbsolutePosition	'indx'	An integer or other constant indicating the position of one or more elements in relation to the beginning or end of their container
formRelativePosition	'rele'	A constant that specifies the element just before or after the container
formTest	'test'	Descriptor records that specify a test
formRange	'rang'	Descriptor records that specify a group of elements between two other elements

A key form of formPropertyID indicates key data that specifies a property. A **property** of an Apple event object is a specific characteristic of that object that can be identified by a constant. The properties associated with the object class for documents include the name of the document and a flag indicating whether the document has been modified since the last save. The properties associated with the object class for words include color, font, point size, and style.

Figure 3-13 shows the structure of a typical object specifier record: four keyword-specified descriptor records that specify the class ID, the container, the key form, and the key data. These four keyword-specified descriptor records are the data for a descriptor record (AEDesc) of descriptor type typeObjectSpecifier. Note the similarities between the object specifier record shown in Figure 3-13 and the Apple event record shown in Figure 3-9 on page 3-19. Like an Apple event record or an AE record, an object specifier record consists of a list of keyword-specified descriptor records.

Figure 3-13 shows the structure of a simple object specifier record that specifies the key form formPropertyID, formName, or formAbsolutePosition. For detailed information about the structure of object specifier records that specify the other key forms, see the chapter "Resolving and Creating Object Specifier Records" in this book.

Figure 3-13 Data structures within a simple object specifier record
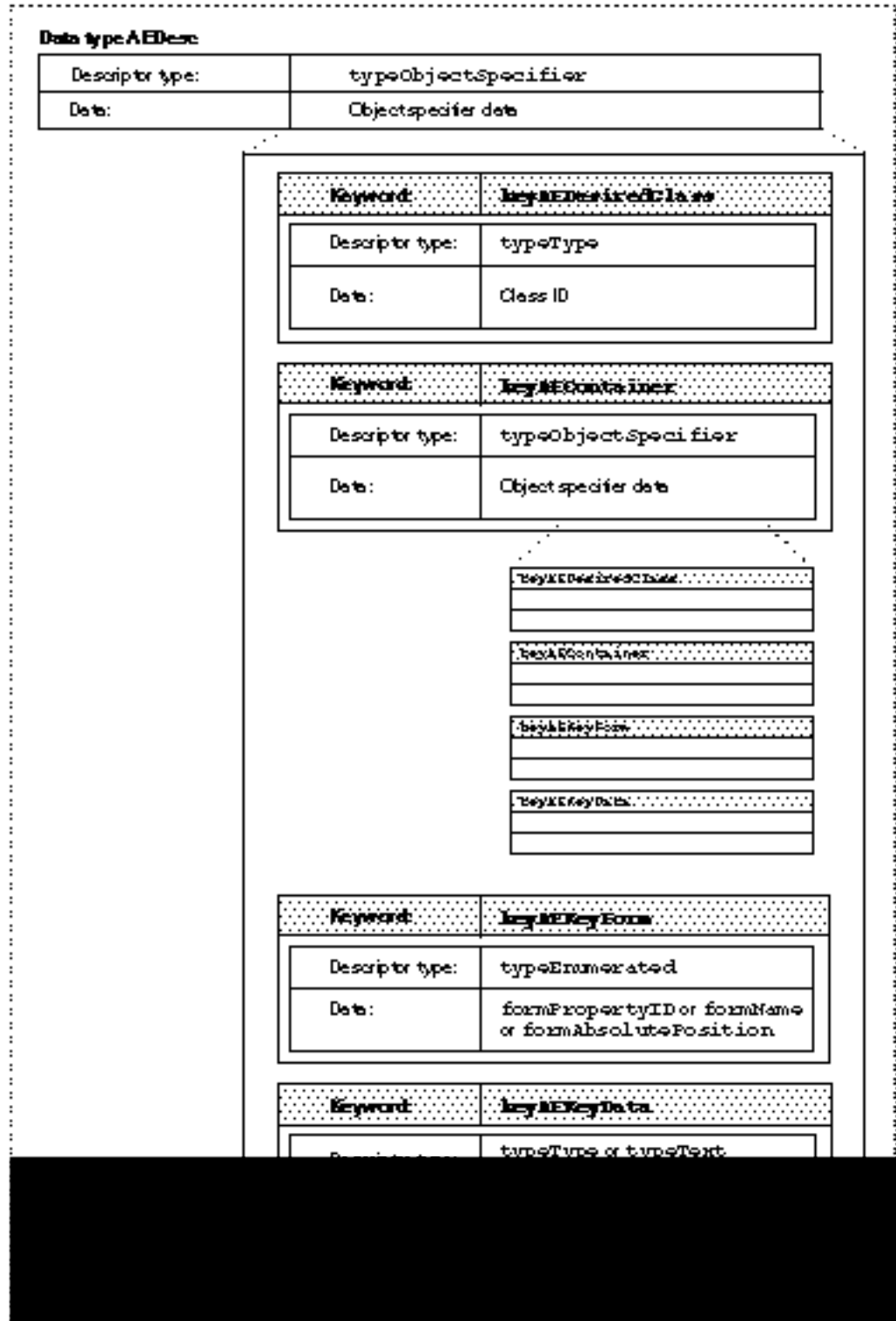
Figure 3-14 shows the object specifier record for the Get Data event previously illustrated in Figure 3-4 on page 3-11. The object class ID tells the SurfWriter application that the requested data is an element of class `cTable`. The container for the table is the document "Sales Report." The key form is `formName`, which tells the server application that the key data identifies the Apple event object by name. The key data is the name of the table.

Figure 3-14   An object specifier record in a Get Data event
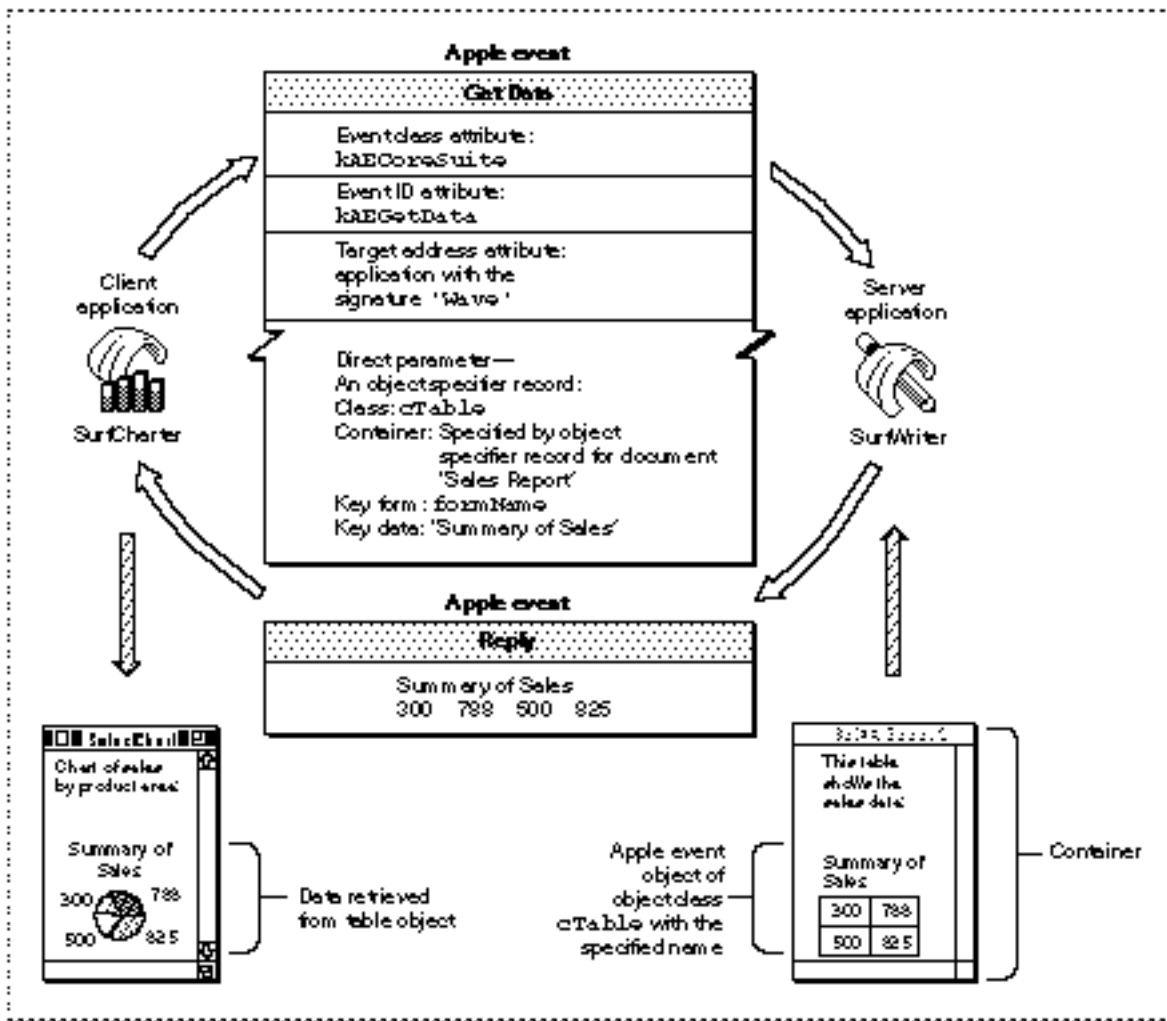
To add an object specifier record to an Apple event as one of its parameters, your application must first create the object specifier record. "Creating Object Specifier Records," which begins on page 6-55, describes the Apple Event Manager routines for creating object specifier records.

To respond to Apple events that include object specifier records, your application should use the standard classification system for Apple event objects and provide functions that can locate those objects within your application or its documents. The next section summarizes the classification of Apple event objects as defined in the *Apple Event Registry: Standard Suites.*

The Classification of Apple Event Objects

To create or resolve object specifier records, your application should use the classification of Apple event objects defined by the *Apple Event Registry: Standard Suites.* This section summarizes the concepts that underlie that classification system. You should have a copy of the *Apple Event Registry: Standard Suites* available for reference purposes while you read this section.

You do not need to write your application in an object-oriented programming language in order to support Apple event objects in your application. However, you must understand the classification system described in this section in order to classify Apple event objects in your application and to write routines that can locate them on the basis of information contained in object specifier records.

Object Classes

Except for the concept of inheritance, Apple event objects are different from the objects used in object-oriented programming languages. Apple event objects are distinct items in a server application or any of its documents that can be specified by an object specifier record in an Apple event sent by a client application. Apple event objects are often, but not always, items that a user can differentiate and manipulate within an application, such as words, paragraphs, shapes, windows, or style formats. Every Apple event object can be classified according to its object class, which defines both its characteristics and its behavior. The object classes listed in the *Apple Event Registry: Standard Suites* provide a method of describing Apple event objects that all applications can understand. Object classes permit more flexibility than simple descriptor types; for example, a word can be defined as a simple string, or it can be defined as an Apple event object with specific characteristics such as font or style.

Note
The definition of an object class only specifies conventions that determine how applications should handle Apple event objects that belong to that class. Your application must make sure that it uses the conventions correctly; they are not enforced by the Apple Event Manager. u

Each object class is identified by a four-character object class ID, which can also be represented by a constant. Constants for object classes always begin with the letter c.

The definition of an object class specifies its **superclass,** which is the object class from which a **subclass** (the class being defined) inherits some of its characteristics. Characteristics can also be inherited from special object classes, called **abstract superclasses,** that are used only in definitions of object classes and do not refer to real Apple event objects. The pattern of inheritance among object classes is called the **object class inheritance hierarchy.** Figure 3-15 shows a portion of this hierarchy. The abstract superclass cObject is at the top of the hierarchy and is therefore the only object class that has no superclass. At the next level are cText, which is a regular object class, and cOpenableObject, which is an abstract superclass. Both are subclasses of cObject and superclasses for their own subclasses. The object classes cWord, cItem, and cChar are all subclasses of cText. Similarly, cWindow, cDocument, and cFile are subclasses of cOpenableObject. Every object class inherits all the characteristics of its superclass and can also add characteristics of its own.

Figure 3-15 Superclasses and subclasses
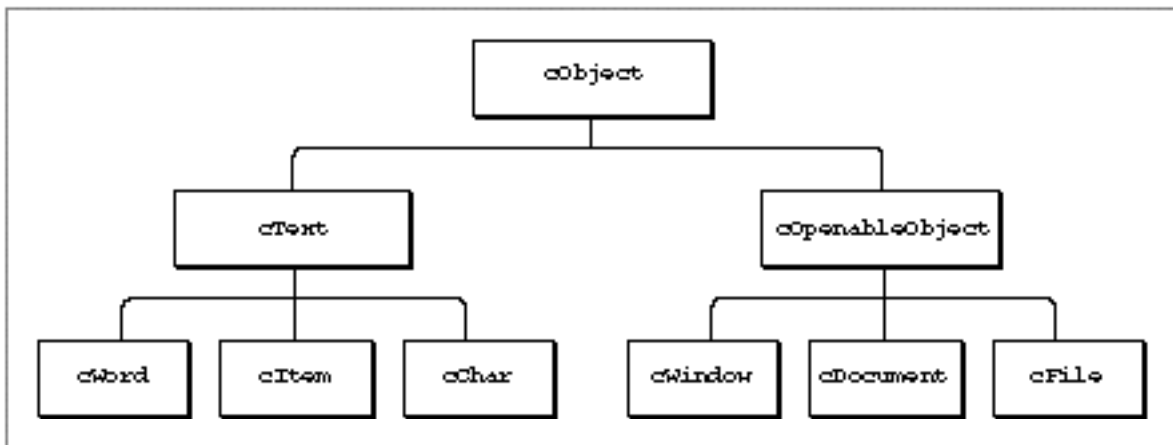
Here are some of the object classes defined for the Core suite:

Class	Class ID	Description
cChar	'cha '	Text characters
cDocument	'docu'	Macintosh documents
cFile	'cfil'	Macintosh files
cSelection	'csel'	User or application selections
cText	'ctxt'	Series of characters
cWindow	'cwin'	Standard Macintosh windows

Here are some of the object classes defined for the Text suite:

Class	Class ID	Description
cChar	'cha '	Text characters
cLine	'clin'	Lines of text
cParagraph	'cpar'	Paragraphs
cText	'ctxt'	Series of characters
cTextFlow	'cflo'	Text flows
cWord	'cwor'	Words

As you can see, some object classes, such as cChar and cText, are defined in more than one suite. For example, the definition of the cText object class in the Text suite is an **extension** of the cText object class defined in the Core suite; it duplicates all the characteristics of the Core suite object class and adds some of its own. Like a word in a dictionary, one object class ID can have several related definitions. You can choose to support the definition that best suits your application; or, if necessary, you can create extensions of your own. The extension of an object class is different from inheritance between object classes. An extension of a standard object class provides additional ways of describing an Apple event object of that class, whereas the object class inheritance hierarchy determines the pattern of characteristics shared by different object classes.

The definition of an object class always specifies a default descriptor type. Suppose, for example, that a client application sends a Get Data, Cut, or Copy event that specifies an Apple event object but does not specify a descriptor type for the returned data. In this case, the server application returns a descriptor record of the default descriptor type for the object class of the specified Apple event object. For example, the default descriptor type for Apple event objects of class cWord is typeIntlText, a descriptor type that specifies an undelimited string of characters in a specific language and script system. The client application can also request that the data be returned in a descriptor record of some other data type.

The definition of an object class includes three lists of characteristics: properties, element classes, and Apple events that support the object class. (The next section describes properties and element classes.) Any or all of these characteristics may be inherited from a superclass. An Apple event is listed for an object class if its parameters can specify objects of that class. The definition for cWindow, for example, lists 12 Apple events, including the Open, Close, and Move events, whose parameters can include object specifier records that specify windows. The cWindow class inherits all of these Apple events from its abstract superclass, cOpenableObject.

The *Apple Event Registry: Standard Suites* also defines **primitive object classes,** which describe Apple event objects that contain a single value. For example, the cBoolean, cLongInteger, and cAlias object classes are all primitive object classes. The object class ID for a primitive object class is the same as the four-character value of its descriptor type. Primitive object classes contain no properties; they contain only the value of the data.

Properties and Elements

The properties listed for an object class can be used to identify characteristics of Apple event objects that belong to that class. Each property is identified by a four-character property ID, which can also be represented by a constant. Constants for properties always begin with the letter p.

Here are constants and property IDs for some properties:

Property	Property ID	Description
pName	'pnam'	Name of an Apple event object
pBounds	'pbnd'	Coordinates of a window
pVisible	'pvis'	Indicates whether a window is visible
pIsModal	'pmod'	Indicates whether a window is modal
pClass	'pcls'	Class ID of an Apple event object
pFont	'font'	Font
pTextStyle	'txst'	Text style
pColor	'colr'	Text color
pTextPointSize	'ptps'	Point size
pScriptTag	'psct'	Script system identifier
pFillColor	'flcl'	Fill color

The property of an Apple event object is itself defined as a single Apple event object whose container is the object to which the property belongs. For example, the pFont property of a word is defined by the name of a font, such as New York; the string that identifies the font is an Apple event object of class cText.

The constant cProperty specifies the object class for any object specifier record that identifies a property.

```
CONST cProperty = 'prop';
```

An object specifier record for a property specifies cProperty as the object class ID, the Apple event object to which the property belongs as the container, formPropertyID as the key form, and a constant such as pFont as the key data.

The **elements** of a specific Apple event object are the other Apple event objects it contains, excluding those that define its properties. An object specifier record for an element specifies the Apple event object in which the element is located as the container and can specify any key form except formPropertyID. Each object class definition in the *Apple Event Registry: Standard Suites* includes a list of **element classes,** which are the object classes of the elements that an Apple event object can contain.

An Apple event object contains exactly one of each of its properties, whereas it can contain no elements or many elements of the same element class. In general, a property of an object describes something about that object; a property can be examined or changed but never deleted. An element can be one or more discrete objects contained in another object and can usually be deleted.

For example, because a paragraph can contain one or more words, one of the element classes listed for the object class cParagraph is the object class cWord. Individual words can be deleted from a paragraph. However, even though a word in a paragraph can be in a different font from the words around it, a paragraph can have only one pFont property. This property is defined as the font of the first character in the paragraph and consists of the name of a font. The paragraph's pFont property can be changed but not removed.

The properties and element classes listed for each object class definition in the *Apple Event Registry: Standard Suites* can be inherited from a superclass, or they can originate with a subclass. Figure 3-16 illustrates the object class inheritance hierarchy for the object class cWindow in the Core suite. Boldface terms in the figure represent those properties, element classes, or Apple events that are not inherited. The object class cWindow includes all the properties and Apple events of its superclass, cOpenableObject, which in turn includes all the properties and Apple events of its superclass, cObject. The object class cWindow also includes 11 properties and one element class that originate with cWindow and are not inherited.

The pClass property—the property that specifies the four-character class ID—originates with cObject. Because the definitions of all object classes are ultimately derived from cObject, pClass is inherited by all object classes. The definition for cObject also lists ten Apple events, which include common events such as Get Data, Move, and Delete Element. Because cObject is at the top of the object class inheritance hierarchy, these ten Apple events can use object specifier records that describe Apple event objects of any object class as a direct parameter. Like all abstract superclasses, cObject does not correspond to a real Apple event object, so its definition does not list any element classes. Unlike any other object class, cObject is at the top of the object class inheritance hierarchy and therefore does not have a superclass.

Figure 3-16   The object class inheritance hierarchy for the object class `cWindow`
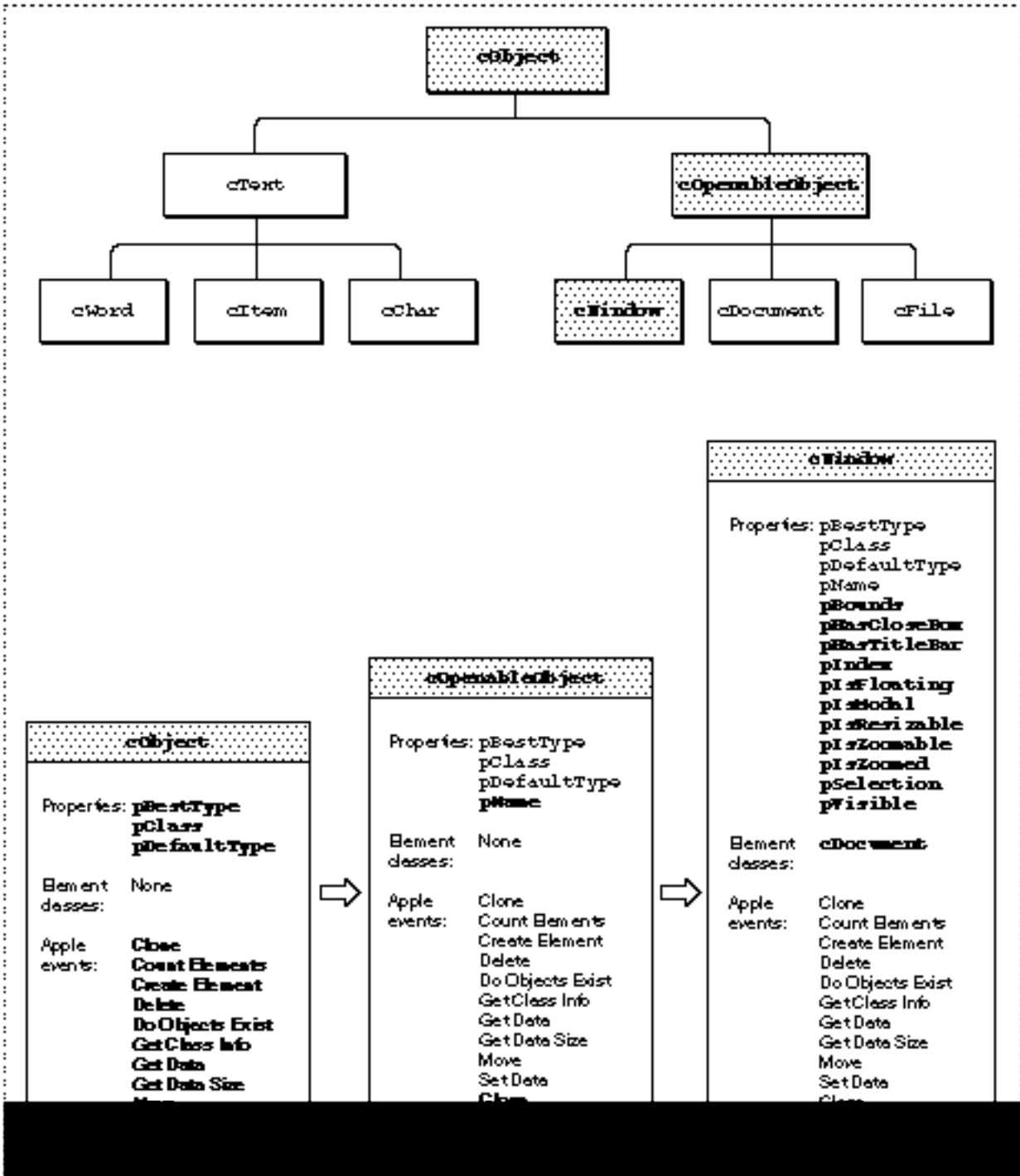
The chain of containers that determine the location of one or more Apple event objects is called the **container hierarchy.** The container hierarchy, which specifies the location of real Apple event objects, is different from the object class inheritance hierarchy, which is an abstract concept that determines which properties, element classes, and Apple events an object class inherits from its superclass. For example, the container hierarchy for an Apple event object of class cWord can vary from one word to another, because various combinations of other Apple event objects, such as a document, a paragraph, a delimited string, or another word, can contain a word.

Applications that support Apple event objects must be able to identify the order of several elements of the same class that are contained within another Apple event object. For example, each word in a paragraph should have an identifiable order, such as the 5th word or the 12th word. This allows other applications to identify Apple event objects by describing their absolute position within a container.

Figure 3-17 shows an Apple event object of object class cWord—the word "Sales"—contained in another Apple event object of object class cParagraph. (Both these object classes are defined in the Text suite.) The figure shows only a portion of the container hierarchy for the word, since a complete description of the word would also include the containers that specify the location of the paragraph.

Your application must take account of the definitions in the *Apple Event Registry: Standard Suites* for any object classes you want to support. For example, the definition for the object class cText lists paragraphs, lines, words, and characters as Apple event objects that can be contained in Apple event objects of class cText. To support Apple events that refer to elements of object class cText, your application should associate the cText object class with paragraphs, lines, words, and characters in its documents. The list of properties defined for class cText includes the properties pColor, pFont, pPointSize, pScriptTag, and pTextStyles. If you want to support Apple events that distinguish a boldface 12-point word of object class cText from an italic 14-point word, for example, your application must associate the point size and style of text in its documents with the properties pPointSize and pTextStyles defined for class cText.

Figure 3-17   An Apple event object of class `cWord` contained in an Apple event object of class `cParagraph`
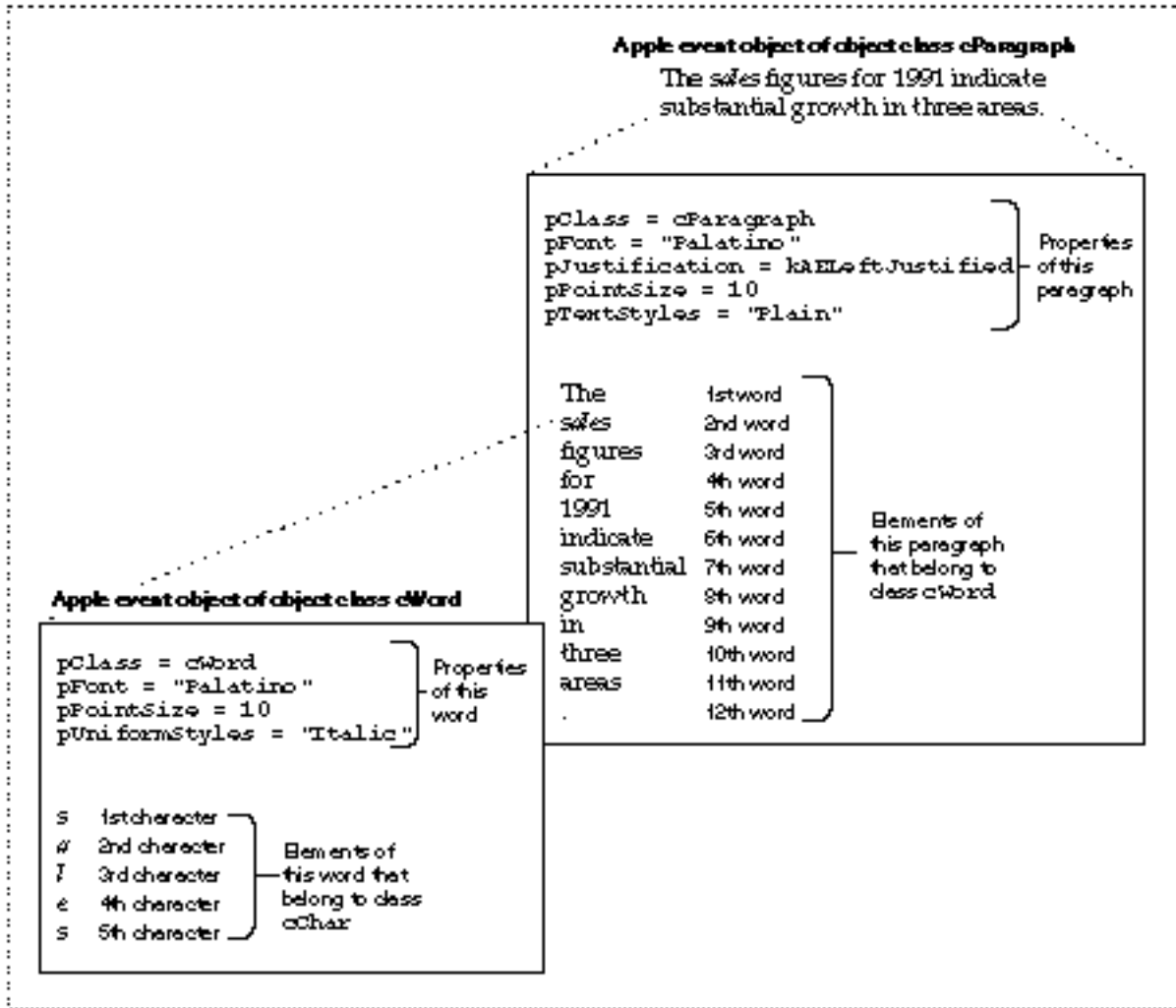
Finding Apple Event Objects

Most of the Apple events in the Core suite and the functional-area suites defined in the *Apple Event Registry: Standard Suites* can include parameters that consist of object specifier records. Your application's handlers for these events can use the `AEResolve` function to resolve object specifier records: that is, to locate the Apple event objects they describe.

The `AEResolve` function parses an object specifier record and performs related tasks that are the same for all object specifier records. When necessary, the `AEResolve` function calls application-defined functions to perform tasks that are unique to the application, such as locating a specific Apple event object in the application's data structures.

Your application can provide two kinds of application-defined functions for use by
`AEResolve`. **Object accessor functions** locate Apple event objects. Every application
that supports simple object specifier records must provide one or more object accessor
functions. **Object callback functions** perform other tasks that only an application can
perform, such as counting, comparing, or marking Apple event objects.

Each time `AEResolve` calls one of your application's object accessor functions
successfully, the object accessor function returns a special descriptor record, called a
token, that identifies either an element in a specified container or a property of a
specified Apple event object. The token can be of any descriptor type, including
descriptor types you define yourself.

You install object accessor functions by using the `AEInstallObjectAccessor`
function. Much like the `AEInstallEventHandler` function,
`AEInstallObjectAccessor` uses an **object accessor dispatch table** to map
requests for Apple event objects to the appropriate object accessor functions in
your application. These requests refer to objects of a specified object class in
containers identified by a token of a specified descriptor type.

Responding to an Apple event that includes an object specifier record typically involves
these steps:

1. After determining that the event is an Apple event, your application calls
   `AEProcessAppleEvent`.

2. The `AEProcessAppleEvent` function uses the Apple event dispatch table to
   dispatch the event to the your application's handler for that event.

3. The Apple event handler extracts the Apple event parameters, and passes the object
   specifier records they contain to `AEResolve`.

4. The `AEResolve` function uses the object accessor dispatch table to call one or more
   object accessor functions, one at a time, that can identify the nested Apple event
   objects described by each object specifier record. Each object accessor function returns
   a token for the object it finds, which in turn helps to determine which object accessor
   function the `AEResolve` function will use to locate the next Apple event object.

5. The `AEResolve` function returns the final token for the requested object to the
   application's handler.

The resolution of an object specifier record always begins with the outermost container it
specifies. For example, to locate a table named "Summary of Sales" in the document
named "Sales Report," the `AEResolve` function first calls an object accessor function
that can locate the document, then uses the token returned by that function to identify an
object accessor function that can locate the table. It then returns the token for the table to
the Apple event handler that called `AEResolve`.

The chapter "Resolving and Creating Object Specifier Records" in this book describes in
detail how to resolve object specifier records and how to write and install object accessor
and object callback functions.

About the Apple Event Manager

You can use the Apple Event Manager to

n respond to Apple events as a server application

n create and send Apple events as a client application

n resolve and create object specifier records

n support Apple event recording

This section briefly summarizes the steps involved in providing each kind of support and tells where to find the relevant information in this book.

Supporting Apple Events as a Server Application

You do not need to implement all Apple events at once. You can begin by supporting just the required events and, if necessary, the events sent by the Edition Manager. The beginning of the section "Handling Apple Events" on page 4-4 describes how to provide this minimal level of support.

It is relatively easy to respond to the required events and the events sent by the Edition Manager. If, however, your application cannot respond to any other Apple events, other applications will not be able to request services that involve locating specific Apple event objects within your application or its documents, and users will not be able to control your application by executing scripts. To respond to Apple events it is likely to receive from other applications or from scripting components, your application must be able to respond to the appropriate core and functional-area Apple events.

Once you have provided the basic level of support for the Required suite and for events sent by the Edition Manager, you should

n decide which other Apple event suites you want to support

n define the hierarchy of Apple event objects within your application that you want scripting components or other applications to be able to identify—that is, which Apple event objects can be contained by other Apple event objects in your application

n write handlers for the Apple events you support, and install corresponding entries in your application's Apple event dispatch table

To decide which Apple event suites you want to support and how to define the hierarchy of Apple event objects in your application, consult the *Apple Event Registry: Standard Suites* and evaluate which Apple events and Apple event object classes make sense for your application. If necessary, you can extend the definitions of the standard Apple events and Apple events objects to cover special requirements of your application. It is better to extend the standard definitions rather than to define your own custom Apple events, because only those applications that choose to support your custom Apple events explicitly will be able to make use of them.

The chapter "Responding to Apple Events" in this book describes how to write Apple event handlers and related routines. The chapter "Resolving and Creating Object Specifier Records" describes how to resolve object specifiers in an Apple event that describes Apple event objects in your application or its documents.

If your application can respond to Apple events, you can make it scriptable simply by adding an `'aete'` resource. Scripting components use your application's `'aete'` resource to obtain information about the Apple events and corresponding human-language terminology that your application supports. The chapter "Apple Event Terminology Resources" in this book describes how to optimize your implementation of Apple events for use by scripting components and how to create an `'aete'` resource.

Supporting Apple Events as a Client Application

Because users can send Apple events to a variety of applications simply by executing a script, many applications have no need to send Apple events. However, if you want to factor your application for recording, or if you want your application to send Apple events directly to other applications, you can use Apple Event Manager routines to create and send Apple events.

To send an Apple event, you must

n create the Apple event

n add parameters and attributes

n use the `AESend` function to send the event

The chapter "Creating and Sending Apple Events" in this book describes how to perform these tasks.

Supporting Apple Event Objects

If your application responds to core and functional-area Apple events, it must also be able to resolve object specifier records that describe the objects on which those Apple events can act. In addition to the tasks described in the chapter "Responding to Apple Events," you must perform the following tasks to handle Apple events that contain object specifier records:

n Write object accessor functions that can locate the Apple event objects you support, and install corresponding entries in your application's object accessor dispatch table.

n Write any object callback functions that you decide to provide. To handle object specifier records that specify a test, your application must provide at least two object callback functions: one that counts objects and one that compares them.

n Call `AEResolve` from your Apple event handlers whenever an Apple event parameter includes an object specifier record.

The chapter "Resolving and Creating Object Specifier Records" describes how to perform these tasks. It also describes how applications that send Apple events to themselves or directly to other applications can create object specifier records.

Supporting Apple Event Recording

If you make your application scriptable, you may also want to make it recordable. Users of recordable applications can record their actions in the form of Apple events that a scripting component translates into a script. When a user executes a recorded script, the scripting component sends the same Apple events to the application in which they were recorded.

To make your application recordable, you should use Apple events to report user actions to the Apple Event Manager in terms of Apple events. One way to do this is to separate the code that implements your application's user interface from the code that actually performs work when the user manipulates the interface. This is called **factoring** your application. A factored application acts as both the client and server application for Apple events it sends to itself in response to user actions. When recording is turned on, the Apple Event Manager sends a copy of every event than an application sends to itself to the scripting component or other process that turned recording on.

The chapter "Introduction to Scripting" in this book provides an overview of how to make your application both scriptable and recordable. The chapter "Recording Apple Events" describes how to factor your application for recording and explains the Apple Event Manager's recording mechanism.

Responding to Apple Events

Contents

This chapter describes how your application can use the Apple Event Manager to respond to Apple events. Your application must be able to respond to the four required Apple events to take advantage of the launching and terminating mechanisms that are part of System 7 and later versions of system software. If your application provides publish and subscribe capabilities, it should also handle the events sent by the Edition Manager. To be scriptable, or capable of responding to Apple events sent by scripting components, your application should handle the appropriate core and functional-area Apple events.

Before you read this chapter, you should be familiar with the chapters "Introduction to Interapplication Communication" and "Introduction to Apple Events" in this book. You should also have a copy of the *Apple Event Registry: Standard Suites* available for reference.

Although the Apple events used by the Edition Manager are discussed in this chapter, you must refer to the chapter "Edition Manager" in this book for a full discussion of how to implement the Edition Manager's publish and subscribe features.

This chapter provides the basic information you need to make your application capable of responding to Apple events. To respond to core and functional-area Apple events, your application must also be able to resolve object specifier records. You should read the chapter "Resolving and Creating Object Specifier Records" before you write Apple event handlers for events that can contain object specifier records.

The section "Handling Apple Events," which begins on page 4-4, describes how to

n   accept and process Apple events

n   install entries in the Apple event dispatch tables

n   handle the required events

n   handle events sent by the Edition Manager

n   get data out of an Apple event

n   write handlers that perform the action requested by an Apple event

n   reply to an Apple event

n   dispose of Apple event data structures

n   write and install coercion handlers

The section "Interacting With the User," which begins on page 4-45, describes

n   how a server application can interact with the user when processing an Apple event

n   how client applications set user interaction preferences

n   how the client application's preferences and the server application's preferences affect user interaction

Handling Apple Events

You do not need to implement all Apple events at once. If you want to begin by supporting only the required Apple events, you must

- n   set bits in the `'SIZE'` resource to indicate that your application supports high-level events

- n   include code to handle high-level events in your main event loop

- n   write routines that handle the required events

- n   install entries for the required Apple events in your application's Apple event dispatch table

The following sections explain how to perform these tasks: "Accepting an Apple Event," which begins on page 4-5, "Installing Entries in the Apple Event Dispatch Tables," which begins on page 4-7, and "Handling the Required Apple Events," which begins on page 4-11.

To respond to the Apple events sent by the Edition Manager in addition to the required events, you must install entries for the Section Read, Section Write, Section Scroll, and Create Publisher events in your application's Apple event dispatch table and write the corresponding handlers, as described in "Handling Apple Events Sent by the Edition Manager" on page 4-20.

To respond to core and functional-area Apple events, you must install entries and write handlers for those events. You must also make sure that your application can locate Apple event objects with the aid of the Apple Event Manager routines described in the chapter "Resolving and Creating Object Specifier Records." These routines are currently available as the Object Support Library (OSL), which you must link with your application when you build it.

The Apple Event Manager (excluding the OSL) is available only in System 7 and later versions of system software. Use the `Gestalt` function with the `gestaltAppleEventsAttr` selector to determine whether the Apple Event Manager is available. In the `response` parameter, the bit defined by the constant `gestaltAppleEventsPresent` is set if the Apple Event Manager is available.

```
CONST gestaltAppleEventsAttr  = 'evnt';   {Gestalt selector}
     gestaltAppleEventsPresent = 0;       {if this bit is set, }
                                          { then the Apple Event }
                                          { Manager is available}
```

To find out which version of the Apple Event Manager is available, you can use the `AEManagerInfo` function; for more information, see page 4-104.

Accepting an Apple Event

To accept or send Apple events (or any other high-level events), you must set the appropriate flags in your application's `'SIZE'` resource and include code to handle high-level events in your application's main event loop.

Two flags in the `'SIZE'` resource determine whether an application receives high-level events:

n   The `isHighLevelEventAware` flag must be set for your application to receive any high-level events.

n   The `localAndRemoteHLEvents` flag must be set for your application to receive high-level events sent from another computer on the network.

Note that in order for your application to respond to Apple events sent from remote computers, the user of your application must also allow network users to link to your application. The user does this by selecting your application in the Finder, choosing Sharing from the File menu, and clicking the Allow Remote Program Linking checkbox. If the user has not yet started program linking, the Sharing command offers to display the Sharing Setup control panel so that the user can start program linking. The user must also authorize remote users for program linking by using the Users & Groups control panel. Program linking and setting up authenticated sessions are described in the chapter "Program-to-Program Communications Toolbox" in this book.

For a complete description of the `'SIZE'` resource, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

Apple events (and other high-level events) are identified by a message class of `kHighLevelEvent` in the `what` field of the event record. You can test the `what` field of the event record to determine whether the event is a high-level event.

Listing 4-1 is an example of a procedure called from an application's main event loop to handle events, including high-level events. The procedure determines the type of event received and then calls another routine to take the appropriate action.

Listing 4-1   A `DoEvent` procedure

```
PROCEDURE DoEvent (event: EventRecord);
BEGIN
   CASE event.what OF          {determine the type of event}
      mouseDown:
         DoMouseDown (event);
      .
      . {handle other kinds of events}
      .
      {handle high-level events, including Apple events}
      kHighLevelEvent: DoHighLevelEvent (event);
   END;
END;
```

Listing 4-2 is an example of a procedure that handles both Apple events and the high-level event identified by the event class `mySpecialHLEventClass` and the event ID `mySpecialHLEventID`. Note that, in most cases, you should use Apple events to communicate with other applications.

Listing 4-2   A `DoHighLevelEvent` procedure for handling Apple events and other high-level events

```
PROCEDURE DoHighLevelEvent (event: EventRecord);
VAR
   myErr: OSErr;
BEGIN
   IF (event.message = LongInt(mySpecialHLEventClass)) AND
      (LongInt(event.where) = LongInt(mySpecialHLEventID))
   THEN
      {it's a high-level event that doesn't use AEIMP}
      myErr := HandleMySpecialHLEvent(event)
   ELSE
      {otherwise, assume that the event is an Apple event}
      myErr := AEProcessAppleEvent(event);

   {check and handle error}
   IF myErr <> noErr THEN DoError(myErr);
END;
```

If your application accepts high-level events that do not follow the Apple Event Interprocess Messaging Protocol (AEIMP), you must dispatch these high-level events before calling `AEProcessAppleEvent`. To dispatch high-level events that do not follow AEIMP, you should check the event class, the event ID, or both for each event to see whether your application can handle the event.

After receiving a high-level event (and, if appropriate, checking whether it is a high-level event other than an Apple event), your application typically calls the `AEProcessAppleEvent` function. The `AEProcessAppleEvent` function determines the type of Apple event received, gets the event buffer that contains the parameters and attributes of the Apple event, and calls the corresponding Apple event handler in your application.

You should provide an Apple event handler for each Apple event that your application supports. This handler is responsible for performing the action requested by the Apple event and if necessary can return data in the reply Apple event.

If the client application requests a reply, the Apple Event Manager passes a default reply Apple event to your handler. If the client application does not request a reply, the Apple Event Manager passes a null descriptor record (a descriptor record of descriptor type `typeNull` and a data handle whose value is `NIL`) to your handler instead of a default reply Apple event.

After your handler finishes processing the Apple event and adds any parameters to the reply Apple event, it must return a result code to `AEProcessAppleEvent`. If the client application is waiting for a reply, the Apple Event Manager returns the reply Apple event to the client.

Installing Entries in the Apple Event Dispatch Tables

When your application receives an Apple event, use the `AEProcessAppleEvent` function to retrieve the data buffer of the event and to route the Apple event to the appropriate Apple event handler in your application. Your application supplies an Apple event dispatch table to map the Apple events your application supports to the Apple event handlers provided by your application.

To install entries in your application's Apple event dispatch table, use the `AEInstallEventHandler` function. You usually install entries for all of the Apple events that your application accepts into your application's Apple event dispatch table.

To install an Apple event handler in your Apple event dispatch table, you must specify

n   the event class of the Apple event

n   the event ID of the Apple event

n   the address of the Apple event handler for the Apple event

n   a reference constant

You provide this information to the `AEInstallEventHandler` function. In addition, you indicate whether the entry should be added to your application's Apple event dispatch table or to the system Apple event dispatch table.

The **system Apple event dispatch table** is a table in the system heap that contains system Apple event handlers—handlers that are available to all applications and processes running on the same computer. The handlers in your application's Apple event dispatch table are available only to your application. If `AEProcessAppleEvent` cannot find a handler for the Apple event in your application's Apple event dispatch table, it looks in the system Apple event dispatch table for a handler (see "How Apple Event Dispatching Works" on page 4-9 for details). If it doesn't find a handler for the event, it returns the `errAEEventNotHandled` result code.

If you add a handler to the system Apple event dispatch table, the handler should reside in the system heap. If there was already an entry in the system Apple event dispatch table for the same event class and event ID, it is replaced unless you chain it to your system handler. See "Creating and Managing the Apple Event Dispatch Tables" on page 4-61 for details.

Installing Entries for the Required Apple Events

Listing 4-3 illustrates how to add entries for the required Apple events to your application's Apple event dispatch table.

Listing 4-3   Adding entries for the required Apple events to an application's Apple event dispatch table

```
myErr := AEInstallEventHandler(kCoreEventClass,
                              kAEOpenApplication,
                              @MyHandleOApp, 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallEventHandler(kCoreEventClass,
                              kAEOpenDocuments,
                              @MyHandleODoc,0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallEventHandler(kCoreEventClass,
                              kAEPrintDocuments,
                              @MyHandlePDoc, 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallEventHandler(kCoreEventClass,
                              kAEQuitApplication,
                              @MyHandleQuit, 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

The code in Listing 4-3 creates entries for all four required Apple events in the Apple event dispatch table. (For examples of handlers that correspond to these entries, see "Handling the Required Apple Events," which begins on page 4-11.) The first entry creates an entry for the Open Application event. The entry indicates the event class and event ID of the Open Application event, supplies the address of the handler for that event, and specifies 0 as the reference constant.

The Apple Event Manager passes the reference constant to your handler each time your handler is called. Your application can use this reference constant for any purpose. If your application doesn't use the reference constant, use 0 as the value.

The last parameter to the `AEInstallEventHandler` function is a Boolean value that determines whether the entry is added to the system Apple event dispatch table or to your application's Apple event dispatch table. To add the entry to your application's Apple event dispatch table, use `FALSE` as the value of this parameter. If you specify `TRUE`, the entry is added to the system Apple event dispatch table. The code shown in Listing 4-3 adds entries to the application's Apple event dispatch table.

Installing Entries for Apple Events Sent by the Edition Manager

If your application supports the Edition Manager, you should also add entries to your application's Apple event dispatch table for the Apple events that your application receives from the Edition Manager. Listing 4-4 shows how to add these entries.

Listing 4-4   Adding entries for Apple events sent by the Edition Manager to an application's Apple event dispatch table

```
myErr := AEInstallEventHandler(sectionEventMsgClass,
                               sectionReadMsgID,
                               @MyHandleSectionReadEvent,
                               0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallEventHandler(sectionEventMsgClass,
                               sectionWriteMsgID,
                               @MyHandleSectionWriteEvent,
                               0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallEventHandler(sectionEventMsgClass,
                               sectionScrollMsgID,
                               @MyHandleSectionScrollEvent,
                               0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

See "Handling Apple Events Sent by the Edition Manager" on page 4-20 for the parameters associated with these events. See the chapter "Edition Manager" in this book for information on how your application should respond to the Apple events sent by the Edition Manager.

How Apple Event Dispatching Works

In addition to the Apple event handler dispatch tables, applications can add entries to a **special handler dispatch table** in either the application heap or the system heap. These dispatch tables are used for various specialized handlers; for more information, see "Creating and Managing the Special Handler Dispatch Tables," which begins on page 4-99.

When an application calls `AEProcessAppleEvent`, the function looks first in the application's special handler dispatch table for an entry that was installed with the constant `keyPreDispatch`. If the application's special handler dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the function looks in the application's Apple event dispatch table for an entry that matches the event class and event ID of the specified Apple event.

If the application's Apple event dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the `AEProcessAppleEvent` function looks in the system special handler dispatch table for an entry that was installed with the constant `keyPreDispatch`. If the system special handler dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the function looks in the system Apple event dispatch table for an entry that matches the event class and event ID of the specified Apple event.

If the system Apple event dispatch table does not include such a handler, the Apple Event Manager returns the result code `errAEEventNotHandled` to the server application and, if the client application is waiting for a reply, to the client application.

s **WARNING**

Before an application calls a system Apple event handler, system software has set up the A5 register for the calling application. For this reason, if you provide a system Apple event handler, it should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system handler may crash. s

For any entry in your Apple event dispatch table, you can specify a wildcard value for the event class, event ID, or both. You specify a wildcard by supplying the `typeWildCard` constant when installing an entry into the Apple event dispatch table. A wildcard value matches all possible values. Wildcards make it possible to supply one Apple event handler that dispatches several related Apple events.

For example, if you specify an entry with the `typeWildCard` event class and the `kAEOpenDocuments` event ID, the Apple Event Manager dispatches Apple events of any event class with an event ID of `kAEOpenDocuments` to the handler for that entry.

If you specify an entry with the `kCoreEventClass` event class and the `typeWildCard` event ID, the Apple Event Manager dispatches Apple events of the `kCoreEventClass` event class with any event ID to the handler for that entry.

If you specify an entry with the `typeWildCard` event class and the `typeWildCard` event ID, the Apple Event Manager dispatches all Apple events of any event class and any event ID to the handler for that entry.

If an Apple event dispatch table contains one entry for an event class and a specific event ID, and also contains another entry that is identical except that it specifies a wildcard value for either the event class or the event ID, the Apple Event Manager dispatches the more specific entry. For example, if an Apple event dispatch table includes one entry that specifies the event class as `kAECoreSuite` and the event ID as `kAEDelete`, and another entry that specifies the event class as `kAECoreSuite` and the event ID as `typeWildCard`, the Apple Event Manager will dispatch the Apple event handler associated with the entry that specifies the event ID as `kAEDelete`.

IMPORTANT

If your application sends Apple events to itself using a
`typeProcessSerialNumber` address descriptor record with the
`lowLongOfPSN` field set to `kCurrentProcess`, the Apple Event
Manager jumps directly to the appropriate Apple event handler without
going through the normal event-processing sequence. For this reason,
your application will not appear to run more slowly when it sends
Apple events to itself. For more information, see "Addressing an Apple
Event for Direct Dispatching" on page 5-13. s

Handling the Required Apple Events

This section describes the required Apple events—the Apple events your application
must support to be compatible with System 7 and later versions of system software—
and the descriptor types for all parameters of the required Apple events. It also describes
how to write the handlers for these events, and it provides sample code.

To support the required Apple events, you must set the necessary flags in the `'SIZE'`
resource of your application, install entries in your application's Apple event dispatch
table, add code to the event loop of your application to recognize high-level events,
and call the `AEProcessAppleEvent` function, as described in "Accepting an Apple
Event," which begins on page 4-5, and "Installing Entries in the Apple Event Dispatch
Tables," which begins on page 4-7. You must also write handlers for each Apple event;
this section describes how to write these handlers.

Required Apple Events

When a user opens or prints a file from the Finder, the Finder sets up the information
your application uses to determine which files to open or print. In System 7 and later
versions, if your application supports high-level events, the Finder communicates this
information to your application through the required Apple events.

The Finder sends these required Apple events to your application to request the
corresponding actions:

Apple event	Requested action
Open Application	Perform tasks your application normally performs when a user opens your application without opening or printing any documents
Open Documents	Open the specified documents
Print Documents	Print the specified documents
Quit Application	Perform tasks—such as releasing memory, requesting the user to save documents, and so on—associated with quitting before the Finder terminates your application

In System 7 and later versions, the Finder uses these events as part of the mechanisms for launching and terminating applications. When the Finder launches your application, the application receives the Open Application, Open Documents, or Print Documents event. When the Finder terminates your application, the application receives the Quit Application event. This method of communicating Finder information to your application replaces the mechanisms used in earlier versions of system software.

Applications that do not support high-level events can still use the `CountAppFiles`, `GetAppFiles`, and `ClrAppFiles` procedures (or the `GetAppParms` procedure) to get the Finder information. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for information on these routines. To make your application compatible with System 7 and with earlier and later versions, you must support both the old and new mechanisms.

Use the `Gestalt` function to determine whether the Apple Event Manager is present. If it is and the `isHighLevelEventAware` flag is set in your application's `'SIZE'` resource, your application receives the Finder information through the required Apple events.

If your application accepts high-level events, it must be able to process the four required Apple events. Your application receives the required Apple events from the Finder in these situations:

n   If your application is not open and the user opens your application from the Finder without opening or printing any documents, the Finder launches your application and sends it the Open Application event.

n   If your application is not open and the user opens one of your application's documents from the Finder, the Finder launches your application and sends it the Open Documents event.

n   If your application is not open and the user prints one of your application's documents from the Finder, the Finder launches your application and sends it the Print Documents event. Your application should print the selected documents and remain open until it receives a Quit Application event from the Finder.

n   If your application is open and the user opens or prints any of your application's documents from the Finder, the Finder sends your application the Open Documents or Print Documents event.

n   If your application is open and the user chooses Restart or Shut Down from the Finder's Special menu, the Finder sends your application the Quit Application event.

Upon receiving any of the required Apple events, your application should perform the action requested by the event. Here is a summary of the contents of the required events and the actions they request applications to perform:

Open Application—perform tasks associated with opening an application

Event class	`kCoreEventClass`
Event ID	`kAEOpenApplication`
Parameters	None
Requested action	Perform any tasks—such as opening an untitled document window—that you would normally perform when a user opens your application without opening or printing any documents.

Open Documents—open the specified documents

Event class	`kCoreEventClass`
Event ID	`kAEOpenDocuments`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeAEList`
Data:	A list of alias records for the documents to be opened
Requested action	Open the documents specified in the `keyDirectObject` parameter.

Print Documents—print the specified documents

Event class	`kCoreEventClass`
Event ID	`kAEPrintDocuments`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeAEList`
Data:	A list of alias records for the documents to be printed
Requested action	Print the documents specified in the `keyDirectObject` parameter without opening windows for the documents.

Quit Application—perform tasks associated with quitting

Event class	`kCoreEventClass`
Event ID	`kAEQuitApplication`
Parameters	None

Quit Application—perform tasks associated with quitting (continued)

Requested action

Perform any tasks that your application would normally perform when the user chooses Quit. Such tasks typically include asking the user whether to save documents that have been changed. When appropriate, the Finder sends this event to an application immediately after sending it a Print Documents event (unless the application was already open) or if the user chooses Restart or Shut Down from the Finder's Special menu.

Your application needs to recognize only two descriptor types to handle the required Apple events: descriptor lists and alias records. The Open Documents event and Print Documents event use descriptor lists to store a list of documents to open. Each document is specified as an alias record in the descriptor list.

You can retrieve the data that specifies the document to open as an alias record, or you can request that the Apple Event Manager coerce the alias record to a file system specification (`FSSpec`) record. The file system specification record provides a standard method of identifying files in System 7 and later versions. See *Inside Macintosh: Files* for a complete description of how to specify files using file system specification records.

Handling the Open Application Event

When the user opens your application, the Finder uses the Process Manager to launch your application. On startup, your application typically performs any needed initialization, and then begins to process events. If your application supports high-level events, and if the user opens your application without selecting any documents to open or print, your application receives the Open Application event.

To handle the Open Application event, your application should do just what the user expects it to do when it is opened. For example, your application might open a new untitled window in response to an Open Application event.

Listing 4-5 shows a handler that processes the Open Application event. This handler first calls an application-defined function called `MyGotRequiredParams`, which checks whether the Apple event contains any required parameters. If so, the handler returns an error, because by definition, the Open Application event should not contain any required parameters. Otherwise, the handler opens a new document window.

Listing 4-5   A handler for the Open Application event

```
FUNCTION MyHandleOApp (theAppleEvent, reply: AppleEvent;
                        handlerRefcon: LongInt): OSErr;
VAR
   myErr: OSErr;
BEGIN
   myErr := MyGotRequiredParams(theAppleEvent);
   IF myErr = noErr THEN
      DoNew;
   MyHandleOApp := myErr;
END;
```

For a description of the `MyGotRequiredParams` function, see Listing 4-11 on page 4-35.
For information about the `reply` and `handlerRefcon` parameters for an Apple event
handler, see "Writing Apple Event Handlers" on page 4-33.

Handling the Open Documents Event

To handle the Open Documents event, your application should open the documents that
the Open Documents event specifies in its direct parameter. Your application extracts this
information and then opens the specified documents. Listing 4-6 shows a handler for the
Open Documents event.

Listing 4-6   A handler for the Open Documents event

```
FUNCTION MyHandleODoc (theAppleEvent, reply: AppleEvent;
                        handlerRefcon: LongInt): OSErr;
VAR
   myFSS:              FSSpec;
   docList:            AEDescList;
   myErr, ignoreErr:   OSErr;
   index, itemsInList: LongInt;
   actualSize:         Size;
   keywd:              AEKeyword;
   returnedType:       DescType;
BEGIN
   {get the direct parameter--a descriptor list--and put it }
   { into docList}
   myErr := AEGetParamDesc(theAppleEvent, keyDirectObject,
                        typeAEList, docList);
```

```
    IF myErr = noErr THEN
    BEGIN
        {check for missing required parameters}
        myErr := MyGotRequiredParams(theAppleEvent);
        IF myErr = noErr THEN
        BEGIN
            {count the number of descriptor records in the list}
            myErr := AECountItems (docList, itemsInList);
            IF myErr = noErr THEN
                {now get each descriptor record from the list, }
                { coerce the returned data to an FSSpec record, and }
                { open the associated file}
                FOR index := 1 TO itemsInList DO
                BEGIN
                    myErr := AEGetNthPtr(docList, index, typeFSS,
                                         keywd, returnedType, @myFSS,
                                         Sizeof(myFSS),actualSize);
                    IF myErr = noErr THEN
                    BEGIN
                        myErr := MyOpenFile(@myFSS);
                        IF myErr <> noErr THEN
                            ; {handle error from MyOpenFile}
                    END
                    ELSE
                        ; {handle error from AEGetNthPtr}
                END; {of For index Do}
        END
        ELSE
            ; {handle error from MyGotRequiredParams}
        ignoreErr := AEDisposeDesc(docList);
    END
    ELSE
        ; {failed to get direct parameter, handle error}
    MyHandleODoc := myErr;
END;
```

The handler in Listing 4-6 first uses the AEGetParamDesc function to get the direct parameter (specified by the keyDirectObject keyword) out of the Apple event. The handler requests that AEGetParamDesc return a descriptor list in the docList variable. The handler then checks that it has retrieved all of the required parameters by calling the MyGotRequiredParams function. (See Listing 4-11 on page 4-35 for a description of this function.)

Once the handler has retrieved the descriptor list from the Apple event, it uses
AECountItems to count the number of descriptors in the list. Using the returned
number as an index, the handler can get the data of each descriptor record in the list.
This handler requests that the AEGetNthPtr function coerce the data in the descriptor
record to a file system specification record. The handler can then use the file system
specification record as a parameter to its own routine for opening files.

For more information on the AEGetParamDesc function, see page 4-69. For more
information on the AEGetNthPtr and AECountItems functions, see "Getting Data Out
of a Descriptor List" on page 4-31.

After extracting the file system specification record that describes the document to open,
your application can use this record to open the file. For example, in Listing 4-6, the code
passes the file system specification record to its routine for opening files, the
MyOpenFile function.

The MyOpenFile function should be designed so that it can be called in response to
both the Open Documents event and to events generated by the user. For example,
when the user chooses Open from the File menu, the code that handles the mouse-down
event uses the StandardGetFile procedure to let the user choose a file; it then
calls MyOpenFile, passing the file system specification record returned by
StandardGetFile. By isolating code that performs a requested action from code that
interacts with the user, you can easily adapt your application to handle Apple events
that request the same action.

Note the use of the AEDisposeDesc function to dispose of the descriptor list when your
handler no longer requires the data in it. Your handler should also return a result code.

Handling the Print Documents Event

To handle the Print Documents event, your application should extract information about
the documents to be printed from the direct parameter, then print the specified
documents.

If your application can interact with the user, it should open windows for the
documents, display a Print dialog box for the first document, and use the settings
entered by the user for the first document to print all the documents. If user interaction is
not allowed, your application may either return the error errAENoUserInteraction
or print the documents using default settings. See "Interacting With the User," which
begins on page 4-45, for information about using the AEInteractWithUser function to
interact with the user.

Note that your application can remain open after processing the Print Documents event;
when appropriate, the Finder sends your application a Quit Application event
immediately after sending it a Print Documents event.

The handler for the Print Documents event shown in Listing 4-7 is similar to the handler
for the Open Documents event, except that it prints the documents referred to in the
direct parameter.

Listing 4-7   A handler for the Print Documents event

```pascal
FUNCTION MyHandlePDoc (theAppleEvent, reply: AppleEvent;
                       handlerRefcon: LongInt): OSErr;
VAR
   myFSS:              FSSpec;
   docList:            AEDescList;
   myErr, ignoreErr:   OSErr;
   index, itemsInList: LongInt;
   actualSize:         Size;
   keywd:              AEKeyword;
   returnedType:       DescType;
BEGIN
   {get the direct parameter--a descriptor list--and put it }
   { into docList}
   myErr := AEGetParamDesc(theAppleEvent, keyDirectObject,
                           typeAEList, docList);
   IF myErr = noErr THEN
   BEGIN
      {check for missing required parameters}
      myErr := MyGotRequiredParams(theAppleEvent);
      IF myErr = noErr THEN
      BEGIN
         {count the number of descriptor records in the list}
         myErr := AECountItems (docList, itemsInList);
         IF myErr = noErr THEN
            {now get each descriptor record from the list, }
            { coerce the returned data to an FSSpec record, and }
            { print the associated file}
            FOR index := 1 TO itemsInList DO
            BEGIN
               myErr := AEGetNthPtr(docList, index, typeFSS,
                                    keywd, returnedType, @myFSS,
                                    Sizeof(myFSS), actualSize);
               IF myErr = noErr THEN
               BEGIN
                  myErr := MyPrintFile(@myFSS);
                  IF myErr <> noErr THEN
                     ; {handle error from MyOpenFile}
               END
               ELSE
                  ; {handle error from AEGetNthPtr}
            END; {of For index Do}
      END
```

```
      ELSE
         ; {handle error from MyGotRequiredParams}
      ignoreErr := AEDisposeDesc(docList);
   END
   ELSE
      ; {failed to get direct parameter, handle error}
   MyHandlePDoc := myErr;
END;
```

Handling the Quit Application Event

To handle the Quit Application event, your application should take any actions that are necessary before it is terminated (such as saving any open documents). Listing 4-8 shows an example of a handler for the Quit Application event.

When appropriate, the Finder sends your application a Quit Application event immediately after a Print Documents event. The Finder also sends your application a Quit Application event if the user chooses Restart or Shut Down from the Finder's Special menu.

Listing 4-8   A handler for the Quit Application event

```
FUNCTION MyHandleQuit (theAppleEvent, reply: AppleEvent;
                       handlerRefcon: LongInt): OSErr;
VAR
   myErr:        OSErr;
   userCanceled: Boolean;
BEGIN
   {check for missing required parameters}
   myErr := MyGotRequiredParams(theAppleEvent);
   IF myErr = noErr THEN
   BEGIN
      userCanceled := MyPrepareToTerminate;
      IF userCanceled THEN
         MyHandleQuit := kUserCanceled
      ELSE
         MyHandleQuit := noErr;
   END
   ELSE
      MyHandleQuit := myErr;
END;
```

The handler in Listing 4-8 calls another function supplied by the application, the `MyPrepareToTerminate` function. This function saves the documents for any open windows and returns a Boolean value that indicates whether the user canceled the Quit operation. This is another example of isolating code for interacting with the user from the code that performs the requested action. By structuring your application in this way, you can use the same routine to respond to a user action (such as choosing the Quit command from the File menu) or to the corresponding Apple event. (For a description of the `MyGotRequiredParams` function, see "Writing Apple Event Handlers" on page 4-33.)

IMPORTANT

When your application is ready to quit, it should call the `ExitToShell` procedure from the main event loop, not from your handler for the Quit Application event. Your application should quit only after the handler returns `noErr` as its function result. s

Handling Apple Events Sent by the Edition Manager

If your application provides publish and subscribe capabilities, it should handle the Apple events sent by the Edition Manager in addition to the required Apple events. Your application should also handle the Create Publisher event, which is described in the "Handling the Create Publisher Event" section on page 4-22.

The Edition Manager sends your application Apple events to communicate information about the publishers and subscribers in your application's documents. Specifically, the Edition Manager uses Apple events to notify your application

n   when the information in an edition is updated

n   when your application needs to write the data from a publisher to an edition

n   when your application should locate a particular publisher and scroll through the document to that location

The Section Read, Section Write, and Section Scroll Events

The following descriptions identify the three Apple events sent by the Edition Manager—
Section Read, Section Write, and Section Scroll—and the actions they tell applications to
perform.

Section Read—read information into the specified section

Event class	`SectionEventMsgClass`
Event ID	`SectionReadMsgID`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeSectionH`
Data:	A handle to the section record of the subscriber whose edition contains updated information
Requested action	Update the subscriber with the new information from the edition.

Section Write—write the specified section to an edition

Event class	`SectionEventMsgClass`
Event ID	`SectionWriteMsgID`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeSectionH`
Data:	A handle to the section record of the publisher
Requested action	Write the publisher's data to its edition.

Section Scroll—scroll through the document to the specified section

Event class	`SectionEventMsgClass`
Event ID	`SectionScrollMsgID`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeSectionH`
Data:	A handle to the section record of the publisher to scroll to
Requested action	Scroll through the document to the publisher identified by the specified section record.

See the chapter "Edition Manager" in this book for details on how your application
should respond to these events.

Handling the Create Publisher Event

If your application supports publish and subscribe capabilities, it should also handle the Create Publisher event.

Create Publisher—create a publisher

Event class	`kAEMiscStdSuite`
Event ID	`kAECreatePublisher`
Required parameter	None
Optional parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeObjectSpecifier`
Data:	An object specifier record that specifies the Apple event object or objects to publish. If this parameter is omitted, publish the current selection.
Optional parameter	
Keyword:	`keyAEEditionFileLoc`
Descriptor type:	`typeAlias`
Data:	An alias record that contains the location of the edition container to create. If this parameter is omitted, use the default edition container.
Requested action	Create a publisher for the specified data using the specified location for the edition container. If the data isn't specified, publish the current selection. If the location of the edition isn't specified, use the default location.

When your application receives the Create Publisher event, it should create a publisher and write the publisher's data to an edition. The data of the publisher, and the location and name of the edition, are defined by the Apple event. If the Create Publisher event includes a `keyDirectObject` parameter, then your application should publish the data contained in the parameter. If the `keyDirectObject` parameter is missing, then your application should publish the current selection. If the document doesn't have a current selection, your handler for the event should return a nonzero result code.

If the Create Publisher event includes a `keyAEEditionFileLoc` parameter, your application should use the location and name contained in the parameter as the default location and name of the edition. If the `keyAEEditionFileLoc` parameter is missing, your application should use the default location and name your application normally uses to specify the edition container.

Listing 4-9 shows a handler for the Create Publisher event. This handler checks for the `keyDirectObject` parameter and the `keyAEEditionFileLoc` parameter. If either of these is not specified, the handler uses default values. The handler uses the application-defined function `DoNewPublisher` to create the publisher and its edition, create a section record, and update other data structures associated with the document. See the chapter "Edition Manager" in this book for an example of the `DoNewPublisher` function.

Listing 4-9   A handler for the Create Publisher event

```
FUNCTION MyHandleCreatePublisherEvent (theAppleEvent,
                                        reply: AppleEvent;
                                        handlerRefcon: LongInt)
                                        : OSErr;
VAR
   myErr:                OSErr;
   returnedType:         DescType;
   thePublisherDataDesc: AEDesc;
   actualSize:           LongInt;
   promptForDialog:      Boolean;
   thisDocument:         MyDocumentInfoPtr;
   preview:              Handle;
   previewFormat:        FormatType;
   defaultLocation:      EditionContainerSpec;
BEGIN
   MyGetDocumentPtr(thisDocument);
   myErr := AEGetParamDesc(theAppleEvent, keyDirectObject,
                           typeObjectSpecifier,
                           thePublisherDataDesc);
   CASE myErr OF
      errAEDescNotFound:
      BEGIN
         {use the current selection as the publisher and set up }
         { info for later when DoNewPublisher displays preview}
         preview := MyGetPreviewForSelection(thisDocument);
         previewFormat := 'TEXT';
      END;
      noErr:
         {use the data in keyDirectObject parameter as the }
         { publisher (which is returned in the }
         { thePublisherDataDesc variable), and set up info for }
         { later when DoNewPublisher displays preview}
         MySetInfoForPreview(thePublisherDataDesc, thisDocument,
                             preview, previewFormat);
      OTHERWISE
      BEGIN
         MyHandleCreatePublisherEvent := myErr;
         Exit(MyHandleCreatePublisherEvent);
      END;
   END;
   myErr := AEDisposeDesc(thePublisherDataDesc);
```

```
    myErr := AEGetParamPtr(theAppleEvent, keyAEEditionFileLoc,
                         typeFSS, returnedType,
                         @defaultLocation.theFile,
                         SizeOf(FSSpec), actualSize);
  CASE myErr OF
     errAEDescNotFound:
        {use the default location as the edition container}
           myErr := MyGetDefaultEditionSpec(thisDocument,
                                            defaultLocation);
     noErr:
     BEGIN                 {the keyAEEditionFileLoc parameter }
                           { contains a default location}
        defaultLocation.thePart := kPartsNotUsed;
        defaultLocation.theFileScript := smSystemScript;
     END;
     OTHERWISE
     BEGIN
        MyHandleCreatePublisherEvent := myErr;
        Exit(MyHandleCreatePublisherEvent);
     END;
  END;
  myErr := MyGotRequiredParams(theAppleEvent);
  IF myErr <> noErr THEN
  BEGIN
     MyHandleCreatePublisherEvent := myErr;
     Exit(MyHandleCreatePublisherEvent);
  END;
  myErr := AEInteractWithUser(kAEDefaultTimeout, gMyNotifyRecPtr,
                              @MyIdleFunction);
  IF myErr = noErr THEN promptForDialog := TRUE
                   ELSE promptForDialog := FALSE;
  myErr := DoNewPublisher(thisDocument, promptForDialog,
                          preview, previewFormat,
                          defaultLocation);
  {add keyErrorNumber and keyErrorString parameters if desired}
END;
```

Note that the MyHandleCreatePublisherEvent handler in Listing 4-9 uses the
AEInteractWithUser function to determine whether user interaction is allowed.
If so, the handler sets the promptForDialog variable to TRUE, indicating that the
DoNewPublisher function should display the publisher dialog box. If not,

the handler sets the `promptForDialog` variable to `FALSE`, and the `DoNewPublisher` function does not prompt the user for the location or name of the edition. For more information about `AEInteractWithUser`, see "Interacting With the User," which begins on page 4-45.

Getting Data Out of an Apple Event

The Apple Event Manager stores the parameters and attributes of an Apple event in a format that is internal to the Apple Event Manager. You use Apple Event Manager functions to retrieve the data from an Apple event and return it to your application in a format your application can use.

Most of the functions that retrieve data from Apple event parameters and attributes are available in two forms: one that returns the desired data in a specified buffer and one that returns a descriptor record containing the same data. For example, the `AEGetParamPtr` function uses a specified buffer to return the data contained in an Apple event parameter, and the `AEGetParamDesc` function returns the descriptor record for a specified parameter.

You can also use Apple Event Manager functions to get data out of descriptor records, descriptor lists, and AE records. You use similar functions to put data into descriptor records, descriptor lists, and AE records.

When your handler receives an Apple event, you typically use the `AEGetParamPtr`, `AEGetAttributePtr`, `AEGetParamDesc`, or `AEGetAttributeDesc` function to get the data out of the Apple event.

Some Apple Event Manager functions let your application request that the data be returned using any descriptor type, even if it is different from the original descriptor type. If the original data is of a different descriptor type, the Apple Event Manager attempts to coerce the data to the requested descriptor type.

For example, the `AEGetParamPtr` function lets you specify the desired descriptor type of the resulting data as follows:

```
VAR
    theAppleEvent:    AppleEvent;
    returnedType:     DescType;
    multResult:       LongInt;
    actualSize:       Size;
    myErr:            OSErr;

myErr := AEGetParamPtr(theAppleEvent, keyMultResult,
                    typeLongInteger, returnedType,
                    @multResult, SizeOf(multResult),
                    actualSize);
```

In this example, the desired type is specified in the third parameter by the `typeLongInteger` descriptor type. This requests that the Apple Event Manager coerce the data to a long integer if it is not already of this type. To prevent coercion and ensure that the descriptor type of the result is of the same type as the original, specify `typeWildCard` for the third parameter.

The Apple Event Manager returns, in the `returnedType` parameter, the descriptor type of the resulting data. This is useful information when you specify `typeWildCard` as the desired descriptor type; you can determine the descriptor type of the resulting data by examining this parameter.

The Apple Event Manager can coerce many different types of data. For example, the Apple Event Manager can convert alias records to file system specification records, integers to Boolean data types, and characters to numeric data types, in addition to other data type conversions. For a complete list of the data types for which the Apple Event Manager provides coercion handling, see Table 4-1 on page 4-43.

To perform data coercions that the Apple Event Manager doesn't perform, you can provide your own coercion handlers. See "Writing and Installing Coercion Handlers," which begins on page 4-41, for information on providing your own coercion handlers.

Apple event parameters are keyword-specified descriptor records. You can use `AEGetParamDesc` to get the descriptor record of a parameter, or you can use `AEGetParamPtr` to get the data out of the descriptor record of a parameter. If an Apple event parameter consists of an object specifier record, you can use `AEResolve` and your own object accessor functions to resolve the object specifier record—that is, to locate the Apple event object it describes. For more information about `AEResolve` and object accessor functions, see "Writing Object Accessor Functions," which begins on page 6-28. Attributes are also keyword-specified descriptor records, and you can use similar routines to get the descriptor record of an attribute or to get the data out of an attribute.

The following sections show how to use the `AEGetParamPtr`, `AEGetAttributePtr`, `AEGetParamDesc`, or `AEGetAttributeDesc` function to get the data out of an Apple event.

Getting Data Out of an Apple Event Parameter

You can use the `AEGetParamPtr` or `AEGetParamDesc` function to get the data out of an Apple event parameter. Use the `AEGetParamPtr` function (or the `AEGetKeyPtr` function, which works the same way) to return the data contained in a parameter. Use the `AEGetParamDesc` function when you need to get the descriptor record of a parameter or to extract the descriptor list from a parameter.

For example, suppose you need to get the data out of a Section Read event. The Edition Manager sends your application a Section Read event to tell your application to read updated information from an edition into the specified subscriber. The direct parameter of the Apple event contains a handle to the section record of the subscriber. You can use the `AEGetParamPtr` function to get the data out of the Apple event.

You specify the Apple event that contains the desired parameter, the keyword of the desired parameter, the descriptor type the function should use to return the data, a buffer to store the data, and the size of this buffer as parameters to the `AEGetParamPtr` function. The `AEGetParamPtr` function returns the descriptor type of the resulting data and the actual size of the data, and it places the requested data in the specified buffer.

```
VAR
    sectionH:        SectionHandle;
    theAppleEvent:   AppleEvent;
    returnedType:    DescType;
    actualSize:      Size;
    myErr:           OSErr;

myErr := AEGetParamPtr(theAppleEvent, keyDirectObject,
                       typeSectionH, returnedType, @sectionH,
                       SizeOf(sectionH), actualSize);
```

In this example, the `keyDirectObject` keyword specifies that the `AEGetParamPtr` function should extract information from the direct parameter; `AEGetParamPtr` returns the data in the buffer specified by the `sectionH` variable.

You can request that the Apple Event Manager return the data using the descriptor type of the original data or you can request that the Apple Event Manager coerce the data into a descriptor type that is different from the original. To prevent coercion, specify the desired descriptor type as `typeWildCard`.

The `typeSectionH` descriptor type specifies that the returned data should be coerced to a handle to a section record. You can use the information returned in the `sectionH` variable to identify the subscriber and read in the information from the edition.

In this example, the `AEGetParamPtr` function returns, in the `returnedType` variable, the descriptor type of the resulting data. The descriptor type of the resulting data matches the requested descriptor type unless the Apple Event Manager wasn't able to coerce the data to the specified descriptor type or you specified the desired descriptor type as `typeWildCard`. If the coercion fails, the Apple Event Manager returns the `errAECoercionFail` result code.

The `AEGetParamPtr` function returns, in the `actualSize` variable, the actual size of the data (that is, the size of coerced data, if any coercion was performed). If the value returned in this variable is greater than the amount your application allocated for the buffer to hold the returned data, your application can increase the size of its buffer to this amount, and get the data again. You can also choose to use the `AEGetParamDesc` function when your application doesn't know the size of the data.

In general, use the `AEGetParamPtr` function to extract data that is of fixed length or known maximum length, and the `AEGetParamDesc` function to extract data that is of variable length. The `AEGetParamDesc` function returns the descriptor record for an Apple event parameter. This function is useful, for example, for extracting a descriptor list from a parameter.

You specify, as parameters to `AEGetParamDesc`, the Apple event that contains the desired parameter, the keyword of the desired parameter, the descriptor type the function should use to return the descriptor record, and a buffer to store the returned descriptor record. The `AEGetParamDesc` function returns the descriptor record using the specified descriptor type.

For example, the direct parameter of the Open Documents event contains a descriptor list that specifies the documents to open. You can use the `AEGetParamDesc` function to get the descriptor list out of the direct parameter.

```
VAR
    docList:          AEDescList;
    theAppleEvent:    AppleEvent;
    myErr:            OSErr;

myErr := AEGetParamDesc(theAppleEvent, keyDirectObject,
                        typeAEList, docList);
```

In this example, the Apple event specified by the variable `theAppleEvent` contains the desired parameter. The `keyDirectObject` keyword specifies that the `AEGetParamDesc` function should get the descriptor record of the direct parameter. The `typeAEList` descriptor type specifies that the descriptor record should be returned as a descriptor list. In this example, the `AEGetParamDesc` function returns a descriptor list in the `docList` variable.

The descriptor list contains a list of descriptor records. To get the descriptor records and their data out of a descriptor list, use the `AECountItems` function to find the number of descriptor records in the list and then make repetitive calls to the `AEGetNthPtr` function to get the data out of each descriptor record. See "Getting Data Out of a Descriptor List" on page 4-31 for more information.

Note that the `AEGetParamDesc` function copies the descriptor record from the parameter. When you're done with a descriptor record that you obtained from `AEGetParamDesc`, you must dispose of it by calling the `AEDisposeDesc` function.

If an Apple event parameter consists of an object specifier record, you can use `AEResolve` to resolve the object specifier record (that is, locate the Apple event object it describes), as explained in "Finding Apple Event Objects" on page 3-46.

Getting Data Out of an Attribute

You can use the `AEGetAttributePtr` or `AEGetAttributeDesc` function to get the data out of the attributes of an Apple event.

You specify, as parameters to AEGetAttributePtr, the Apple event that contains the desired attribute, the keyword of the desired attribute, the descriptor type the function should use to return the data, a buffer to store the data, and the size of this buffer. The AEGetAttributePtr function returns the descriptor type of the returned data and the actual size of the data and places the requested data in the specified buffer.

For example, this code gets the data out of the keyEventSourceAttr attribute of an Apple event.

```
VAR
    theAppleEvent:    AppleEvent;
    returnedType:     DescType;
    sourceOfAE:       Integer;
    actualSize:       Size;
    myErr:            OSErr;

myErr := AEGetAttributePtr(theAppleEvent, keyEventSourceAttr,
                            typeShortInteger, returnedType,
                            @sourceOfAE, SizeOf(sourceOfAE),
                            actualSize);
```

The keyEventSourceAttr keyword specifies the attribute from which to get the data. The typeShortInteger descriptor type specifies that the data should be returned as a short integer; the returnedType variable contains the actual descriptor type that is returned. You also must specify a buffer to hold the returned data and specify the size of this buffer. If the data is not already a short integer, the Apple Event Manager coerces it as necessary before returning it. The AEGetAttributePtr function returns, in the actualSize variable, the actual size of the returned data after coercion has taken place. You can check this value to make sure you got all the data.

As with the AEGetParamPtr function, you can request that AEGetAttributePtr return the data using the descriptor type of the original data, or you can request that the Apple Event Manager coerce the data into a descriptor type that is different from the original.

In this example, the AEGetAttributePtr function returns the requested data as a short integer in the sourceOfAE variable, and you can get information about the source of the Apple event by examining this value. You can test the returned value against the values defined by the data type AEEventSource.

```
TYPE AEEventSource = (kAEUnknownSource, kAEDirectCall,
                        kAESameProcess, kAELocalProcess,
                        kAERemoteProcess);
```

The constants defined by the data type `AEEventSource` have the following meanings:

Constant	Meaning
kAEUnknownSource	Source of Apple event unknown
kAEDirectCall	A direct call that bypassed the PPC Toolbox
kAESameProcess	Target application is also the source application
kAELocalProcess	Source application is another process on the same computer as the target application
kAERemoteProcess	Source application is a process on a remote computer on the network

The next example shows how to use the `AEGetAttributePtr` function to get data out of the `keyMissedKeywordAttr` attribute. After your handler extracts all known parameters from an Apple event, it should check whether the `keyMissedKeywordAttr` attribute exists. If it does, then your handler did not get all of the required parameters.

Note that if `AEGetAttributePtr` returns the `errAEDescNotFound` result code, then the `keyMissedKeywordAttr` attribute does not exist—that is, your application has extracted all of the required parameters. If `AEGetAttributePtr` returns `noErr`, then the `keyMissedKeywordAttr` attribute does exist—that is, your handler did not get all of the required parameters.

```
myErr := AEGetAttributePtr(theAppleEvent, keyMissedKeywordAttr,
                           typeWildCard, returnedType, NIL, 0,
                           actualSize);
```

The data in the `keyMissedKeywordAttr` attribute contains the keyword of the first required parameter, if any, that your handler didn't retrieve. If you want this data returned, specify a buffer to hold it and specify the buffer size. Otherwise, as in this example, specify `NIL` as the buffer and 0 as the size of the buffer.

This example shows how to use the `AEGetAttributePtr` function to get the address of the sender of an Apple event from the `keyAddressAttr` attribute of the Apple event:

```
VAR
   theAppleEvent: AppleEvent;
   returnedType:  DescType;
   addressOfAE:   TargetID;
   actualSize:    Size;
   myErr:         OSErr;

myErr := AEGetAttributePtr(theAppleEvent, keyAddressAttr,
                           typeTargetID, returnedType,
                           @addressOfAE, SizeOf(addressOfAE),
                           actualSize);
```

The keyAddressAttr keyword specifies the attribute to get the data from. The typeTargetID descriptor type specifies that the data should be returned as a target ID record; the returnedType variable contains the actual descriptor type that is returned. You can examine the address returned in the addressOfAE variable to determine the sender of the Apple event.

The target ID record returned in the addressOfAE variable contains the sender's port name, port location, and session reference number. To get the process serial number for a process on the local machine, pass the port name returned in the target ID record to the GetProcessSerialNumberFromPortName function. You can then pass the process serial number to the GetProcessInformation function to find the creator signature for a given process. (For more information about these functions, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.)

For more information about target addresses, see "Specifying a Target Address" on page 5-10.

Getting Data Out of a Descriptor List

You can use the AECountItems function to count the number of items in a descriptor list, and you can use AEGetNthDesc or AEGetNthPtr to get a descriptor record or its data out of a descriptor list.

The Open Documents event contains a direct parameter that specifies the list of documents to open. The list of documents is contained in a descriptor list. After extracting the descriptor list from the parameter, you can determine the number of items in the list and then extract each descriptor record from the descriptor list. See Figure 3-9 on page 3-19 for a depiction of the Open Documents event.

For example, when your handler receives an Open Documents event, you can use the AEGetParamDesc function to return the direct parameter as a descriptor list. You can then use AECountItems to return the number of descriptor records in the list.

```
VAR
   theAppleEvent:     AppleEvent;
   docList:           AEDescList;
   itemsInList:       LongInt;
   myErr:             OSErr;

myErr := AEGetParamDesc(theAppleEvent, keyDirectObject,
                        typeAEList, docList);
myErr := AECountItems(docList, itemsInList);
```

The AEGetParamDesc function returns, in the docList variable, a copy of the descriptor list from the direct parameter of the Open Documents event. You specify this list to the AECountItems function.

You specify the descriptor list whose items you want to count in the first parameter to
`AECountItems`. The Apple Event Manager returns, in the second parameter, the
number of items in the list. When extracting the descriptor records from a list, you often
use the number of items as a loop index. Here's an example:

```
FOR index := 1 TO itemsInList DO
   BEGIN
   {for each descriptor record in the list, get its data}
   END;
```

The format of the descriptor records in a descriptor list is private to the Apple Event
Manager. You must use the `AEGetNthPtr` or `AEGetNthDesc` function to extract
descriptor records from a descriptor list.

You specify the descriptor list that contains the desired descriptor records and an index
as parameters to the `AEGetNthPtr` function. The index represents a specific descriptor
record in the descriptor list. The `AEGetNthPtr` function returns the data for the
descriptor record represented by the specified index.

You also specify the descriptor type the function should use to return the data, a buffer
to store the data, and the size of this buffer. If the specified descriptor record exists, the
`AEGetNthPtr` function returns the keyword of the parameter, the descriptor type of the
returned data, and the actual size of the data, and it places the requested data in the
specified buffer.

Here's an example that uses the `AEGetNthPtr` function to extract an item from the
descriptor list in the direct parameter of the Open Documents event:

```
myErr := AEGetNthPtr(docList, index, typeFSS, keywd,
                     returnedType, @myFSS, Sizeof(myFSS),
                     actualSize);
```

The `docList` variable specifies the descriptor list from the direct parameter of the
Open Documents event. The `index` variable specifies the index of the descriptor record
to extract. You can use the `typeFSS` descriptor type, as in this example, to specify that
the data be returned as a file system specification record. The Apple Event Manager
automatically coerces the original data type of the descriptor record from an alias
record to a file system specification record. The `AEGetNthPtr` function returns the
keyword of the parameter and the descriptor type of the resulting data in the `keywd`
and `returnedType` variables, respectively.

You also specify a buffer to hold the desired data and the size (in bytes) of the buffer. In
this example, the `myFSS` variable specifies the buffer. The function returns the actual size
of the data in the `actualSize` variable. If this size is larger than the size of the buffer
you provided, you know that you didn't get all of the data for the descriptor record.

Listing 4-10 shows a more complete example of extracting the items from a descriptor
list in the Open Documents event.

Listing 4-10   Extracting items from a descriptor list

```
VAR
    index:          LongInt;
    itemsInList:    LongInt;
    docList:        AEDescList;
    keywd:          AEKeyword;
    returnedType:   DescType;
    myFSS:          FSSpec;
    actualSize:     Size;
    myErr:          OSErr;

FOR index := 1 TO itemsInList DO
    BEGIN
        myErr := AEGetNthPtr(docList, index, typeFSS, keywd,
                             returnedType, @myFSS, Sizeof(myFSS),
                             actualSize);
        IF myErr <> noErr THEN DoError(myErr);
        myErr := MyOpenFile(@myFSS);
        IF myErr <> noErr THEN DoError(myErr);
    END;
myErr := AEDisposeDesc(docList);
```

Writing Apple Event Handlers

For each Apple event your application supports, you must provide a function called
an Apple event handler. The `AEProcessAppleEvent` function calls one of your Apple
event handlers when it processes an Apple event. Your Apple event handlers should
perform any action requested by the Apple event, add parameters to the reply Apple
event if appropriate, and return a result code.

The Apple Event Manager uses dispatch tables to route Apple events to the appropriate
Apple event handler. You must supply an Apple event handler for each entry in your
application's Apple event dispatch table. Each handler must be a function that uses
this syntax:

```
FUNCTION MyEventHandler (theAppleEvent: AppleEvent;
                         reply: AppleEvent;
                         handlerRefcon: LongInt): OSErr;
```

The parameter `theAppleEvent` is the Apple event to handle. Your handler uses Apple
Event Manager functions to extract any parameters and attributes from the Apple event
and then performs the necessary processing. If any of the parameters include object
specifier records, your handler should call `AEResolve` to resolve them—that is, to locate
the Apple event objects they describe. For more information, see the chapter "Resolving
and Creating Object Specifier Records" in this book.

The `reply` parameter is the default reply provided by the Apple Event Manager. ("Replying to an Apple Event," which begins on page 4-36, describes how to add parameters to the default reply.) The `handlerRefcon` parameter is the reference constant stored in the Apple event dispatch table entry for the Apple event. Your handler can check the reference constant, if necessary, for information about the Apple event.

You can use the reference constant for anything you wish. For example, if you want to use the same handler for several Apple events, you can install entries for each event in your application's Apple event dispatch table that specify the same handler but different reference constants. Your handler can then use the reference constant to distinguish the different Apple events it handles.

To provide an Apple event handler in C, be sure to include the Pascal declaration before the handler declaration. This is the syntax for an Apple event handler in C:

```
pascal OSErr MyEventHandler (const AppleEvent *theAppleEvent,
                             const AppleEvent *reply,
                             long handlerRefcon);
```

After extracting all known parameters from the Apple event, every handler should determine whether the Apple event contains any further required parameters. Your handler can determine whether it retrieved all the required parameters by checking whether the `keyMissedKeywordAttr` attribute exists. If the attribute exists, then your handler has not retrieved all the required parameters and should immediately return an error. If the attribute does not exist, then the Apple event does not contain any more required parameters, although it may contain additional optional parameters.

The Apple Event Manager determines which parameters are optional according to the keywords listed in the `keyOptionalKeywordAttr` attribute. The source application is responsible for adding these keywords to the `keyOptionalKeywordAttr` attribute, but is not required to do so, even if that parameter is listed in the *Apple Event Registry: Standard Suites* as an optional parameter. If the source application does not add the necessary keyword to the `keyOptionalKeywordAttr` attribute, the target application treats the parameter as required for that Apple event. If the target application supports the parameter, it should handle the Apple event as the source application expects. If the target application does not support the parameter and checks whether it has received all the required parameters, it finds that there's another parameter that the client application considered required, and should return the result code `errAEParamMissed` without attempting to handle the event.

Listing 4-11 shows a function that checks for a `keyMissedKeywordAttr` attribute. A handler calls this function after getting all the required parameters it knows about from an Apple event.

Listing 4-11 A function that checks for a `keyMissedKeywordAttr` attribute

```
FUNCTION MyGotRequiredParams (theAppleEvent: AppleEvent): OSErr;
VAR
    myErr:          OSErr;
    returnedType:   DescType;
    actualSize:     Size;
BEGIN
    myErr := AEGetAttributePtr(theAppleEvent,
                                keyMissedKeywordAttr,
                                typeWildCard, returnedType,
                                NIL, 0, actualSize);
    IF myErr = errAEDescNotFound THEN
        {you got all the required parameters}
        MyGotRequiredParams := noErr
    ELSE IF myErr = noErr THEN
        {you missed a required parameter}
        MyGotRequiredParams := errAEParamMissed;
END;
```

The code in Listing 4-11 uses the `AEGetAttributePtr` function to get the
`keyMissedKeywordAttr` attribute. This attribute contains the first required parameter,
if any, that your handler didn't retrieve. If `AEGetAttributePtr` returns the
`errAEDescNotFound` result code, the Apple event doesn't contain a
`keyMissedKeywordAttr` attribute. If the Apple event doesn't contain this attribute,
then your handler has extracted all of the parameters that the client application
considered required.

If the `AEGetAttributePtr` function returns `noErr` as the result code, then the
attribute does exist, meaning that your handler has not extracted all of the required
parameters. In this case, your handler should return an error and not process the
Apple event.

The first remaining required parameter is specified by the data of the
`keyMissedKeywordAttr` attribute. If you want this data returned, specify a buffer to
hold the data. Otherwise, specify `NIL` as the buffer and 0 as the size of the buffer. If you
specify a buffer to hold the data, you can check the value of the `actualSize` parameter
to see if the data is larger than the buffer you allocated.

For more information about specifying Apple event parameters as optional or required,
see "Specifying Optional Parameters for an Apple Event" beginning on page 5-7.

Replying to an Apple Event

Your handler routine for a particular Apple event is responsible for performing the action requested by the Apple event, and can optionally return data in a reply Apple event. The Apple Event Manager passes a default reply Apple event to your handler. The default reply Apple event has no parameters when it is passed to your handler. Your handler can add parameters to the reply Apple event. If the client application requested a reply, the Apple Event Manager returns the reply Apple event to the client.

The reply Apple event is identified by the `kCoreEventClass` event class and by the `kAEAnswer` event ID. If the client application specified the `kAENoReply` flag in the `reply` parameter of the `AESend` function, the Apple Event Manager passes a null descriptor record (a descriptor record of type `typeNull` whose data handle has the value `NIL`) to your handler instead of a default reply Apple event. Your handler should check the descriptor type of the reply Apple event before attempting to add any attributes or parameters to it. An attempt to add an Apple event attribute or parameter to a null descriptor record generates an error.

If the client application requests a reply, the Apple Event Manager prepares a reply Apple event for the client by passing a default reply Apple event to your handler. The default reply Apple event has no parameters when it is passed to your handler. Your handler can add any parameters to the reply Apple event. If your application is a spelling checker, for example, you can return a list of misspelled words in a parameter.

When your handler finishes processing an Apple event, it returns a result code to `AEProcessAppleEvent`, which returns this result code as its function result. If your handler returns a nonzero result code, and if you have not added your own `keyErrorNumber` parameter, the Apple Event Manager also returns this result code to the client application by putting the result code into a `keyErrorNumber` parameter for the reply Apple event. The client can check for the existence of this parameter to determine whether the handler performed the requested action.

The client application specifies whether it wants a reply Apple event or not by specifying flags (represented by constants) in the `sendMode` parameter of the `AESend` function.

If the client specifies the `kAEWaitReply` flag in the `sendMode` parameter, the `AESend` function does not return until the timeout specified by the `timeoutInTicks` parameter expires or the server application returns a reply. When the server application returns a reply, the `reply` parameter to `AESend` contains the reply Apple event that your handler returned to the `AEProcessAppleEvent` function. When the client application no longer needs the original Apple event and the reply event, it must dispose of them, but the Apple Event Manager disposes of both the Apple event and the reply event for the server application when the server's handler returns to `AEProcessAppleEvent`.

If the client specified the `kAEQueueReply` flag, the client receives the reply event at a later time during its normal processing of other events.

Your handler should always set its function result to `noErr` if it successfully handles the Apple event. If an error occurs, your handler should return either `errAEEventNotHandled` or some other nonzero result code. If the error occurs because your application cannot understand the event, return `errAEEventNotHandled`. This allows the Apple Event Manager to look for a handler in the system special handler or system Apple event dispatch tables that might be able to handle the event. If the error occurs because the event is impossible to handle as specified, return the result code returned by whatever function caused the failure, or whatever other result code is appropriate.

For example, suppose your application receives a Get Data event requesting the name of the current printer, and your application cannot handle such an event. In this situation, you should return `errAEEventNotHandled` in case another handler available to the Apple Event Manager can handle the Get Data event. This strategy allows users to take advantage of system capabilities from within your application via system handlers.

However, if your application cannot handle a Get Data event that requests the fifth paragraph in a document because the document contains only four paragraphs, you should return some other nonzero error, because further attempts to handle the event are pointless.

If your Apple event handler calls the `AEResolve` function and `AEResolve` calls an object accessor function in the system object accessor dispatch table, your Apple event handler may not recognize the descriptor type of the token returned by the function. In this case, your handler should return the result code `errAEUnknownObjectType`. When your handler returns this result code, the Apple Event Manager attempts to locate a system Apple event handler that can recognize the token. For more information, see "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

The Apple Event Manager automatically adds any nonzero result code that your handler returns to a `keyErrorNumber` parameter in the reply Apple event. In addition to returning a result code, your handler can also return an error string in the `keyErrorString` parameter of the reply Apple event. Your handler should provide meaningful text in the `keyErrorString` parameter, so that the client can display this string to the user if desired.

Listing 4-12 shows how to add the `keyErrorString` parameter to the reply Apple event. See "Adding Parameters to an Apple Event" on page 5-5 for a description of the `AEPutParamPtr` function.

Listing 4-12   Adding the `keyErrorString` parameter to the reply Apple event

```
FUNCTION MyHandler (theAppleEvent: AppleEvent; reply: AppleEvent;
                    handlerRefcon: LongInt): OSErr;
VAR
   myErr:       OSErr;
   errStr:      Str255;
BEGIN
   {handle your Apple event here}

   {if an error occurs when handling an Apple event, set the }
   { function result and error string accordingly}
   IF myErr <> noErr THEN
   BEGIN
      MyHandler := myErr; {result code to be returned--the }
                          { Apple Event Manager adds this }
                          { result code to the reply Apple }
                          { event as the keyErrorNumber }
                          { parameter}
      IF (reply.dataHandle <> NIL) THEN
      {add error string parameter to the default reply}
      BEGIN
         {strings should normally be stored in resources}
         errStr := 'Why error occurred';
         myErr := AEPutParamPtr(reply, keyErrorString,
                                typeIntlText, @errStr[1],
                                length(errStr));
      END;
   END
   ELSE
      MyHandler := noErr;
END;
```

If your handler needs to return data to the client, it can add parameters to the reply Apple event. Listing 4-13 shows how a handler for the Multiply event (an imaginary Apple event that asks the server to multiply two numbers) might return the results of the multiplication to the client.

Listing 4-13   Adding parameters to the reply Apple event

```
FUNCTION MyMultHandler (theAppleEvent: AppleEvent;
                        reply: AppleEvent;
                        handlerRefcon: LongInt): OSErr;
VAR
   myErr:            OSErr;
   number1,number2:  LongInt;
   replyResult:      LongInt;
   actualSize:       Size;
   returnedType:     DescType;
BEGIN
   {get the numbers to multiply from the parameters of the }
   { Apple event; put the numbers in the number1 and number2 }
   { variables and then perform the requested multiplication}
   myErr := MyDoMultiply(theAppleEvent, number1,
                            number2, replyResult);
   IF myErr = noErr THEN
      IF (reply.dataHandle <> NIL) THEN
         {return result of the multiplication in the reply Apple }
         { event}
         myErr := AEPutParamPtr(reply, keyDirectObject,
                                   typeLongInteger, @replyResult,
                                   SizeOf(replyResult));
   MyMultHandler := myErr;
   {if an error occurs, set the error string }
   { accordingly, as shown in Listing 4-12}
END;
```

Disposing of Apple Event Data Structures

Whenever a client application uses Apple Event Manager functions to create a descriptor record, descriptor list, or Apple event record, the Apple Event Manager allocates memory for these data structures in the client's application heap. Likewise, when a server application extracts a descriptor record from an Apple event by using Apple Event Manager functions, the Apple Event Manager creates a copy of the descriptor record, including the data to which its handle refers, in the server's application heap.

Whenever you finish using a descriptor record or descriptor list that you have created or extracted from an Apple event, you should dispose of the descriptor record—and thereby deallocate the memory it uses—by calling the AEDisposeDesc function. If the descriptor record you pass to AEDisposeDesc (such as an Apple event record or an AE record) includes other nested descriptor records, one call to AEDisposeDesc will dispose of them all.

When a client application adds a descriptor record to an Apple event (for example, when it creates a descriptor record by calling AECreateDesc and then puts a copy of it into a parameter of an Apple event by calling AEPutParamDesc), it is still responsible for disposing of the original descriptor record. After a client application has finished using both the Apple event specified in the AESend function and the reply Apple event, it should dispose of their descriptor records by calling AEDisposeDesc. The client application should dispose of them even if AESend returns a nonzero result code.

The Apple event that a server application's handler receives is a copy of the original event created by the client application. When a server application's handler returns to AEProcessAppleEvent, the Apple Event Manager disposes of the server's copy (in the server's application heap) of both the Apple event and the reply event. The server application is responsible for disposing of any descriptor records created while extracting data from the Apple event or adding data to the reply event.

In general, outputs from Apple Event Manager functions are your application's responsibility. Once you finish using them, you should use AEDisposeDesc to dispose of any Apple event data structures created or returned by these functions:

AECoerceDesc	AEDuplicateDesc
AECoercePtr	AEGetAttributeDesc
AECreateAppleEvent	AEGetKeyDesc
AECreateDesc	AEGetNthDesc
AECreateList	AEGetParamDesc

If you attempt to dispose of descriptor records returned by successful calls to these functions without using AEDisposeDesc, your application may not be compatible with future versions of the Apple Event Manager. However, if any of these functions return a nonzero result code, they return a null descriptor record, which does not need to be disposed of.

Outputs from functions, such as AEGetKeyPtr, that use a buffer rather than a descriptor record to return data do not require the use of AEDisposeDesc. It is therefore preferable to use these functions for any data that is not identified by a handle.

Some of the functions described in the chapter "Resolving and Creating Object Specifier Records" in this book also create descriptor records. If you set the disposeInputs parameter to FALSE for any of the following functions, you should dispose of any Apple event data structures that they create or return:

CreateCompDescriptor	CreateObjSpecifier
CreateLogicalDescriptor	CreateRangeDescriptor

Your application is also responsible for disposing of some of the tokens it creates in the process of resolving an object specifier record. For information about token disposal, see "Defining Tokens" on page 6-39.

Writing and Installing Coercion Handlers

When your application extracts data from a parameter, it can request that the Apple Event Manager return the data using a descriptor type that is different from the original descriptor type. For example, when extracting data from the direct parameter of the Open Documents event, you can request that the alias records be returned as file system specification records. The Apple Event Manager can automatically coerce many different types of data from one to another. Table 4-1 on page 4-43 shows descriptor types and the kinds of coercion that the Apple Event Manager can perform.

You can also provide your own routines, referred to as **coercion handlers,** to coerce data into any other descriptor type. To install your own coercion handlers, use the `AEInstallCoercionHandler` function. You specify as parameters to this function

n   the descriptor type of the data coerced by the handler

n   the descriptor type of the resulting data

n   the address of the coercion handler for this descriptor type

n   a reference constant

n   a Boolean value that indicates whether your coercion handler expects the data to be specified as a descriptor record or as a pointer to the actual data

n   a Boolean value that indicates whether your coercion handler should be added to your application's coercion dispatch table or the system coercion dispatch table

The **system coercion dispatch table** is a table in the system heap that contains coercion handlers available to all applications and processes running on the same computer. The coercion handlers in your application's coercion dispatch table are available only to your application. When attempting to coerce data, the Apple Event Manager first looks for a coercion handler in your application's coercion dispatch table. If it cannot find a handler for the descriptor type, it looks in the system coercion dispatch table for a handler. If it doesn't find a handler there, it attempts to use the default coercion handling described by Table 4-1 on page 4-43. If it can't find an appropriate default coercion handler, it returns the `errAECoercionFail` result code.

Any handler that you add to the system coercion dispatch table should reside in the system heap. If there was already an entry in the system coercion dispatch table for the same descriptor type, it is replaced. Therefore, if there is an entry in the system coercion dispatch table for the same descriptor type, you should chain it to your system coercion handler as explained in "Creating and Managing the Coercion Handler Dispatch Tables," which begins on page 4-96.

s   **WARNING**
Before an application calls a system coercion handler, system software has set up the A5 register for the calling application. For this reason, if you provide a system coercion handler, it should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system coercion handler may crash. s

You can provide a coercion handler that expects to receive the data in a descriptor record or a buffer referred to by a pointer. When you install your coercion handler, you specify how your handler wishes to receive the data. Whenever possible, you should write your coercion handler so that it can accept a pointer to the data, because it's more efficient for the Apple Event Manager to provide your coercion handler with a pointer to the data.

A coercion handler that accepts a pointer to data must be a function with the following syntax:

```
FUNCTION MyCoercePtr (typeCode: DescType; dataPtr: Ptr;
                      dataSize: Size; toType: DescType;
                      handlerRefcon: LongInt;
                      VAR result: AEDesc): OSErr;
```

The `typeCode` parameter is the descriptor type of the original data. The `dataPtr` parameter is a pointer to the data to coerce; the `dataSize` parameter is the length, in bytes, of the data. The `toType` parameter is the desired descriptor type of the resulting data. The `handlerRefcon` parameter is a reference constant stored in the coercion table entry for the handler and passed to the handler by the Apple Event Manager whenever the handler is called. The `result` parameter is the descriptor record returned by your coercion handler.

Your coercion handler should coerce the data to the desired descriptor type and return the data in the descriptor record specified by the `result` parameter. If your handler successfully performs the coercion, it should return the `noErr` result code; otherwise, it should return a nonzero result code.

A coercion handler that accepts a descriptor record must be a function with the following syntax:

```
FUNCTION MyCoerceDesc (theAEDesc: AEDesc; toType: DescType;
                       handlerRefcon: LongInt;
                       VAR result: AEDesc): OSErr;
```

The parameter `theAEDesc` is the descriptor record that contains the data to be coerced. The `toType` parameter is the descriptor type of the resulting data. The `handlerRefcon` parameter is a reference constant stored in the coercion table entry for the handler and passed to the handler by the Apple Event Manager whenever the handler is called. The `result` parameter is the resulting descriptor record.

Your coercion handler should coerce the data in the descriptor record to the desired descriptor type and return the data in the descriptor record specified by the `result` parameter. Your handler should return an appropriate result code.

Note

To ensure that no coercion is performed and that the descriptor type of the result is of the same descriptor type as the original, specify `typeWildCard` for the desired type. u

Table 4-1 lists the descriptor types for which the Apple Event Manager provides coercion.

Table 4-1   Coercion handling provided by the Apple Event Manager

Original descriptor type of data to be coerced	Desired descriptor type	Description
typeChar	typeInteger typeLongInteger typeSMInt typeSMFloat typeShortInteger typeFloat typeLongFloat typeShortFloat typeExtended typeComp typeMagnitude	Any string that is a valid representation of a number can be coerced into an equivalent numeric value.
typeInteger typeLongInteger typeSMInt typeSMFloat typeShortInteger typeFloat typeLongFloat typeShortFloat typeExtended typeComp typeMagnitude	typeChar	Any numeric descriptor type can be coerced into the equivalent text string.
typeInteger typeLongInteger typeSMInt typeSMFloat typeShortInteger typeFloat typeLongFloat typeShortFloat typeExtended typeComp typeMagnitude	typeInteger typeLongInteger typeSMInt typeSMFloat typeShortInteger typeFloat typeLongFloat typeShortFloat typeExtended typeComp typeMagnitude	Any numeric descriptor type can be coerced into any other numeric descriptor type.
typeChar	typeType typeEnumerated typeKeyword typeProperty	Any four-character string can be coerced to one of these descriptor types.
typeEnumerated typeKeyword typeProperty typeType	typeChar	Any of these descriptor types can be coerced to the equivalent text string.

continued

Table 4-1 Coercion handling provided by the Apple Event Manager (continued)

Original descriptor type of data to be coerced	Desired descriptor type	Description
typeIntlText	typeChar	The result contains text only, without the script code or language code from the original descriptor record.
typeTrue	typeBoolean	The result is the Boolean value TRUE.
typeFalse	typeBoolean	The result is the Boolean value FALSE.
typeEnumerated	typeBoolean	The enumerated value 'true' becomes the Boolean value TRUE. The enumerated value 'fals' becomes the Boolean value FALSE.
typeBoolean	typeEnumerated	The Boolean value FALSE becomes the enumerated value 'fals'. The Boolean value TRUE becomes the enumerated value 'true'.
typeShortInteger typeSMInt	typeBoolean	A value of 1 becomes the Boolean value TRUE. A value of 0 becomes the Boolean value FALSE.
typeBoolean	typeShortInteger typeSMInt	A value of FALSE becomes 0. A value of TRUE becomes 1.
typeAlias	typeFSS	An alias record is coerced into a file system specification record.
typeAppleEvent	typeAppParameters	An Apple event is coerced into a list of application parameters for the LaunchParamBlockRec parameter block.
any descriptor type	typeAEList	A descriptor record is coerced into a descriptor list containing a single item.
typeAEList	*type of list item*	A descriptor list containing a single descriptor record is coerced into a descriptor record.

NOTE   Some of the descriptor types listed in this table are synonyms; for example, the constants typeSMInt and typeShortInteger have the same four-character code, 'shor'.

Interacting With the User

When your application receives an Apple event, it may need to interact with the user. For example, it may need to display a dialog box asking the user for additional information or confirmation. You must use the AEInteractWithUser function to make sure your application is in the foreground before it actually interacts with the user.

Both the client application and the server application specify their preferences for user interaction. The AEInteractWithUser function checks the user interaction preferences set by each application. If both the client and the server allow user interaction, AEInteractWithUser usually posts a notification request, and the Notification Manager brings the server to the foreground after the user responds to the notification request.

The AEInteractWithUser function can also bring the server application directly to the foreground, but only if the client application is the active application on the same computer and has set two flags in the sendMode parameter of the AESend function: the kAEWaitReply flag, which indicates that it is waiting for a reply, and the kAECanSwitchLayer flag, which indicates that it wants the server application to come directly to the foreground rather than posting a notification request.

To specify its preferences for how the server application should interact with the user, the client application sets various flags in the sendMode parameter to AESend. The Apple Event Manager sets the corresponding flags in the keyInteractLevelAttr attribute of the Apple event.

The server application sets its preferences with the AESetInteractionAllowed function. This function lets your application specify whether it allows interaction with the user as a result of receiving an Apple event from itself; from itself and other processes on the local computer; or from itself, local processes, and processes from another computer on the network.

Your application calls the AEInteractWithUser function before interacting with the user. If AEInteractWithUser returns the noErr result code, then your application is currently in the front and free to interact with the user. If AEInteractWithUser returns the errAENoUserInteraction result code, the conditions didn't allow user interaction and your application should not interact with the user.

The rest of this section explains how to set user interactions for the client and server applications and the practical effect these settings have when a server needs to interact with a user.

Setting the Client Application's User Interaction Preferences

The client application sets its user interaction preferences by setting flags in the `sendMode` parameter to the `AESend` function. The Apple Event Manager automatically adds the specified flags to the `keyInteractLevelAttr` attribute of the Apple event. These flags are represented by the following constants:

Flag	Description
`kAENeverInteract`	The server application should never interact with the user in response to the Apple event. If this flag is set, `AEInteractWithUser` returns the `errAENoUserInteraction` result code. This flag is the default when an Apple event is sent to a remote application.
`kAECanInteract`	The server application can interact with the user in response to the Apple event—by convention, if the user needs to supply information to the server. If this flag is set and the server allows interaction, `AEInteractWithUser` either brings the server application to the foreground or posts a notification request. This flag is the default when an Apple event is sent to a local application.
`kAEAlwaysInteract`	The server application can interact with the user in response to the Apple event—by convention, whenever the server application normally asks a user to confirm a decision or interact in any other way, even if no additional information is needed from the user. If this flag is set and the server allows interaction, `AEInteractWithUser` either brings the server application to the foreground or posts a notification request.

For example, suppose a client application sends a Set Data event to a database application to change a customer's address. The database application is configured to request user confirmation of changes to a customer's record. In this case the client sets the `kAECanInteract` flag before sending the event. Thus, the database application attempts to interact with the user if interaction is allowed. If interaction is not allowed, the database makes the correction anyway without consulting the user. However, if the client application sends a Delete event to delete a customer's record entirely and sets the `kAEAlwaysInteract` flag, the database application deletes the specified record only if it can interact with the user first and receives confirmation of the decision to delete a record. If interaction with the user is not allowed, the database application returns an error. By setting the `kAEAlwaysInteract` flag, the client application ensures that the entire record won't be lost if the user sends the Delete event by mistake.

If the client application doesn't specify any of the three user interaction flags, the Apple Event Manager sets either the `kAENeverInteract` or the `kAECanInteract` flag in the `keyInteractLevelAttr` attribute of the Apple event, depending on the location of the server application. If the server application is on a remote computer,

the Apple Event Manager sets the `kAENeverInteract` flag as the default. If the server application is on the local computer, the Apple Event Manager sets the `kAECanInteract` flag as the default.

In addition to the three user interaction flags, the client application can set another flag in the `sendMode` parameter to `AESend` to request that the Apple Event Manager immediately bring the server application directly to the foreground instead of posting a notification request:

Flag	Description
`kAECanSwitchLayer`	If both the client and server allow interaction, and if the client application is the active application on the local computer and is waiting for a reply (that is, it has set the `kAEWaitReply` flag), `AEInteractWithUser` brings the server directly to the foreground. Otherwise, `AEInteractWithUser` uses the Notification Manager to request that the user bring the server application to the foreground.

Note that although the `kAECanSwitchLayer` flag must be set for the Apple Event Manager to bring the server application directly to the foreground, setting it does not guarantee that the Apple Event Manager will bypass the notification request if user interaction is permitted. Another flag, the `kAEWaitReply` flag, must also be set in the `sendMode` parameter, and the client application must provide an idle function.

The `kAEWaitReply` flag is one of three flags in the `sendMode` parameter that a client application can set to specify whether and how the client should wait for a reply. (For a description of these flags, see "Sending an Apple Event and Handling the Reply" on page 3-30.) If the client application is not waiting for a reply, the user may have continued with other work. An application switch at this point might be unexpected and would thus violate the principle of user control as described in *Macintosh Human Interface Guidelines.*

If the client application sets the `kAEWaitReply` flag, it should also provide an idle function when it calls `AESend` so that it can handle events such as update events that it receives while waiting for the reply. Idle functions are described in "Writing an Idle Function," which begins on page 5-21.

When a server application calls `AEInteractWithUser`, the function first checks whether the `kAENeverInteract` flag in the `keyInteractLevelAttr` attribute of the Apple event is set. (The Apple Event Manager sets this attribute according to the flags specified in the `sendMode` parameter of `AESend`.) If the `kAENeverInteract` flag is set, `AEInteractWithUser` immediately returns the `errAENoUserInteraction` result code. If the client specified `kAECanInteract` or `kAEAlwaysInteract`, `AEInteractWithUser` checks the server's preferences for user interaction.

Setting the Server Application's User Interaction Preferences

The server sets its user interaction preferences by using the
`AESetInteractionAllowed` function. This function specifies the conditions
under which your application is willing to interact with the user.

```
myErr := AESetInteractionAllowed(level);
```

The `level` parameter is of type `AEInteractAllowed`.

```
TYPE  AEInteractAllowed = (kAEInteractWithSelf,
                           kAEInteractWithLocal,
                           kAEInteractWithAll);
```

You can specify one of these values for the interaction level:

Flag	Description
`kAEInteractWithSelf`	Your server application can interact with the user in response to an Apple event only when your application is also the client application—that is, only when your application is sending the Apple event to itself.
`kAEInteractWithLocal`	Your server application can interact with the user in response to an Apple event only if the client application is on the same computer as your application. This is the default if the server application does not call the function `AESetInteractionAllowed`.
`kAEInteractWithAll`	Your server application can interact with the user in response to an Apple event sent by any client application on any computer.

If the server application does not set the user interaction level, `AEInteractWithUser`
uses `kAEInteractWithLocal` as the value.

If the application sends itself an Apple event (that is, if the application is both the client
and the server) without setting the `kAENeverInteract` flag, `AEInteractWithUser`
always allows user interaction. If the client application is a process on the local computer
and specifies `kAECanInteract` or `kAEAlwaysInteract`, and if the server has set the
interaction level to `kAEInteractWithLocal` or `kAEInteractWithAll`, then
`AEInteractWithUser` allows user interaction. If the client is a process on a remote
computer on the network and specifies `kAECanInteract` or `kAEAlwaysInteract`,
`AEInteractWithUser` allows user interaction only if the server specified
the `kAEInteractWithAll` flag for the interaction level. In all other cases,
`AEInteractWithUser` does not allow user interaction.

Requesting User Interaction

If your server application needs to interact with the user for any reason, it must call the
`AEInteractWithUser` function to make sure it is in the foreground before it actually
interacts with the user. When `AEInteractWithUser` allows user interaction (based on
the client's and server's preferences), `AEInteractWithUser` brings the server
application to the foreground—either directly or after the user responds to a notification
request—and then returns a `noErr` result code. If `AEInteractWithUser` brings the
server to the foreground directly, the client returns to the foreground immediately after
the server has finished interacting with the user. If `AEInteractWithUser` brings the
server to the foreground after the user responds to a notification request, the server
remains in the foreground after completing the user interaction.

The `AEInteractWithUser` function specifies how long your handler is willing to wait
for a response from the user. For example, if the timeout value is 900 ticks (15 seconds)
and the Apple Event Manager posts a notification request, the Notification Manager
begins to display a blinking icon in the upper-right corner of the screen, then removes
the notification request (and the blinking icon) if the user does not respond within
15 seconds. (The discussion that follows describes some restrictions on the icons that can
be displayed in this situation.)

Note that the timeout value passed to the `AEInteractWithUser` function is separate
from the timeout value passed to the `AESend` function, which specifies how long the
client application is willing to wait for the reply or return receipt from the server
application. If `AEInteractWithUser` does not receive a response from the user within
the specified time, `AEInteractWithUser` returns `errAETimeout`.

You may want to give the user a method of setting the interaction level. For example,
some users may not want to be interrupted while background processing of an Apple
event occurs, or they may not want to respond to dialog boxes when your application is
handling Apple events sent from another computer.

Listing 4-14 illustrates the use of the `AEInteractWithUser` function. You call this function before your application displays a dialog box or otherwise interacts with the user when processing an Apple event. You specify a timeout value, a pointer to a Notification Manager record, and the address of an idle function as parameters to `AEInteractWithUser`.

Listing 4-14   Using the `AEInteractWithUser` function

```
myErr := AEInteractWithUser(kAEDefaultTimeout, gMyNotifyRecPtr,
                            @MyIdleFunction);
IF myErr <> noErr THEN
    {the attempt to interact failed; do any error handling}
    DoError(myErr)
ELSE
    {interact with the user by displaying a dialog box }
    { or by interacting in any other way that is necessary}
    DisplayMyDialogBox;
```

You can set a timeout value, in ticks, in the first parameter to `AEInteractWithUser`. Use the `kAEDefaultTimeout` constant if you want the Apple Event Manager to use a default value for the timeout value. The Apple Event Manager uses a timeout value of about one minute if you specify this constant. You can also specify the `kNoTimeOut` constant if your application is willing to wait an indefinite amount of time for a response from the user. Usually you should provide a timeout value, so that your application can complete processing of the Apple event in a reasonable amount of time.

If you specify `NIL` instead of a Notification Manager record in the second parameter of `AEInteractWithUser`, the Apple Event Manager looks for an application icon with the ID specified by the application's bundle (`'BNDL'`) resource and the application's file reference (`'FREF'`) resource. The Apple Event Manager first looks for an `'SICN'` resource with the specified ID; if it can't find an `'SICN'` resource, it looks for the `'ICN#'` resource and compresses the icon to fit in the menu bar. The Apple Event Manager won't look for any members of an icon family other than the icon specified in the `'ICN#'` resource.

If the application doesn't have `'SICN'` or `'ICN#'` resources, or if it doesn't have a file reference resource, the Apple Event Manager passes `NIL` to the Notification Manager, and no icon appears in the upper-right corner of the screen. Therefore, if you want to display any icon other than those of type `'SICN'` or `'ICN#'`, you must specify a notification record as the second parameter to the `AEInteractWithUser` function.

Note

If you want the Notification Manager to use a color icon when it posts a notification request, you should provide a Notification Manager record that specifies a `'cicn'` resource. u

The `AEInteractWithUser` function posts a notification request only when user interaction is allowed and the `kAECanSwitchLayer` flag in the `keyInteractLevelAttr` attribute is not set.

The last parameter to `AEInteractWithUser` specifies an idle function provided by your application. Your idle function should handle any update events, null events, operating-system events, or activate events while your application is waiting to be brought to the front. See "Writing an Idle Function" on page 5-21 for more information.

Figure 4-1 illustrates a situation in which a client application (a forms application) might request a service from a server application (a database application). To perform this service, the server application must interact with the user.

Figure 4-1   A document with a button that triggers a Get Data event
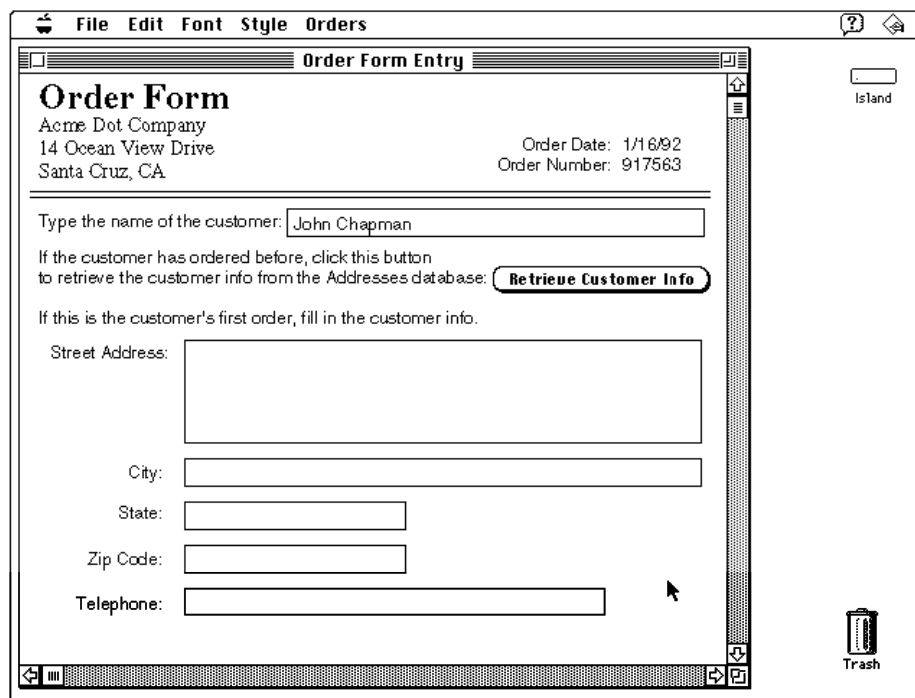
Figure 4-1 shows part of an electronic form used to enter information about an order received by telephone. If the customer has ordered from the company before, the user can quickly retrieve the customer's address and telephone number by clicking the Retrieve Customer Info button. In response, the forms application sends a Get Data event to a database application (SurfDB) currently open on the same computer. The Get Data event sent by the forms application (the client application for the ensuing transaction) asks SurfDB (the server) to locate the customer's name in a table of addresses and return the customer's address. When the forms application receives the reply Apple event, it can add the address data to the appropriate fields in the order form.

If SurfDB, as the server application, locates more than one entry for the specified customer name, it needs to interact with the user to determine which data to return in the reply Apple event. To interact with the user, the server application must be in the foreground, so that it can display a dialog box like the one shown in Figure 4-2.

Figure 4-2   A server application displaying a dialog box that requests information from the user
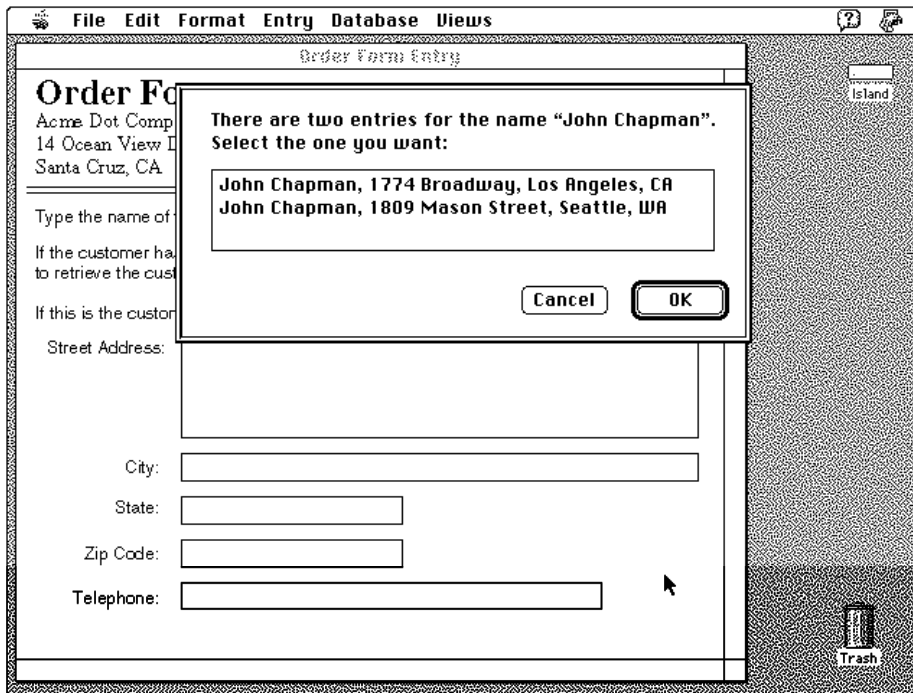
Figure 4-3, Figure 4-4, and Figure 4-5 illustrate two methods of dealing with this situation. Figure 4-3 shows the behavior of the server application that is common to both methods. In both cases, the server uses AESetInteractionAllowed to set its own interaction level to kAEInteractWithLocal. After calling AEResolve to locate the requested data, the server application discovers that two addresses match the name the user typed into the electronic form. The server then calls AEInteractWithUser with a timeout value of kAEDefaultTimeout so it can find out which address the user wants.

Figure 4-3 Handling user interaction
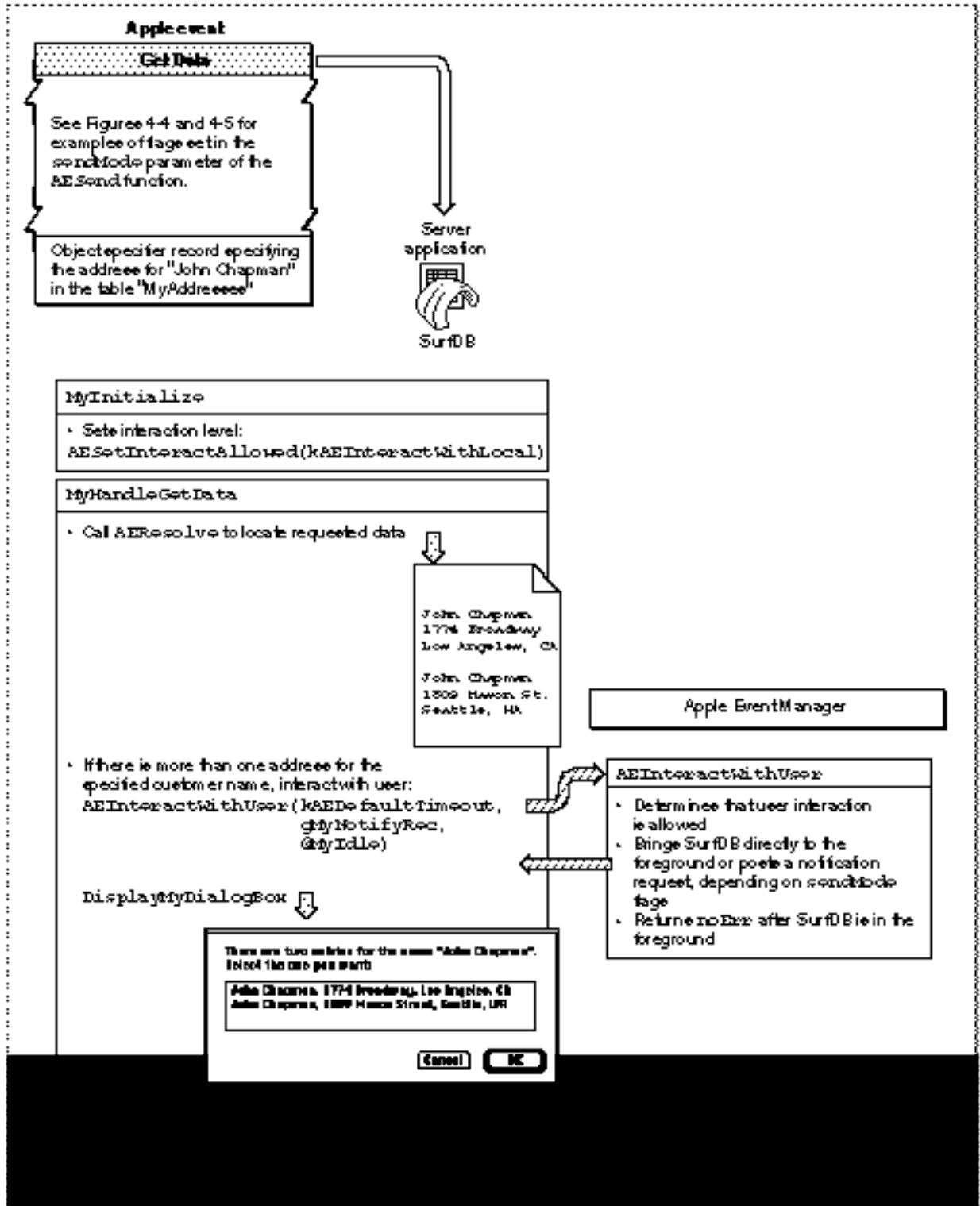
Figure 4-4 shows the circumstances in which the server application's call to
AEInteractWithUser shown in Figure 4-3 will cause the Apple Event Manager to
bring the server application directly to the foreground. The client application sets the
kAECanInteract, kAECanSwitchLayer, and kAEWaitReply flags in the sendMode
parameter of the AESend function when it sends the Get Data event shown in the figure.
These flags indicate that the client application expects the user to wait until the address
appears in the appropriate fields of the electronic form before continuing with any other
work. In this case, an automatic layer switch will not surprise the user and will avoid
the additional user action required to respond to a notification request, so
AEInteractWithUser brings the server application directly to the foreground
and returns a noErr result code. The server application then displays the dialog box
requesting that the user select the desired customer.

After the user selects the desired customer and clicks OK, the server application's
Get Data event handler returns. The Apple Event Manager immediately brings the client
application to the foreground, and the client application displays the requested customer
information in the appropriate fields.

Figure 4-4   Handling user interaction with the kAEWaitReply flag set
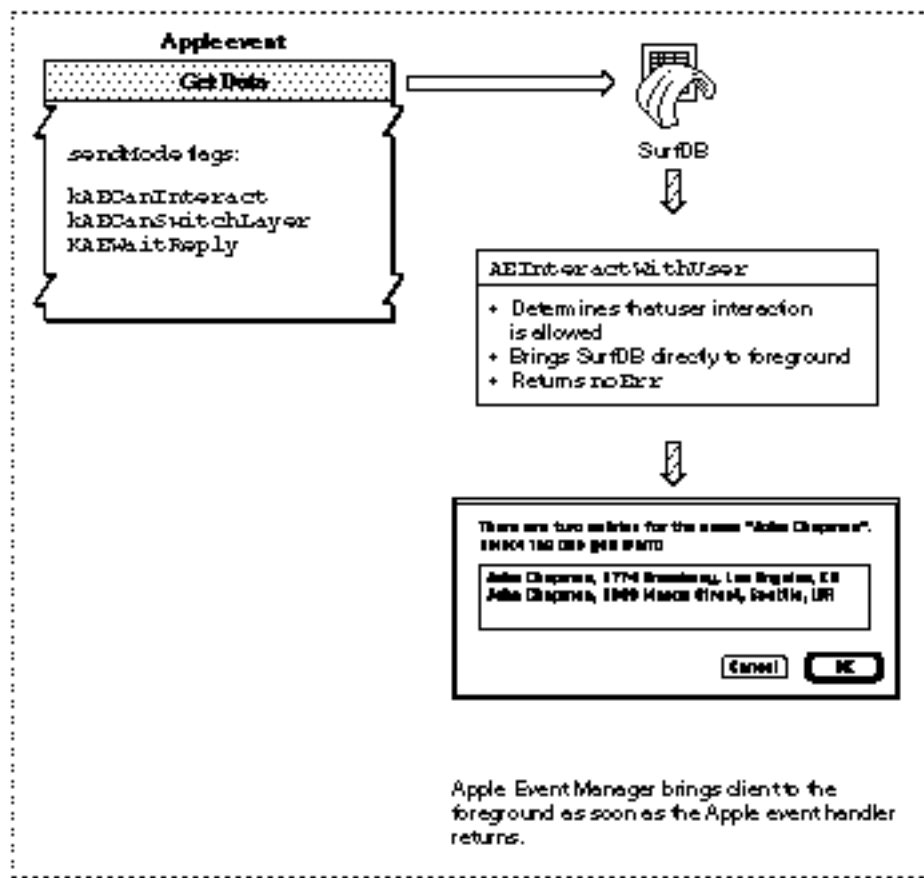
Figure 4-5 shows the circumstances in which the server application's call to
`AEInteractWithUser` in Figure 4-3 will cause the Apple Event Manager to post a
notification request rather than bringing the server application directly to the foreground.

Figure 4-5   Handling user interaction with the `kAEQueueReply` flag set
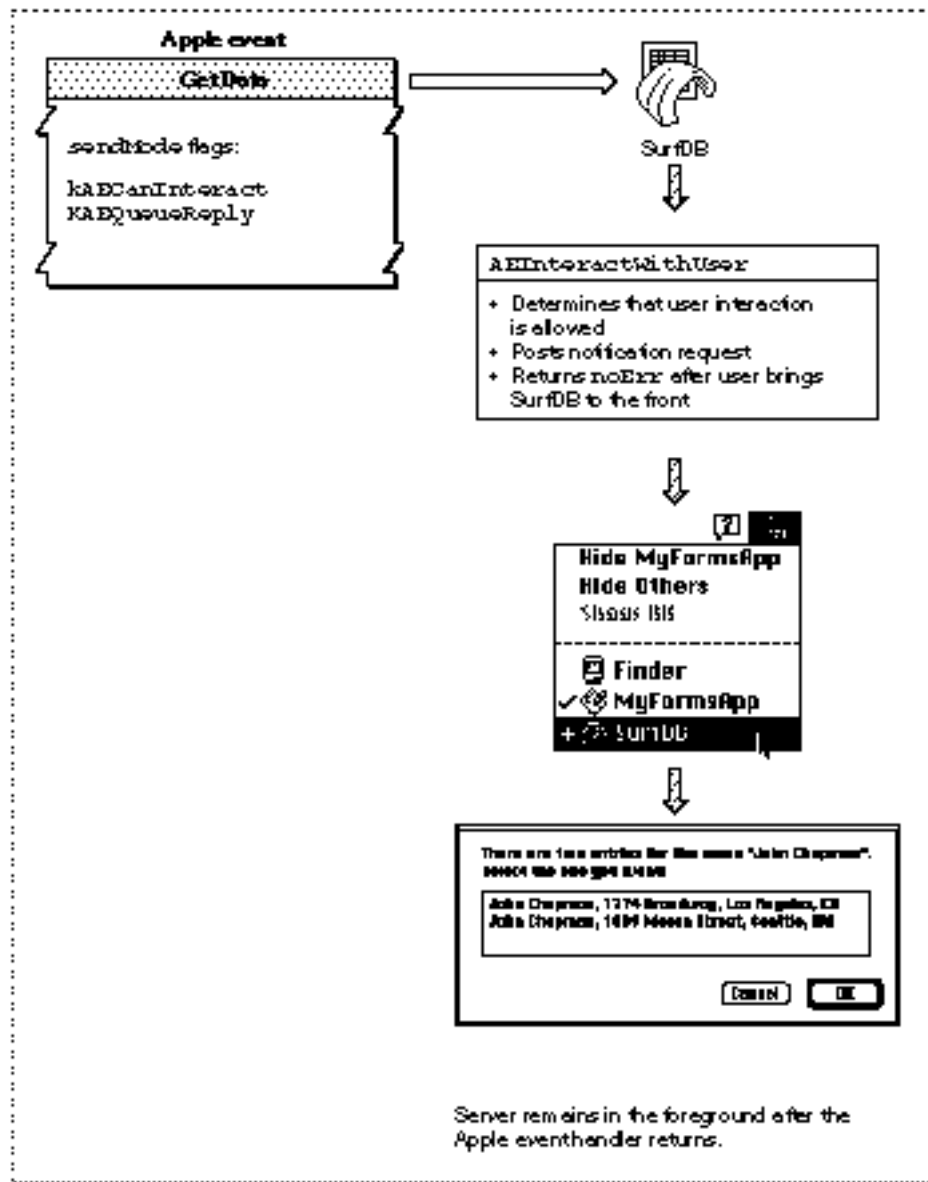
The only difference between the Get Data event shown in Figure 4-4 and the Get Data event shown in Figure 4-5 is that the client application has set the `kAEQueueReply` flag instead of the `kAEWaitReply` flag in the `sendMode` parameter of `AESend` and has not set the `kAECanSwitchLayer` flag. This combination of flags indicates that the client application expects the user to continue filling in other parts of the form, such as the items being ordered; the address will just appear after a while, provided there is no duplicate name. In this case, an automatic layer switch would disrupt the user's work. Instead of bringing the server application directly to the foreground, `AEInteractWithUser` uses the Notification Manager to post a notification request.

After the user has responded to the request and has brought the server application to the foreground, `AEInteractWithUser` returns a `noErr` result code, and the server application displays the dialog box requesting that the user select the desired customer. When the user selects a customer and clicks OK, the server application's Get Data event handler returns. Because the user brought the server to the foreground manually, the server remains in the foreground after the handler returns.

Reference to Responding to Apple Events

This section describes the basic Apple Event Manager data structures and routines that your application can use to respond to Apple events. It also describes the syntax for application-defined Apple event handlers and coercion handlers that your application can provide for use by the Apple Event Manager.

For information about routines used to create and send Apple events, see the chapter "Creating and Sending Apple Events" in this book. For information about routines and data structures used with object specifier records, see the chapter "Resolving and Creating Object Specifier Records" in this book.

Data Structures Used by the Apple Event Manager

This section summarizes the major data structures used by the Apple Event Manager. For an overview of the relationships among these data structures, see "Data Structures Within Apple Events," which begins on page 3-12.

Descriptor Records and Related Data Structures

Descriptor records are the fundamental data structures from which Apple events are constructed. A descriptor record is a data structure of type `AEDesc`.

```
TYPE AEDesc =
   RECORD                              {descriptor record}
      descriptorType:   DescType;    {type of data being passed}
      dataHandle:       Handle;      {handle to data being passed}
   END;
```

Field descriptions

descriptorType

A four-character string of type DescType that indicates the type of data being passed.

dataHandle   A handle to the data being passed.

The descriptor type is a structure of type DescType, which in turn is of data type ResType—that is, a four-character code. Constants, rather than these four-character codes, are usually used to refer to descriptor types. Table 4-2 lists the constants for the basic descriptor types used by the Apple Event Manager.

Table 4-2   Descriptor types used by the Apple Event Manager (excluding those used with object specifier records)

Descriptor type	Value	Description
typeBoolean	'bool'	Boolean value
typeChar	'TEXT'	Unterminated string
typeLongInteger	'long'	32-bit integer
typeInteger	'long'	32-bit integer
typeShortInteger	'shor'	16-bit integer
typeSMInt	'shor'	16-bit integer
typeLongFloat	'doub'	SANE double
typeFloat	'doub'	SANE double
typeShortFloat	'sing'	SANE single
typeSMFloat	'sing'	SANE single
typeExtended	'exte'	SANE extended
typeComp	'comp'	SANE comp
typeMagnitude	'magn'	Unsigned 32-bit integer
typeAEList	'list'	List of descriptor records
typeAERecord	'reco'	List of keyword-specified descriptor records
typeAppleEvent	'aevt'	Apple event record
typeTrue	'true'	TRUE Boolean value

Table 4-2   Descriptor types used by the Apple Event Manager (excluding those used with object specifier records) (continued)

Descriptor type	Value	Description
typeFalse	'fals'	FALSE Boolean value
typeAlias	'alis'	Alias record
typeEnumerated	'enum'	Enumerated data
typeType	'type'	Four-character code for event class or event ID
typeAppParameters	'appa'	Process Manager launch parameters
typeProperty	'prop'	Apple event property
typeFSS	'fss '	File system specification
typeKeyword	'keyw'	Apple event keyword
typeSectionH	'sect'	Handle to a section record
typeWildCard	'****'	Matches any type
typeApplSignature	'sign'	Application signature
typeSessionID	'ssid'	Session reference number
typeTargetID	'targ'	Target ID record
typeProcessSerialNumber	'psn '	Process serial number
typeNull	'null'	Nonexistent data (data handle is NIL)

For information about descriptor records and descriptor types used with object specifier records, see the chapter "Resolving and Creating Object Specifier Records" in this book.

Apple event attributes, Apple event parameters, object specifier records, tokens, and most of the other data structures used by the Apple Event Manager are constructed from one or more descriptor records. The Apple Event Manager identifies the various parts of an Apple event by means of keywords associated with the corresponding descriptor records. The AEKeyword data type is defined as a four-character code.

```
TYPE  AEKeyword = PACKED ARRAY[1..4] OF Char;
```

Constants are typically used for keywords. A keyword combined with a descriptor record forms a keyword-specified descriptor record, which is defined by a data structure of type AEKeyDesc.

```
TYPE   AEKeyDesc =
    RECORD
        descKey:       AEKeyword;      {keyword}
        descContent:   AEDesc;         {descriptor record}
    END;
```

Field descriptions

descKey   A four-character code of type AEKeyword that identifies the data in
                   the descContent field.

descContent   A descriptor record of type AEDesc.

Every Apple event includes an attribute that contains the address of the target
application. A descriptor record that contains an application's address is called an
address descriptor record.

```
TYPE   AEAddressDesc = AEDesc;       {address descriptor record}
```

Many Apple Event Manager functions take or return lists of descriptor records in a
special descriptor record called a descriptor list. A descriptor list is a structure of data
type AEDescList whose data consists of a list of other descriptor records.

```
TYPE   AEDescList = AEDesc;          {list of descriptor records}
```

Other Apple Event Manager functions take or return lists of keyword-specified
descriptor records in the form of an AE record. An AE record is a structure of data type
AERecord whose data handle refers to a list of keyword-specified descriptor records.

```
TYPE   AERecord = AEDescList;        {list of keyword-specified }
                                     { descriptor records}
```

The handle for a descriptor list of data type AERecord refers to a list of
keyword-specified descriptor records that specify Apple event parameters; they cannot
specify Apple event attributes.

Finally, a full-fledged Apple event, including both attributes and parameters, is an
Apple event record, which is a structure of data type AppleEvent.

```
TYPE   AppleEvent = AERecord;        {list of attributes and }
                                     { parameters for an Apple }
                                     { event}
```

The event class and event ID of an Apple event are specified in Apple Event Manager
routines by structures of data types AEEventClass and AEEventID, respectively.

```
TYPE AEEventClass = PACKED ARRAY[1..4] OF Char;

TYPE AEEventID = PACKED ARRAY[1..4] OF Char;
```

For more information about descriptor records and the other data structures described in
this section, see "Data Structures Within Apple Events," which begins on page 3-12.

With the exception of array data records, which are described in the next section, the
other Apple Event Manager data structures used in responding to Apple events are
described in "Routines for Responding to Apple Events," beginning on page 4-61, under
the descriptions of the routines that use them.

Apple Event Array Data Types

The AEGetArray function (see page 4-77) creates a Pascal or C array that corresponds to an **Apple event array** in a descriptor list, and the AEPutArray function (see page 5-31) adds data specified in a buffer to a descriptor list as an Apple event array.

You can use the data type AEArrayType to define the type of Apple event array you want to add to or obtain from a descriptor list.

```
TYPE AEArrayType = (kAEDataArray, kAEPackedArray, kAEHandleArray,
                    kAEDescArray, kAEKeyDescArray);
```

When your application adds an Apple event array to a descriptor list, it provides the data for an Apple event array in an array data record, which is defined by the data type AEArrayData.

```
TYPE AEArrayData =
   RECORD                          {data for an Apple event array}
      CASE AEArrayType OF
      kAEDataArray:
         (AEDataArray:    ARRAY[0..0] OF Integer);
      kAEPackedArray:
         (AEPackedArray:  PACKED ARRAY[0..0] OF Char);
      kAEHandleArray:
         (AEHandleArray:  ARRAY[0..0] OF Handle);
      kAEDescArray:
         (AEDescArray:    ARRAY[0..0] OF AEDesc);
      kAEKeyDescArray:
         (AEKeyDescArray: ARRAY[0..0] OF AEKeyDesc);
   END;
```

The type of array depends on the data for the array:

Array type	Description of Apple event array
kAEDataArray	Array items consist of data of the same size and same type, and are aligned on word boundaries.
kAEPackedArray	Array items consist of data of the same size and same type, and are packed without regard for word boundaries.
kAEHandleArray	Array items consist of handles to data of variable size and the same type.
kAEDescArray	Array items consist of descriptor records of different descriptor types with data of variable size.
kAEKeyDescArray	Array items consist of keyword-specified descriptor records with different keywords, different descriptor types, and data of variable size.

Array items in Apple event arrays of type `kAEDataArray`, `kAEPackedArray`, or `kAEHandleArray` must be factored—that is, contained in a factored descriptor list. Before adding array items to a factored descriptor list, you should provide both a pointer to the data that is common to all array items and the size of that common data when you first call `AECreateList` to create a factored descriptor list. When you call `AEPutArray` to add the array data to such a descriptor list, the Apple Event Manager automatically isolates the common data you specified in the call to `AECreateList`.

When you call `AEGetArray` or `AEPutArray`, you specify a pointer of data type `AEArrayDataPointer` that points to a buffer containing the data for the array.

```
TYPE AEArrayDataPointer = ^AEArrayData;
```

For more information about using `AECreateList` to create factored descriptor lists for arrays, see page 5-28. For information about using `AEGetArray` and `AEPutArray`, see page 4-77 and page 5-31, respectively.

Routines for Responding to Apple Events

This section describes the Apple Event Manager routines you can use to create and manage the Apple event dispatch tables, dispatch Apple events, extract information from Apple events, request user interaction, request more time to respond to Apple events, suspend and resume Apple event handling, delete descriptor records, deallocate memory for descriptor records, create and manage the coercion handler and special handler dispatch tables, and get information about the Apple Event Manager.

Because the Apple Event Manager uses the services of the Event Manager, which in turn uses the services of the PPC Toolbox, the routines described in this section may return Event Manager and PPC Toolbox result codes in addition to the Apple Event Manager result codes listed.

Creating and Managing the Apple Event Dispatch Tables

An Apple event dispatch table contains entries that specify the event class and event ID that refer to one or more Apple events, the address of the handler routine that handles those Apple events, and a reference constant. You can use the `AEInstallEventHandler` function to add entries to the Apple event dispatch table. This function sets up the initial mapping between the handlers in your application and the Apple events that they handle.

To get the address of a handler currently in the Apple event dispatch table, use the `AEGetEventHandler` function. If you need to remove any of your Apple event handlers after the mapping between handlers and Apple events is established, you can use the `AERemoveEventHandler` function.

AEInstallEventHandler

You can use the AEInstallEventHandler function to add an entry to either your application's Apple event dispatch table or the system Apple event dispatch table.

```
FUNCTION AEInstallEventHandler (theAEEventClass: AEEventClass;
                                theAEEventID: AEEventID;
                                handler: EventHandlerProcPtr;
                                handlerRefcon: LongInt;
                                isSysHandler: Boolean): OSErr;
```

theAEEventClass
: The event class for the Apple event or events to be dispatched for this entry. The AEEventClass data type is defined as a four-character code:

```
TYPE AEEventClass = PACKED ARRAY[1..4] OF Char;
```

theAEEventID
: The event ID for the Apple event or events to be dispatched for this entry. The AEEventID data type is defined as a four-character code:

```
TYPE AEEventID = PACKED ARRAY[1..4] OF Char;
```

handler
: A pointer to an Apple event handler for this dispatch table entry. Note that a handler in the system dispatch table must reside in the system heap; this means that if the value of the isSysHandler parameter is TRUE, the handler parameter should point to a location in the system heap. Otherwise, if you put your system handler code in your application heap, you must use AERemoveEventHandler to remove the handler before your application terminates.

handlerRefcon
: A reference constant that is passed by the Apple Event Manager to the handler each time the handler is called. If your handler doesn't use a reference constant, use 0 as the value of this parameter.

isSysHandler
: Specifies the dispatch table to which you want to add the handler. If the value of isSysHandler is TRUE, the Apple Event Manager adds the handler to the system Apple event dispatch table. Entries in the system dispatch table are available to all applications. If the value of isSysHandler is FALSE, the Apple Event Manager adds the handler to your application's Apple event dispatch table. The application's dispatch table is searched first; the system dispatch table is searched only if the necessary handler is not found in your application's dispatch table.

DESCRIPTION

The `AEInstallEventHandler` function creates an entry in the Apple event dispatch table. You must supply parameters that specify the event class, the event ID, the address of the handler that handles Apple events of the specified event class and event ID, and whether the handler is to be added to the system Apple event dispatch table or your application's Apple event dispatch table. You can also specify a reference constant that the Apple Event Manager passes to your handler whenever your handler processes an Apple event.

The parameters `theAEEventClass` and `theAEEventID` specify the event class and event ID of the Apple events to be handled by the handler for this dispatch table entry. For these parameters, you must provide one of the following combinations:

n   the event class and event ID of a single Apple event to be dispatched to the handler

n   the `typeWildCard` constant for `theAEEventClass` and an event ID for `theAEEventID`, which indicate that Apple events from all event classes whose event IDs match `theAEEventID` should be dispatched to the handler

n   an event class for `theAEEventClass` and the `typeWildCard` constant for `theAEEventID`, which indicate that all events from the specified event class should be dispatched to the handler

n   the `typeWildCard` constant for both the `theAEEventClass` and `theAEEventID` parameters, which indicates that all Apple events should be dispatched to the handler

IMPORTANT

If you use the `typeWildCard` constant for either the `theAEEventClass` or the `theAEEventID` parameter (or for both parameters), the corresponding handler must return the error `errAEEventNotHandled` if it does not handle a particular event. s

If there was already an entry in the specified dispatch table for the same event class and event ID, it is replaced. Therefore, before installing a handler for a particular Apple event in the system dispatch table, use the `AEGetEventHandler` function (described next) to determine whether the table already contains a handler for that event. If an entry exists, `AEGetEventHandler` returns a reference constant and a pointer to that event handler. Chain the existing handler to your handler by providing pointers to the previous handler and its reference constant in the `handlerRefcon` parameter of `AEInstallEventHandler`. When your handler is done, use these pointers to call the previous handler. If you remove your system Apple event handler, be sure to reinstall the chained handler.

SPECIAL CONSIDERATIONS

Before an application calls a system Apple event handler, system software has set up the A5 register for the calling application. For this reason, if you provide a system Apple event handler, it should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system handler may crash.

RESULT CODES

noErr	**0**	No error
paramErr	**−50**	Parameter error (handler pointer is NIL or odd)
memFullErr	**−108**	Not enough room in heap zone

SEE ALSO

For more information about installing Apple event handlers, see "Installing Entries in the Apple Event Dispatch Tables," which begins on page 4-7.

AEGetEventHandler

You can use the AEGetEventHandler function to get an entry from an Apple event dispatch table.

```
FUNCTION AEGetEventHandler (theAEEventClass: AEEventClass;
                            theAEEventID: AEEventID;
                            VAR handler: EventHandlerProcPtr;
                            VAR handlerRefcon: LongInt;
                            isSysHandler: Boolean): OSErr;
```

theAEEventClass
> The value of the event class field of the dispatch table entry for the desired handler.

theAEEventID
> The value of the event ID field of the dispatch table entry for the desired handler.

handler   The AEGetEventHandler function returns, in this parameter, a pointer to the specified handler.

handlerRefcon
> The AEGetEventHandler function returns, in this parameter, the reference constant from the dispatch table entry for the specified handler.

isSysHandler

Specifies the Apple event dispatch table from which to get the handler. If the value of isSysHandler is TRUE, the AEGetEventHandler function returns the handler from the system dispatch table. If the value is FALSE, AEGetEventHandler returns the handler from your application's dispatch table.

DESCRIPTION

The AEGetEventHandler function returns, in the handler parameter, a pointer to the handler for the Apple event dispatch table entry you specify in the parameters theAEEventClass and theAEEventID. You can use the typeWildCard constant for either or both of these parameters; however, AEGetEventHandler returns an error unless an entry exists that specifies typeWildCard in exactly the same way. For example, if you specify typeWildCard in both the theAEEventClass parameter and the theAEEventID parameter, the Apple Event Manager will not return the first handler for any event class and event ID in the dispatch table; instead, the dispatch table must contain an entry that specifies type typeWildCard for both the event class and the event ID.

RESULT CODES

noErr	0	No error
errAEHandlerNotFound	–1717	No handler found for an Apple event

SEE ALSO

For an explanation of wildcard values, see the description of the AEInstallEventHandler function on page 4-62.

AERemoveEventHandler

You can use the AERemoveEventHandler function to remove an entry from an Apple event dispatch table.

```
FUNCTION AERemoveEventHandler (theAEEventClass: AEEventClass;
                               theAEEventID: AEEventID;
                               handler: EventHandlerProcPtr;
                               isSysHandler: Boolean): OSErr;
```

theAEEventClass

The event class for the handler whose entry you want to remove from the dispatch table.

theAEEventID
: The event ID for the handler whose entry you want to remove from the Apple event dispatch table.

handler
: A pointer to the handler to be removed. Although the parameters theAEEventClass and theAEEventID would be sufficient to identify the handler to be removed, providing the handler parameter is a recommended safeguard that ensures that you remove the correct handler. If the value of this parameter is NIL, the Apple Event Manager relies solely on the event class and event ID to identify the handler to be removed.

isSysHandler
: Specifies the dispatch table from which to remove the handler. If the value of isSysHandler is TRUE, AERemoveEventHandler removes the handler from the system dispatch table. If the value is FALSE, AERemoveEventHandler removes the handler from your application's dispatch table.

DESCRIPTION

The AERemoveEventHandler function removes the Apple event dispatch table entry you specify in the parameters theAEEventClass, theAEEventID, and handler. You can use the typeWildCard constant for the theAEEventClass or the theAEEventID parameter, or for both parameters; however, AERemoveEventHandler returns an error unless an entry exists that specifies typeWildCard in exactly the same way. For example, if you specify typeWildCard in both the theAEEventClass parameter and the theAEEventID parameter, the Apple Event Manager will not remove the first handler for any event class and event ID in the dispatch table; instead, the dispatch table must contain an entry that specifies type typeWildCard for both the event class and the event ID.

RESULT CODES

noErr	0	No error
errAEHandlerNotFound	–1717	No handler found for an Apple event

SEE ALSO

For an explanation of wildcard values, see the description of the AEInstallEventHandler function on page 4-62.

Dispatching Apple Events

After receiving a high-level event (and optionally determining whether it is a type of high-level event other than an Apple event that your application might support), your application typically calls the AEProcessAppleEvent function to determine the type of Apple event received and call the corresponding handler.

AEProcessAppleEvent

You can use the `AEProcessAppleEvent` function to call the appropriate handler for a specified Apple event.

```
FUNCTION AEProcessAppleEvent
                        (theEventRecord: EventRecord): OSErr;
```

`theEventRecord`
    The event record for the Apple event.

DESCRIPTION

The `AEProcessAppleEvent` function looks first in the application's special handler dispatch table for an entry that was installed with the constant `keyPreDispatch`. If the application's special handler dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the function looks in the application's Apple event dispatch table for an entry that matches the event class and event ID of the specified Apple event.

If the application's Apple event dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the `AEProcessAppleEvent` function looks in the system special handler dispatch table for an entry that was installed with the constant `keyPreDispatch`. If the system special handler dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the function looks in the system Apple event dispatch table for an entry that matches the event class and event ID of the specified Apple event.

If the system Apple event dispatch table does not include such a handler, the Apple Event Manager returns the result code `errAEEventNotHandled` to the server application and, if the client application is waiting for a reply, to the client application.

If `AEProcessAppleEvent` finds an entry in one of the dispatch tables that matches the event class and event ID of the specified Apple event, it calls the corresponding handler.

SPECIAL CONSIDERATIONS

If an Apple event dispatch table contains one entry for an event class and a specific event ID, and also contains another entry that is identical except that it specifies a wildcard value for either the event class or the event ID, the Apple Event Manager dispatches the more specific entry. For example, if an Apple event dispatch table includes one entry that specifies the event class as `kAECoreSuite` and the event ID as `kAEDelete`, and another entry that specifies the event class as `kAECoreSuite` and the event ID as `typeWildCard`, the Apple Event Manager dispatches the Apple event handler associated with the entry that specifies the event ID as `kAEDelete`.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
bufferIsSmall	–607	Buffer is too small
noOutstandingHLE	–608	No outstanding high-level event
errAECorruptData	–1702	Data in an Apple event could not be read
errAENewerVersion	–1706	Need a newer version of the Apple Event Manager
errAEEventNotHandled	–1708	Event wasn't handled by an Apple event handler

SEE ALSO

For more information about event processing, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

Getting Data or Descriptor Records Out of Apple Event Parameters and Attributes

The Apple Event Manager provides four functions that allow you to get data from Apple event parameters and attributes. The AEGetParamPtr and AEGetParamDesc functions get data from a specified Apple event parameter. The AEGetAttributePtr and AEGetAttributeDesc functions get data from a specified Apple event attribute.

AEGetParamPtr

You can use the AEGetParamPtr function to get a pointer to a buffer that contains the data from a specified Apple event parameter.

```
FUNCTION AEGetParamPtr (theAppleEvent: AppleEvent;
                        theAEKeyword: AEKeyword;
                        desiredType: DescType;
                        VAR typeCode: DescType; dataPtr: Ptr;
                        maximumSize: Size;
                        VAR actualSize: Size): OSErr;
```

theAppleEvent
          The Apple event containing the desired parameter.

theAEKeyword
          The keyword that specifies the desired parameter.

desiredType
          The desired descriptor type for the data to be returned; if the requested
          Apple event parameter is not of this type, the Apple Event Manager
          attempts to coerce it to this type. If the value of desiredType

is `typeWildCard`, no coercion is performed, and the descriptor type of the returned data is the same as the descriptor type of the Apple event parameter.

typeCode   The descriptor type of the returned data.

dataPtr   A pointer to the buffer in which the returned data is stored.

maximumSize
           The maximum length, in bytes, of the data to be returned. You must allocate at least this amount of storage for the buffer specified by the `dataPtr` parameter.

actualSize
           The length, in bytes, of the data for the specified Apple event parameter. If this value is larger than the value of the `maximumSize` parameter, not all of the data for the parameter was returned.

DESCRIPTION

The `AEGetParamPtr` function uses a buffer to return the data from a specified Apple event parameter, which it attempts to coerce to the descriptor type specified by the `desiredType` parameter.

RESULT CODES

noErr	0	No error
memFullErr	−108	Not enough room in heap zone
errAECoercionFail	−1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	−1701	Descriptor record was not found
errAEWrongDataType	−1703	Wrong descriptor type
errAENotAEDesc	−1704	Not a valid descriptor record
errAEReplyNotArrived	−1718	Reply has not yet arrived

SEE ALSO

For examples of the use of `AEGetParamPtr`, see "Getting Data Out of an Apple Event," which begins on page 4-25.

AEGetParamDesc

You can use the `AEGetParamDesc` function to get the descriptor record for a specified Apple event parameter.

```
FUNCTION AEGetParamDesc (theAppleEvent: AppleEvent;
                         theAEKeyword: AEKeyword;
                         desiredType: DescType;
                         VAR result: AEDesc): OSErr;
```

theAppleEvent
                The Apple event containing the desired parameter.
theAEKeyword
                The keyword that specifies the desired parameter.
desiredType
                The desired descriptor type for the descriptor record to be returned; if the
                requested Apple event parameter is not of this type, the Apple Event
                Manager attempts to coerce it to this type. If the value of desiredType is
                typeWildCard, no coercion is performed, and the descriptor type of the
                returned data is the same as the descriptor type of the Apple event
                parameter.
result   The descriptor record from the desired Apple event parameter coerced to
                the descriptor type specified in desiredType.

DESCRIPTION

The AEGetParamDesc function returns, in the result parameter, the descriptor record
for a specified Apple event parameter, which it attempts to coerce to the descriptor type
specified by the desiredType parameter. Your application should call the
AEDisposeDesc function to dispose of the resulting descriptor record after your
application has finished using it.

If AEGetParamDesc returns a nonzero result code, it returns a null descriptor record
unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

noErr	0	No error
memFullErr	−108	Not enough room in heap zone
errAECoercionFail	−1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	−1701	Descriptor record was not found
errAENotAEDesc	−1704	Not a valid descriptor record
errAEReplyNotArrived	−1718	Reply has not yet arrived

SEE ALSO

For an example of the use of AEGetParamDesc, see "Getting Data Out of an Apple
Event Parameter," which begins on page 4-26.

AEGetAttributePtr

You can use the AEGetAttributePtr function to get a pointer to a buffer that contains the data from a specified Apple event attribute.

```
FUNCTION AEGetAttributePtr (theAppleEvent: AppleEvent;
                            theAEKeyword: AEKeyword;
                            desiredType: DescType;
                            VAR typeCode: DescType; dataPtr: Ptr;
                            maximumSize: Size;
                            VAR actualSize: Size): OSErr;
```

theAppleEvent
              The Apple event containing the desired attribute.

theAEKeyword
              The keyword that specifies the desired attribute.

              TYPE AEKeyword = PACKED ARRAY[1..4] OF Char;

              The keyword can be any of the constants listed in the description that follows.

desiredType
              The desired descriptor type for the data to be returned; if the requested Apple event attribute is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of desiredType is typeWildCard, no coercion is performed, and the descriptor type of the returned data is the same as the descriptor type of the Apple event attribute.

typeCode   The descriptor type of the returned data.

dataPtr   A pointer to the buffer in which the returned data is stored.

maximumSize
              The maximum length, in bytes, of the data to be returned. You must allocate at least this amount of storage for the buffer specified by the dataPtr parameter.

actualSize
              The length, in bytes, of the data for the specified Apple event attribute. If this value is larger than the value of the maximumSize parameter, not all of the data for the attribute was returned.

DESCRIPTION

The AEGetAttributePtr function uses a buffer to return the data from an Apple event attribute with the specified keyword, which it attempts to coerce to the descriptor type specified by the desiredType parameter. You can specify the parameter theAEKeyWord using any of these constants:

```
CONST
   keyAddressAttr            = 'addr'; {address of target or }
                                       { client application}
   keyEventClassAttr         = 'evcl'; {event class}
   keyEventIDAttr            = 'evid'; {event ID}
   keyEventSourceAttr        = 'esrc'; {nature of source }
                                       { application}
   keyInteractLevelAttr      = 'inte'; {settings to allow the }
                                       { Apple Event Manager to }
                                       { bring server application }
                                       { to the foreground}
   keyMissedKeywordAttr      = 'miss'; {first required parameter }
                                       { remaining in Apple event}
   keyOptionalKeywordAttr    = 'optk'; {list of optional }
                                       { parameters for Apple }
                                       { event}
   keyOriginalAddressAttr    = 'from'; {address of original source }
                                       { of Apple event; available }
                                       { beginning with version }
                                       { 1.01 of Apple Event }
                                       { Manager}
   keyReturnIDAttr           = 'rtid'; {return ID for reply Apple }
                                       { event}
   keyTimeoutAttr            = 'timo'; {length of time in ticks }
                                       { that client will wait }
                                       { for reply or result from }
                                       { the server}
   keyTransactionIDAttr      = 'tran'; {transaction ID identifying }
                                       { a series of Apple events}
```

RESULT CODES

noErr	0	No error
memFullErr	−108	Not enough room in heap zone
errAECoercionFail	−1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	−1701	Descriptor record was not found
errAENotAEDesc	−1704	Not a valid descriptor record
errAEReplyNotArrived	−1718	Reply has not yet arrived

SEE ALSO

For an example of the use of the AEGetAttributePtr function, see "Getting Data Out of an Attribute" and "Writing Apple Event Handlers," which begin on page 4-28 and page 4-33, respectively.

AEGetAttributeDesc

You can use the AEGetAttributeDesc function to get the descriptor record for a specified Apple event attribute.

```
FUNCTION AEGetAttributeDesc (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword;
                             desiredType: DescType;
                             VAR result: AEDesc): OSErr;
```

theAppleEvent
: The Apple event containing the desired attribute.

theAEKeyword
: The keyword that specifies the desired attribute.

    TYPE AEKeyword = PACKED ARRAY[1..4] OF Char;

    The keyword can be any of the constants listed in the description of AEGetAttributePtr on page 4-71.

desiredType
: The desired descriptor type for the descriptor record to be returned; if the requested Apple event attribute is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of desiredType is typeWildCard, no coercion is performed, and the descriptor type of the returned data is the same as the descriptor type of the Apple event attribute.

result
: A copy of the descriptor record from the desired attribute coerced to the descriptor type specified by the desiredType parameter.

DESCRIPTION

The AEGetAttributeDesc function returns, in the result parameter, the descriptor record for the Apple event attribute with the specified keyword. Your application should call the AEDisposeDesc function to dispose of the resulting descriptor record after your application has finished using it.

If AEGetAttributeDesc returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	–1701	Descriptor record was not found
errAENotAEDesc	–1704	Not a valid descriptor record
errAEReplyNotArrived	–1718	Reply has not yet arrived

Counting the Items in Descriptor Lists

The AECountItems function counts the number of descriptor records in any descriptor list, including an Apple event record.

AECountItems

You can use the AECountItems function to count the number of descriptor records in any descriptor list.

```
FUNCTION AECountItems (theAEDescList: AEDescList;
                       VAR theCount: LongInt): OSErr;
```

theAEDescList
        The descriptor list to be counted.
theCount   The AECountItems function returns the number of descriptor records in the specified descriptor list in this parameter.

RESULT CODES

noErr	0	No error
errAENotAEDesc	–1704	Not a valid descriptor record

SEE ALSO

Getting Items From Descriptor Lists

The Apple Event Manager provides three functions that allow you to get items from any descriptor list, including an Apple event record. The AEGetNthPtr and AEGetNthDesc functions give you access to the data in a descriptor list. The AEGetArray function gets data from an array contained in a descriptor list.

AEGetNthPtr

You can use the AEGetNthPtr function to get a pointer to a buffer that contains a copy of a descriptor record from any descriptor list.

```
FUNCTION AEGetNthPtr (theAEDescList: AEDescList; index: LongInt;
                      desiredType: DescType;
                      VAR theAEKeyword: AEKeyword;
                      VAR typeCode: DescType; dataPtr: Ptr;
                      maximumSize: Size;
                      VAR actualSize: Size): OSErr;
```

theAEDescList
              The descriptor list containing the desired descriptor record.

index   The position of the desired descriptor record in the list (for example, 2 specifies the second descriptor record).

desiredType
              The desired descriptor type for the copy of the descriptor record to be returned; if the desired descriptor record is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of desiredType is typeWildCard, no coercion is performed, and the descriptor type of the copied descriptor record is the same as the descriptor type of the original descriptor record.

theAEKeyword
              The keyword of the specified descriptor record, if you are getting data from a list of keyword-specified descriptor records; otherwise, AEGetNthPtr returns the value typeWildCard.

typeCode   The descriptor type of the returned descriptor record.

dataPtr   A pointer to the buffer in which the returned descriptor record is stored.

maximumSize
              The maximum length, in bytes, of the data to be returned. You must allocate at least this amount of storage for the buffer specified by the dataPtr parameter.

actualSize
              The length, in bytes, of the data for the specified descriptor record. If this value is larger than the value of the maximumSize parameter, not all of the data for the descriptor record was returned.

DESCRIPTION

The AEGetNthPtr function uses a buffer to return a specified descriptor record from a specified descriptor list; the function attempts to coerce the descriptor record to the descriptor type specified by the desiredType parameter.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	–1701	Descriptor record was not found
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEReplyNotArrived	–1718	Reply has not yet arrived

SEE ALSO

For an example of the use of AEGetNthPtr, see Listing 4-10 on page 4-33.

AEGetNthDesc

You can use the AEGetNthDesc function to get a copy of a descriptor record from any descriptor list.

```
FUNCTION AEGetNthDesc (theAEDescList: AEDescList; index: LongInt;
                       desiredType: DescType;
                       VAR theAEKeyword: AEKeyword;
                       VAR result: AEDesc): OSErr;
```

theAEDescList
: The descriptor list containing the desired descriptor record.

index
: The position of the desired descriptor record in the list (for example, 2 specifies the second descriptor record).

desiredType
: The desired descriptor type for the copy of the descriptor record to be returned; if the desired descriptor record is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of desiredType is typeWildCard, no coercion is performed, and the descriptor type of the copied descriptor record is the same as the descriptor type of the original descriptor record.

theAEKeyword
: The keyword of the specified descriptor record, if you are getting data from a list of keyword-specified descriptor records; otherwise, AEGetNthDesc returns the value typeWildCard.

result
: A copy of the desired descriptor record coerced to the descriptor type specified by the desiredType parameter.

DESCRIPTION

The `AEGetNthDesc` function returns a specified descriptor record from a specified descriptor list. Your application should call the `AEDisposeDesc` function to dispose of the resulting descriptor record after your application has finished using it.

If `AEGetNthDesc` returns a nonzero result code, it returns a descriptor record of descriptor type `typeNull`. A descriptor record of this type does not contain any data.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	–1701	Descriptor record was not found
errAENotAEDesc	–1704	Not a valid descriptor record
errAEReplyNotArrived	–1718	Reply has not yet arrived

AEGetArray

You can use the `AEGetArray` function to convert an Apple event array (an array created with the `AEPutArray` function and stored in a descriptor list) to the corresponding Pascal or C array and place the converted array in a buffer for which you have provided a pointer.

```
FUNCTION AEGetArray (theAEDescList: AEDescList;
                     arrayType: AEArrayType;
                     arrayPtr: AEArrayDataPointer;
                     maximumSize: Size;
                     VAR itemType: DescType; VAR itemSize: Size;
                     VAR itemCount: LongInt): OSErr;
```

theAEDescList
: A descriptor list containing the desired array. If the array is of type `kAEDataArray`, `kAEPackedArray`, or `kAEHandleArray`, the descriptor list must be factored.

arrayType
: The Apple event array type to be converted. This is specified by one of the following constants: `kAEDataArray`, `kAEPackedArray`, `kAEHandleArray`, `kAEDescArray`, or `kAEKeyDescArray`.

arrayPtr
: A pointer to the buffer for storing the array.

maximumSize
: The maximum length, in bytes, of the buffer for storing the array.

itemType
: For arrays of type `kAEDataArray`, `kAEPackedArray`, or `kAEHandleArray`, the `AEGetArray` function returns the descriptor type of the returned array items in this parameter.

itemSize   For arrays of type `kAEDataArray` or `kAEPackedArray`, the
`AEGetArray` function returns the size (in bytes) of the returned array
items in this parameter.

itemCount   The `AEGetArray` function returns the number of items in the resulting
array in this parameter.

DESCRIPTION

The `AEGetArray` function uses a buffer identified by the pointer in the `arrayPtr`
parameter to return the converted data for the Apple event array specified by the
`theAEDescList` parameter. Even if the descriptor list that contains the array is
factored, the converted data for each array item includes the data common to all the
descriptor records in the list. The Apple Event Manager automatically reconstructs
the common data for each item when you call `AEGetArray`.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEReplyNotArrived	–1718	Reply has not yet arrived

SEE ALSO

For more information about data types and constants used with `AEGetArray`, see
"Apple Event Array Data Types" on page 4-60.

For information about creating and factoring descriptor lists for Apple event arrays, see
the description of `AECreateList` on page 5-28. For information about adding an Apple
event array to a descriptor list, see the description of `AEPutArray` on page 5-31.

Getting Data and Keyword-Specified Descriptor Records Out of AE Records

The Apple Event Manager provides two functions, `AEGetKeyPtr` and `AEGetKeyDesc`,
that allow you to get data and descriptor records out of an AE record or an Apple event
record.

AEGetKeyPtr

You can use the AEGetKeyPtr function to get a pointer to a buffer that contains the data from a keyword-specified descriptor record. You can use this function to get data from an AE record or an Apple event record.

```
FUNCTION AEGetKeyPtr (theAERecord: AERecord;
                      theAEKeyword: AEKeyword;
                      desiredType: DescType;
                      VAR typeCode: DescType;
                      dataPtr: Ptr; maximumSize: Size;
                      VAR actualSize: Size): OSErr;
```

theAERecord
:   The AE record containing the desired data.

theAEKeyword
:   The keyword that specifies the desired descriptor record.

desiredType
:   The desired descriptor type for the data to be returned; if the requested data is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of desiredType is typeWildCard, no coercion is performed, and the descriptor type of returned data is the same as the descriptor type of the original data.

typeCode   The descriptor type of the returned data.

dataPtr   A pointer to the buffer for storing the data.

maximumSize
:   The maximum length, in bytes, of the data to be returned. You must allocate at least this amount of storage for the buffer specified by the dataPtr parameter.

actualSize
:   The length, in bytes, of the data for the keyword-specified descriptor record. If this value is larger than the value of the maximumSize parameter, not all of the data for the parameter was returned.

DESCRIPTION

The AEGetKeyPtr function uses a buffer to return the data from a keyword-specified Apple event parameter, which the function attempts to coerce to the descriptor type specified by the desiredType parameter.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	–1701	Descriptor record was not found
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEReplyNotArrived	–1718	Reply has not yet arrived

AEGetKeyDesc

You can use the `AEGetKeyDesc` function to get the descriptor record for a keyword-specified descriptor record. You can use this function to get a descriptor record out of an AE record or an Apple event record.

```
FUNCTION AEGetKeyDesc (theAERecord: AERecord;
                       theAEKeyword: AEKeyword;
                       desiredType: DescType;
                       VAR result: AEDesc): OSErr;
```

theAERecord
The AE record containing the desired descriptor record.

theAEKeyword
The keyword that specifies the desired descriptor record.

desiredType
The desired descriptor type for the descriptor record to be returned; if the requested descriptor record is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of `desiredType` is `typeWildCard`, no coercion is performed, and the descriptor type of the returned descriptor record is the same as the descriptor type of the original descriptor record.

result
A copy of the keyword-specified descriptor record, coerced to the descriptor type specified in the `desiredType` parameter.

DESCRIPTION

The `AEGetKeyDesc` function returns a copy of the descriptor record for a keyword-specified descriptor record. Your application should call the `AEDisposeDesc` function to dispose of the resulting descriptor record after your application has finished using it.

If `AEGetKeyDesc` returns a nonzero result code, it returns a descriptor record of descriptor type `typeNull`. A descriptor record of this type does not contain any data.

RESULT CODES

noErr	0	No error
memFullErr	−108	Not enough room in heap zone
errAECoercionFail	−1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	−1701	Descriptor record was not found
errAENotAEDesc	−1704	Not a valid descriptor record
errAEReplyNotArrived	−1718	Reply has not yet arrived

Requesting User Interaction

The Apple Event Manager provides three functions that allow you to set or request user interaction levels and to initiate user interaction when your application is the server application. The AESetInteractionAllowed and AEGetInteractionAllowed functions specify and return, respectively, the current user interaction preferences. Your application should call the AEInteractWithUser function before actually interacting with the user in response to an Apple event.

AESetInteractionAllowed

You can use the AESetInteractionAllowed function to specify your application's user interaction preferences for responding to an Apple event.

```
FUNCTION AESetInteractionAllowed
                          (level: AEInteractAllowed): OSErr;
```

level   The user interaction level to be set.

DESCRIPTION

The AESetInteractionAllowed function sets the user interaction level for a server application's response to an Apple event. The level parameter must be one of three flags: kAEInteractWithSelf, kAEInteractWithLocal, or kAEInteractWithAll.

Specifying the kAEInteractWithSelf flag allows the server application to interact with the user in response to an Apple event only when the client application and server application are the same—that is, only when the application is sending the Apple event to itself.

Specifying the kAEInteractWithLocal flag allows the server application to interact with the user in response to an Apple event only if the client application is on the same computer as the server application; this is the default if the AESetInteractionAllowed function is not used.

Specifying the kAEInteractWithAll flag allows the server application to interact with the user in response to an Apple event sent from any client application on any computer.

RESULT CODE

noErr   **0**   No error

SEE ALSO

For more information about setting user preferences for a server application, see "Setting the Server Application's User Interaction Preferences" on page 4-48.

AEGetInteractionAllowed

You can use the AEGetInteractionAllowed function to get the current user interaction preferences for responding to an Apple event.

```
FUNCTION AEGetInteractionAllowed
                        (VAR level: AEInteractAllowed): OSErr;
```

level   The current user interaction level, using the data type
               AEInteractAllowed.

               TYPE AEInteractAllowed = (kAEInteractWithSelf,
                                         kAEInteractWithLocal,
                                         kAEInteractWithAll);

DESCRIPTION

The AEGetInteractionAllowed function returns, in the level parameter, a value that indicates the user interaction preferences for responding to an Apple event. The value, set by a previous call to AESetInteractionAllowed, is one of the following flags: kAEInteractWithSelf, kAEInteractWithLocal, or kAEInteractWithAll. The default value of kAEInteractWithLocal is returned if your application has not used AESetInteractionAllowed to set the interaction level explicitly.

The kAEInteractWithSelf flag indicates that the server application may interact with the user in response to an Apple event only when the client application and server application are the same—that is, only when the application is sending the Apple event to itself.

The kAEInteractWithLocal flag indicates that the server application may interact with the user in response to an Apple event only if the client application is on the same computer as the server application. This is the default if your application has not used the AESetInteractionAllowed function to set the interaction level explicitly.

The kAEInteractWithAll flag indicates that the server application may interact with the user in response to an Apple event sent from any client application on any computer.

RESULT CODE

noErr   **0**   No error

AEInteractWithUser

You can use the AEInteractWithUser function to initiate interaction with the user when your application is a server application responding to an Apple event.

```
FUNCTION AEInteractWithUser (timeOutInTicks: LongInt;
                             nmReqPtr: NMRecPtr;
                             idleProc: IdleProcPtr): OSErr;
```

timeOutInTicks

The amount of time (in ticks) that your handler is willing to wait for a response from the user. You can specify a number of ticks or use one of the following constants:

```
CONST kAEDefaultTimeout = -1; {value determined }
                              { by AEM}
      kNoTimeOut        = -2; {wait until reply }
                              { comes back}
```

nmReqPtr   A pointer to a Notification Manager record provided by your application. You can specify NIL for this parameter to get the default notification handling provided by the Apple Event Manager.

idleProc   A pointer to your application's idle function, which handles events while waiting for the Apple Event Manager to return control.

DESCRIPTION

Your application should call the AEInteractWithUser function before displaying a dialog box or alert box or otherwise interacting with the user in response to an Apple event. If the user interaction preference settings permit the application to come to the foreground, this function brings your application to the front, either directly or by posting a notification request.

Your application should normally pass a notification record in the nmReqPtr parameter rather than specifying NIL for default notification handling. If you specify NIL, the Apple Event Manager looks for an application icon with the ID specified by the application's bundle ('BNDL') resource and the application's file reference ('FREF') resource. The Apple Event Manager first looks for an 'SICN' resource with the specified ID; if it can't find an 'SICN' resource, it looks for the 'ICN#' resource and compresses the icon to fit in the menu bar. The Apple Event Manager won't look for any members of an icon family other than the icon specified in the 'ICN#' resource.

If the application doesn't have 'SICN' or 'ICN#' resources, or if it doesn't have a file reference resource, the Apple Event Manager passes NIL to the Notification Manager, and no icon appears in the upper-right corner of the screen. Therefore, if you want to display any icon other than those of type 'SICN' or 'ICN#', you must specify a notification record as the second parameter to the AEInteractWithUser function.

Note

If you want the Notification Manager to use a color icon when it posts a notification request, you should provide a Notification Manager record that specifies a `'cicn'` resource. u

The `AEInteractWithUser` function checks whether the client application set the `kAENeverInteract` flag for the Apple event and, if so, returns an error. If not, then the `AEInteractWithUser` function checks the server application's preference set by the `AESetInteractionAllowed` function and compares it against the source of the Apple event—that is, whether it came from the same application, another process on the same computer, or a process running on another computer. The `AEInteractWithUser` function returns the `errAENoUserInteraction` result code if the user interaction preferences don't allow user interaction. If user interaction is allowed, the Apple Event Manager brings your application to the front, either directly or by posting a notification request. If `AEInteractWithUser` returns the `noErr` result code, then your application is in the foreground and is free to interact with the user.

RESULT CODES

noErr	0	No error
errAETimeout	–1712	Apple event timed out
errAENoUserInteraction	–1713	No user interaction allowed

SEE ALSO

For information about idle functions, see "Writing an Idle Function" on page 5-21.

For examples of the use of the `AEInteractWithUser` function, see "Interacting With the User," which begins on page 4-45.

Requesting More Time to Respond to Apple Events

The `AEResetTimer` function resets the timeout value for an Apple event to its starting value. A server application can call this function when it knows it cannot fulfill a client application's request (either by returning a result or by sending back a reply Apple event) before the client application is due to time out.

AEResetTimer

You can use the `AEResetTimer` function to reset the timeout value for an Apple event to its starting value.

```
FUNCTION AEResetTimer (reply: AppleEvent): OSErr;
```

reply   The default reply for an Apple event, provided by the Apple Event
               Manager.

DESCRIPTION

When your application calls `AEResetTimer`, the Apple Event Manager for the server
application uses the default reply to send a Reset Timer event to the client application;
the Apple Event Manager for the client application's computer intercepts this Apple
event and resets the client application's timer for the Apple event. (The Reset Timer
event is never dispatched to a handler, so the client application does not need a handler
for it.)

RESULT CODE

noErr   **0**   No error
errAEReplyNotValid   **–1709**   `AEResetTimer` was passed an invalid reply

Suspending and Resuming Apple Event Handling

When your application calls `AEProcessAppleEvent` and one of your event handlers is
invoked, the Apple Event Manager normally assumes that your application has finished
handling the event when the event handler returns. At this point, the Apple Event
Manager disposes of the event. However, some applications, such as multi-session
servers or any applications that implement their own internal event queueing, may need
to defer handling of the event.

The `AESuspendTheCurrentEvent`, `AEResumeTheCurrentEvent`,
`AESetTheCurrentEvent`, and `AEGetTheCurrentEvent` functions described in this
section allow you to suspend and resume Apple event handling, specify the Apple event
to be handled, and identify an Apple event that is currently being handled.

AESuspendTheCurrentEvent

You can use the `AESuspendTheCurrentEvent` function to suspend the processing of
the Apple event that is currently being handled.

```
FUNCTION AESuspendTheCurrentEvent
                            (theAppleEvent: AppleEvent): OSErr;
```

theAppleEvent
               The Apple event whose handling is to be suspended. Although the Apple
               Event Manager doesn't need this parameter to identify the Apple event
               currently being handled, providing it is a safeguard that you are
               suspending the correct Apple event.

DESCRIPTION

After a server application makes a successful call to the `AESuspendTheCurrentEvent` function, it is no longer required to return a result or a reply for the Apple event that was being handled. It can, however, return a result if it later calls the `AEResumeTheCurrentEvent` function to resume event processing.

The Apple Event Manager does not automatically dispose of Apple events that have been suspended or their default replies. (The Apple Event Manager does, however, automatically dispose of a previously suspended Apple event and its default reply if the server later resumes processing of the Apple event by calling the `AEResumeTheCurrentEvent` function.) If your server application does not resume processing of a suspended Apple event, it is responsible for using the `AEDisposeDesc` function to dispose of both the Apple event and its default reply when your application has finished using them.

SPECIAL CONSIDERATIONS

If your application suspends handling of an Apple event it sends to itself, the Apple Event Manager immediately returns from the `AESend` call with the error code `errAETimeout`, regardless of whether the `kAEQueueReply`, `kAEWaitReply`, or `kAENoReply` flags were set, even if the `timeout` parameter is set to `kNoTimeOut`. The routine calling `AESend` should take the timeout error as confirmation that the event was sent.

As with other calls to `AESend` that return a timeout error, the handler continues to process the event nevertheless. The handler's reply, if any, is provided in the reply event when the handling is completed. The Apple Event Manager provides no notification that the reply is ready. If no data has yet been placed in the reply event, the Apple Event Manager returns `errAEReplyNotArrived` when your application attempts to extract data from the reply.

RESULT CODE

`noErr`   **0**   No error

AEResumeTheCurrentEvent

You can use the `AEResumeTheCurrentEvent` function to inform the Apple Event Manager that your application wants to resume the handling of a previously suspended Apple event or that it has completed the handling of the Apple event.

```
FUNCTION AEResumeTheCurrentEvent
                        (theAppleEvent, reply: AppleEvent;
                         dispatcher: EventHandlerProcPtr;
                         handlerRefcon: LongInt): OSErr;
```

theAppleEvent
>   The Apple event to be resumed.

reply   The default reply provided by the Apple Event Manager for the
>   Apple event.

dispatcher
>   One of the following:

>   n   a pointer to a routine for handling the event

>   n   the kAEUseStandardDispatch constant, which tells the Apple
>   Event Manager to dispatch the resumed event using the standard
>   dispatching scheme it uses for other Apple events

>   n   the kAENoDispatch constant, which tells the Apple Event Manager
>   that the Apple event has been completely processed and need not be
>   dispatched

handlerRefcon
>   If the value of the dispatcher parameter is not
>   kAEUseStandardDispatch, this parameter is the reference
>   constant passed to the handler when the handler is called. If the
>   value of the dispatcher parameter is kAEUseStandardDispatch,
>   the Apple Event Manager ignores the handlerRefcon parameter
>   and instead passes the reference constant stored in the Apple event
>   dispatch table entry for the Apple event. (You may wish to pass the
>   same reference constant that is stored in the Apple event dispatch
>   table. If so, call the AEGetEventHandler function.)

DESCRIPTION

When your application calls the AEResumeTheCurrentEvent function, the Apple
Event Manager resumes handling the specified Apple event using the handler specified
in the dispatcher parameter, if any. If kAENoDispatch is specified in the
dispatcher parameter, AEResumeTheCurrentEvent simply informs the Apple
Event Manager that the specified event has been handled.

SPECIAL CONSIDERATIONS

An Apple event handler that suspends an event should not immediately call
AEResumeTheCurrentEvent, or else the handler will generate an error. Instead, the
handler should return just after suspending the event.

When your application calls AEResumeTheCurrentEvent for an event that was not
directly dispatched, the Apple Event Manager disposes of the event and the reply,
just as it normally does, after the event handler returns to AEProcessAppleEvent.
Make sure all processing involving the event or the reply has been completed
before your application calls AEResumeTheCurrentEvent. Do not call
AEResumeTheCurrentEvent for an event that was not suspended.

When your application calls `AEResumeTheCurrentEvent` for an event that was directly dispatched, your application is responsible for disposing of the original event and the reply, since it is acts as both the server and the client.

RESULT CODE

noErr   **0**   No error

AESetTheCurrentEvent

You can use the `AESetTheCurrentEvent` function to specify the Apple event to be handled.

```
FUNCTION AESetTheCurrentEvent (theAppleEvent: AppleEvent): OSErr;
```

theAppleEvent
    The Apple event to be handled.

DESCRIPTION

There is usually no reason for your application to use the `AESetTheCurrentEvent` function. Instead of calling this function, your application should let the Apple Event Manager set the current Apple event through the dispatch tables.

If you need to avoid the dispatch tables, you must use the `AESetTheCurrentEvent` function only in the following way:

1. Your application suspends handling of an Apple event by calling the `AESuspendTheCurrentEvent` function.

2. Your application calls the `AESetTheCurrentEvent` function. This informs the Apple Event Manager that your application is handling the suspended Apple event. In this way, any routines that call the `AEGetTheCurrentEvent` function can ascertain which event is currently being handled.

3. When your application finishes handling the Apple event, it calls the `AEResumeTheCurrentEvent` function with the value `kAENoDispatch` to tell the Apple Event Manager that the event has been processed and need not be dispatched.

RESULT CODE

noErr   **0**   No error

AEGetTheCurrentEvent

You can use the AEGetTheCurrentEvent function to get the Apple event that is currently being handled.

```
FUNCTION AEGetTheCurrentEvent
                        (VAR theAppleEvent: AppleEvent): OSErr;
```

theAppleEvent
            The Apple event that is currently being handled; if no Apple event is currently being handled, AEGetTheCurrentEvent returns a null descriptor record in this parameter.

DESCRIPTION

In many applications, the handling of an Apple event involves one or more long chains of calls to internal routines. The AEGetTheCurrentEvent function makes it unnecessary for these calls to include the current Apple event as a parameter; the routines can simply call AEGetTheCurrentEvent to get the current Apple event when it is needed.

You can also use the AEGetTheCurrentEvent function to make sure that no Apple event is currently being handled. For example, suppose your application always uses an application-defined routine to delete a file. That routine can first call AEGetTheCurrentEvent and delete the file only if AEGetTheCurrentEvent returns a null descriptor record (that is, only if no Apple event is currently being handled).

RESULT CODE

noErr   0   No error

Getting the Sizes and Descriptor Types of Descriptor Records

The Apple Event Manager provides four routines that allow you to get the sizes and descriptor types of descriptor records that are not part of an Apple event record. The AESizeOfNthItem function returns the size and descriptor type of a descriptor record in a descriptor list. The AESizeOfKeyDesc function returns the size and descriptor type of a keyword-specified descriptor record in an AE record. You can get the size and descriptor type of an Apple event parameter or Apple event attribute using the AESizeOfParam and AESizeOfAttribute functions.

AESizeOfNthItem

You can use the `AESizeOfNthItem` function to get the size and descriptor type of a descriptor record in a descriptor list.

```
FUNCTION AESizeOfNthItem (theAEDescList: AEDescList;
                          index: LongInt; VAR typeCode: DescType;
                          VAR dataSize: Size): OSErr;
```

theAEDescList
            The descriptor list containing the descriptor record.

index   The position of the descriptor record in the list (for example, 2 specifies
            the second descriptor record).

typeCode   The descriptor type of the descriptor record.

dataSize   The length (in bytes) of the data in the descriptor record.

RESULT CODES

noErr	0	No error
errAEDescNotFound	–1701	Descriptor record was not found
errAEReplyNotArrived	–1718	Reply has not yet arrived

AESizeOfKeyDesc

You can use the `AESizeOfKeyDesc` function to get the size and descriptor type of a keyword-specified descriptor record in an AE record.

```
FUNCTION AESizeOfKeyDesc (theAERecord: AERecord;
                          theAEKeyword: AEKeyword;
                          VAR typeCode: DescType;
                          VAR dataSize: Size): OSErr;
```

theAERecord
            The AE record containing the desired keyword-specified descriptor
            record.

theAEKeyword
            The keyword that specifies the desired descriptor record.

typeCode   The descriptor type of the keyword-specified descriptor record.

dataSize   The length, in bytes of the data in the keyword-specified descriptor record.

RESULT CODES

noErr	0	No error
errAEDescNotFound	–1701	Descriptor record was not found
errAENotAEDesc	–1704	Not a valid descriptor record
errAEReplyNotArrived	–1718	Reply has not yet arrived

AESizeOfParam

You can use the AESizeOfParam function to get the size and descriptor type of an Apple event parameter.

```
FUNCTION AESizeOfParam (theAppleEvent: AppleEvent; theAEKeyword:
                        AEKeyword; VAR typeCode: DescType;
                        VAR dataSize: Size): OSErr;
```

theAppleEvent
            The Apple event containing the parameter.

theAEKeyword
            The keyword that specifies the desired parameter.

typeCode   The descriptor type of the Apple event parameter.

dataSize   The length, in bytes, of the data in the Apple event parameter.

RESULT CODES

noErr	0	No error
errAEDescNotFound	–1701	Descriptor record was not found
errAENotAEDesc	–1704	Not a valid descriptor record
errAEReplyNotArrived	–1718	Reply has not yet arrived

AESizeOfAttribute

You can use the AESizeOfAttribute function to get the size and descriptor type of an Apple event attribute.

```
FUNCTION AESizeOfAttribute (theAppleEvent: AppleEvent;
                            theAEKeyword: AEKeyword;
                            VAR typeCode: DescType;
                            VAR dataSize: Size): OSErr;
```

theAppleEvent
            The Apple event containing the desired attribute.

theAEKeyword
:   The keyword that specifies the attribute.

typeCode   The descriptor type of the attribute.

dataSize   The length, in bytes, of the data in the attribute.

RESULT CODES

noErr	0	No error
errAEDescNotFound	–1701	Descriptor record was not found
errAENotAEDesc	–1704	Not a valid descriptor record
errAEReplyNotArrived	–1718	Reply has not yet arrived

Deleting Descriptor Records

The Apple Event Manager provides three functions that allow you to delete descriptor records. The AEDeleteItem, AEDeleteKeyDesc, and AEDeleteParam functions allow you to delete descriptor records from a descriptor list, an AE record, and an Apple event parameter, respectively.

AEDeleteItem

You can use the AEDeleteItem function to delete a descriptor record from a descriptor list. All subsequent descriptor records will then move up one place.

```
FUNCTION AEDeleteItem (theAEDescList: AEDescList;
                       index: LongInt): OSErr;
```

theAEDescList
:   The descriptor list containing the descriptor record to be deleted.

index   The position of the descriptor record to delete (for example, 2 specifies the second item).

RESULT CODES

noErr	0	No error
errAEDescNotFound	–1701	Descriptor record was not found
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed

AEDeleteKeyDesc

You can use the `AEDeleteKeyDesc` function to delete a keyword-specified descriptor record from an AE record.

```
FUNCTION AEDeleteKeyDesc (theAERecord: AERecord;
                          theAEKeyword: AEKeyword): OSErr;
```

theAERecord
> The AE record containing the keyword-specified descriptor record to be deleted.

theAEKeyword
> The keyword that specifies the descriptor record to be deleted.

RESULT CODES

noErr	0	No error
errAEDescNotFound	–1701	Descriptor record was not found
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed

AEDeleteParam

You can use the `AEDeleteParam` function to delete an Apple event parameter.

```
FUNCTION AEDeleteParam (theAppleEvent: AppleEvent;
                        theAEKeyword: AEKeyword): OSErr;
```

theAppleEvent
> The Apple event containing the parameter to be deleted.

theAEKeyword
> The keyword that specifies the parameter to be deleted.

RESULT CODES

noErr	0	No error
errAEDescNotFound	–1701	Descriptor record was not found
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed

Deallocating Memory for Descriptor Records

The `AEDisposeDesc` function deallocates the memory used by a descriptor record. Because all Apple event structures (except for keyword-specified descriptor records) are descriptor records, you can use `AEDisposeDesc` for any of them.

AEDisposeDesc

You can use the `AEDisposeDesc` function to deallocate the memory used by a descriptor record.

```
FUNCTION AEDisposeDesc (VAR theAEDesc: AEDesc): OSErr;
```

theAEDesc   The descriptor record to deallocate. The function returns a null descriptor record in this parameter. If you pass a null descriptor record in this parameter, `AEDisposeDesc` returns `noErr`.

RESULT CODE

noErr   **0**   No error

SEE ALSO

For more information about using `AEDisposeDesc`, see "Disposing of Apple Event Data Structures," which begins on page 4-39.

Coercing Descriptor Types

The Apple Event Manager provides two functions that allow you to coerce descriptor types. The `AECoercePtr` function takes a pointer to data and a desired descriptor type and attempts to coerce the data to a descriptor record of the desired descriptor type. The `AECoerceDesc` function attempts to coerce the data in an existing descriptor record to another descriptor type.

AECoercePtr

You can use the `AECoercePtr` function to coerce data to a desired descriptor type. If successful, it creates a descriptor record containing the newly coerced data.

```
FUNCTION AECoercePtr (typeCode: DescType; dataPtr: Ptr;
                      dataSize: Size; toType: DescType;
                      VAR result: AEDesc): OSErr;
```

typeCode   The descriptor type of the source data.
dataPtr   A pointer to the data to be coerced.
dataSize   The length, in bytes, of the data to be coerced.
toType   The desired descriptor type of the resulting descriptor record.
result   The resulting descriptor record.

DESCRIPTION

The AECoercePtr function creates a new descriptor record by coercing the specified data to a descriptor record of the specified descriptor type. You should use the AEDisposeDesc function to dispose of the resulting descriptor record once you are finished using it.

If AECoercePtr returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to the requested descriptor type

SEE ALSO

For a description of the AEDisposeDesc function, see page 4-94.

AECoerceDesc

You can use the AECoerceDesc function to coerce the data in a descriptor record to another descriptor type.

```
FUNCTION AECoerceDesc (theAEDesc: AEDesc; toType: DescType;
                        VAR result: AEDesc): OSErr;
```

theAEDesc   The descriptor record whose data is to be coerced.

toType   The desired descriptor type of the resulting descriptor record.

result   The resulting descriptor record.

DESCRIPTION

The AECoerceDesc function attempts to create a new descriptor record by coercing the specified descriptor record. Your application is responsible for using the AEDisposeDesc function to dispose of the resulting descriptor record once you are finished using it.

If AECoerceDesc returns a nonzero result code, it returns a null descriptor record (a descriptor record of type typeNull, which does not contain any data) unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

noErr	0	No error
memFullErr	−108	Not enough room in heap zone
errAECoercionFail	−1700	Data could not be coerced to requested descriptor type

SEE ALSO

For a list of the descriptor types for which the Apple Event Manager provides coercions, see Table 4-1, which begins on page 4-43.

Creating and Managing the Coercion Handler Dispatch Tables

The Apple Event Manager provides three functions that allow you to create and manage the coercion handler dispatch tables. The AEInstallCoercionHandler function installs a coercion handler routine in either the application or system coercion dispatch table. The AEGetCoercionHandler function returns the handler for a specified descriptor type coercion. The AERemoveCoercionHandler function removes a coercion handler from either the application or system coercion table.

AEInstallCoercionHandler

You can use the AEInstallCoercionHandler function to install a coercion handler routine in either the application or system coercion handler dispatch table.

```
FUNCTION AEInstallCoercionHandler (fromType: DescType;
                                   toType: DescType;
                                   handler: ProcPtr;
                                   handlerRefcon: LongInt;
                                   fromTypeIsDesc: Boolean;
                                   isSysHandler: Boolean): OSErr;
```

fromType   The descriptor type of the data coerced by the handler.

toType   The descriptor type of the resulting data. If there was already an entry in the specified coercion handler table for the same source descriptor type and result descriptor type, the existing entry is replaced.

handler   A pointer to the coercion handler. Note that a handler in the system coercion table must reside in the system heap; thus, if the value of the isSysHandler parameter is TRUE, the handler parameter should point to a location in the system heap. Otherwise, if you put your system handler code in your application heap, you should use AERemoveCoercionHandler to remove the handler when your application quits.

handlerRefcon

> A reference constant passed by the Apple Event Manager to the handler each time the handler is called. If your handler doesn't expect a reference constant, use 0 as the value of this parameter.

fromTypeIsDesc

> Specifies the form of the data to be coerced. If the value of this parameter is TRUE, the coercion handler expects the data to be passed as a descriptor record. If the value is FALSE, the coercion handler expects a pointer to the data. Because it is more efficient for the Apple Event Manager to provide a pointer to data than to a descriptor record, all coercion routines should accept a pointer to data if possible.

isSysHandler

> Specifies the coercion table to which the handler is added. If the value of this parameter is TRUE, the handler is added to the system coercion table and made available to all applications. If the value is FALSE, the handler is added to the application coercion table. Note that a handler in the system coercion table must reside in the system heap; thus, if the value of the isSysHandler parameter is TRUE, the handler parameter must point to a location in the system heap.

DESCRIPTION

Before using AEInstallCoercionHandler to install a handler for a particular descriptor type into the system coercion handler dispatch table, use the AEGetCoercionHandler function to determine whether the table already contains a coercion handler for that descriptor type. If an entry exists, AEGetCoercionHandler returns a reference constant and a pointer to that handler. Chain these to your coercion handler by providing, in the handlerRefcon parameter of AEInstallCoercionHandler, pointers to the previous handler and its reference constant. If your coercion handler returns the error errAECoercionFail, use these pointers to call the previous handler. If you remove your system coercion handler, be sure to reinstall the chained handlers.

SPECIAL CONSIDERATIONS

Before an application calls a system coercion handler, system software has set up the A5 register for the calling application. For this reason, if you provide a system coercion handler, it should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system handler may crash.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone

AEGetCoercionHandler

You can use the AEGetCoercionHandler function to get the handler for a specified descriptor type coercion.

```
FUNCTION AEGetCoercionHandler (fromType: DescType;
                               toType: DescType;
                               VAR handler: ProcPtr;
                               VAR handlerRefcon: LongInt;
                               VAR fromTypeIsDesc: Boolean;
                               isSysHandler: Boolean): OSErr;
```

fromType   The descriptor type of the data coerced by the handler.

toType   The descriptor type of the resulting data.

handler   A pointer to the desired coercion handler.

handlerRefcon
               The reference constant for the desired handler. The Apple Event Manager
               passes this reference constant to the handler each time the handler
               is called.

fromTypeIsDesc
               If the AEGetCoercionHandler function returns TRUE in this parameter,
               the coercion handler expects the data to be passed as a descriptor record.
               If the function returns FALSE, the coercion handler expects a pointer to
               the data.

isSysHandler
               Specifies the coercion table from which to get the handler. If the value of
               this parameter is TRUE, the handler is taken from the system coercion
               table. If the value is FALSE, the handler is taken from the application
               coercion table.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAEHandlerNotFound	–1717	No coercion handler found

AERemoveCoercionHandler

You can use the AERemoveCoercionHandler function to remove a coercion handler from either the application or system coercion handler dispatch table.

```
FUNCTION AERemoveCoercionHandler (fromType: DescType;
                                  toType: DescType;
                                  handler: ProcPtr;
                                  isSysHandler: Boolean): OSErr;
```

fromType   The descriptor type of the data coerced by the handler.

toType   The descriptor type of the resulting data.

handler   A pointer to the coercion handler. Although the fromType and toType parameters would be sufficient to identify the handler to be removed, providing the handler parameter is a safeguard to ensure that you remove the correct handler.

isSysHandler
                The coercion table from which to remove the handler. If the value of this parameter is TRUE, the handler is removed from the system coercion table. If the value is FALSE, the handler is removed from the application coercion dispatch table.

RESULT CODES

noErr	0	No error
memFullErr	−108	Not enough room in heap zone
errAEHandlerNotFound	−1717	No coercion handler found

Creating and Managing the Special Handler Dispatch Tables

The Apple Event Manager provides three functions that allow you to create and manage the special handler dispatch tables. The AEInstallSpecialHandler function installs an entry for a special handler in either the application or system special handler dispatch table. The AEGetSpecialHandler function returns the handler for a specified special handler. The AERemoveSpecialHandler function removes a special handler from either the application or system special handler dispatch table.

You can also use the AEInstallSpecialHandler, AEGetSpecialHandler, and AERemoveSpecialHandler functions to install, get, and remove object callback functions—including system object callback functions, which cannot be installed with the AESetObjectCallbacks function. When calling any of these three functions, use one of the following constants as the value of the functionClass parameter to specify the object callback function:

Object callback function	Constant
Object-counting function	keyAECountProc
Object-comparison function	keyAECompareProc
Token disposal function	keyDiposeTokenProc
Error callback function	keyAEGetErrDescProc
Mark token function	keyAEMarkTokenProc
Object-marking function	keyAEMarkProc
Mark-adjusting function	keyAEAdjustMarksProc

You can also use the AERemoveSpecialHandler function to disable all the Apple Event Manager routines that support object specifier records. To do this, specify the constant keySelectProc in the functionClass parameter as described on page 4-102.

AEInstallSpecialHandler

You can use the AEInstallSpecialHandler function to install a special handler in either the application or system special handler dispatch table.

```
FUNCTION AEInstallSpecialHandler (functionClass: AEKeyword;
                                  handler: ProcPtr;
                                  isSysHandler: Boolean): OSErr;
```

functionClass
    The keyword for the special handler that is installed. The keyPreDispatch constant identifies a handler with the same parameters as an Apple event handler called immediately before the Apple Event Manager dispatches an Apple event. Any of the constants for object callback functions listed above can also be specified in this parameter. If there was already an entry in the specified special handler dispatch table for the same value of functionClass, it is replaced.

handler   A pointer to the special handler. Note that a handler in the system special handler dispatch table must reside in the system heap; thus, if the value of the isSysHandler parameter is TRUE, the handler parameter should point to a location in the system heap. Otherwise, if you put your system handler code in your application heap, use AERemoveSpecialHandler to remove the handler when your application quits.

isSysHandler

The special handler dispatch table to which to add the handler. If the value of this parameter is TRUE, the handler is added to the system handler dispatch table and made available to all applications. If the value is FALSE, the handler is added to the application handler table.

DESCRIPTION

The AEInstallSpecialHandler function creates an entry in either your application's special handler dispatch table or the system special handler dispatch table. You must supply parameters that specify the keyword for the special handler that is installed, the handler routine, and whether the handler is to be added to the system special handler dispatch table or your application's special handler dispatch table.

SPECIAL CONSIDERATIONS

Before an application calls a system special handler, system software has set up the A5 register for the calling application. For this reason, a system special handler should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system handler may crash.

RESULT CODES

noErr	0	No error
paramErr	–50	Parameter error (handler pointer is NIL or odd)
memFullErr	–108	Not enough room in heap zone
errAENotASpecialFunction	–1714	Wrong keyword for a special function

AEGetSpecialHandler

You can use the AEGetSpecialHandler function to get a specified special handler.

```
FUNCTION AEGetSpecialHandler (functionClass: AEKeyword;
                              VAR handler: ProcPtr;
                              isSysHandler: Boolean): OSErr;
```

functionClass

The keyword for the special handler that is installed. The keyPreDispatch constant identifies a handler with the same parameters as an Apple event handler that is called immediately before the Apple Event Manager dispatches an Apple event. Any of the constants for object callback functions listed on page 4-100 can also be specified in this parameter.

handler   A pointer to the special handler.

isSysHandler

Specifies the special handler dispatch table from which to get the handler. If the value of this parameter is TRUE, the handler is taken from the system special handler dispatch table. If the value is FALSE, the handler is taken from the application's special handler dispatch table.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAENotASpecialFunction	–1714	Wrong keyword for a special handler

AERemoveSpecialHandler

You can use the AERemoveSpecialHandler function to remove a handler from a special handler table.

```
FUNCTION AERemoveSpecialHandler (functionClass: AEKeyword;
                                handler: ProcPtr;
                                isSysHandler: Boolean): OSErr;
```

functionClass

The keyword for the special handler to be removed. In addition to the constants for object callback functions listed on page 4-100, two other values are allowed for the functionClass parameter: keyPreDispatch and keySelectProc. The keyPreDispatch constant identifies a handler with the same parameters as an Apple event handler that is called immediately before the Apple Event Manager dispatches an Apple event. The keySelectProc constant indicates that you want to disable the Object Support Library—that is, all the routines described in the chapter "Resolving and Creating Object Specifier Records" in this book (see the description that follows for more information).

handler   A pointer to the special handler to be removed. Although the functionClass parameter would be sufficient to identify the handler to be removed, providing the handler parameter is a safeguard that you remove the correct handler.

isSysHandler

Specifies the special handler dispatch table from which to remove the handler. If the value of this parameter is TRUE, the handler is taken from the system special handler dispatch table. If the value is FALSE, the handler is removed from the application special handler dispatch table.

DESCRIPTION

In addition to using the `AERemoveSpecialHandler` function to remove specific special handlers, you can use the function to disable, within your application only, all Apple Event Manager routines that support Apple event objects—that is, all the routines available to your application as a result of linking the Object Support Library (OSL) and calling the `AEObjectInit` function.

An application that expects its copy of the OSL to move after it is installed—for example, an application that keeps it in a stand-alone code resource—would need to do this. When an application calls `AEObjectInit` to initialize the OSL, the OSL installs the addresses of its routines as extensions to the pack. If those routines move, the addresses become invalid.

To disable the OSL, you should pass the keyword `keySelectProc` in the `functionClass` parameter, `NIL` in the `handler` parameter, and `FALSE` in the `isSysHandler` parameter. Once you have called the `AERemoveSpecialHandler` function with these parameters, subsequent calls by your application to any of the Apple Event Manager routines that support Apple event objects will return errors. To initialize the OSL after disabling it with the `AERemoveSpecialHandler` function, your application must call `AEObjectInit` again.

If you expect to initialize the OSL and disable it several times, you should call `AERemoveObjectAccessor` to remove your application's object accessor functions from your application's object accessor dispatch table before you call `AERemoveSpecialHandler`.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAENotASpecialFunction	–1714	Wrong keyword for a special function

Getting Information About the Apple Event Manager

The `AEManagerInfo` routine allows you to get two kinds of information related to Apple events on the current computer: the number of processes currently recording Apple events and the version of the Apple Event Manager. If you decide to make your application recordable, this information may be useful when your application is responding to Apple events that it sends to itself.

You can find out whether the Apple Event Manager is available in system software by using the `Gestalt` function. See page 4-4 for details.

AEManagerInfo

You can use the AEManagerInfo function to obtain information about the version of the Apple Event Manager currently available or the number of processes that are currently recording Apple events. This function is available only in version 1.01 and later versions of the Apple Event Manager.

```
FUNCTION AEManagerInfo (keyword: AEKeyword;
                        VAR result: LongInt): OSErr;
```

keyword   A value that determines what kind of information AEManagerInfo returns. The value can be represented by one of these constants:

```
CONST keyAERecorderCount        = 'recr';
      keyAEVersion              = 'vers';
```

result   If the value of the keyword parameter is keyAERecorderCount, this parameter is an integer that indicates the number of processes that are currently recording Apple events. If the value of the keyword parameter is keyAEVersion, this parameter is an integer that provides information about the version of the Apple Event Manager available on the current computer, using the same format as a 'vers' resource.

RESULT CODE

noErr   **0**   No error

SEE ALSO

For information about using the AEManagerInfo function to check whether Apple event recording is on or not, see the chapter "Recording Apple Events" in this book.

For information about using Gestalt to determine whether the Apple Event Manager is available, see "Handling Apple Events" on page 4-4.

For information about the 'vers' resource, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials.*

Application-Defined Routines

For each Apple event your application supports, you must provide an Apple event handler. The AEProcessAppleEvent function calls one of your Apple event handlers when it processes an Apple event. An Apple event handler (MyEventHandler) should perform any action described by the Apple event, add parameters to the reply Apple event if appropriate, and return a result code.

You can also provide your own coercion handlers to coerce data to descriptor types other than those for which the Apple Event Manager provides coercion handling. The MyCoercePtr function accepts a pointer to data and returns a descriptor record, and the MyCoerceDesc function accepts a descriptor record and returns a descriptor record.

MyEventHandler

An Apple event handler has the following syntax:

```
FUNCTION MyEventHandler (theAppleEvent: AppleEvent;
                         reply: AppleEvent;
                         handlerRefcon: LongInt): OSErr;
```

theAppleEvent
: The Apple event to handle.

reply
: The default reply Apple event provided by the Apple Event Manager.

handlerRefcon
: The reference constant stored in the Apple event dispatch table for the Apple event.

DESCRIPTION

An Apple event handler should extract any parameters and attributes from the Apple event, perform the requested action, and add parameters to the reply Apple event if appropriate.

Your handler should always set its function result to `noErr` if it successfully handles the Apple event. If an error occurs, your handler should return either `errAEEventNotHandled` or some other nonzero result code. If the error occurs because your application cannot understand the event, return `errAEEventNotHandled`, in case a handler in the system special handler or system Apple event dispatch tables might be able to handle the event. If the error occurs because the event is impossible to handle as specified, return the result code returned by whatever function caused the failure, or whatever other result code is appropriate.

For example, suppose your application receives a Get Data event that requests the name of the current printer, and your application cannot handle such an event. In this situation, you should return `errAEEventNotHandled` in case another handler available to the Apple Event Manager can handle the event. This strategy allows users to take advantage of system capabilities from within your application via system handlers.

However, if your application cannot handle a Get Data event that requests the fifth paragraph in a document because the document contains only four paragraphs, you should return some other nonzero error, because further attempts to handle the event are pointless.

If your Apple event handler calls the `AEResolve` function and `AEResolve` calls an object accessor function in the system object accessor dispatch table, your Apple event handler may not recognize the descriptor type of the token returned by the function. In this case, your handler should return the result code `errAEUnknownObjectType`. When your handler returns this result code, the Apple Event Manager attempts to locate a system Apple event handler that can recognize the token.

For more information about Apple event handlers, see "Writing Apple Event Handlers" on page 4-33.

For a discussion of the dispatching of object accessor functions and the use of the result code errAEUnknownObjectType, see "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

MyCoercePtr

A coercion handler that accepts a pointer to data has the following syntax:

```
FUNCTION MyCoercePtr (typeCode: DescType; dataPtr: Ptr;
                      dataSize: Size; toType: DescType;
                      handlerRefcon: LongInt;
                      VAR result: AEDesc): OSErr;
```

typeCode   The descriptor type of the original data.

dataPtr   A pointer to the data to coerce.

dataSize   The length, in bytes, of the data to coerce.

toType   The desired descriptor type for the resulting descriptor record.

handlerRefcon
                A reference constant that is stored in the coercion dispatch table entry for the handler and passed to the handler by the Apple Event Manager whenever the handler is called.

result   The resulting descriptor record.

DESCRIPTION

Your coercion handler should coerce the data to the desired descriptor type and return the resulting data in the descriptor record specified by the result parameter. Your handler should return the noErr result code if your handler successfully performs the coercion, and a nonzero result code otherwise.

SEE ALSO

For more information, see "Writing and Installing Coercion Handlers" on page 4-41.

MyCoerceDesc

A coercion handler that accepts a descriptor record has the following syntax:

```
FUNCTION MyCoerceDesc (theAEDesc: AEDesc; toType: DescType;
                       handlerRefcon: LongInt;
                       VAR result: AEDesc): OSErr;
```

theAEDesc   The descriptor record that contains the data to be coerced.

toType   The desired descriptor type for the resulting descriptor record.

handlerRefcon
             A reference constant that is stored in the coercion dispatch table entry for the handler and passed to the handler by the Apple Event Manager whenever the handler is called.

result   The resulting descriptor record.

DESCRIPTION

Your coercion handler should coerce the data in the descriptor record to the desired descriptor type and return the resulting data in the descriptor record specified by the result parameter. Your handler should return an appropriate result code.

SEE ALSO

For more information, see "Writing and Installing Coercion Handlers" on page 4-41.

Summary of Responding to Apple Events

Pascal Summary

Constants

```
CONST
  gestaltAppleEventsAttr      = 'evnt';    {selector for Apple events}
  gestaltAppleEventsPresent  = 0;          {if this bit is set, then Apple }
                                           { Event Manager is available}

  {Apple event descriptor types}
  typeBoolean              = 'bool';     {1-byte Boolean value}
  typeChar                 = 'TEXT';     {unterminated string}
  typeSMInt                = 'shor';     {16-bit integer}
  typeInteger              = 'long';     {32-bit integer}
  typeSMFloat              = 'sing';     {SANE single}
  typeFloat                = 'doub';     {SANE double}
  typeLongInteger          = 'long';     {32-bit integer}
  typeShortInteger         = 'shor';     {16-bit integer}
  typeLongFloat            = 'doub';     {SANE double}
  typeShortFloat           = 'sing';     {SANE single}
  typeExtended             = 'exte';     {SANE extended}
  typeComp                 = 'comp';     {SANE comp}
  typeMagnitude            = 'magn';     {unsigned 32-bit integer}
  typeAEList               = 'list';     {list of descriptor records}
  typeAERecord             = 'reco';     {list of keyword-specified }
                                         { descriptor records}
  typeAppleEvent           = 'aevt';     {Apple event record}
  typeTrue                 = 'true';     {TRUE Boolean value}
  typeFalse                = 'fals';     {FALSE Boolean value}
  typeAlias                = 'alis';     {alias record}
  typeEnumerated           = 'enum';     {enumerated data}
  typeType                 = 'type';     {four-character code for }
                                         { event class or event ID}
  typeAppParameters        = 'appa';     {Process Manager launch parameters}
  typeProperty             = 'prop';     {Apple event property}
  typeFSS                  = 'fss ';     {file system specification}
```

```
typeKeyword            = 'keyw';      {Apple event keyword}
typeSectionH           = 'sect';      {handle to a section record}
typeWildCard           = '****';      {matches any type}
typeApplSignature      = 'sign';      {application signature}
typeSessionID          = 'ssid';      {session reference number}
typeTargetID           = 'targ';      {target ID record}
typeProcessSerialNumber = 'psn ';     {process serial number}
typeNull               = 'null';      {NULL or nonexistent data}

{keywords for Apple event parameters}
keyDirectObject        = '----';      {direct parameter}
keyErrorNumber         = 'errn';      {error number parameter}
keyErrorString         = 'errs';      {error string parameter}
keyProcessSerialNumber = 'psn ';      {process serial number param}

{keywords for Apple event attributes}
keyTransactionIDAttr   = 'tran';      {transaction ID}
keyReturnIDAttr        = 'rtid';      {return ID}
keyEventClassAttr      = 'evcl';      {event class}
keyEventIDAttr         = 'evid';      {event ID}
keyAddressAttr         = 'addr';      {address of target or }
                                      { client application}
keyOptionalKeywordAttr = 'optk';      {list of optional parameters }
                                      { for the Apple event}
keyTimeoutAttr         = 'timo';      {number of ticks the client }
                                      { will wait}
keyInteractLevelAttr   = 'inte';      {settings to allow Apple Event }
                                      { Manager to bring server }
                                      { to foreground}
keyEventSourceAttr     = 'esrc';      {nature of source }
                                      { application}
keyMissedKeywordAttr   = 'miss';      {first required parameter }
                                      { remaining in an Apple event}
keyOriginalAddressAttr = 'from';      {address of original source; }
                                      { available only in version }
                                      { 1.01 and later versions of }
                                      { the Apple Event Manager}

{keywords for special handlers}
keyPreDispatch         = 'phac';      {identifies a handler routine }
                                      { called immediately before the }
                                      { Apple Event Manager dispatches }
                                      { an Apple event}
```

```
keySelectProc              = 'selh';        {selector used with }
                                            { AERemoveSpecialHandler to }
                                            { disable the OSL}


{keywords for use with AEManagerInfo; available only in version }
{ 1.0.1 and later versions of the Apple Event Manager}
keyAERecorderCount         = 'recr';        {keyword for recording info}
keyAEVersion               = 'vers';        {keyword for version info}


{event class}
kCoreEventClass            = 'aevt';        {event class for required Apple }
                                            { events}


{event IDs for required Apple events}
kAEOpenApplication         = 'oapp';        {event ID for Open }
                                            { Application event}
kAEOpenDocuments           = 'odoc';        {event ID for Open Documents event}
kAEPrintDocuments          = 'pdoc';        {event ID for Print Documents }
                                            { event}
kAEQuitApplication         = 'quit';        {event ID for Quit Application }
                                            { event}
kAEAnswer                  = 'ansr';        {event ID for Apple event replies}
kAEApplicationDied         = 'obit';        {event ID for Application Died }
                                            { event}


{constants for setting the sendMode parameter of AESend}
kAENoReply                 = $00000001;     {client doesn't want reply}
kAEQueueReply              = $00000002;     {client wants server to }
                                            { reply in event queue}
kAEWaitReply               = $00000003;     {client wants a reply and }
                                            { will give up processor}
kAENeverInteract           = $00000010;     {server application should }
                                            { not interact with user }
                                            { for this Apple event}
kAECanInteract             = $00000020;     {server may interact with }
                                            { user for this Apple event }
                                            { to supply information}

kAEAlwaysInteract          = $00000030;     {server may interact with user }
                                            { for this Apple event even if }
                                            { no information is required}
kAECanSwitchLayer          = $00000040;     {server should come directly }
                                            { to foreground when appropriate}
```

```
kAEDontReconnect        = $00000080;   {don't reconnect if there }
                                       { is a PPC session closed error}
kAEWantReceipt          = nReturnReceipt; {client wants return }
                                       { receipt}
kAEDontRecord           = $00001000;   {don't record this event}
kAEDontExecute          = $00002000;   {don't excecute this event}


{constants for setting the sendPriority parameter of AESend}
kAENormalPriority       = $00000000;   {put event at the back of }
                                       { event queue}
kAEHighPriority         = nAttnMsg;    {put event at the front of }
                                       { the event queue}


{event IDs for recording events; available only in version 1.01 and }
{ later versions of the Apple Event Manager}
kAEStartRecording       = 'reca';      {event ID for Start Recording }
                                       { event}
kAEStopRecording        = 'recc';      {event ID for Stop Recording }
                                       { event}
kAENotifyStartRecording = 'rec1';      {event ID for Recording On event}
kAENotifyStopRecording  = 'rec0';      {event ID for Recording Off event}
kAENotifyRecording      = 'recr';      {event ID for Receive Recordable }
                                       { Event event}


{constant for the returnID parameter of AECreateAppleEvent}
kAutoGenerateReturnID   = -1;          {tells Apple Event Manager to }
                                       { generate a unique return ID}


{constant for transaction IDs}
kAnyTransactionID       = 0;           {the Apple event is not }
                                       { part of a transaction}


{constants for timeout durations}
kAEDefaultTimeout       = -1;          {use default timeout value}
kNoTimeOut              = -2;          {never time out}


{constants for the dispatcher parameter of AEResumeTheCurrentEvent}
kAENoDispatch           = 0;           {don't redispatch the Apple event}
kAEUseStandardDispatch  = -1;          {redispatch the Apple event }
                                       { by using its entry in the }
                                       { Apple event dispatch table}
```

Data Types

```
TYPE
  AEEventClass =
    PACKED ARRAY[1..4] OF Char;              {event class for a high-level }
                                             { event}
  AEEventID =
    PACKED ARRAY[1..4] OF Char;              {event ID for a high-level }
                                             { event}

  AEKeyword =
    PACKED ARRAY[1..4] OF Char;              {keyword for a descriptor }
                                             { record}

  DescType                 = ResType;        {descriptor type}

  AEDesc =                                   {descriptor record}
  RECORD
    descriptorType:        DescType;         {type of data being passed}
    dataHandle:            Handle;           {handle to data being passed}
  END;

  AEKeyDesc =                                {keyword-specified }
  RECORD                                     { descriptor record}
    descKey:               AEKeyword;        {keyword}
    descContent:           AEDesc;           {descriptor record}
  END;

  AEAddressDesc            = AEDesc;         {address descriptor record}

  AEDescList              = AEDesc;         {list of descriptor records}

  AERecord                = AEDescList;     {list of keyword-specified }
                                             { descriptor records}

  AppleEvent              = AERecord;       {list of attributes and }
                                             { parameters necessary for }
                                             { an Apple event}

  AESendMode              = LongInt;        {flags that determine how }
                                             { an Apple event is sent}

  AESendPriority          = Integer;        {send priority of an Apple }
                                             { event}
```

```
AEInteractAllowed = (kAEInteractWithSelf, kAEInteractWithLocal,
                      kAEInteractWithAll); {what processes may }
                                           { interact with the user}


AEEventSource = (kAEUnknownSource, kAEDirectCall, kAESameProcess,
                 kAELocalProcess, kAERemoteProcess);
                                           {the source of an Apple }
                                           { event}


AEArrayType = (kAEDataArray, kAEPackedArray, kAEHandleArray,
               kAEDescArray, kAEKeyDescArray);
                                           {type of an Apple event array}
AEArrayData =
RECORD                                     {data for an Apple event array}
   CASE AEArrayType OF
   kAEDataArray:
      (AEDataArray: ARRAY[0..0] OF Integer);
   kAEPackedArray:
      (AEPackedArray: PACKED ARRAY[0..0] OF Char);
   kAEHandleArray:
      (AEHandleArray: ARRAY[0..0] OF Handle);
   kAEDescArray:
      (AEDescArray: ARRAY[0..0] OF AEDesc);
   kAEKeyDescArray:
      (AEKeyDescArray: ARRAY[0..0] OF AEKeyDesc);
END;


AEArrayDataPointer = ^AEArrayData;


EventHandlerProcPtr = ProcPtr;            {pointer to an Apple event }
                                          { handler}
IdleProcPtr = ProcPtr;                    {pointer to an application's }
                                          { idle function}
EventFilterProcPtr = ProcPtr;             {pointer to an application's }
                                          { filter function}
```

Routines for Responding to Apple Events

Creating and Managing the Apple Event Dispatch Tables

```
FUNCTION AEInstallEventHandler
                              (theAEEventClass: AEEventClass;
                               theAEEventID: AEEventID;
                               handler: EventHandlerProcPtr;
                               handlerRefcon: LongInt;
                               isSysHandler: Boolean): OSErr;
FUNCTION AEGetEventHandler   (theAEEventClass: AEEventClass;
                               theAEEventID: AEEventID;
                               VAR handler: EventHandlerProcPtr;
                               VAR handlerRefcon: LongInt;
                               isSysHandler: Boolean): OSErr;
FUNCTION AERemoveEventHandler
                              (theAEEventClass: AEEventClass; theAEEventID:
                               AEEventID; handler: EventHandlerProcPtr;
                               isSysHandler: Boolean): OSErr;
```

Dispatching Apple Events

```
FUNCTION AEProcessAppleEvent
                              (theEventRecord: EventRecord): OSErr;
```

Getting Data or Descriptor Records Out of Apple Event Parameters and Attributes

```
FUNCTION AEGetParamPtr       (theAppleEvent: AppleEvent;
                               theAEKeyword: AEKeyword;
                               desiredType: DescType;
                               VAR typeCode: DescType;
                               dataPtr: Ptr; maximumSize: Size;
                               VAR actualSize: Size): OSErr;
FUNCTION AEGetParamDesc      (theAppleEvent: AppleEvent;
                               theAEKeyword: AEKeyword; desiredType: DescType;
                               VAR result: AEDesc): OSErr;
FUNCTION AEGetAttributePtr   (theAppleEvent: AppleEvent;
                               theAEKeyword: AEKeyword; desiredType: DescType;
                               VAR typeCode: DescType;
                               dataPtr: Ptr; maximumSize: Size;
                               VAR actualSize: Size): OSErr;
FUNCTION AEGetAttributeDesc  (theAppleEvent: AppleEvent;
                               theAEKeyword: AEKeyword; desiredType: DescType;
                               VAR result: AEDesc): OSErr;
```

Counting the Items in Descriptor Lists

```
FUNCTION AECountItems       (theAEDescList: AEDescList;
                             VAR theCount: LongInt): OSErr;
```

Getting Items From Descriptor Lists

```
FUNCTION AEGetNthPtr        (theAEDescList: AEDescList; index: LongInt;
                             desiredType: DescType;
                             VAR theAEKeyword: AEKeyword;
                             VAR typeCode: DescType; dataPtr: Ptr;
                             maximumSize: Size;
                             VAR actualSize: Size): OSErr;
FUNCTION AEGetNthDesc       (theAEDescList: AEDescList; index: LongInt;
                             desiredType: DescType;
                             VAR theAEKeyword: AEKeyword;
                             VAR result: AEDesc): OSErr;
FUNCTION AEGetArray         (theAEDescList: AEDescList;
                             arrayType: AEArrayType;
                             arrayPtr: AEArrayDataPointer;
                             maximumSize: Size;
                             VAR itemType: DescType; VAR itemSize: Size;
                             VAR itemCount: LongInt): OSErr;
```

Getting Data and Keyword-Specified Descriptor Records Out of AE Records

```
FUNCTION AEGetKeyPtr        (theAERecord: AERecord;
                             theAEKeyword: AEKeyword;
                             desiredType: DescType; VAR typeCode: DescType;
                             dataPtr: Ptr; maximumSize: Size;
                             VAR actualSize: Size): OSErr;
FUNCTION AEGetKeyDesc       (theAERecord: AERecord;
                             theAEKeyword: AEKeyword;
                             desiredType: DescType;
                             VAR result: AEDesc): OSErr;
```

Requesting User Interaction

```
FUNCTION AESetInteractionAllowed
                            (level: AEInteractAllowed): OSErr;
FUNCTION AEGetInteractionAllowed
                            (VAR level: AEInteractAllowed): OSErr;
FUNCTION AEInteractWithUser (timeOutInTicks: LongInt; nmReqPtr: NMRecPtr;
                             idleProc: IdleProcPtr): OSErr;
```

Requesting More Time to Respond to Apple Events

```
FUNCTION AEResetTimer          (reply: AppleEvent): OSErr;
```

Suspending and Resuming Apple Event Handling

```
FUNCTION AESuspendTheCurrentEvent
                               (theAppleEvent: AppleEvent): OSErr;
FUNCTION AEResumeTheCurrentEvent
                               (theAppleEvent, reply: AppleEvent;
                                dispatcher: EventHandlerProcPtr;
                                handlerRefcon: LongInt): OSErr;
FUNCTION AESetTheCurrentEvent
                               (theAppleEvent: AppleEvent): OSErr;
FUNCTION AEGetTheCurrentEvent
                               (VAR theAppleEvent: AppleEvent): OSErr;
```

Getting the Sizes and Descriptor Types of Descriptor Records

```
FUNCTION AESizeOfNthItem       (theAEDescList: AEDescList; index: LongInt;
                                VAR typeCode: DescType;
                                VAR dataSize: Size): OSErr;
FUNCTION AESizeOfKeyDesc       (theAERecord: AERecord;
                                theAEKeyword: AEKeyword;
                                VAR typeCode: DescType;
                                VAR dataSize: Size): OSErr;
FUNCTION AESizeOfParam         (theAppleEvent: AppleEvent;
                                theAEKeyword: AEKeyword;
                                VAR typeCode: DescType;
                                VAR dataSize: Size): OSErr;
FUNCTION AESizeOfAttribute     (theAppleEvent: AppleEvent;
                                theAEKeyword: AEKeyword;
                                VAR typeCode: DescType;
                                VAR dataSize: Size): OSErr;
```

Deleting Descriptor Records

```
FUNCTION AEDeleteItem          (theAEDescList: AEDescList;
                                index: LongInt): OSErr;
FUNCTION AEDeleteKeyDesc       (theAERecord: AERecord;
                                theAEKeyword: AEKeyword): OSErr;
FUNCTION AEDeleteParam         (theAppleEvent: AppleEvent;
                                theAEKeyword: AEKeyword): OSErr;
```

Deallocating Memory for Descriptor Records

```
FUNCTION AEDisposeDesc        (VAR theAEDesc: AEDesc): OSErr;
```

Coercing Descriptor Types

```
FUNCTION AECoercePtr          (typeCode: DescType; dataPtr: Ptr;
                               dataSize: Size; toType: DescType;
                               VAR result: AEDesc): OSErr;
FUNCTION AECoerceDesc         (theAEDesc: AEDesc; toType: DescType;
                               VAR result: AEDesc): OSErr;
```

Creating and Managing the Coercion Handler Dispatch Tables

```
FUNCTION AEInstallCoercionHandler
                              (fromType: DescType; toType: DescType;
                               handler: ProcPtr; handlerRefcon: LongInt;
                               fromTypeIsDesc: Boolean;
                               isSysHandler: Boolean): OSErr;
FUNCTION AEGetCoercionHandler
                              (fromType: DescType; toType: DescType;
                               VAR handler: ProcPtr;
                               VAR handlerRefcon: LongInt;
                               VAR fromTypeIsDesc: Boolean;
                               isSysHandler: Boolean): OSErr;
FUNCTION AERemoveCoercionHandler
                              (fromType: DescType; toType: DescType;
                               handler: ProcPtr;
                               isSysHandler: Boolean): OSErr;
```

Creating and Managing the Special Handler Dispatch Tables

```
FUNCTION AEInstallSpecialHandler
                              (functionClass: AEKeyword; handler: ProcPtr;
                               isSysHandler: Boolean): OSErr;
FUNCTION AEGetSpecialHandler
                              (functionClass: AEKeyword;
                               VAR handler: ProcPtr;
                               isSysHandler: Boolean): OSErr;
FUNCTION AERemoveSpecialHandler
                              (functionClass: AEKeyword; handler: ProcPtr;
                               isSysHandler: Boolean): OSErr;
```

Getting Information About the Apple Event Manager

```
{available only in version 1.01 and later versions of Apple Event Manager}

FUNCTION AEManagerInfo        (keyword: AEKeyword;
                               VAR result: LongInt): OSErr;
```

Application-Defined Routines

```
FUNCTION MyEventHandler       (theAppleEvent: AppleEvent; reply: AppleEvent;
                               handlerRefcon: LongInt): OSErr;
FUNCTION MyCoercePtr          (typeCode: DescType; dataPtr: Ptr;
                               dataSize: Size; toType: DescType;
                               handlerRefcon: LongInt;
                               VAR result: AEDesc): OSErr;
FUNCTION MyCoerceDesc         (theAEDesc: AEDesc; toType: DescType;
                               handlerRefcon: LongInt;
                               VAR result: AEDesc): OSErr;
```

C Summary

Constants

```
enum {
     #define gestaltAppleEventsAttr  'evnt' /*selector for Apple events*/
     gestaltAppleEventsPresent        = 0   /*if this bit is set, then */
                                            /* Apple Event Manager is */
};                                          /* available*/


     /*Apple event descriptor types*/
enum   {
     typeBoolean             = 'bool',      /*1-byte Boolean value*/
     typeChar                = 'TEXT',      /*unterminated string*/
     typeSMInt               = 'shor',      /*16-bit integer*/
     typeInteger             = 'long',      /*32-bit integer*/
     typeSMFloat             = 'sing',      /*SANE single*/
     typeFloat               = 'doub',      /*SANE double*/
     typeLongInteger         = 'long',      /*32-bit integer*/
     typeShortInteger        = 'shor',      /*16-bit integer*/
     typeLongFloat           = 'doub',      /*SANE double*/
     typeShortFloat          = 'sing',      /*SANE single*/
     typeExtended            = 'exte',      /*SANE extended*/
```

```
        typeComp                = 'comp',       /*SANE comp*/
        typeMagnitude           = 'magn',       /*unsigned 32-bit integer*/
        typeAEList              = 'list',       /*list of descriptor records*/
        typeAERecord            = 'reco',       /*list of keyword-specified */
                                                /* descriptor records*/
        typeAppleEvent          = 'aevt',       /*Apple event record*/
        typeTrue                = 'true',       /*TRUE Boolean value*/
        typeFalse               = 'fals',       /*FALSE Boolean value*/
        typeAlias               = 'alis',       /*alias record*/
        typeEnumerated          = 'enum'        /*enumerated data*/
};


enum {
        typeType                = 'type',       /*four-character code for */
                                                /* event class or event ID*/
        typeAppParameters       = 'appa',       /*Process Manager launch */
                                                /* parameters*/
        typeProperty            = 'prop',       /*Apple event property*/
        typeFSS                 = 'fss ',       /*file system specification*/
        typeKeyword             = 'keyw',       /*Apple event keyword*/

        typeSectionH            = 'sect',       /*handle to a section record*/
        typeWildCard            = '****',       /*matches any type*/
        typeApplSignature       = 'sign',       /*application signature*/
        typeSessionID           = 'ssid',       /*session ID*/
        typeTargetID            = 'targ',       /*target ID record*/
        typeProcessSerialNumber = 'psn ',       /*process serial number*/
        typeNull                = 'null'        /*NULL or nonexistent data*/
};


        /*keywords for Apple event parameters*/
enum {
        keyDirectObject         = '----',       /*direct parameter*/
        keyErrorNumber          = 'errn',       /*error number parameter*/
        keyErrorString          = 'errs',       /*error string parameter*/
        keyProcessSerialNumber  = 'psn '        /*process serial number param*/
};


        /*keywords for Apple event attributes*/
enum {
        keyTransactionIDAttr    = 'tran',       /*transaction ID*/
        keyReturnIDAttr         = 'rtid',       /*return ID*/
        keyEventClassAttr       = 'evcl',       /*event class*/
```

```
      keyEventIDAttr            = 'evid',      /*event ID*/
      keyAddressAttr            = 'addr',      /*address of target or */
                                              /* client application*/
      keyOptionalKeywordAttr = 'optk',        /*list of optional parameters */
                                              /* for the Apple event*/
      keyTimeoutAttr            = 'timo',      /*number of ticks the client */
                                              /* will wait*/
      keyInteractLevelAttr      = 'inte',      /*settings to allow Apple */
                                              /* Event Mgr to bring */
                                              /* server to foreground*/
      keyEventSourceAttr        = 'esrc',      /*nature of source */
                                              /* application*/
      keyMissedKeywordAttr      = 'miss',      /*first required parameter */
                                              /* remaining in an Apple */
                                              /* event*/
      keyOriginalAddressAttr = 'from'         /*address of original source; */
                                              /* available only in version */
                                              /* 1.01 and later versions of */
                                              /* the Apple Event Manager*/
};


      /*keywords for special handlers*/
enum {
      keyPreDispatch            = 'phac',      /*identifies a handler */
                                              /* routine that is called */
                                              /* immediately before the */
                                              /* Apple Event Manager */
                                              /* dispatches an Apple event*/
      keySelectProc             = 'selh',      /*selector used with */
                                              /* AERemoveSpecialHandler to */
                                              /* disable the OSL*/

      /*keywords for use with AEManagerInfo, available only in version */
      /* 1.0.1 and later versions of the Apple Event Manager*/
      keyAERecorderCount        = 'recr',      /*keyword for recording info*/
      keyAEVersion              = 'vers',      /*keyword for version info*/

      /*event class*/
      kCoreEventClass           = 'aevt'      /*event class for required */
                                              /* Apple events*/
};
```

```
      /*event IDs for required Apple events*/
enum {
      kAEOpenApplication       = 'oapp',      /*event ID for Open */
                                              /* Application event*/
      kAEOpenDocuments         = 'odoc',      /*event ID for Open */
                                              /* Documents event*/

      kAEPrintDocuments        = 'pdoc',      /*event ID for Print */
                                              /* Documents event*/
      kAEQuitApplication       = 'quit',      /*event ID for Quit */
                                              /* Application event*/
      kAEAnswer                = 'ansr',      /*event ID for Apple event */
                                              /* replies*/
      kAEApplicationDied       = 'obit'       /*event ID for Application */
                                              /* Died event*/
};
      /*constants for setting the sendMode parameter of AESend*/
enum {
      kAENoReply               = 0x00000001,  /*client doesn't want reply*/
      kAEQueueReply            = 0x00000002,  /*client wants server to */
                                              /* reply in event queue*/
      kAEWaitReply             = 0x00000003,  /*client wants a reply and */
                                              /* will give up processor*/
      kAENeverInteract         = 0x00000010,  /*server application should */
                                              /* not interact with user */
                                              /* for this Apple event*/
      kAECanInteract           = 0x00000020,  /*server may interact with */
                                              /* user for this Apple event */
                                              /* to supply information*/
      kAEAlwaysInteract        = 0x00000030,  /*server may interact with */
                                              /* user for this Apple event */
                                              /* even if no information */
                                              /* is required*/
      kAECanSwitchLayer        = 0x00000040,  /*server should come */
                                              /* directly to foreground */
                                              /* when appropriate*/
      kAEDontReconnect         = 0x00000080,  /*don't reconnect if there */
                                              /* is a PPC session closed */
                                              /* error*/
      kAEWantReceipt           = nReturnReceipt, /*client wants return */
                                              /* receipt*/
      kAEDontRecord            = 0x00001000,  /*don't record this event*/
      kAEDontExecute           = 0x00002000,  /*don't excecute this event*/
```

```
      /*constants for setting the sendPriority parameter of AESend*/
      kAENormalPriority        = 0x00000000,  /*post message at end of */
                                              /* event queue*/
      kAEHighPriority          = nAttnMsg      /*post message at front of */
                                              /* event queue*/
};


      /*event IDs for recording events; available only in version 1.01 and */
      /* later versions of the Apple Event Manager*/
enum {
      kAEStartRecording        = 'reca',      /*event ID for Start */
                                              /* Recording event*/
      kAEStopRecording         = 'recc',      /*event ID for Stop */
                                              /* Recording event*/
      kAENotifyStartRecording  = 'rec1',      /*event ID for Recording On*/
                                              /* event*/
      kAENotifyStopRecording   = 'rec0',      /*event ID for Recording Off */
                                              /* event*/
      kAENotifyRecording       = 'recr'       /*event ID for Receive */
                                              /* Recordable Event event*/
};
enum {
      /*constant for the returnID parameter of AECreateAppleEvent*/
      kAutoGenerateReturnID    = -1,          /*tells Apple Event Manager */
                                              /* to generate a unique */
                                              /* return ID*/

      /*constant for transaction IDs*/
      kAnyTransactionID        = 0,           /*the Apple event is not */
                                              /* part of a transaction*/

      /*constants for timeout durations*/
      kAEDefaultTimeout        = -1,          /*use default timeout value*/
      kNoTimeOut               = -2,          /*never time out*/

      /*constants for the dispatcher parameter of AEResumeTheCurrentEvent*/
      kAENoDispatch            = 0,           /*don't redispatch the */
                                              /* Apple event*/
      kAEUseStandardDispatch   = -1           /*redispatch the Apple event */
                                              /* by using its entry in the */
                                              /* Apple event dispatch table*/
};
```

Data Types

```
typedef unsigned long AEEventClass;         /*event class for a */
                                            /* high-level event*/
typedef unsigned long AEEventID;            /*event ID for a high-level */
                                            /* event*/

typedef unsigned long AEKeyword;            /*keyword for a descriptor */
                                            /* record*/
typedef ResType DescType;                   /*descriptor type*/

struct AEDesc {                             /*descriptor record*/
   DescType descriptorType;                 /*type of data being passed*/
   Handle dataHandle;                       /*handle to data being passed*/
};
typedef struct AEDesc AEDesc;

struct AEKeyDesc {                          /*keyword-specified */
                                            /* descriptor record*/

   AEKeyword descKey;                       /*keyword*/
   AEDesc descContent;                      /*descriptor record*/
};
typedef struct AEKeyDesc AEKeyDesc;

typedef AEDesc AEAddressDesc;               /*address descriptor record*/
typedef AEDesc AEDescList;                  /*list of descriptor records*/
typedef AEDescList AERecord;                /*list of keyword-specified */
                                            /* descriptor records*/
typedef AERecord AppleEvent;                /*list of attributes and */
                                            /* parameters necessary for */
                                            /* an Apple event*/

typedef long AESendMode;                    /*flags that determine how */
                                            /* an Apple event is sent*/

typedef short AESendPriority;               /*send priority of an Apple */
                                            /* event*/

enum { kAEInteractWithSelf, kAEInteractWithLocal,
      kAEInteractWithAll };                 /*what processes may */
typedef unsigned char AEInteractAllowed;    /* interact with the user*/
```

```
enum { kAEUnknownSource, kAEDirectCall, kAESameProcess, kAELocalProcess,
      kAERemoteProcess };                    /*the source of an Apple */
typedef unsigned char AEEventSource;         /* event*/
enum { kAEDataArray, kAEPackedArray, kAEHandleArray,
      kAEDescArray, kAEKeyDescArray };       /*type of an Apple event */
typedef unsigned char AEArrayType;           /* array*/

union AEArrayData {                                    /*data for an Apple event */
   short        kAEDataArray[1];                       /* array*/
   char         kAEPackedArray[1];
   Handle       kAEHandleArray[1];
   AEDesc       kAEDescArray[1];
   AEKeyDesc    kAEKeyDescArray[1];
};
typedef union AEArrayData AEArrayData;

typedef AEArrayData *AEArrayDataPointer;

typedef ProcPtr EventHandlerProcPtr;         /*pointer to an Apple event */
                                             /* handler*/
typedef ProcPtr IdleProcPtr;                 /*pointer to an application's */
                                             /* idle function*/
typedef ProcPtr EventFilterProcPtr;          /*pointer to an application's */
                                             /* filter function*/
```

Routines for Responding to Apple Events

Creating and Managing the Apple Event Dispatch Tables

```
pascal OSErr AEInstallEventHandler
                        (AEEventClass theAEEventClass,
                         AEEventID theAEEventID,
                         EventHandlerProcPtr handler,
                         long handlerRefcon, Boolean isSysHandler);
pascal OSErr AEGetEventHandler
                        (AEEventClass theAEEventClass,
                         AEEventID theAEEventID,
                         EventHandlerProcPtr *handler,
                         long *handlerRefcon, Boolean isSysHandler);
pascal OSErr AERemoveEventHandler
                        (AEEventClass theAEEventClass,
                         AEEventID theAEEventID,
                         EventHandlerProcPtr handler,
                         Boolean isSysHandler);
```

Dispatching Apple Events

```
pascal  OSErr  AEProcessAppleEvent
                              (const EventRecord *theEventRecord);
```

Getting Data or Descriptor Records Out of Apple Event Parameters and Attributes

```
pascal OSErr AEGetParamPtr  (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType desiredType,
                             DescType *typeCode, void* dataPtr,
                             Size maximumSize, Size *actualSize);
pascal OSErr AEGetParamDesc (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType desiredType,
                             AEDesc *result);
pascal OSErr AEGetAttributePtr
                            (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType desiredType,
                             DescType *typeCode, void* dataPtr,
                             Size maximumSize, Size *actualSize);
pascal OSErr AEGetAttributeDesc
                            (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType desiredType,
                             AEDesc *result);
```

Counting the Items in Descriptor Lists

```
pascal OSErr AECountItems   (const AEDescList *theAEDescList,
                             long *theCount);
```

Getting Items From Descriptor Lists

```
pascal OSErr AEGetNthPtr    (const AEDescList *theAEDescList, long index,
                             DescType desiredType, AEKeyword *theAEKeyword,
                             DescType *typeCode, void* dataPtr,
                             Size maximumSize, Size *actualSize);
pascal OSErr AEGetNthDesc   (const AEDescList *theAEDescList, long index,
                             DescType desiredType, AEKeyword *theAEKeyword,
                             AEDesc *result);
pascal OSErr AEGetArray     (const AEDescList *theAEDescList,
                             AEArrayType arrayType,
                             AEArrayDataPointer arrayPtr, Size maximumSize,
                             DescType *itemType, Size *itemSize,
                             long *itemCount);
```

Getting Data and Keyword-Specified Descriptor Records Out of AE Records

```
pascal OSErr AEGetKeyPtr      (const AERecord *theAERecord,
                               AEKeyword theAEKeyword, DescType desiredType,
                               DescType *typeCode, void* dataPtr,
                               Size maximumSize, Size *actualSize);
pascal OSErr AEGetKeyDesc     (const AERecord *theAERecord,
                               AEKeyword theAEKeyword, DescType desiredType,
                               AEDesc *result);
```

Requesting User Interaction

```
pascal OSErr AESetInteractionAllowed
                              (AEInteractAllowed level);
pascal OSErr AEGetInteractionAllowed
                              (AEInteractAllowed *level);
pascal OSErr AEInteractWithUser
                              (long timeOutInTicks, NMRecPtr nmReqPtr,
                               IdleProcPtr idleProc);
```

Requesting More Time to Respond to Apple Events

```
pascal OSErr AEResetTimer     (const AppleEvent *reply);
```

Suspending and Resuming Apple Event Handling

```
pascal OSErr AESuspendTheCurrentEvent
                              (const AppleEvent *theAppleEvent);
pascal OSErr AEResumeTheCurrentEvent
                              (const AppleEvent *theAppleEvent,
                               const AppleEvent *reply,
                               EventHandlerProcPtr dispatcher,
                               long handlerRefcon);
pascal OSErr AESetTheCurrentEvent
                              (const AppleEvent *theAppleEvent);
pascal OSErr AEGetTheCurrentEvent
                              (AppleEvent *theAppleEvent);
```

Getting the Sizes and Descriptor Types of Descriptor Records

```
pascal OSErr AESizeOfNthItem
                              (const AEDescList *theAEDescList, long index,
                               DescType *typeCode, Size *dataSize);
pascal OSErr AESizeOfKeyDesc
                              (const AERecord *theAERecord,
                               AEKeyword theAEKeyword, DescType *typeCode,
                               Size *dataSize);
```

```
pascal OSErr AESizeOfParam   (const AppleEvent *theAppleEvent,
                              AEKeyword theAEKeyword, DescType *typeCode,
                              Size *dataSize);
pascal OSErr AESizeOfAttribute
                             (const AppleEvent *theAppleEvent,
                              AEKeyword theAEKeyword, DescType *typeCode,
                              Size *dataSize);
```

Deleting Descriptor Records

```
pascal OSErr AEDeleteItem   (const AEDescList *theAEDescList, long index);
pascal OSErr AEDeleteKeyDesc
                             (const AERecord *theAERecord,
                              AEKeyword theAEKeyword);
pascal OSErr AEDeleteParam  (const AppleEvent *theAppleEvent,
                              AEKeyword theAEKeyword);
```

Deallocating Memory for Descriptor Records

```
pascal OSErr AEDisposeDesc   (AEDesc *theAEDesc);
```

Coercing Descriptor Types

```
pascal OSErr AECoercePtr    (DescType typeCode, const void* dataPtr,
                             Size dataSize, DescType toType,
                             AEDesc *result);
pascal OSErr AECoerceDesc   (const AEDesc *theAEDesc, DescType toType,
                             AEDesc *result);
```

Creating and Managing the Coercion Handler Dispatch Tables

```
pascal OSErr AEInstallCoercionHandler
                             (DescType fromType, DescType toType,
                              ProcPtr handler, long handlerRefcon,
                              Boolean fromTypeIsDesc, Boolean isSysHandler);
pascal OSErr AEGetCoercionHandler
                             (DescType fromType, DescType toType,
                              ProcPtr *handler, long *handlerRefcon,
                              Boolean *fromTypeIsDesc,
                              Boolean isSysHandler);
pascal OSErr AERemoveCoercionHandler
                             (DescType fromType, DescType toType,
                              ProcPtr handler, Boolean isSysHandler);
```

Creating and Managing the Special Handler Dispatch Tables

```
pascal OSErr AEInstallSpecialHandler
                        (AEKeyword functionClass, ProcPtr handler,
                         Boolean isSysHandler);
pascal OSErr AEGetSpecialHandler
                        (AEKeyword functionClass, ProcPtr *handler,
                         Boolean isSysHandler);
pascal OSErr AERemoveSpecialHandler
                        (AEKeyword functionClass, ProcPtr handler,
                         Boolean isSysHandler);
```

Getting Information About the Apple Event Manager

```
/*available only in version 1.01 and later versions of Apple Event Manager*/

pascal OSErr AEManagerInfo  (AEKeyword keyword, long *result);
```

Application-Defined Routines

```
pascal OSErr MyEventHandler (const AppleEvent *theAppleEvent,
                             const AppleEvent *reply, long handlerRefcon);
pascal OSErr MyCoercePtr        (DescType typeCode, const void* dataPtr,
                             Size dataSize, DescType toType,
                             long handlerRefcon, AEDesc *result);
pascal OSErr MyCoerceDesc       (const AEDesc *theAEDesc, DescType toType, long
                             handlerRefcon, AEDesc *result);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

```
_Pack8
```

Selector	Routine
$011E	AESetInteractionAllowed
$0204	AEDisposeDesc
$0219	AEResetTimer
$021A	AEGetTheCurrentEvent
$021B	AEProcessAppleEvent

Selector	Routine
$021D	AEGetInteractionAllowed
$022B	AESuspendTheCurrentEvent
$022C	AESetTheCurrentEvent
$0407	AECountItems
$040E	AEDeleteItem
$0413	AEDeleteKeyDesc
$0413	AEDeleteParam
$0441	AEManagerInfo
$0500	AEInstallSpecialHandler
$0501	AERemoveSpecialHandler
$052D	AEGetSpecialHandler
$0603	AECoerceDesc
$061C	AEInteractWithUser
$0720	AERemoveEventHandler
$0723	AERemoveCoercionHandler
$0812	AEGetKeyDesc
$0812	AEGetParamDesc
$0818	AEResumeTheCurrentEvent
$0826	AEGetAttributeDesc
$0828	AESizeOfAttribute
$0829	AESizeOfKeyDesc
$0829	AESizeOfParam
$082A	AESizeOfNthItem
$091F	AEInstallEventHandler
$0921	AEGetEventHandler
$0A02	AECoercePtr
$0A22	AEInstallCoercionHandler
$0A0B	AEGetNthDesc
$0B24	AEGetCoercionHandler
$0D0C	AEGetArray
$0E11	AEGetKeyPtr
$0E11	AEGetParamPtr
$0E15	AEGetAttributePtr
$100A	AEGetNthPtr

Result Codes

noErr	0	No error
paramErr	–50	Parameter error (for example, value of handler pointer is NIL or odd)
eLenErr	–92	Buffer too big to send
memFullErr	–108	Not enough room in heap zone
userCanceledErr	–128	User canceled an operation
procNotFound	–600	No eligible process with specified process serial number
bufferIsSmall	–607	Buffer is too small
noOutstandingHLE	–608	No outstanding high-level event
connectionInvalid	–609	Nonexistent signature or session ID
noUserInteractionAllowed	–610	Background application sends event requiring authentication
noPortErr	–903	Client hasn't set 'SIZE' resource to indicate awareness of high-level events
destPortErr	–906	Server hasn't set 'SIZE' resource to indicate awareness of high-level events, or else is not present
sessClosedErr	–917	The kAEDontReconnect flag in the sendMode parameter was set, and the server quit and then restarted
errAECoercionFail	–1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	–1701	Descriptor record was not found
errAECorruptData	–1702	Data in an Apple event could not be read
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed
errAENewerVersion	–1706	Need a newer version of the Apple Event Manager
errAENotAppleEvent	–1707	Event is not an Apple event
errAEEventNotHandled	–1708	Event wasn't handled by an Apple event handler
errAEReplyNotValid	–1709	AEResetTimer was passed an invalid reply
errAEUnknownSendMode	–1710	Invalid sending mode was passed
errAEWaitCanceled	–1711	User canceled out of wait loop for reply or receipt
errAETimeout	–1712	Apple event timed out
errAENoUserInteraction	–1713	No user interaction allowed
errAENotASpecialFunction	–1714	The keyword is not valid for a special function
errAEParamMissed	–1715	Handler cannot understand a parameter the client considers required
errAEUnknownAddressType	–1716	Unknown Apple event address type
errAEHandlerNotFound	–1717	No handler found for an Apple event or a coercion, or no object callback function found
errAEReplyNotArrived	–1718	Reply has not yet arrived
errAEIllegalIndex	–1719	Not a valid list index
errAEImpossibleRange	–1720	The range is not valid because it is impossible for a range to include the first and last objects that were specified; an example is a range in which the offset of the first object is greater than the offset of the last object
errAEWrongNumberArgs	–1721	The number of operands provided for the kAENot logical operator is not 1

errAEAccessorNotFound	−1723	There is no object accessor function for the specified object class and token descriptor type
errAENoSuchLogical	−1725	The logical operator in a logical descriptor record is not kAEAnd, kAEOr, or kAENot
errAEBadTestKey	−1726	The descriptor record in a test key is neither a comparison descriptor record nor a logical descriptor record
errAENotAnObjectSpec	−1727	The objSpecifier parameter of AEResolve is not an object specifier record
errAENoSuchObject	−1728	A run-time resolution error, for example: object specifier record asked for the third element, but there are only two
errAENegativeCount	−1729	Object-counting function returned negative value
errAEEmptyListContainer	−1730	The container for an Apple event object is specified by an empty list
errAEUnknownObjectType	−1731	Descriptor type of token returned by AEResolve is not known to server application
errAERecordingIsAlreadyOn	−1732	Attempt to turn recording on when it is already on

Creating and Sending Apple Events

Contents

This chapter describes how your application can use the Apple Event Manager to create and send Apple events. If you want to factor your application for recording, or if you want your application to send Apple events directly to other applications, you can use Apple Event Manager routines to create and send Apple events.

Before you read this chapter, you should be familiar with the chapters "Introduction to Interapplication Communication," "Introduction to Apple Events," and "Responding to Apple Events" in this book. If you are factoring your application, you should also be familiar with the chapter "Recording Apple Events" in this book.

This chapter provides the basic information you need to create and send Apple events from your application. To send core and functional-area Apple events, your application must also be able to create object specifier records. For information about object specifier records, see the chapter "Resolving and Creating Object Specifier Records" in this book.

To allow your application to send Apple events to applications on other computers, you may wish to use the `PPCBrowser` function, which is described in the chapter "Program-to-Program Communications Toolbox" in this book.

The first section in this chapter, "Creating an Apple Event," describes how to

n   create an Apple event

n   add parameters to an Apple event

n   specify optional Apple event parameters

n   specify a target address

The section "Sending an Apple Event" describes how to

n   send an Apple event

n   deal with timeouts

n   write an idle function

n   write a reply filter function

Creating an Apple Event

You create an Apple event by using the `AECreateAppleEvent` function. You supply parameters that specify the event class and event ID, the target address, the return ID, and the transaction ID, and a buffer for the returned Apple event. The `AECreateAppleEvent` function creates and returns, in the buffer you specify, an Apple event with the attributes set as your application requested. You should not directly manipulate the contents of the Apple event; rather, use Apple Event Manager functions to add additional attributes or parameters to it.

The example that follows creates an imaginary Multiply event using the `AECreateAppleEvent` function.

```
myErr := AECreateAppleEvent(kArithmeticClass, kMultEventID,
                            targetAddress, kAutoGenerateReturnID,
                            kAnyTransactionID, theAppleEvent);
```

The event class, specified by the `kArithmeticClass` constant, identifies this event as belonging to a class of Apple events for arithmetic operations. The event ID specifies the particular Apple event within the class—in this case, an Apple event that performs multiplication.

You specify the target of the Apple event in the third parameter to `AECreateAppleEvent`. The target address can identify an application on the local computer or another computer on the network. You can specify the address using a target ID record or session reference number. For processes on the local computer, you can also use a process serial number or application signature to specify the address. See "Specifying a Target Address" on page 5-10 for more information.

In the fourth parameter, you specify the return ID of the Apple event, which associates this Apple event with the server's reply. The `AECreateAppleEvent` function assigns the specified return ID value to the `keyReturnIDAttr` attribute of the Apple event. If a server returns a standard reply Apple event (that is, an event of event class `'aevt'` and event ID `'ansr'`) in response to this event, the Apple Event Manager assigns the reply event the same return ID. When you receive a reply Apple event, you can check the `keyReturnIDAttr` attribute to determine which outstanding Apple event the reply is responding to. You can use the `kAutoGenerateReturnID` constant to request that the Apple Event Manager generate a return ID that is unique to this session for the Apple event. Otherwise, you are responsible for making it unique.

The fifth parameter specifies the transaction ID attribute of the Apple event. A **transaction** is a sequence of Apple events that are sent back and forth between the client and server applications, beginning with the client's initial request for a service. All Apple events that are part of one transaction must have the same transaction ID.

You can use a transaction ID to indicate that an Apple event is one of a sequence of Apple events related to a single transaction. The `kAnyTransactionID` constant indicates that the Apple event is not part of a transaction.

The `AECreateAppleEvent` function creates an Apple event with only the specified attributes and no parameters. To add parameters or additional attributes, you can use other Apple Event Manager functions.

Adding Parameters to an Apple Event

You can use the `AEPutParamPtr` or `AEPutParamDesc` function to add parameters to an Apple event. When you use either of these functions, the Apple Event Manager adds the specified parameter to the Apple event.

Use the `AEPutParamPtr` function when you want to add data specified in a buffer as the parameter of an Apple event. You specify the Apple event, the keyword of the parameter to add, the descriptor type, a buffer that contains the data, and the size of this buffer as parameters to the `AEPutParamPtr` function. The `AEPutParamPtr` function adds the data to the Apple event as a parameter with the specified keyword.

For example, this code adds a parameter to the Multiply event using the `AEPutParamPtr` function:

```
CONST keyOperand1 = 'OPN1';
VAR
    number1:          LongInt;
    theAppleEvent:    AppleEvent;
    myErr:            OSErr;

number1 := 10;
myErr := AEPutParamPtr(theAppleEvent, keyOperand1,
                       typeLongInteger, @number1,
                       SizeOf(number1));
```

In this example, the Apple Event Manager adds the parameter containing the first number to the specified Apple event.

Use the `AEPutParamDesc` function to add a descriptor record to an Apple event. The descriptor record you specify must already exist. To create or get a descriptor record, you can use the `AECreateDesc`, `AEDuplicateDesc`, and other Apple Event Manager functions that return a descriptor record.

When you create a descriptor record using the `AECreateDesc` function, you specify the descriptor type, a buffer that contains the data, and the size of this buffer as parameters. The `AECreateDesc` function returns the descriptor record that describes the data.

This example creates a descriptor record for the second parameter of the Multiply event:

```
VAR
    number2:            LongInt;
    multParam2Desc:    AEDesc;
    myErr:              OSErr;

number2 := 8;
myErr := AECreateDesc(typeLongInteger, @number2, SizeOf(number2),
                        multParam2Desc);
```

In this example, the `AECreateDesc` function creates a descriptor record with the `typeLongInteger` descriptor type and the data identified in the `number2` variable.

Once you have created a descriptor record, you can use `AEPutParamDesc` to add the data to an Apple event parameter. You specify the Apple event to add the parameter to, the keyword of the parameter, and the descriptor record of the parameter as parameters to the `AEPutParamDesc` function.

This example adds a second parameter to the Multiply event using the `AEPutParamDesc` function:

```
CONST keyOperand2 = 'OPN2';

myErr := AEPutParamDesc(theAppleEvent, keyOperand2,
                        multParam2Desc);
```

This example adds the `keyOperand2` keyword and the descriptor record created in the previous example as the second parameter to the specified Apple event.

You can also create a descriptor record without using Apple Event Manager routines. For example, this example generates an alias descriptor record from an existing alias handle:

```
WITH myAliasDesc DO
    BEGIN
        descriptorType := typeAlias;
        dataHandle := myAliasHandle;
    END;
```

Whatever method you use to create a descriptor record, you can add it to an Apple event parameter by using `AEPutParamDesc`.

After adding parameters to an Apple event, you can send the Apple event using the `AESend` function. See "Sending an Apple Event," which begins on page 5-13, for information about using this function.

Specifying Optional Parameters for an Apple Event

The parameters for a given Apple event are listed in the *Apple Event Registry: Standard Suites* as either required or optional. Your application does not usually have to include Apple event parameters that are listed as optional; the target application uses default values for parameters that are listed as optional if your application does not provide them. The *Apple Event Registry: Standard Suites* defines the default value a target application should use for each optional parameter of a specific Apple event.

The guidelines listed in the *Apple Event Registry: Standard Suites* for which parameters should be considered optional and which should be considered required are not enforced by the Apple Event Manager. Instead, the source application indicates which Apple event parameters it considers optional by listing the keywords for those parameters in the `keyOptionalKeywordAttr` attribute.

The `keyOptionalKeywordAttr` attribute does not contain the optional parameters; it simply lists the keywords of any parameters for the Apple event that the source application wants to identify as optional. Although the source application is responsible for providing this information in the `keyOptionalKeywordAttr` attribute of an Apple event, it is not required to provide this attribute.

If a keyword for an Apple event parameter is not included in the `keyOptionalKeywordAttr` attribute, the source application expects the target application to understand the Apple event parameter identified by that keyword. If a target application cannot understand the parameter, it should return the result code `errAEParamMissed` and should not attempt to handle the event.

If a keyword for an Apple event parameter is included in the `keyOptionalKeywordAttr` attribute, the source application does not require the target application to understand the Apple event parameter identified by that keyword. If the target application cannot understand a parameter whose keyword is included in the `keyOptionalKeywordAttr` attribute, it should ignore that parameter and attempt to handle the Apple event as it normally does.

A source application can choose not to list the keyword for an Apple event parameter in the `keyOptionalKeywordAttr` attribute even if that parameter is listed in the *Apple Event Registry: Standard Suites* as an optional parameter. This has the effect of forcing the target application to treat the parameter as required for a particular Apple event. If the target application supports the parameter, it should handle the Apple event as the client application expects. If the target application does not support the parameter and calls an application-defined routine such as `MyGotRequiredParams` to check whether it has received all the required parameters, it finds that there's another parameter that the client application considered required, and should return the result code `errAEParamMissed`.

If a source application wants a target application to attempt to handle an Apple event regardless of whether the target application supports a particular Apple event parameter included in that Apple event, the source application should list the keyword for that parameter in the `keyOptionalKeywordAttr` attribute.

It is up to the source application to decide whether to list a parameter that is described as optional in the *Apple Event Registry: Standard Suites* in the `keyOptionalKeywordAttr` attribute of an Apple event. For example, suppose a source application has extended the definition of the Print event to include an optional `keyColorOrGrayscale` parameter that specifies printing in color or gray scale rather than black and white. The source application might decide whether or not to list the keyword `keyColorOrGrayscale` in the `keyOptionalKeywordAttr` attribute according to the characteristics of the print request. If the source application requires the target application to print a document in color, the source application could choose not to add the keyword `keyColorOrGrayscale` to the `keyOptionalKeywordAttr` attribute; in this case, only target applications that supported the `keyColorOrGrayscale` parameter would attempt to handle the event. If the source application does not require the document printed in color, it could choose to add the keyword `keyColorOrGrayscale` to the `keyOptionalKeywordAttr` attribute; in this case, the target application will attempt to handle the event regardless of whether it supports the `keyColorOrGrayscale` parameter.

Your application can add optional parameters to an Apple event the same way it adds required parameters, using the `AECreateDesc`, `AEPutParamPtr`, and `AEPutParamDesc` functions as described in the previous section, "Adding Parameters to an Apple Event." If your application chooses to provide the `keyOptionalKeywordAttr` attribute for an Apple event, it should first create a descriptor list that specifies the keywords of the optional parameters, then add it to the Apple event as a `keyOptionalKeywordAttr` attribute.

Listing 5-1 shows an application-defined routine, MyCreateOptionalKeyword, that
creates the keyOptionalKeywordAttr attribute for the Create Publisher event.

Listing 5-1   Creating the optional keyword for the Create Publisher event

```
FUNCTION MyCreateOptionalKeyword
                        (VAR createPubAppleEvent: AppleEvent)
                         : OSErr;
VAR
   optionalList:  AEDescList;
   myOptKeyword1: AEKeyword;
   myOptKeyword2: AEKeyword;
   myErr:         OSErr;
   ignoreErr:     OSErr;
BEGIN
   myOptKeyword1 := keyDirectObject;
   {create an empty descriptor list}
   myErr := AECreateList(NIL, 0, FALSE, optionalList);
   IF myErr = noErr THEN
   BEGIN
      {add the keyword of the first optional parameter}
      myErr := AEPutPtr(optionalList, 1, typeKeyword,
                        @myOptKeyword1, SizeOf(myOptKeyword1));
      IF myErr = noErr THEN
      BEGIN
         {add the keyword of the next optional parameter}
         myOptKeyword2 := keyAEEditionFileLoc;
         myErr := AEPutPtr(optionalList, 2, typeKeyword,
                           @myOptKeyword2, SizeOf(myOptKeyword2));
      END;
      IF myErr = noErr THEN
         {create the keyOptionalKeywordAttr attribute and add it }
         { to the Create Publisher event}
         myErr := AEPutAttributeDesc(createPubAppleEvent,
                                     keyOptionalKeywordAttr,
                                     optionalList);
   END;
   ignoreErr := AEDisposeDesc(optionalList);
   MyCreateOptionalKeyword := myErr;
END;
```

The `MyCreateOptionalKeyword` function shown in Listing 5-1 adds to a descriptor list the keyword of each parameter that the source application considers optional. Each keyword is added as a descriptor record with the descriptor type `typeKeyword`. The function specifies that the target application can handle the Create Publisher event without supporting parameters identified by the keywords `keyDirectObject` and `keyAEEditionFileLoc`. (These are the parameters that specify the Apple event object to publish and the location of the edition container; if these parameters are missing, the target application creates a publisher for the current selection using the application's default edition container.) After adding these keywords to the descriptor list, the function creates the `keyOptionalKeywordAttr` attribute using the `AEPutAttributeDesc` function.

Typically a target application does not examine the `keyOptionalKeywordAttr` attribute directly. Instead, a target application that supports a parameter listed as optional in the *Apple Event Registry: Standard Suites* attempts to extract it from the Apple event (using `AEGetParamDesc`, for example). If it can't extract the parameter, the target application uses the default value, if any, listed in the *Apple Event Registry.* A target application can use the `keyMissedKeywordAttr` attribute to return the first required parameter (that is, considered required by the source application), if any, that it did not retrieve from the Apple event. The `keyMissedKeywordAttr` attribute does not return any parameters whose keywords are listed in the `keyOptionalKeywordAttr` attribute of the Apple event.

Specifying a Target Address

When you create an Apple event, you must specify the address of the target. The **target address** identifies the particular application or process to which you want to send the Apple event. You can send Apple events to applications on the local computer or on remote computers on the network.

These are the descriptor types that identify the four methods of addressing an Apple event:

`typeApplSignature`	The application signature of the target
`typeSessionID`	The session reference number of the target
`typeTargetID`	The target ID record of the target
`typeProcessSerialNumber`	The process serial number of the target

To address an Apple event to a target on a remote computer on the network, you must use either the `typeSessionID` or `typeTargetID` descriptor type.

If your application sends an Apple event to itself, it should address the Apple event using a process serial number of `kCurrentProcess`. This is the fastest way for your application to send an Apple event to itself. For more information, see "Addressing an Apple Event for Direct Dispatching" on page 5-13.

You can use any of the four address types when sending an Apple event to another application on the local computer. The chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* describes all four types of addresses. Your application can

also use another address type if it provides a coercion handler that coerces the address type into one of the four address types that the Apple Event Manager recognizes. See "Writing and Installing Coercion Handlers," which begins on page 4-41, for more information.

To allow the user to choose the target of an Apple event, use the PPCBrowser function. This function presents a standard user interface for choosing a target application, much as the Standard File Package provides a standard user interface for opening and saving files. The PPCBrowser function returns, in a target ID record, information about the application the user chose. Listing 5-3 on page 5-12 shows how to use the PPCBrowser function to let the user choose a target.

Creating an Address Descriptor Record

You specify the address using an address descriptor record (a descriptor record of data type AEAddressDesc). You must create a descriptor record of this type and then add it to the Apple event using the AECreateAppleEvent function.

You can use the AECreateDesc function to create address descriptor records for any of the four types of target addresses. Listing 5-2 shows four possible ways to create an address, each using a different address type.

Listing 5-2   Creating a target address

```
PROCEDURE MySetTargetAddresses(VAR targetAddress1,
                                   targetAddress2, targetAddress3,
                                   targetAddress4: AEAddressDesc;
                                   toTargetID: TargetID;
                                   thePSN: ProcessSerialNumber;
                                   theSignature: OSType;
                                   theSessionRef: PPCSessRefNum);
VAR
   myErr:   OSErr;
BEGIN
   myErr := AECreateDesc(typeTargetID, @toTargetID,
                         SizeOf(toTargetID), targetAddress1);
   myErr := AECreateDesc(typeProcessSerialNumber, @thePSN,
                         SizeOf(thePSN), targetAddress2);
   myErr := AECreateDesc(typeApplSignature, @theSignature,
                         SizeOf(theSignature), targetAddress3);
   myErr := AECreateDesc(typeSessionID, @theSessionRef,
                         SizeOf(theSessionRef), targetAddress4);
   {add your own error checking}
END;
```

To create an address descriptor record, specify the following as parameters to AECreateDesc: the descriptor type for the address, a pointer to the buffer containing the address, and the size of the buffer. The AECreateDesc function returns an address descriptor record with the specified characteristics.

After creating an address, you can specify it as a parameter to the AECreateAppleEvent function. See "Creating an Apple Event," which begins on page 5-3, for an example using the AECreateAppleEvent function.

When you specify an address to the AECreateAppleEvent function, the Apple Event Manager stores the address in the keyAddressAttr attribute of the Apple event.

If you use the PPCBrowser function to allow the user to choose an Apple event's target, your application must create a target ID record based on the user's choice. Listing 5-3 shows how to create a target ID record using the information returned from the PPCBrowser function and create an address descriptor record using the AECreateDesc function.

Listing 5-3 Specifying a target address in an Apple event by using the PPCBrowser function

```
FUNCTION MyGetTargetAddress (myPrompt: Str255; myAppStr: Str255;
                            VAR myPortInfo: PortInfoRec;
                            VAR targetAddress: AEAddressDesc;
                            VAR toTargetID: targetID): OSErr;
VAR
   myErr:   OSErr;
BEGIN
   {use PPCBrowser to let user choose the target}
   myErr := PPCBrowser(myPrompt, myAppStr, FALSE,
                       toTargetID.location,
                       myPortInfo, NIL, '');
   MyGetTargetAddress := myErr;
   IF myErr <> noErr THEN Exit(MyGetTargetAddress);

   toTargetID.name := myPortInfo.name;

   {create the descriptor record for the target address}
   MyGetTargetAddress := AECreateDesc(typeTargetID, @toTargetID,
                                   SizeOf(toTargetID),
                                   targetAddress);
END;
```

See the chapter "Program-to-Program Communications Toolbox" in this book for more information on using the PPCBrowser function.

Addressing an Apple Event for Direct Dispatching

As described in the chapter "Recording Apple Events" in this book, a recordable application must send itself Apple events in response to user actions. Your application can send itself Apple events by using an address descriptor record of descriptor type `typeProcessSerialNumber` with the `lowLongOfPSN` field set to `kCurrentProcess` and the `highLongOfPSN` set to 0. The Apple Event Manager processes such Apple events immediately, executing the appropriate Apple event handler directly without going through the normal event-processing sequence. For this reason, your application will not appear to run more slowly when it sends Apple events to itself.

Apple events your application sends to itself this way do not appear in your application's high-level event queue. This not only speeds up delivery of the event but also avoids situations in which an Apple event sent in response to a user action arrives in the event queue after some other event that really occurred later than the user action. For example, suppose a user chooses Cut from the Edit menu and then clicks in another window. If the Cut event arrives in the queue after the window activate event, a selection in the wrong window might be cut.

Your application can send events to itself using other forms of addressing, such as the true process serial number returned by `GetCurrentProcess`. Because direct dispatching avoids event sequence problems, applications should generally send events to themselves by using an address descriptor record of descriptor type `typeProcessSerialNumber` with the `kCurrentProcess` constant rather than using a true process serial number or an application signature.

IMPORTANT

When Apple event recording has been turned on, the Apple Event Manager records every event that your application sends to itself unless you specify the `kAEDontRecord` flag in the `sendMode` parameter of the `AESend` function. s

Sending an Apple Event

To send an Apple event, you first create an Apple event, add parameters and attributes to it, and then use the `AESend` function to send it.

When you send an Apple event, you specify various options to indicate how the server should handle the Apple event. You request a user interaction level from the server and specify whether the server can switch directly to the foreground if user interaction is needed, whether your application is willing to wait for a reply Apple event, whether reconnection is allowed, and whether your application wants a return receipt for the Apple event.

You specify these options by setting flags in the `sendMode` parameter for `AESend`. Here are the constants that represent these flags:

```
CONST kAENoReply       = $00000001;{client doesn't want reply}
      kAEQueueReply    = $00000002;{client wants Apple Event }
                                   { Manager to return }
                                   { reply in event queue}
      kAEWaitReply     = $00000003;{client wants a reply and }
                                   { will give up processor}


      kAENeverInteract = $00000010;{server application }
                                   { should not interact }
                                   { with user for this }
                                   { Apple event}
      kAECanInteract   = $00000020;{server may interact with }
                                   { user for this Apple }
                                   { event to supply }
                                   { information}
      kAEAlwaysInteract = $00000030;{server may interact with }
                                   { user for this Apple }
                                   { event even if no }
                                   { information is required}


      kAECanSwitchLayer = $00000040;{server should come }
                                   { directly to foreground }
                                   { when appropriate}
      kAEDontReconnect = $00000080;{don't reconnect if there }
                                   { is a PPC session closed }
                                   { error}
      kAEWantReceipt   = nReturnReceipt;{client wants return }
                                       { receipt}
      kAEDontRecord    = $00001000;{don't record this event}
      kAEDontExecute   = $00002000;{don't execute this event}
```

If you want your application to receive a reply Apple event, specify the `kAEQueueReply` or `kAEWaitReply` flag. If you want your application to receive the reply Apple event in its event queue, use `kAEQueueReply`. If you want your application to receive the reply Apple event in the `reply` parameter for `AESend` and you are willing to give up the processor while it is waiting for the reply, use `kAEWaitReply`. If you don't want your application to receive a reply Apple event and your application doesn't need to wait for the server to handle the Apple event, specify `kAENoReply`.

Note

Before the Apple Event Manager sends a reply event back to the client application, the `keyAddressAttr` attribute contains the address of the client application. After the client receives the reply event, the `keyAddressAttr` attribute contains the address of the server application. u

If you specify `kAENoReply` or `kAEQueueReply`, the `AESend` function returns immediately after using the Event Manager to send the event. In this case, a `noErr` result code from `AESend` indicates that the Event Manager sent the Apple event; it does not mean that the server accepted or handled the Apple event.

When `AESend` returns, the `reply` parameter does not contain valid data if your application specifies `kAENoReply` or `kAEQueueReply`. The `kAENoReply` flag indicates that the Apple Event Manager will not return the reply Apple event to your application. The `kAEQueueReply` flag indicates that you want your application to receive the reply via its event queue rather than the `reply` parameter of `AESend`. If you specify `kAEQueueReply`, you must install a handler for the reply Apple event (event class `kCoreEventClass` and event ID `kAEAnswer`).

If you specify `kAEWaitReply`, the Apple Event Manager uses the Event Manager to send the event. The Apple Event Manager then calls the `WaitNextEvent` function on behalf of your application, causing your application to yield the processor and giving the server application a chance to receive and handle the Apple event. Your application continues to yield the processor until the server handles the Apple event or the request times out.

If you specify `kAEWaitReply`, you must provide an idle function. This function should process any update events, null events, operating-system events, or activate events that occur while your application is waiting for a reply. See "Writing an Idle Function," which begins on page 5-21, for sample code that shows an idle function.

You use one of the three flags—`kAENeverInteract`, `kAECanInteract`, and `kAEAlwaysInteract`—to specify whether the server should interact with the user when handling the Apple event. Specify `kAENeverInteract` if the server should not interact with the user when handling the Apple event. You might specify this constant if you don't want the user to be interrupted while the server is handling the Apple event.

Use the `kAECanInteract` flag if the server should interact with the user when the user needs to supply information to the server. Use the `kAEAlwaysInteract` flag if the server should interact with the user whenever the server normally asks a user to confirm a decision or interact in any other way, even if no additional information is needed from the user. Note that it is the responsibility of the server and client applications to agree on how to interpret the `kAEAlwaysInteract` flag.

If the client application does not set any one of the user interaction flags, the Apple Event Manager sets a default, depending on the location of the target of the Apple event. If the server application is on a remote computer, the Apple Event Manager sets the `kAENeverInteract` flag as the default. If the target of the Apple event is on the local computer, the Apple Event Manager sets the `kAECanInteract` flag as the default.

The server application should call `AEInteractWithUser` if it needs to interact with the user. If both the client and the server allow user interaction, the Apple Event Manager attempts to bring the server to the foreground if it is not already the foreground process. If both the `kAECanSwitchLayer` and the `kAEWaitReply` flags are set, and if the client application is the active application on the local computer, the Apple Event Manager brings the server application directly to the front. Otherwise, the Apple Event Manager posts a notification request asking the user to bring the server application to the front, regardless of whether the `kAECanSwitchLayer` flag is set. This ensures that the user will not be interrupted by an unexpected application switch.

You should specify the `kAECanSwitchLayer` flag only when the client and server applications reside on the same computer. In general, you should not set this flag if it would be confusing or inconvenient to the user for the server application to come to the front unexpectedly. This flag is ignored if you are sending an Apple event to a remote computer.

Specify the `kAEDontReconnect` flag if the Apple Event Manager should not reconnect if it receives a session closed error from the PPC Toolbox. If you don't set this flag, the Apple Event Manager automatically attempts to reconnect and reestablish the session.

Specify the `kAEWantReceipt` flag if your application wants notification that the server application has accepted the Apple event. If you specify this flag, your application receives a return receipt as a high-level event.

If you specify the `kAEWantReceipt` flag and the server application does not accept the Apple event within the time specified by the `timeOutInTicks` parameter to `AESend`, the `AESend` function returns a timeout error. Note that `AESend` also returns a timeout error if your application sets the `kAEWaitReply` flag and does not receive the reply Apple event within the time specified by the `timeOutInTicks` parameter.

Specify the `kAEDontRecord` flag if your application is sending an Apple event to itself that you don't want to be recorded. When Apple event recording has been turned on, every event that your application sends to itself will be automatically recorded by the Apple Event Manager except those sent with the `kAEDontRecord` flag set.

Specify the `kAEDontExecute` flag if your application is sending an Apple event to itself for recording purposes only—that is, if you want the Apple Event Manager to send a copy of the event to the recording process but you do not want your application actually to receive the event. (For more information about when to use the `kAEDontExecute` flag, see the chapter "Recording Apple Events" in this book.)

Listing 5-4 illustrates how to send a Multiply event (an imaginary Apple event for multiplying two long integers). It first creates an Apple event, adds parameters containing the numbers to multiply, then sends it, specifying various options. It also illustrates how to handle the reply Apple event that contains the result.

Note

If you want to send Apple events, your application must set flags in its
`'SIZE'` resource indicating that it can handle high-level events, and it
must provide handlers for the required Apple events. See "Accepting an
Apple Event" on page 4-5 for information on setting the appropriate
flags in the `'SIZE'` resource and "Handling the Required Apple
Events" on page 4-11 for information on supporting the required
Apple events. u

Listing 5-4   Sending an Apple event

```
FUNCTION MySendMultiplyEvent (serverAddress: AEAddressDesc;
                                firstOperand: LongInt; secondOperand: LongInt;
                                VAR replyResultLongInt: LongInt): OSErr;
CONST
   kArithmeticClass   =  'ARTH';   {event class for arithmetic }
                                   { Apple events}
   kMultiplyEventID   =  'MULT';   {event ID for Multiply event}
   keyMultOperand1    =  'OPN1';   {keyword for first parameter}
   keyMultOperand2    =  'OPN2';   {keyword for second parameter}
VAR
   theAppleEvent:     AppleEvent;
   reply:             AppleEvent;
   returnedType:      DescType;
   actualSize:        LongInt;
   myErr:             OSErr;
   ignoreErr:         OSErr;
   errStr:            Str255;
   errNumber:         LongInt;
BEGIN
   myErr := AECreateAppleEvent(kArithmeticClass, kMultiplyEventID,
                               serverAddress, kAutoGenerateReturnID,
                               kAnyTransactionID, theAppleEvent);
   IF myErr = noErr THEN
      {add the first operand}
      myErr := AEPutParamPtr(theAppleEvent, keyMultOperand1,
                             typeLongInteger, @firstOperand,
                             SizeOf(firstOperand));
   {add the second operand with the proper keyword}
   IF myErr = noErr THEN
      myErr := AEPutParamPtr(theAppleEvent, keyMultOperand2,
                             typeLongInteger, @secondOperand,
                             SizeOf(secondOperand));
```

```
IF myErr = noErr THEN
   myErr := AESend(theAppleEvent, reply, kAEWaitReply + kAENeverInteract,
               kAENormalPriority, 120, @MyIdleFunction, NIL);
IF myErr = noErr THEN {Apple event successfully sent}
BEGIN    {Check whether it was successfully handled-- }
         { get result code returned by the server's handler}
   myErr := AEGetParamPtr(reply, keyErrorNumber, typeLongInteger,
                        returnedType, @errNumber, SizeOf(errNumber),
                        actualSize);

   IF (myErr = errAEDescNotFound) OR (errNumber = noErr) THEN
   {if keyErrorNumber doesn't exist or server returned noErr }
   { then the Apple event was successfully handled--the reply Apple }
   { event contains the result in the direct parameter}
      myErr := AEGetParamPtr(reply, keyDirectObject, typeLongInteger,
                        returnedType, @replyResultLongInt,
                        SizeOf(replyResultLongInt), actualSize)
   ELSE
   BEGIN    {server returned an error, so get error string}
      myErr := AEGetParamPtr(reply, keyErrorString, typeChar,
                        returnedType, @errStr[1], SizeOf(errStr)-1,
                        actualSize);
      IF myErr = noErr THEN
      BEGIN
         IF actualSize > 255 THEN
            actualSize := 255;
         errStr[0] := chr(actualSize);
         MyDisplayError(errStr);
      END;
   END;
   ignoreErr := AEDisposeDesc(reply);
END
ELSE
BEGIN
   {the Apple event wasn't successfully dispatched, }
   { the request timed out, the user canceled, or other error}
END;
ignoreErr := AEDisposeDesc(theAppleEvent);
MySendMultiplyEvent := myErr;
END;
```

The code in Listing 5-4 first creates an Apple event with `kArithmeticClass` as the event class and `kMultiplyEventID` as the event ID. It also specifies the server of the Apple event. See "Specifying a Target Address" on page 5-10 for information on specifying a target address and "Creating an Apple Event," which begins on page 5-3, for more information on creating an Apple event.

The Multiply event shown in Listing 5-4 contains two parameters, each specifying a number to multiply. See "Adding Parameters to an Apple Event" on page 5-5 for examples of how to specify the parameters for the `AEPutParamPtr` function.

After adding the parameters to the event, the code uses `AESend` to send the event. The first parameter to `AESend` specifies the Apple event to send—in this example, the Multiply event. The next parameter specifies the reply Apple event.

This example specifies `kAEWaitReply` in the third parameter, indicating that the client is willing to yield the processor for the specified timeout value (120 ticks, or 2 seconds). The `kAENeverInteract` flag indicates that the server should not interact with the user when processing the Apple event. The fourth parameter specifies that the Multiply event is to be sent using normal priority (that is, placed at the end of the event queue). You can specify the `kAEHighPriority` flag to place the event in the front of the event queue, but this is not usually recommended.

The next to last parameter specifies the address of an idle function. If you specify `kAEWaitReply`, you must provide an idle function. This function should process any update events, null events, operating-system events, or activate events that occur while your application is waiting for a reply. See "Writing an Idle Function," which begins on page 5-21, for sample code that shows an idle function.

The last parameter to `AESend` specifies a filter function. You can supply a filter function to filter high-level events that your application may receive while waiting for a reply Apple event. You can specify `NIL` for this parameter if you do not need to filter high-level events while waiting for a reply. See "Writing a Reply Filter Function" on page 5-24 for more information.

If you specify `kAEWaitReply`, a `noErr` result code from `AESend` indicates that the Apple event was sent successfully, not that the server has completed the requested action successfully. Therefore, you should find out whether a result code was returned from the handler by checking the reply Apple event for the existence of either the `keyErrorNumber` or `keyErrorString` parameter. If the `keyErrorNumber` parameter does not exist or contains the `noErr` result code, you can use `AEGetParamPtr` to get the parameter you're interested in from the reply Apple event.

The `MySendMultiplyEvent` function in Listing 5-4 checks the function result of `AESend`. If it is `noErr`, `MySendMultiplyEvent` checks the `keyErrorNumber` parameter of the reply Apple event to determine whether the server successfully handled the Apple event. If this parameter exists and indicates that an error occurred, `MySendMultiplyEvent` gets the error string out of the `keyErrorString` parameter. Otherwise, the server performed the request, and the reply Apple event contains the answer to the multiplication request.

When you have finished using the Apple event specified in the `AESend` function and no longer need the reply Apple event, you must dispose of both the original event and the reply by calling the `AEDisposeDesc` function.

IMPORTANT

If your application sends Apple events to itself using a `typeProcessSerialNumber` address descriptor record with the `lowLongOfPSN` field set to `kCurrentProcess`, the Apple Event Manager jumps directly to the appropriate Apple event handler without going through the normal event-processing sequence. For this reason, your application will not appear to run more slowly when it sends Apple events to itself. For more information, see "Addressing an Apple Event for Direct Dispatching" on page 5-13. s

Dealing With Timeouts

When your application calls `AESend` and chooses to wait for the server application to handle the Apple event, it can also specify the maximum amount of time it is willing to wait for a response. You can specify a timeout value in the `timeOutInTicks` parameter to `AESend`. You can either specify a particular length of time, in ticks, that your application is willing to wait, or you can specify the `kNoTimeOut` constant or the `kAEDefaultTimeout` constant.

Use the `kNoTimeOut` constant to indicate that your application is willing to wait forever for a response from the server. You should use this value only if you are sure that the server will respond in a reasonable amount of time. You should also implement a method of checking whether the user wants to cancel. The idle function that you specify as a parameter to `AESend` should check the event queue for any instances of Command-period and immediately return `TRUE` as its function result if it finds a request to cancel in the event queue.

Use the `kAEDefaultTimeout` constant if you want the Apple Event Manager to use a default timeout value. The Apple Event Manager uses a timeout value of about one minute if you specify this constant.

If you set the `kAEWaitReply` flag and the server doesn't have a handler for the Apple event, the server immediately returns the `errAEEventNotHandled` result code. If the server doesn't respond within the length of time specified by the timeout value, `AESend` returns the `errAETimeout` result code and a reply Apple event that contains no data. This result code does not necessarily mean that the server failed to perform the requested action; it means only that the server did not complete processing within the specified time. The server might still be processing the Apple event, and it might still send a reply.

If the server finishes processing the Apple event sometime after the time specified in the
`keyTimeoutAttr` attribute has expired, it returns a reply Apple event to
`AEProcessAppleEvent`. The Apple Event Manager then adds the actual data to the
reply. Thus, your application can continue to check the reply Apple event to see if the
server has responded, even after the time expires. If the server has not yet sent the reply
when the client attempts to extract data from the reply Apple event, the Apple Event
Manager functions return the `errAEReplyNotArrived` result code. After the reply
Apple event returns from the server, the client can extract the data in the reply.

Additionally, the server can examine the `keyTimeoutAttr` attribute of the Apple event
to determine the timeout value specified by the client. You can use the value of this
attribute as a rough estimate of how much time your handler has to respond. You can
assume that your handler has less time to respond than the timeout value, because
transmitting the Apple event uses some of the available time, as does transmitting the
reply Apple event back to the client, and the event may have been in the queue for a
while already.

If you set the `kAENoReply` or `kAEQueueReply` flag, the Apple Event Manager ignores
any timeout value you specify, because your application is not waiting for the reply. An
attempt by the server to examine the `keyTimeoutAttr` attribute in this situation
generates the error `errAEDescNotFound`.

If your handler needs more time than is specified in the `keyTimeoutAttr` attribute, you
can reset the timer by using the `AEResetTimer` function. This function resets the
timeout value of an Apple event to its starting value.

Writing an Idle Function

This section describes how to write an idle function for use with the `AESend` or
`AEInteractWithUser` function.

When your application sends an Apple event, you can set one of three flags in the
`sendMode` parameter to `AESend` that specify how you want to deal with the reply:
`kAENoReply` if you don't want your application to receive a reply, `kAEQueueReply` if
you want it to receive the reply in its event queue, or `kAEWaitReply` if you want the
reply returned in the `reply` parameter of `AESend` and you are willing to give up
the processor while your application is waiting for the reply.

If you specify `kAENoReply` or `kAEQueueReply`, the `AESend` function returns
immediately after using the Event Manager to send the event. If you specify
`kAEWaitReply`, the `AESend` function does not return until either the server application
finishes handling the Apple event or a specified amount of time expires. In this case the
`AESend` function calls `WaitNextEvent` on behalf of your application. This yields the
processor to other processes, so that the server has an opportunity to receive and process
the Apple event sent by your application. While your application is waiting for a reply, it
cannot receive events unless it provides an idle function.

If you provide a pointer to an idle function as a parameter to the `AESend` function, `AESend` calls your idle function whenever an update event, null event, operating-system event, or activate event is received for your application. To allow your application to process high-level events that it receives while waiting for a reply, provide a reply filter function. See the next section, "Writing a Reply Filter Function," for more information.

Your application can yield the processor in a similar manner when it calls the `AEInteractWithUser` function. If `AEInteractWithUser` needs to post a notification request to bring your application to the front, your application yields the processor until the user brings your application to the front. To receive events while waiting for the user to bring your application to the front, you must provide an idle function.

If you provide a pointer to an idle function as a parameter to the `AEInteractWithUser` function, `AEInteractWithUser` calls your idle function whenever an update event, null event, operating-system event, or activate event is received for your application.

An idle function must use this syntax:

```
FUNCTION MyIdleFunction (VAR event: EventRecord;
                         VAR sleepTime: LongInt;
                         VAR mouseRgn: RgnHandle): Boolean;
```

The `event` parameter is the event record of the event to process. The `sleepTime` parameter and `mouseRgn` parameter are values that your idle function sets the first time it is called; thereafter they contain the values your function set. Your idle function should return a Boolean value that indicates whether your application wishes to continue waiting. Set the function result to `TRUE` if your application is no longer willing to wait for a reply from the server or for the user to bring the application to the front. Set the function result to `FALSE` if your application is still willing to wait.

You use the `sleepTime` and `mouseRgn` parameters in the same way as the `sleep` and `mouseRgn` parameters of the `WaitNextEvent` function. Specify in the `sleepTime` parameter the amount of time (in ticks) during which your application agrees to relinquish the processor if no events are pending for it.

In the `mouseRgn` parameter, you specify a screen region that determines the conditions under which your application is to receive notice of mouse-moved events. Your idle function receives mouse-moved events only if your application is the front application and the cursor strays outside the region you specify.

Your idle function receives only update events, null events, operating-system events, and activate events. When your idle function receives a null event, it can use the idle time to update a status dialog box, animate cursors, or perform similar tasks. If your idle function receives any of the other events, it should handle the event as it normally would if received in its event loop.

Listing 5-5 shows an example of an idle function for use with AESend or AEInteractWithUser. The idle function processes update events, null events, operating-system events, and activate events. The first time the function is called it receives a null event. At this time, it sets the sleepTime and mouseRgn parameters. The function continues to process events until the server finishes handling the Apple event or the user brings the application to the front.

Your application should implement a method of checking whether the user wants to cancel. The MyCancelInQueue function in Listing 5-5 checks the event queue for any instances of Command-period and immediately returns TRUE as its function result if it finds a request to cancel in the event queue.

Listing 5-5   An idle function

```
FUNCTION MyIdleFunction (VAR event: EventRecord;
                         VAR sleeptime: LongInt;
                         VAR mouseRgn: RgnHandle): Boolean;
BEGIN
   MyIdleFunction := FALSE;
   {the MyCancelInQueue function checks for Command-period}
   IF MyCancelInQueue THEN
      BEGIN
         MyIdleFunction := TRUE;
         Exit(MyIdleFunction);
      END;
   CASE event.what OF
      updateEvt,
      activateEvt,    {every idle function should handle }
      osEvt:          { these kinds of events}
         BEGIN
            MyAdjustCursor(event.where, gCursorRgn);
            DoEvent(event);
         END;
      nullEvent:
         BEGIN
            {set the sleepTime and mouseRgn parameters}
            mouseRgn := gCursorRgn;
            sleeptime := 10;  {use the correct value for your }
                              { app}
            DoIdle;           {the application's idle handling}
         END;
   END; {of CASE}
END;
```

CHAPTER 5

Creating and Sending Apple Events

Writing a Reply Filter Function

If your application calls `AESend` and chooses to yield the processor to other processes while waiting for a reply, you can provide an idle function to process update, null, operating-system, and activate events, and you can provide a reply filter function to process high-level events. The previous section describes how an idle function processes events.

Your reply filter function can process any high-level events that it is willing to handle while waiting for a reply Apple event. For example, your application can choose to handle Apple events from other processes while waiting. Note, however, that your application must maintain any necessary state information. Your reply filter function must not accept any Apple events that can change the state of your application and make it impossible to return to its previous state.

A reply filter function must use this syntax:

```
FUNCTION MyReplyFilter (VAR event: EventRecord;
                        returnID: LongInt;
                        transactionID: LongInt;
                        sender: AEAddressDesc): Boolean;
```

The `event` parameter is the event record for a high-level event. The next three parameters contain valid information only if the event is an Apple event. The `returnID` parameter is the return ID for the Apple event. The `transactionID` parameter is the transaction ID for the Apple event. The `sender` parameter contains the address of the application or process that sent the Apple event.

Your reply filter function should return `TRUE` as the function result if you want to accept the Apple event; otherwise, it should return `FALSE`. If your filter function returns `TRUE`, the Apple Event Manager calls the `AEProcessAppleEvent` function on behalf of your application, and your handler routine is called to process the Apple event. In this case, make sure your handler is not called while it is still being used by an earlier call.

Reference to Creating and Sending Apple Events

This section describes the basic Apple Event Manager routines that your application can use to create and send Apple events. It also describes application-defined idle functions and reply filter functions that your application can provide for use by the Apple Event Manager.

For information about data structures used with the routines described in this chapter, see the section "Data Structures Used by the Apple Event Manager," which begins on page 4-56.

5-24   Reference to Creating and Sending Apple Events

Routines for Creating and Sending Apple Events

This section describes the Apple Event Manager routines you can use to create Apple events, create and duplicate descriptor records, create and add items to descriptor lists and AE records, add parameters and attributes to Apple events, and send Apple events. The section "Routines for Responding to Apple Events," which begins on page 4-61, describes other Apple Event Manager routines used for both responding to and creating Apple events.

Creating Apple Events

The `AECreateAppleEvent` function allows you to create an Apple event.

AECreateAppleEvent

You can use the `AECreateAppleEvent` function to create an Apple event with several important attributes but no parameters. You add parameters to the Apple event after you create it.

```
FUNCTION AECreateAppleEvent (theAEEventClass: AEEventClass;
                             theAEEventID: AEEventID;
                             target: AEAddressDesc;
                             returnID: Integer;
                             transactionID: LongInt;
                             VAR result: AppleEvent): OSErr;
```

theAEEventClass
: The event class of the Apple event to be created.

theAEEventID
: The event ID of the Apple event to be created.

target
: The address of the server application.

returnID
: The return ID for the Apple event; if you specify `kAutoGenerateReturnID`, the Apple Event Manager assigns a return ID that is unique to the current session.

transactionID
: The transaction ID for this Apple event. A transaction is a sequence of Apple events that are sent back and forth between the client and server applications, beginning with the client's initial request for a service. All Apple events that are part of a transaction must have the same transaction ID.

result
: The `AECreateAppleEvent` function returns, in this parameter, the Apple event that it creates.

DESCRIPTION

The `AECreateAppleEvent` function creates an Apple event.Your application is responsible for using the `AEDisposeDesc` function to dispose of the Apple event when you no longer need it.

If `AECreateAppleEvent` returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone

SEE ALSO

See "Creating an Apple Event," which begins on page 5-3, for more information on how to create an Apple event.

See "Specifying a Target Address" on page 5-10 for information on how to address an Apple event.

Creating and Duplicating Descriptor Records

The `AECreateDesc` function converts data into a descriptor record, and the `AEDuplicateDesc` function makes a copy of a descriptor record.

AECreateDesc

You can use the `AECreateDesc` function to convert data into a descriptor record.

```
FUNCTION AECreateDesc (typeCode: DescType; dataPtr: Ptr;
                       dataSize: Size; VAR result: AEDesc): OSErr;
```

typeCode	The descriptor type for the descriptor record.
dataPtr	A pointer to the data for the descriptor record.
dataSize	The length, in bytes, of the data for the descriptor record.
result	The descriptor record that the `AECreateDesc` function creates.

DESCRIPTION

The `AECreateDesc` function creates a new descriptor record that incorporates the specified data. Your application is responsible for using the `AEDisposeDesc` function to dispose of the resulting descriptor record when you no longer need it. You normally do this after receiving a result code from the `AESend` function.

If `AECreateDesc` returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

`noErr`	**0**	No error
`memFullErr`	**–108**	Not enough room in heap zone

SEE ALSO

For examples of the use of `AECreateDesc`, see "Adding Parameters to an Apple Event," which begins on page 5-5, and Listing 5-2 on page 5-11.

AEDuplicateDesc

You can use the `AEDuplicateDesc` function to make a copy of a descriptor record.

```
FUNCTION AEDuplicateDesc (theAEDesc: AEDesc;
                          VAR result: AEDesc): OSErr;
```

`theAEDesc`   The descriptor record to be duplicated.
`result`   The duplicate descriptor record.

DESCRIPTION

The `AEDuplicateDesc` function creates a new descriptor record by copying the descriptor record from the parameter `theAEDesc`. Your application is responsible for using the `AEDisposeDesc` function to dispose of the resulting descriptor record when you no longer need it. You normally do this after receiving a result code from the `AESend` function.

If `AEDuplicateDesc` returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

It is common for applications to send Apple events that have one or more attributes or parameters in common. For example, if you send a series of Apple events to the same application, the address attribute is the same. In these cases, the most efficient way to create the necessary Apple events is to make a template Apple event that you can then copy—by calling the `AEDuplicateDesc` function—as needed. You then fill in or change the remaining parameters and attributes of the copy, send the copy by calling `AESend`, and dispose of the copy—by calling `AEDisposeDesc`—after `AESend` returns a result code.

RESULT CODES

`noErr`	**0**	No error
`memFullErr`	**–108**	Not enough room in heap zone

Creating Descriptor Lists and AE Records

The `AECreateList` function allows you to create an empty descriptor list or AE record.

AECreateList

You can use the `AECreateList` function to create an empty descriptor list or AE record.

```
FUNCTION AECreateList (factoringPtr: Ptr; factoredSize: Size;
                       isRecord: Boolean;
                       VAR resultList: AEDescList): OSErr;
```

factoringPtr
: A pointer to the data at the beginning of each descriptor that is the same for all descriptor records in the list. If there is no common data, or if you decide not to isolate the common data, specify `NIL` as the value of this parameter.

factoredSize
: The size of the common data. If there is no common data, or if you decide not to isolate the common data, the value of `factoredSize` must be 0. (See the description that follows for more information.)

isRecord
: A Boolean value that specifies the kind of list to create. If you set it to `TRUE`, the Apple Event Manager creates an AE record. If you set it to `FALSE`, the Apple Event Manager creates a descriptor list.

resultList
: The descriptor list or AE record that the `AECreateList` function creates.

DESCRIPTION

The `AECreateList` function creates an empty descriptor list or AE record. Your application is responsible for using the `AEDisposeDesc` function to dispose of the resulting descriptor record when you no longer need it. You normally do this after receiving a result code from the `AESend` function.

If you intend to use a descriptor list for a factored Apple event array, you must provide, in the `factoringPtr` parameter, a pointer to the data shared by all items in the array and, in the `factoredSize` parameter, the size of the common data. The common data must be 4, 8, or more than 8 bytes in length because it always consists of (a) the descriptor type (4 bytes); (b) the descriptor type (4 bytes) and the size of each item's data (4 bytes); or (c) the descriptor type (4 bytes), the size of each item's data (4 bytes), and some portion of the data itself (1 or more bytes).

If `AECreateList` returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

noErr	**0**	No error
paramErr	**–50**	Parameter error (value of handler pointer is NIL or odd)
memFullErr	**–108**	Not enough room in heap zone

SEE ALSO

For an example of the use of `AECreateList`, see Listing 5-1 on page 5-9.

For information about data types used with Apple event arrays, see "Apple Event Array Data Types" on page 4-60.

Adding Items to Descriptor Lists

The Apple Event Manager provides three routines that allow you to add descriptor records to any descriptor list, including an Apple event record. The `AEPutPtr` function converts data specified in a buffer to a descriptor record and adds the descriptor record to a descriptor list. The `AEPutDesc` function adds a descriptor record to a descriptor list. The `AEPutArray` function puts the data for an Apple event array into a descriptor list.

AEPutPtr

You can use the `AEPutPtr` routine to add data specified in a buffer to any descriptor list as a descriptor record.

```
FUNCTION AEPutPtr (theAEDescList: AEDescList; index: LongInt;
                    typeCode: DescType; dataPtr: Ptr;
                    dataSize: Size): OSErr;
```

theAEDescList
: The descriptor list to which to add a descriptor record.

index
: The position of the descriptor record in the descriptor list. (For example, the value 2 specifies the second descriptor record in the list.) If there is already a descriptor record in the specified position, it is replaced. If the value of index is 0, the descriptor record is added to the end of the list.

typeCode
: The descriptor type for the resulting descriptor record.

dataPtr
: A pointer to the data for the descriptor record.

dataSize
: The length, in bytes, of the data for the descriptor record.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed
errAEIllegalIndex	–1719	Not a valid list index

SEE ALSO

For an example of the use of `AEPutPtr`, see Listing 5-1 on page 5-9.

AEPutDesc

You can use the `AEPutDesc` function to add a descriptor record to any descriptor list.

```
FUNCTION AEPutDesc (theAEDescList: AEDescList; index: LongInt;
                    theAEDesc: AEDesc): OSErr;
```

theAEDescList
The descriptor list to which to add a descriptor record.

index   The position of the descriptor record in the descriptor list. (For example, the value 2 specifies the second descriptor record in the list.) If there is already a descriptor record in the specified position, it is replaced. If the value of `index` is 0, the descriptor record is added to the end of the list.

theAEDesc   The descriptor record to be added to the list.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed
errAEIllegalIndex	–1719	Not a valid list index

AEPutArray

You can use the `AEPutArray` function to put the data for an Apple event array into any descriptor list.

```
FUNCTION AEPutArray (theAEDescList: AEDescList;
                     arrayType: AEArrayType;
                     arrayPtr: AEArrayDataPointer;
                     itemType: DescType;
                     itemSize: Size; itemCount: LongInt): OSErr;
```

theAEDescList
: The descriptor list into which to put the Apple event array. If there are any items already in the descriptor list, they are replaced.

arrayType
: The Apple event array type to be created. This is specified by one of the following constants: `kAEDataArray`, `kAEPackedArray`, `kAEHandleArray`, `kAEDescArray`, or `kAEKeyDescArray`.

arrayPtr
: A pointer to the buffer containing the array.

itemType
: For arrays of type `kAEDataArray`, `kAEPackedArray`, or `kAEHandleArray`, the descriptor type of array items to be created.

itemSize
: For arrays of type `kAEDataArray` or `kAEPackedArray`, the size (in bytes) of the array items to be created.

itemCount
: The number of elements in the array.

DESCRIPTION

When you use `AEPutArray` to put an array into a factored descriptor list, each array item must include the data that is common to all the descriptor records in the list. The Apple Event Manager automatically isolates the data you specified in the call to `AECreateList` that is common to all the elements of the array.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record

SEE ALSO

For information about data types and constants used with `AEPutArray`, see "Apple Event Array Data Types" on page 4-60.

For more information about creating descriptor lists for Apple event arrays, see the description of `AECreateList` on page 5-28.

Adding Data and Descriptor Records to AE Records

The Apple Event Manager provides two routines that allow you to add data and descriptor records to AE records. The AEPutKeyPtr function takes a pointer to data, a descriptor type, and a keyword and converts them into a keyword-specified descriptor record that it adds to an AE record. The AEPutKeyDesc function takes a descriptor record and a keyword and converts them into a keyword-specified descriptor record that it adds to an AE record.

AEPutKeyPtr

You can use the AEPutKeyPtr function to add a pointer to data, a descriptor type, and a keyword to an AE record as a keyword-specified descriptor record.

```
FUNCTION AEPutKeyPtr (theAERecord: AERecord;
                      theAEKeyword: AEKeyword;
                      typeCode: DescType; dataPtr: Ptr;
                      dataSize: Size): OSErr;
```

theAERecord
            The AE record to which to add a keyword-specified
            descriptor record.

theAEKeyword
            The keyword that identifies the descriptor record. If the AE record already
            includes a descriptor record with this keyword, it is replaced.

typeCode   The descriptor type for the keyword-specified descriptor record.

dataPtr   A pointer to the data for the keyword-specified descriptor record.

dataSize   The length, in bytes, of the data for the keyword-specified descriptor
            record.

RESULT CODES

noErr	0	No error
memFullErr	−108	Not enough room in heap zone
errAEWrongDataType	−1703	Wrong descriptor type
errAENotAEDesc	−1704	Not a valid descriptor record
errAEBadListItem	−1705	Operation involving a list item failed

AEPutKeyDesc

You can use the AEPutKeyDesc function to add a descriptor record and a keyword to an AE record as a keyword-specified descriptor record.

```
FUNCTION AEPutKeyDesc (theAERecord: AERecord;
                       theAEKeyword: AEKeyword;
                       theAEDesc: AEDesc): OSErr;
```

theAERecord
              The AE record to which to add the keyword-specified descriptor record.

theAEKeyword
              The keyword specifying the descriptor record. If there was already a keyword-specified descriptor record with this keyword, it is replaced.

theAEDesc   The descriptor record for the keyword-specified descriptor record.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed

Adding Parameters and Attributes to Apple Events

The Apple Event Manager provides four functions that allow you to add Apple event parameters and attributes to an Apple event. The AEPutParamPtr and AEPutParamDesc functions add parameters to a specified Apple event. The AEPutAttributePtr and AEPutAttributeDesc functions add attributes to a specified Apple event.

AEPutParamPtr

You can use the AEPutParamPtr function to add a pointer to data, a descriptor type, and a keyword to an Apple event as an Apple event parameter.

```
FUNCTION AEPutParamPtr (theAppleEvent: AppleEvent;
                        theAEKeyword: AEKeyword;
                        typeCode: DescType; dataPtr: Ptr;
                        dataSize: Size): OSErr;
```

theAppleEvent
              The Apple event to which to add a parameter.

theAEKeyword
: The keyword for the parameter to be added. If the Apple event already included a parameter with this keyword, the parameter is replaced.

typeCode : The descriptor type for the parameter.

dataPtr : A pointer to the data for the parameter.

dataSize : The length, in bytes, of the data for the parameter.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed

SEE ALSO

For an example of the use of `AEPutParamPtr`, see "Adding Parameters to an Apple Event," which begins on page 5-5.

AEPutParamDesc

You can use the `AEPutParamDesc` function to add a descriptor record and a keyword to an Apple event as an Apple event parameter.

```
FUNCTION AEPutParamDesc (theAppleEvent: AppleEvent;
                         theAEKeyword: AEKeyword;
                         theAEDesc: AEDesc): OSErr;
```

theAppleEvent
: The Apple event to which to add a parameter.

theAEKeyword
: The keyword for the parameter to be added. If the Apple event already included a parameter with this keyword, the parameter is replaced.

theAEDesc : The descriptor record for the parameter.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed

AEPutAttributePtr

You can use the `AEPutAttributePtr` function to add a pointer to data, a descriptor type, and a keyword to an Apple event as an attribute.

```
FUNCTION AEPutAttributePtr (theAppleEvent: AppleEvent;
                            theAEKeyword: AEKeyword;
                            typeCode: DescType;
                            dataPtr: Ptr; dataSize: Size): OSErr;
```

theAppleEvent
:   The Apple event to which to add an attribute.

theAEKeyword
:   The keyword for the attribute to be added.

    `TYPE AEKeyword = PACKED ARRAY[1..4] OF Char;`

    The keyword can be any of the constants listed in the description that follows. If the Apple event already included an attribute with this keyword, the attribute is replaced.

typeCode
:   The descriptor type for the attribute.

dataPtr
:   A pointer to the buffer containing the data to be assigned to the attribute.

dataSize
:   The length, in bytes, of the data to be assigned to the attribute.

DESCRIPTION

The `AEPutAttributePtr` function adds the specified pointer to data, descriptor type, and keyword to the specified Apple event as an attribute. You can specify the parameter `theAEKeyWord` using any of the following constants:

```
CONST
   keyAddressAttr          = 'addr'; {address of target }
                                     { application}
   keyEventClassAttr       = 'evcl'; {event class}
   keyEventIDAttr          = 'evid'; {event ID}
   keyEventSourceAttr      = 'esrc'; {source application}
   keyInteractLevelAttr    = 'inte'; {settings to allow the }
                                     { Apple Event Manager to }
                                     { bring server application }
                                     { to the foreground}
```

```
keyMissedKeywordAttr    = 'miss'; {first required parameter }
                                  { remaining in Apple event}
keyOptionalKeywordAttr  = 'optk'; {list of optional }
                                  { parameters for Apple }
                                  { event}
keyOriginalAddressAttr  = 'from'; {address of original source }
                                  { of Apple event}
keyReturnIDAttr         = 'rtid'; {return ID for reply Apple }
                                  { event}
keyTimeoutAttr          = 'timo'; {length of time in ticks }
                                  { that client will wait }
                                  { for reply or result from }
                                  { the server}
keyTransactionIDAttr    = 'tran'; {transaction ID identifying }
                                  { a series of Apple events}
```

RESULT CODES

noErr	0	No error
memFullErr	−108	Not enough room in heap zone
errAECoercionFail	−1700	Data could not be coerced to the requested descriptor type
errAENotAEDesc	−1704	Not a valid descriptor record

AEPutAttributeDesc

You can use the AEPutAttributeDesc function to add a descriptor record and a keyword to an Apple event as an attribute.

```
FUNCTION AEPutAttributeDesc (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword;
                             theAEDesc: AEDesc): OSErr;
```

theAppleEvent
        The Apple event to which you are adding an attribute.

theAEKeyword
        The keyword for the attribute to be added.

        TYPE AEKeyword = PACKED ARRAY[1..4] OF Char;

        The keyword can be any of the constants listed in the description of
        AEPutAttributePtr on page 5-35. If the Apple event already included
        an attribute with this keyword, the attribute is replaced.

theAEDesc   The descriptor record to be assigned to the attribute. The descriptor type of the specified descriptor record should match the defined descriptor type for that attribute. For example, the `keyEventSourceAttr` attribute has the `typeShortInteger` descriptor type.

DESCRIPTION

The `AEPutAttributeDesc` function takes a descriptor record and a keyword and adds them to an Apple event as an attribute. If the descriptor type required for the attribute is different from the descriptor type of the descriptor record, the Apple Event Manager attempts to coerce the descriptor record into the required type, with one exception: the Apple Event Manager does not attempt to coerce the data for an address attribute, thereby allowing applications to use their own address types.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to the requested descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record

SEE ALSO

For an example of the use of `AEPutAttributeDesc`, see Listing 5-1 on page 5-9.

Sending Apple Events

The `AESend` function allows you to send an Apple event that you have previously created with the `AECreateAppleEvent` function.

AESend

You can use the `AESend` function to send an Apple event.

```
FUNCTION AESend (theAppleEvent: AppleEvent;
                 VAR reply: AppleEvent; sendMode: AESendMode;
                 sendPriority: AESendPriority;
                 timeOutInTicks: LongInt; idleProc: IdleProcPtr;
                 filterProc: EventFilterProcPtr): OSErr;
```

theAppleEvent
          The Apple event to be sent.

reply   The reply Apple event returned by the `AESend` function if you specify
                 the `kAEWaitReply` flag in the `sendMode` parameter. (If you specify the
                 `kAEQueueReply` flag in the `sendMode` parameter, you receive the reply
                 Apple event in your event queue.) If you specify `kAENoReply` flag, the
                 reply Apple event returned by this function is a null descriptor record. If
                 you specify `kAEWaitReply` in the `sendMode` parameter, your
                 application is responsible for using the `AEDisposeDesc` function to
                 dispose of the descriptor record returned in the `reply` parameter.

sendMode   Specifies the following: the reply mode for the Apple event (set with one
                 of the constants `kAENoReply`, `kAEQueueReply`, or `kAEWaitReply`);
                 the interaction level (set with one of the constants `kAENeverInteract`,
                 `kAECanInteract`, or `kAEAlwaysInteract`, which represent flags in
                 the `keyInteractLevelAttr` attribute); the application switch mode
                 (set with the `kAECanSwitchLayer` constant); the reconnection mode (set
                 with the `kAEDontReconnect` constant); and the return receipt
                 mode (set with the `kAEWantReceipt` constant). You obtain the value
                 for this parameter by adding the appropriate constants. (The description
                 that follows provides more details about the `sendMode` flags.)

sendPriority
                 An integer of data type `AESendPriority` that specifies whether the
                 Apple event is put at the back of the event queue (indicated by the
                 `kAENormalPriority` flag) or at the front of the queue (indicated by
                 the `kAEHighPriority` flag).

timeOutInTicks
                 If the reply mode specified in the `sendMode` parameter is
                 `kAEWaitReply`, or if a return receipt is requested, this parameter
                 specifies the length of time (in ticks) that the client application is willing
                 to wait for the reply or return receipt from the server application before
                 timing out. Most applications should use the `kAEDefaultTimeout`
                 constant, which tells the Apple Event Manager to provide an appropriate
                 timeout duration. If the value of this parameter is `kNoTimeOut`, the
                 Apple event never times out.

idleProc   A pointer to a function that handles events (such as update,
                 operating-system, activate, and null events) that your application receives
                 while waiting for a reply. Your application can also perform other tasks
                 (such as displaying a wristwatch or spinning beachball cursor) while
                 waiting for a reply or a return receipt. Your application must provide an
                 idle function if it specifies the `kAEWaitReply` flag in the `sendMode`
                 parameter.

filterProc
                 A pointer to a function that accepts certain incoming Apple events that
                 are received while the handler waits for a reply or a return receipt and
                 filters out the rest.

DESCRIPTION

You can use one of the following flags in the `sendMode` parameter to specify the reply mode for an Apple event. Only one of these flags may be set.

Flag	Description
kAENoReply	Your application does not want a reply Apple event; the server processes your Apple event as soon as it has the opportunity.
kAEQueueReply	Your application wants a reply Apple event; the reply appears in your event queue as soon as the server has the opportunity to process and respond to your Apple event.
kAEWaitReply	Your application wants a reply Apple event and is willing to give up the processor while waiting for the reply; for example, if the server application is on the same computer as your application, your application yields the processor to allow the server to respond to your Apple event. If you specify kAEWaitReply, you should provide an idle function.

You can communicate your user interaction preferences to the server application by specifying one of the following flags in the `sendMode` parameter. Only one of these flags may be set.

Flag	Description
kAENeverInteract	The server application should never interact with the user in response to the Apple event. If this flag is set, AEInteractWithUser returns the errAENoUserInteraction result code. This flag is the default when an Apple event is sent to a remote application.
kAECanInteract	The server application can interact with the user in response to the Apple event—by convention, if the user needs to supply information to the server. If this flag is set and the server allows interaction, AEInteractWithUser either brings the server application to the foreground or posts a notification request. This flag is the default when an Apple event is sent to a local application.
kAEAlwaysInteract	The server application can interact with the user in response to the Apple event—by convention, whenever the server application normally asks a user to confirm a decision or interact in any other way, even if no additional information is needed from the user. If this flag is set and the server allows interaction, AEInteractWithUser either brings the server application to the foreground or posts a notification request.

The flags in the following list specify the application switch mode, the reconnection mode, and the return receipt mode. Any of these flags may be set.

Flag	Description
kAECanSwitchLayer	If both the client and server allow interaction, and if the client application is the active application on the local computer and is waiting for a reply (that is, it has set the kAEWaitReply flag), AEInteractWithUser brings the server directly to the foreground. Otherwise, AEInteractWithUser uses the Notification Manager to request that the user bring the server application to the foreground.
kAEDontReconnect	The Apple Event Manager must not automatically try to reconnect if it receives a sessClosedErr result code from the PPC Toolbox.
kAEWantReceipt	The sender wants to receive a return receipt for this Apple event from the Event Manager. (A return receipt means only that the receiving application accepted the Apple event; the Apple event may or may not be handled successfully after it is accepted.) If the receiving application does not send a return receipt before the request times out, AESend returns errAETimeout as its function result.

If the Apple Event Manager cannot find a handler for an Apple event in either the application or system Apple event dispatch table, it returns the result code errAEEventNotHandled to the server application (as the result of the AEProcessAppleEvent function). If the client application is waiting for a reply, the Apple Event Manager also returns this result code to the client.

The AESend function returns noErr as its function result if the Apple event was successfully sent by the Event Manager. A noErr result from AESend does not indicate that the Apple event was handled successfully; it indicates only that the Apple event was successfully sent by the Event Manager. If the handler returns a result code other than noErr, and if the client is waiting for a reply, it is returned in the keyErrorNumber parameter of the reply Apple event.

If your application is sending an event to itself, you can set one of these flags to prevent the event from being recorded or to ask the Apple Event Manager to record the event without your application actually receiving it. Only one of these flags may be set.

Flag	Description
kAEDontRecord	Your application is sending an event to itself but does not want the event recorded. When Apple event recording is on, the Apple Event Manager records a copy of every event your application sends to itself except for those events for which this flag is set.
kAEDontExecute	Your application is sending an Apple event to itself for recording purposes only—that is, you want the Apple Event Manager to send a copy of the event to the recording process but you do not want your application actually to receive the event.

RESULT CODES

`noErr`	0	No error
`eLenErr`	–92	Buffer too big to send
`memFullErr`	–108	Not enough room in heap zone
`userCanceledErr`	–128	User canceled an operation
`procNotFound`	–600	No eligible process with specified process serial number
`connectionInvalid`	–609	Nonexistent signature or session ID
`noUserInteractionAllowed`	–610	Background application sends event requiring authentication
`noPortErr`	–903	Client hasn't set `'SIZE'` resource to indicate awareness of high-level events
`destPortErr`	–906	Server hasn't set `'SIZE'` resource to indicate awareness of high-level events, or else is not present
`sessClosedErr`	–917	The `kAEDontReconnect` flag in the `sendMode` parameter was set and the server quit, then restarted
`errAEEventNotHandled`	–1708	Event wasn't handled by an Apple event handler
`errAEUnknownSendMode`	–1710	Invalid sending mode was passed
`errAEWaitCanceled`	–1711	User canceled out of wait loop for reply or receipt
`errAETimeout`	–1712	Apple event timed out
`errAEUnknownAddressType`	–1716	Unknown Apple event address type

SEE ALSO

For more information on sending Apple events, see "Sending an Apple Event," which begins on page 5-13.

For information on writing an idle function, see "Writing an Idle Function," which begins on page 5-21.

For information on writing a reply filter function, see "Writing a Reply Filter Function," which begins on page 5-24.

For information on when to use the `kAEDontExecute` flag, see the chapter "Recording Apple Events" in this book.

Application-Defined Routines

If your application sends an Apple event using `AESend` and is waiting for a reply, or if it calls `AEInteractWithUser`, you can provide an idle function to handle update events, null events, operating-system events, and activate events. You can also provide a reply filter function that can handle any high-level events that you want your application to handle while it is waiting for a reply or for user interaction.

MyIdleFunction

An idle function has the following syntax:

```
FUNCTION MyIdleFunction (VAR event: EventRecord;
                         VAR sleepTime: LongInt;
                         VAR mouseRgn: RgnHandle): Boolean;
```

event   The event record of the event to process.

sleepTime   Amount of time (in ticks) during which your application agrees to relinquish the processor if no events are pending.

mouseRgn   A screen region that determines the conditions under which your application is to receive notice of mouse-moved events.

DESCRIPTION

If your application provides a pointer to an idle function (`MyIdleFunction`) as a parameter to `AESend` or `AEInteractWithUser`, the Apple Event Manager will call the idle function to handle any update event, null event, operating-system event, or activate event received for your application while it is waiting for a reply.

Set the function result to `TRUE` if your application is no longer willing to wait for a reply from the server or for the user to bring the application to the front. Set the function result to `FALSE` if your application is still willing to wait.

SEE ALSO

For more information, see "Writing an Idle Function," which begins on page 5-21.

MyReplyFilter

A reply filter function has the following syntax:

```
FUNCTION MyReplyFilter (VAR event: EventRecord;
                        returnID: LongInt;
                        transactionID: LongInt;
                        sender: AEAddressDesc): Boolean;
```

event   The event record for a high-level event. The next three parameters contain valid information only if the event is an Apple event.

returnID   Return ID for the Apple event.

transactionID
               Transaction ID for the Apple event.

sender   Address of process that sent the Apple event.

DESCRIPTION

If your application provides a pointer to a reply filter function as a parameter to the `AESend` function, the reply filter function can process any high-level events that it is willing to handle while your application is waiting for a reply.

Your reply filter function should return `TRUE` as the function result if you want to accept the Apple event; otherwise, it should return `FALSE`.

SEE ALSO

For more information, see "Writing a Reply Filter Function" on page 5-24.

Summary of Creating and Sending Apple Events

Pascal Summary

Constants

```
CONST
  gestaltAppleEventsAttr     = 'evnt';   {selector for Apple events}
  gestaltAppleEventsPresent  = 0;        {if this bit is set, then Apple }
                                         { Event Manager is available}


  {Apple event descriptor types}
  typeBoolean            = 'bool';   {1-byte Boolean value}
  typeChar               = 'TEXT';   {unterminated string}
  typeSMInt              = 'shor';   {16-bit integer}
  typeInteger            = 'long';   {32-bit integer}
  typeSMFloat            = 'sing';   {SANE single}
  typeFloat              = 'doub';   {SANE double}
  typeLongInteger        = 'long';   {32-bit integer}
  typeShortInteger       = 'shor';   {16-bit integer}
  typeLongFloat          = 'doub';   {SANE double}
  typeShortFloat         = 'sing';   {SANE single}
  typeExtended           = 'exte';   {SANE extended}
  typeComp               = 'comp';   {SANE comp}
  typeMagnitude          = 'magn';   {unsigned 32-bit integer}
  typeAEList             = 'list';   {list of descriptor records}
  typeAERecord           = 'reco';   {list of keyword-specified }
                                     { descriptor records}
  typeAppleEvent         = 'aevt';   {Apple event record}
  typeTrue               = 'true';   {TRUE Boolean value}
  typeFalse              = 'fals';   {FALSE Boolean value}
  typeAlias              = 'alis';   {alias record}
  typeEnumerated         = 'enum';   {enumerated data}
  typeType               = 'type';   {four-character code for }
                                     { event class or event ID}
  typeAppParameters      = 'appa';   {Process Manager launch parameters}
  typeProperty           = 'prop';   {Apple event property}
  typeFSS                = 'fss ';   {file system specification}
```

```
typeKeyword            = 'keyw';      {Apple event keyword}
typeSectionH           = 'sect';      {handle to a section record}
typeWildCard           = '****';      {matches any type}
typeApplSignature      = 'sign';      {application signature}
typeSessionID          = 'ssid';      {session reference number}
typeTargetID           = 'targ';      {target ID record}
typeProcessSerialNumber = 'psn ';     {process serial number}
typeNull               = 'null';      {NULL or nonexistent data}

{keywords for Apple event parameters}
keyDirectObject        = '----';      {direct parameter}
keyErrorNumber         = 'errn';      {error number parameter}
keyErrorString         = 'errs';      {error string parameter}
keyProcessSerialNumber = 'psn ';      {process serial number param}

{keywords for Apple event attributes}
keyTransactionIDAttr   = 'tran';      {transaction ID}
keyReturnIDAttr        = 'rtid';      {return ID}
keyEventClassAttr      = 'evcl';      {event class}
keyEventIDAttr         = 'evid';      {event ID}
keyAddressAttr         = 'addr';      {address of target or }
                                      { client application}
keyOptionalKeywordAttr = 'optk';      {list of optional parameters }
                                      { for the Apple event}
keyTimeoutAttr         = 'timo';      {number of ticks the client }
                                      { will wait}
keyInteractLevelAttr   = 'inte';      {settings to allow Apple Event }
                                      { Manager to bring server }
                                      { to foreground}
keyEventSourceAttr     = 'esrc';      {nature of source }
                                      { application}
keyMissedKeywordAttr   = 'miss';      {first required parameter }
                                      { remaining in an Apple event}
keyOriginalAddressAttr = 'from';      {address of original source; }
                                      { available only in version }
                                      { 1.01 and later versions of }
                                      { the Apple Event Manager}

{keywords for special handlers}
keyPreDispatch         = 'phac';      {identifies a handler routine }
                                      { called immediately before the }
                                      { Apple Event Manager dispatches }
                                      { an Apple event}
```

```
keySelectProc            = 'selh';        {selector used with }
                                          { AERemoveSpecialHandler to }
                                          { disable the OSL}


{keywords for use with AEManagerInfo; available only in version }
{ 1.0.1 and later versions of the Apple Event Manager}
keyAERecorderCount       = 'recr';        {keyword for recording info}
keyAEVersion             = 'vers';        {keyword for version info}


{event class}
kCoreEventClass          = 'aevt';        {event class for required Apple }
                                          { events}


{event IDs for required Apple events}
kAEOpenApplication       = 'oapp';        {event ID for Open }
                                          { Application event}
kAEOpenDocuments         = 'odoc';        {event ID for Open Documents event}
kAEPrintDocuments        = 'pdoc';        {event ID for Print Documents }
                                          { event}
kAEQuitApplication       = 'quit';        {event ID for Quit Application }
                                          { event}
kAEAnswer                = 'ansr';        {event ID for Apple event replies}
kAEApplicationDied       = 'obit';        {event ID for Application Died }
                                          { event}


{constants for setting the sendMode parameter of AESend}
kAENoReply               = $00000001;     {client doesn't want reply}
kAEQueueReply            = $00000002;     {client wants server to }
                                          { reply in event queue}
kAEWaitReply             = $00000003;     {client wants a reply and }
                                          { will give up processor}
kAENeverInteract         = $00000010;     {server application should }
                                          { not interact with user }
                                          { for this Apple event}
kAECanInteract           = $00000020;     {server may interact with }
                                          { user for this Apple event }
                                          { to supply information}

kAEAlwaysInteract        = $00000030;     {server may interact with user }
                                          { for this Apple event even if }
                                          { no information is required}
```

```
kAECanSwitchLayer       = $00000040;   {server should come directly }
                                       { to foreground when appropriate}
kAEDontReconnect        = $00000080;   {don't reconnect if there }
                                       { is a PPC session closed error}
kAEWantReceipt          = nReturnReceipt; {client wants return }
                                       { receipt}
kAEDontRecord           = $00001000;   {don't record this event}
kAEDontExecute          = $00002000;   {don't excecute this event}

{constants for setting the sendPriority parameter of AESend}
kAENormalPriority       = $00000000;   {put event at back of }
                                       { event queue}
kAEHighPriority         = nAttnMsg;    {put event at front of }
                                       { the event queue}

{event IDs for recording events; available only in version 1.01 and }
{ later versions of the Apple Event Manager}
kAEStartRecording       = 'reca';      {event ID for Start Recording }
                                       { event}
kAEStopRecording        = 'recc';      {event ID for Stop Recording }
                                       { event}
kAENotifyStartRecording = 'rec1';      {event ID for Recording On event}
kAENotifyStopRecording  = 'rec0';      {event ID for Recording Off event}
kAENotifyRecording      = 'recr';      {event ID for Receive Recordable }
                                       { Event event}

{constant for the returnID parameter of AECreateAppleEvent}
kAutoGenerateReturnID   = -1;          {tells Apple Event Manager to }
                                       { generate a unique return ID}

{constant for transaction IDs}
kAnyTransactionID       = 0;           {the Apple event is not }
                                       { part of a transaction}

{constants for timeout durations}
kAEDefaultTimeout       = -1;          {use default timeout value}
kNoTimeOut              = -2;          {never time out}

{constants for the dispatcher parameter of AEResumeTheCurrentEvent}
kAENoDispatch           = 0;           {don't redispatch the Apple event}
kAEUseStandardDispatch  = -1;          {redispatch the Apple event }
                                       { by using its entry in the }
                                       { Apple event dispatch table}
```

Data Types

```
TYPE
   AEEventClass =
      PACKED ARRAY[1..4] OF Char;          {event class for a high-level }
                                           { event}
   AEEventID =
      PACKED ARRAY[1..4] OF Char;          {event ID for a high-level }
                                           { event}

   AEKeyword =
      PACKED ARRAY[1..4] OF Char;          {keyword for a descriptor }
                                           { record}

   DescType               = ResType;       {descriptor type}

   AEDesc =                                {descriptor record}
   RECORD
      descriptorType:      DescType;       {type of data being passed}
      dataHandle:          Handle;         {handle to data being passed}
   END;

   AEKeyDesc =                             {keyword-specified }
   RECORD                                  { descriptor record}
      descKey:             AEKeyword;      {keyword}
      descContent:         AEDesc;         {descriptor record}
   END;

   AEAddressDesc          = AEDesc;        {address descriptor record}

   AEDescList             = AEDesc;        {list of descriptor records}

   AERecord               = AEDescList;    {list of keyword-specified }
                                           { descriptor records}

   AppleEvent             = AERecord;      {list of attributes and }
                                           { parameters necessary for }
                                           { an Apple event}

   AESendMode             = LongInt;       {flags that determine how }
                                           { an Apple event is sent}

   AESendPriority         = Integer;       {send priority of an Apple }
                                           { event}
```

```
AEInteractAllowed = (kAEInteractWithSelf, kAEInteractWithLocal,
                     kAEInteractWithAll); {what processes may }
                                          { interact with the user}


AEEventSource = (kAEUnknownSource, kAEDirectCall, kAESameProcess,
                 kAELocalProcess, kAERemoteProcess);
                                          {the source of an Apple }
                                          { event}


AEArrayType = (kAEDataArray, kAEPackedArray, kAEHandleArray,
               kAEDescArray, kAEKeyDescArray);
                                          {type of an Apple event array}
AEArrayData =
RECORD                                    {data for an Apple event array}
   CASE AEArrayType OF
   kAEDataArray:
      (AEDataArray: ARRAY[0..0] OF Integer);
   kAEPackedArray:
      (AEPackedArray: Packed Array[0..0] OF Char);
   kAEHandleArray:
      (AEHandleArray: Array[0..0] OF Handle);
   kAEDescArray:
      (AEDescArray: Array[0..0] OF AEDesc);
   kAEKeyDescArray:
      (AEKeyDescArray: Array[0..0] OF AEKeyDesc);
END;


AEArrayDataPointer = ^AEArrayData;


EventHandlerProcPtr = ProcPtr;           {pointer to an Apple event }
                                         { handler}
IdleProcPtr = ProcPtr;                   {pointer to an application's }
                                         { idle function}
EventFilterProcPtr = ProcPtr;            {pointer to an application's }
                                         { filter function}
```

Routines for Creating and Sending Apple Events

Creating Apple Events

```
FUNCTION AECreateAppleEvent (theAEEventClass: AEEventClass;
                             theAEEventID: AEEventID;
                             target: AEAddressDesc; returnID: Integer;
                             transactionID: LongInt;
                             VAR result: AppleEvent): OSErr;
```

Creating and Duplicating Descriptor Records

```
FUNCTION AECreateDesc         (typeCode: DescType; dataPtr: Ptr;
                               dataSize: Size; VAR result: AEDesc): OSErr;
FUNCTION AEDuplicateDesc      (theAEDesc: AEDesc; VAR result: AEDesc): OSErr;
```

Creating Descriptor Lists and AE Records

```
FUNCTION AECreateList         (factoringPtr: Ptr; factoredSize: Size;
                               isRecord: Boolean;
                               VAR resultList: AEDescList): OSErr;
```

Adding Items to Descriptor Lists

```
FUNCTION AEPutPtr             (theAEDescList: AEDescList; index: LongInt;
                               typeCode: DescType; dataPtr: Ptr;
                               dataSize: Size): OSErr;
FUNCTION AEPutDesc            (theAEDescList: AEDescList; index: LongInt;
                               theAEDesc: AEDesc): OSErr;
FUNCTION AEPutArray           (theAEDescList: AEDescList;
                               arrayType: AEArrayType;
                               arrayPtr: AEArrayDataPointer;
                               itemType: DescType; itemSize: Size;
                               itemCount: LongInt): OSErr;
```

Adding Data and Descriptor Records to AE Records

```
FUNCTION AEPutKeyPtr          (theAERecord: AERecord;
                               theAEKeyword: AEKeyword; typeCode: DescType;
                               dataPtr: Ptr; dataSize: Size): OSErr;
FUNCTION AEPutKeyDesc         (theAERecord: AERecord;
                               theAEKeyword: AEKeyword;
                               theAEDesc: AEDesc): OSErr;
```

Adding Parameters and Attributes to Apple Events

```
FUNCTION AEPutParamPtr       (theAppleEvent: AppleEvent;
                              theAEKeyword: AEKeyword; typeCode: DescType;
                              dataPtr: Ptr; dataSize: Size): OSErr;
FUNCTION AEPutParamDesc      (theAppleEvent: AppleEvent;
                              theAEKeyword: AEKeyword;
                              theAEDesc: AEDesc): OSErr;
FUNCTION AEPutAttributePtr   (theAppleEvent: AppleEvent;
                              theAEKeyword: AEKeyword; typeCode: DescType;
                              dataPtr: Ptr; dataSize: Size): OSErr;
FUNCTION AEPutAttributeDesc  (theAppleEvent: AppleEvent;
                              theAEKeyword: AEKeyword;
                              theAEDesc: AEDesc): OSErr;
```

Sending Apple Events

```
FUNCTION AESend              (theAppleEvent: AppleEvent;
                              VAR reply: AppleEvent; sendMode: AESendMode;
                              sendPriority: AESendPriority;
                              timeOutInTicks: LongInt;
                              idleProc: IdleProcPtr;
                              filterProc: EventFilterProcPtr): OSErr;
```

Application-Defined Routines

```
FUNCTION MyIdleFunction      (VAR event: EventRecord;
                              VAR sleepTime: LongInt;
                              VAR mouseRgn: RgnHandle): Boolean;
FUNCTION MyReplyFilter       (VAR event: EventRecord;
                              returnID: LongInt; transactionID: LongInt;
                              sender: AEAddressDesc): Boolean;
```

C Summary

Constants

```
enum {
    #define gestaltAppleEventsAttr    'evnt' /*selector for Apple events*/
    gestaltAppleEventsPresent       = 0   /*if this bit is set, then */
                                          /* Apple Event Manager is */
};                                        /* available*/
```

```
      /*Apple event descriptor types*/
enum  {
      typeBoolean            = 'bool',       /*1-byte Boolean value*/
      typeChar               = 'TEXT',       /*unterminated string*/
      typeSMInt              = 'shor',       /*16-bit integer*/
      typeInteger            = 'long',       /*32-bit integer*/
      typeSMFloat            = 'sing',       /*SANE single*/
      typeFloat              = 'doub',       /*SANE double*/
      typeLongInteger        = 'long',       /*32-bit integer*/
      typeShortInteger       = 'shor',       /*16-bit integer*/
      typeLongFloat          = 'doub',       /*SANE double*/
      typeShortFloat         = 'sing',       /*SANE single*/
      typeExtended           = 'exte',       /*SANE extended*/
      typeComp               = 'comp',       /*SANE comp*/
      typeMagnitude          = 'magn',       /*unsigned 32-bit integer*/
      typeAEList             = 'list',       /*list of descriptor records*/
      typeAERecord           = 'reco',       /*list of keyword-specified */
                                             /* descriptor records*/
      typeAppleEvent         = 'aevt',       /*Apple event record*/
      typeTrue               = 'true',       /*TRUE Boolean value*/
      typeFalse              = 'fals',       /*FALSE Boolean value*/
      typeAlias              = 'alis',       /*alias record*/
      typeEnumerated         = 'enum'        /*enumerated data*/
};

enum {
      typeType               = 'type',       /*four-character code for */
                                             /* event class or event ID*/
      typeAppParameters      = 'appa',       /*Process Manager launch */
                                             /* parameters*/
      typeProperty           = 'prop',       /*Apple event property*/
      typeFSS                = 'fss ',       /*file system specification*/
      typeKeyword            = 'keyw',       /*Apple event keyword*/

      typeSectionH           = 'sect',       /*handle to a section record*/
      typeWildCard           = '****',       /*matches any type*/
      typeApplSignature      = 'sign',       /*application signature*/
      typeSessionID          = 'ssid',       /*session ID*/
      typeTargetID           = 'targ',       /*target ID record*/
      typeProcessSerialNumber = 'psn ',      /*process serial number*/
      typeNull               = 'null'        /*NULL or nonexistent data*/
};
```

```
      /*keywords for Apple event parameters*/
enum {
      keyDirectObject        = '----',        /*direct parameter*/
      keyErrorNumber         = 'errn',        /*error number parameter*/
      keyErrorString         = 'errs',        /*error string parameter*/
      keyProcessSerialNumber = 'psn '         /*process serial number param*/
};
      /*keywords for Apple event attributes*/
enum {
      keyTransactionIDAttr   = 'tran',        /*transaction ID*/
      keyReturnIDAttr        = 'rtid',        /*return ID*/
      keyEventClassAttr      = 'evcl',        /*event class*/
      keyEventIDAttr         = 'evid',        /*event ID*/
      keyAddressAttr         = 'addr',        /*address of target or */
                                              /* client application*/
      keyOptionalKeywordAttr = 'optk',        /*list of optional parameters */
                                              /* for the Apple event*/
      keyTimeoutAttr         = 'timo',        /*number of ticks the client */
                                              /* will wait*/
      keyInteractLevelAttr   = 'inte',        /*settings to allow Apple */
                                              /* Event Mgr to bring */
                                              /* server to foreground*/
      keyEventSourceAttr     = 'esrc',        /*nature of source */
                                              /* application*/
      keyMissedKeywordAttr   = 'miss',        /*first required parameter */
                                              /* remaining in an Apple */
                                              /* event*/
      keyOriginalAddressAttr = 'from'         /*address of original source; */
                                              /* available only in version */
                                              /* 1.01 and later versions of */
                                              /* the Apple Event Manager*/
};


      /*keywords for special handlers*/
enum {
      keyPreDispatch         = 'phac',        /*identifies a handler */
                                              /* routine that is called */
                                              /* immediately before the */
                                              /* Apple Event Manager */
                                              /* dispatches an Apple event*/
      keySelectProc          = 'selh',        /*selector used with */
                                              /* AERemoveSpecialHandler to */
                                              /* disable the OSL*/
```

```
      /*keywords for use with AEManagerInfo, available only in version */
      /* 1.0.1 and later versions of the Apple Event Manager*/
      keyAERecorderCount      = 'recr',      /*keyword for recording info*/
      keyAEVersion            = 'vers',      /*keyword for version info*/


      /*event class*/
      kCoreEventClass         = 'aevt'       /*event class for required */
                                             /* Apple events*/
};


      /*event IDs for required Apple events*/
enum {
      kAEOpenApplication      = 'oapp',      /*event ID for Open */
                                             /* Application event*/
      kAEOpenDocuments        = 'odoc',      /*event ID for Open */
                                             /* Documents event*/

      kAEPrintDocuments       = 'pdoc',      /*event ID for Print */
                                             /* Documents event*/
      kAEQuitApplication      = 'quit',      /*event ID for Quit */
                                             /* Application event*/
      kAEAnswer               = 'ansr',      /*event ID for Apple event */
                                             /* replies*/
      kAEApplicationDied      = 'obit'       /*event ID for Application */
                                             /* Died event*/
};
      /*constants for setting the sendMode parameter of AESend*/
enum {
      kAENoReply              = 0x00000001,  /*client doesn't want reply*/
      kAEQueueReply           = 0x00000002,  /*client wants server to */
                                             /* reply in event queue*/
      kAEWaitReply            = 0x00000003,  /*client wants a reply and */
                                             /* will give up processor*/
      kAENeverInteract        = 0x00000010,  /*server application should */
                                             /* not interact with user */
                                             /* for this Apple event*/
      kAECanInteract          = 0x00000020,  /*server may interact with */
                                             /* user for this Apple event */
                                             /* to supply information*/
      kAEAlwaysInteract       = 0x00000030,  /*server may interact with */
                                             /* user for this Apple event */
                                             /* even if no information */
                                             /* is required*/
```

```
kAECanSwitchLayer       = 0x00000040,  /*server should come */
                                       /* directly to foreground */
                                       /* when appropriate*/
kAEDontReconnect        = 0x00000080,  /*don't reconnect if there */
                                       /* is a PPC session closed */
                                       /* error*/
kAEWantReceipt          = nReturnReceipt, /*client wants return */
                                       /* receipt*/
kAEDontRecord           = 0x00001000,  /*don't record this event*/
kAEDontExecute          = 0x00002000,  /*don't excecute this event*/

/*constants for setting the sendPriority parameter of AESend*/
kAENormalPriority       = 0x00000000,  /*post message at end of */
                                       /* event queue*/
kAEHighPriority         = nAttnMsg     /*post message at front of */
                                       /* event queue*/
};
/*event IDs for recording events; available only in version 1.01 and */
/* later versions of the Apple Event Manager*/
enum {
kAEStartRecording       = 'reca',      /*event ID for Start */
                                       /* Recording event*/
kAEStopRecording        = 'recc',      /*event ID for Stop */
                                       /* Recording event*/
kAENotifyStartRecording = 'rec1',      /*event ID for Recording On */
                                       /* event*/
kAENotifyStopRecording  = 'rec0',      /*event ID for Recording Off */
                                       /* event*/
kAENotifyRecording      = 'recr'       /*event ID for Receive */
                                       /* Recordable Event event*/
};
enum {
/*constant for the returnID parameter of AECreateAppleEvent*/
kAutoGenerateReturnID   = -1,          /*tells Apple Event Manager */
                                       /* to generate a unique */
                                       /* return ID*/
/*constant for transaction IDs*/
kAnyTransactionID       = 0,           /*the Apple event is not */
                                       /* part of a transaction*/

/*constants for timeout durations*/
kAEDefaultTimeout       = -1,          /*use default timeout value*/
kNoTimeOut              = -2,          /*never time out*/
```

```
      /*constants for the dispatcher parameter of AEResumeTheCurrentEvent*/
      kAENoDispatch          = 0,               /*don't redispatch the */
                                                /* Apple event*/
      kAEUseStandardDispatch  = -1             /*redispatch the Apple event */
                                                /* by using its entry in the */
                                                /* Apple event dispatch table*/
};
```

Data Types

```
typedef unsigned long AEEventClass;           /*event class for a */
                                              /* high-level event*/
typedef unsigned long AEEventID;              /*event ID for a high-level */
                                              /* event*/
typedef unsigned long AEKeyword;              /*keyword for a descriptor */
                                              /* record*/
typedef ResType DescType;                     /*descriptor type*/

struct AEDesc {                               /*descriptor record*/
   DescType descriptorType;                   /*type of data being passed*/
   Handle dataHandle;                         /*handle to data being passed*/
};
typedef struct AEDesc AEDesc;

struct AEKeyDesc {                            /*keyword-specified */
                                              /* descriptor record*/
   AEKeyword descKey;                         /*keyword*/
   AEDesc descContent;                        /*descriptor record*/
};
typedef struct AEKeyDesc AEKeyDesc;

typedef AEDesc AEAddressDesc;                 /*address descriptor record*/
typedef AEDesc AEDescList;                    /*list of descriptor records*/
typedef AEDescList AERecord;                  /*list of keyword-specified */
                                              /* descriptor records*/
typedef AERecord AppleEvent;                  /*list of attributes and */
                                              /* parameters necessary for */
                                              /* an Apple event*/
typedef long AESendMode;                      /*flags that determine how */
                                              /* an Apple event is sent*/
```

```
typedef short AESendPriority;                    /*send priority of an Apple */
                                                 /* event*/


enum { kAEInteractWithSelf, kAEInteractWithLocal,
       kAEInteractWithAll };                     /*what processes may */
typedef unsigned char AEInteractAllowed;         /* interact with the user*/


enum { kAEUnknownSource, kAEDirectCall, kAESameProcess, kAELocalProcess,
       kAERemoteProcess };                       /*the source of an Apple */
typedef unsigned char AEEventSource;             /* event*/
enum { kAEDataArray, kAEPackedArray, kAEHandleArray,
       kAEDescArray, kAEKeyDescArray };          /*type of an Apple event */
typedef unsigned char AEArrayType;               /* array*/


union AEArrayData {                              /*data for an Apple event */
   short kAEDataArray[1];                        /* array*/
   char kAEPackedArray[1];
   Handle kAEHandleArray[1];
   AEDesc kAEDescArray[1];
   AEKeyDesc kAEKeyDescArray[1];
};
typedef union AEArrayData AEArrayData;


typedef AEArrayData *AEArrayDataPointer;


typedef ProcPtr EventHandlerProcPtr;             /*pointer to an Apple event */
                                                 /* handler*/
typedef ProcPtr IdleProcPtr;                     /*pointer to an application's */
                                                 /* idle function*/
typedef ProcPtr EventFilterProcPtr;              /*pointer to an application's */
                                                 /* filter function*/
```

Routines for Creating and Sending Apple Events

Creating Apple Events

```
pascal OSErr AECreateAppleEvent
                        (AEEventClass theAEEventClass,
                         AEEventID theAEEventID,
                         const AEAddressDesc *target, short returnID,
                         long transactionID, AppleEvent *result);
```

Creating and Duplicating Descriptor Records

```
pascal OSErr AECreateDesc    (DescType typeCode, const void* dataPtr,
                              Size dataSize, AEDesc *result);
pascal OSErr AEDuplicateDesc
                             (const AEDesc *theAEDesc, AEDesc *result);
```

Creating Descriptor Lists and AE Records

```
pascal OSErr AECreateList    (const void* factoringPtr, Size factoredSize,
                              Boolean isRecord, AEDescList *resultList);
```

Adding Items to Descriptor Lists

```
pascal OSErr AEPutPtr        (const AEDescList *theAEDescList, long index,
                              DescType typeCode, const void* dataPtr,
                              Size dataSize);
pascal OSErr AEPutDesc       (const AEDescList *theAEDescList, long index,
                              const AEDesc *theAEDesc);
pascal OSErr AEPutArray      (const AEDescList *theAEDescList,
                              AEArrayType arrayType,
                              const AEArrayDataPointer *arrayPtr,
                              DescType itemType, Size itemSize,
                              long itemCount);
```

Adding Data and Descriptor Records to AE Records

```
pascal OSErr AEPutKeyPtr     (const AERecord *theAERecord,
                              AEKeyword theAEKeyword, DescType typeCode,
                              const void* dataPtr, Size dataSize);
pascal OSErr AEPutKeyDesc    (const AERecord *theAERecord,
                              AEKeyword theAEKeyword,
                              const AEDesc *theAEDesc);
```

Adding Parameters and Attributes to Apple Events

```
pascal OSErr AEPutParamPtr   (const AppleEvent *theAppleEvent,
                              AEKeyword theAEKeyword, DescType typeCode,
                              const void* dataPtr, Size dataSize);
pascal OSErr AEPutParamDesc  (const AppleEvent *theAppleEvent,
                              AEKeyword theAEKeyword,
                              const AEDesc *theAEDesc);
pascal OSErr AEPutAttributePtr
                             (const AppleEvent *theAppleEvent,
                              AEKeyword theAEKeyword, DescType typeCode,
                              const void* dataPtr, Size dataSize);
```

```
pascal OSErr AEPutAttributeDesc
                          (const AppleEvent *theAppleEvent,
                           AEKeyword theAEKeyword,
                           const AEDesc *theAEDesc);
```

Sending Apple Events

```
pascal OSErr AESend       (const AppleEvent *theAppleEvent,
                           AppleEvent *reply, AESendMode sendMode,
                           AESendPriority sendPriority,
                           long timeOutInTicks, IdleProcPtr idleProc,
                           EventFilterProcPtr filterProc);
```

Application-Defined Routines

```
pascal Boolean MyIdleFunction
                          (const EventRecord *event,
                           long *sleepTime, RgnHandle *mouseRgn);
pascal Boolean MyReplyFilter
                          (const EventRecord *event,
                           long returnID, long transactionID,
                           AEAddressDesc sender);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

_Pack8

Selector	Routine
$0405	AEDuplicateDesc
$0609	AEPutDesc
$0610	AEPutKeyDesc
$0610	AEPutParamDesc
$0627	AEPutAttributeDesc
$0706	AECreateList
$0825	AECreateDesc
$0A08	AEPutPtr
$0A0F	AEPutKeyPtr

Selector	Routine
$0A0F	AEPutParamPtr
$0A16	AEPutAttributePtr
$0B0D	AEPutArray
$0B14	AECreateAppleEvent
$0D17	AESend

Result Codes

noErr	0	No error
paramErr	−50	Parameter error (for example, value of handler pointer is NIL or odd)
eLenErr	−92	Buffer too big to send
memFullErr	−108	Not enough room in heap zone
userCanceledErr	−128	User canceled an operation
procNotFound	−600	No eligible process with specified process serial number
bufferIsSmall	−607	Buffer is too small
noOutstandingHLE	−608	No outstanding high-level event
connectionInvalid	−609	Nonexistent signature or session ID
noUserInteractionAllowed	−610	Background application sends event requiring authentication
noPortErr	−903	Client hasn't set 'SIZE' resource to indicate awareness of high-level events
destPortErr	−906	Server hasn't set 'SIZE' resource to indicate awareness of high-level events, or else is not present
sessClosedErr	−917	The kAEDontReconnect flag in the sendMode parameter was set, and the server quit and then restarted
errAECoercionFail	−1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	−1701	Descriptor record was not found
errAECorruptData	−1702	Data in an Apple event could not be read
errAEWrongDataType	−1703	Wrong descriptor type
errAENotAEDesc	−1704	Not a valid descriptor record
errAEBadListItem	−1705	Operation involving a list item failed
errAENewerVersion	−1706	Need a newer version of the Apple Event Manager
errAENotAppleEvent	−1707	Event is not an Apple event
errAEEventNotHandled	−1708	Event wasn't handled by an Apple event handler
errAEReplyNotValid	−1709	AEResetTimer was passed an invalid reply
errAEUnknownSendMode	−1710	Invalid sending mode was passed
errAEWaitCanceled	−1711	User canceled out of wait loop for reply or receipt
errAETimeout	−1712	Apple event timed out
errAENoUserInteraction	−1713	No user interaction allowed
errAENotASpecialFunction	−1714	The keyword is not valid for a special function
errAEParamMissed	−1715	Handler cannot understand a parameter the client considers required
errAEUnknownAddressType	−1716	Unknown Apple event address type

errAEHandlerNotFound	–1717	No handler found for an Apple event or a coercion, or no object callback function found
errAEReplyNotArrived	–1718	Reply has not yet arrived
errAEIllegalIndex	–1719	Not a valid list index
errAEImpossibleRange	–1720	The range is not valid because it is impossible for a range to include the first and last objects that were specified; an example is a range in which the offset of the first object is greater than the offset of the last object
errAEWrongNumberArgs	–1721	The number of operands provided for the kAENot logical operator is not 1
errAEAccessorNotFound	–1723	There is no object accessor function for the specified object class and token descriptor type
errAENoSuchLogical	–1725	The logical operator in a logical descriptor record is not kAEAnd, kAEOr, or kAENot
errAEBadTestKey	–1726	The descriptor record in a test key is neither a comparison descriptor record nor a logical descriptor record
errAENotAnObjectSpec	–1727	The objSpecifier parameter of AEResolve is not an object specifier record
errAENoSuchObject	–1728	A run-time resolution error, for example: object specifier record asked for the third element, but there are only two
errAENegativeCount	–1729	Object-counting function returned negative value
errAEEmptyListContainer	–1730	The container for an Apple event object is specified by an empty list
errAEUnknownObjectType	–1731	Descriptor type of token returned by AEResolve is not known to server application
errAERecordingIsAlreadyOn	–1732	Attempt to turn recording on when it is already on

Resolving and Creating Object Specifier Records

Contents

Resolving and Creating Object Specifier Records

This chapter describes how your application can use the Apple Event Manager and application-defined functions to resolve object specifier records. Your application must be able to resolve object specifier records to respond to core and functional-area Apple events defined in the *Apple Event Registry: Standard Suites.*

For example, after receiving a Get Data event that requests a table in a document, your application can use the Apple Event Manager and application-defined functions to parse the object specifier record in the direct parameter, locate the requested table, and send a reply Apple event containing the table's data back to the application that requested it.

This chapter also describes how your application can use the Apple Event Manager to create object specifier records. If you want to factor your application for Apple event recording, or if you want to send Apple events directly to other applications, you need to know how to create object specifier records.

To use this chapter, you should be familiar with the chapters "Introduction to Apple Events" and "Responding to Apple Events" in this book. The section "Working With Object Specifier Records," which begins on page 3-32, provides a general introduction to the subject.

If you plan to create object specifier records, you should also be familiar with the chapter "Creating and Sending Apple Events." If you are factoring your application, you should read the chapter "Recording Apple Events" before you write code for resolving or creating object specifier records.

This chapter begins with an overview of the way your application works with the Apple Event Manager to resolve object specifier records. It then describes

n   how the data in an object specifier record is organized

n   how to install entries in the object accessor tables

n   how to write object accessor and object callback functions

n   how to create an object specifier record

IMPORTANT

Versions 1.0 and 1.01 of the Apple Event Manager do not include the routines for resolving and creating object specifier records described in this chapter. To use these routines with those versions of the Apple Event Manager, you must link the Object Support Library (OSL) with your application when you build it, and call the `AEObjectInit` function before calling any of the routines. s

Resolving Object Specifier Records

If an Apple event parameter consists of an object specifier record, your handler for the Apple event should **resolve** the object specifier record: that is, locate the Apple event objects it describes. The first step is to call the `AEResolve` function with the object specifier record as a parameter.

The `AEResolve` function performs tasks that are required to resolve any object specifier record, such as parsing its contents, keeping track of the results of tests, and handling memory management. When necessary, `AEResolve` calls application-defined functions to perform tasks that are unique to the application, such as locating a specific Apple event object in the application's data structures or counting the number of Apple event objects in a container.

Note

Object specifier records are only valid while the Apple event that contains them is being handled. For example, if an application receives an Apple event asking it to cut row 5 of a table, what was row 6 then becomes row 5, and the original object specifier record that referred to row 5 no longer refers to the same row. u

The `AEResolve` function can call two kinds of application-defined functions. **Object accessor functions** locate Apple event objects. **Object callback functions** perform other tasks that only an application can perform, such as counting, comparing, or marking Apple event objects. This section provides an overview of the way `AEResolve` calls object accessor and object callback functions when it resolves object specifier records.

Each time `AEResolve` calls one of your application's object accessor functions successfully, the object accessor function should return a special descriptor record created by your application, called a **token,** that identifies either an element in a specified container or a property of a specified Apple event object. The Apple Event Manager examines the token's descriptor type but does nothing with the token's data. When it needs to refer to the object the token identifies, the Apple Event Manager simply passes the token back to your application.

Each object accessor function provided by your application should either find elements of a given object class in a container identified by a token of a given descriptor type, or find properties of an Apple event object identified by a token of a specified descriptor type. The Apple Event Manager uses the object class ID and the descriptor type of the token that identifies the object's container to determine which object accessor function to call.

It is up to you to decide how many object accessor functions you need to write for your application. You can write one object accessor function that locates Apple event objects of several different object classes, or you can write separate object accessor functions for certain object classes. Similarly, you may want to use only one descriptor type for all the tokens returned by your object accessor functions, or you may want to use several descriptor types. The way you define your tokens depends on the needs of your application.

You can use the `AEInstallObjectAccessor` function to create an **object accessor dispatch table** that the Apple Event Manager uses to map requests for Apple event objects to the appropriate object accessor function in your application. The Apple Event Manager uses the object class of each requested object and the descriptor type of the token that identifies the object's container to determine which object accessor function to call. Depending on the container hierarchy for a given object specifier record and the way your application defines its object accessor functions, the Apple Event Manager may need to call a series of object accessor functions to resolve the nested object specifier records that describe an Apple event object's container. For information about creating and using the object accessor dispatch table, see "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

Figure 6-1 illustrates the major steps involved in resolving an object specifier record. The SurfWriter application shown in Figure 6-1 receives a Get Data event whose direct parameter is an object specifier record for a table named "Summary of Sales" in a document named "Sales Report." The SurfWriter application's handler for the Get Data event calls the `AEResolve` function with the object specifier record as a parameter. The `AEResolve` function begins to parse the object specifier record. The first object accessor function that `AEResolve` calls is usually the function that can identify the Apple event object in the application's **default container**—the outermost container in the container hierarchy. In Figure 6-1, the object specifier record for the document "Sales Report" specifies the default container, so the Apple Event Manager calls the object accessor function in the SurfWriter application that can locate a document in a container identified by a descriptor record of descriptor type `typeNull`.

Figure 6-1 Resolving an object specifier record for a table in a document
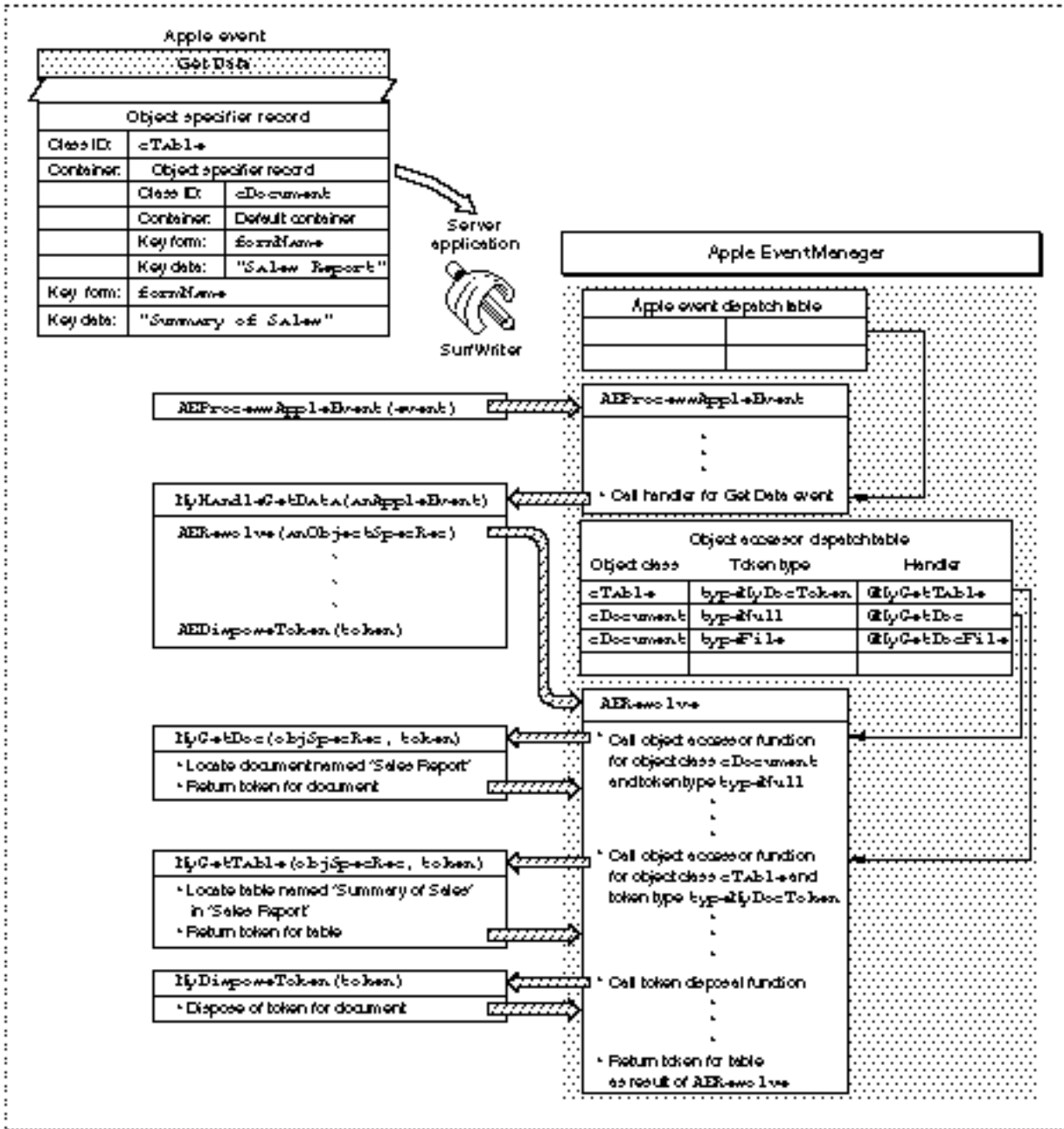
After locating the document named "Sales Report," the SurfWriter application returns a token to the Apple Event Manager—that is, a descriptor record that SurfWriter uses to identify the document. The Apple Event Manager examines the descriptor type of the token but does not need to know anything about the token's data to continue parsing the object specifier record. Next, the Apple Event Manager calls the object accessor function that can identify a table in a container identified by a token of descriptor type typeMyDocToken. When the Apple Event Manager calls this object accessor function, it uses the token that describes the document to identify the table's container. After the SurfWriter application has located the table named "Summary of Sales" in the document named "Sales Report," it returns a token describing that table to the Apple Event Manager.

After your application has successfully located an Apple event object, the Apple Event Manager disposes of all previous tokens returned during resolution of the object specifier record for the object. The Apple Event Manager disposes of tokens by calling either the AEDisposeDesc function or your application's **token disposal function,** if you have provided one, which is an object callback function that disposes of a token. In Figure 6-1, the AEResolve function calls the SurfWriter application's token disposal function to dispose of the token for the document after AEResolve receives the token for the table. After the SurfWriter application has disposed of the token for the document, the AEResolve function returns the result of the resolution—that is, the token for the requested table—to the handler in the SurfWriter application that originally called AEResolve.

The Apple Event Manager can complete the cycle of parsing the object specifier record and calling the appropriate object accessor function to obtain a token as many times as necessary to identify every container in the container hierarchy and finish resolving an object specifier record, including disposing of the tokens for the containers. However, one token will always be left over—the token that identifies the requested Apple event object. After AEResolve returns this final token and your application performs the action requested by the Apple event, it is up to your application to dispose of the token. Your application can do so by calling the AEDisposeToken function, which in turn calls either AEDisposeDesc or your application's token disposal function.

You need to provide a token disposal function only if a call to AEDisposeDesc is not sufficient by itself to dispose of a token or if you provide **marking callback functions,** which are three object callback functions that allow your application to use its own marking scheme rather than tokens when identifying large groups of Apple event objects. Your application is not required to provide marking callback functions.

To handle object specifier records that specify a test, your application must provide two object callback functions: (a) an **object-counting function,** which counts the number of elements of a given object class in a given container so that the Apple Event Manager can determine how many elements it must test to find the element or elements that meet a specified condition, and (b) an **object-comparison function,** which compares one element to another element or to a descriptor record and returns TRUE or FALSE.

Your application may also provide an **error callback function** that can identify which descriptor record caused the resolution of an object specifier record to fail. Your application is not required to provide an error callback function.

If your application resolves object specifier records without the help of the Apple Event Manager, it must extract the equivalent descriptor records and coerce them as necessary to get access to their data. The Apple Event Manager includes coercion handlers for these coercions; for information about this default coercion handling, see Table 4-1 on page 4-43.

For more information about object accessor functions, see "Writing Object Accessor Functions," which begins on page 6-28. For more information about object callback functions, see "Writing Object Callback Functions," which begins on page 6-45.

The next section, "Descriptor Records Used in Object Specifier Records," describes how the data in an object specifier record is interpreted by the Apple Event Manager.

Descriptor Records Used in Object Specifier Records

An object specifier record is a coerced AE record of descriptor type `typeObjectSpecifier`. The data to which its data handle refers consists of four keyword-specified descriptor records:

Keyword	Value	Description of data
keyAEDesiredClass	'want'	A four-character code for the object class
keyAEContainer	'from'	An object specifier record (or in some cases a descriptor record with a handle whose value is NIL) that identifies the container for the requested objects
keyAEKeyForm	'form'	A four-character code for the key form
keyAEKeyData	'seld'	Data or nested descriptor records that specify a property, name, position, range, or test, depending on the key form

This section describes the descriptor types and data associated with each of these keywords. You need this information if your application resolves or creates object specifier records.

For a summary of the descriptor types and key forms discussed in this section, see Table 6-11 on page 6-76. For an overview of object specifier records, see "Working With Object Specifier Records," which begins on page 3-32.

Object Class

The object class of the requested objects is identified by an object class ID. The corresponding keyword-specified descriptor record takes this form:

Keyword	Descriptor type	Data
keyAEDesiredClass	typeType	Object class ID

The *Apple Event Registry: Standard Suites* defines constants for the standard object class IDs.

Container

The container for the requested objects is usually the object in which they are located. It can be identified in one of four ways:

Keyword	Descriptor type	Data
keyAEContainer	typeObjectSpecifier	Object specifier record.
	typeNull	Value of data handle is NIL. Specifies the default container at the top of the container hierarchy.
	typeObjectBeingExamined	Value of data handle is NIL. Specifies the container for elements that are tested one at a time; used only within key data for key form formTest.
	typeCurrentContainer	Value of data handle is NIL. Specifies a container for an element that demarcates one boundary in a range. Used only within key data for key form formRange.

The data that describes a container usually consists of another object specifier record. The ability to nest one object specifier record within another in this way makes it possible to identify a chain of containers that fully describes the location of one or more Apple event objects.

For example, Figure 6-2 shows nested object specifier records that specify the first row of a table named "Summary of Sales" in a document named "Sales Report." The container specified by the object specifier record at the bottom of the figure describes the outermost container in the container hierarchy—the container for the document "Sales Report."

Because a container must be specified for each Apple event object in a container hierarchy, a **null descriptor record** (that is, a descriptor record whose descriptor type is typeNull and whose data handle has the value NIL) is used to specify the application's default container—the outermost container for any container hierarchy in that application.

Figure 6-2 Nested object specifier records that specify a container hierarchy
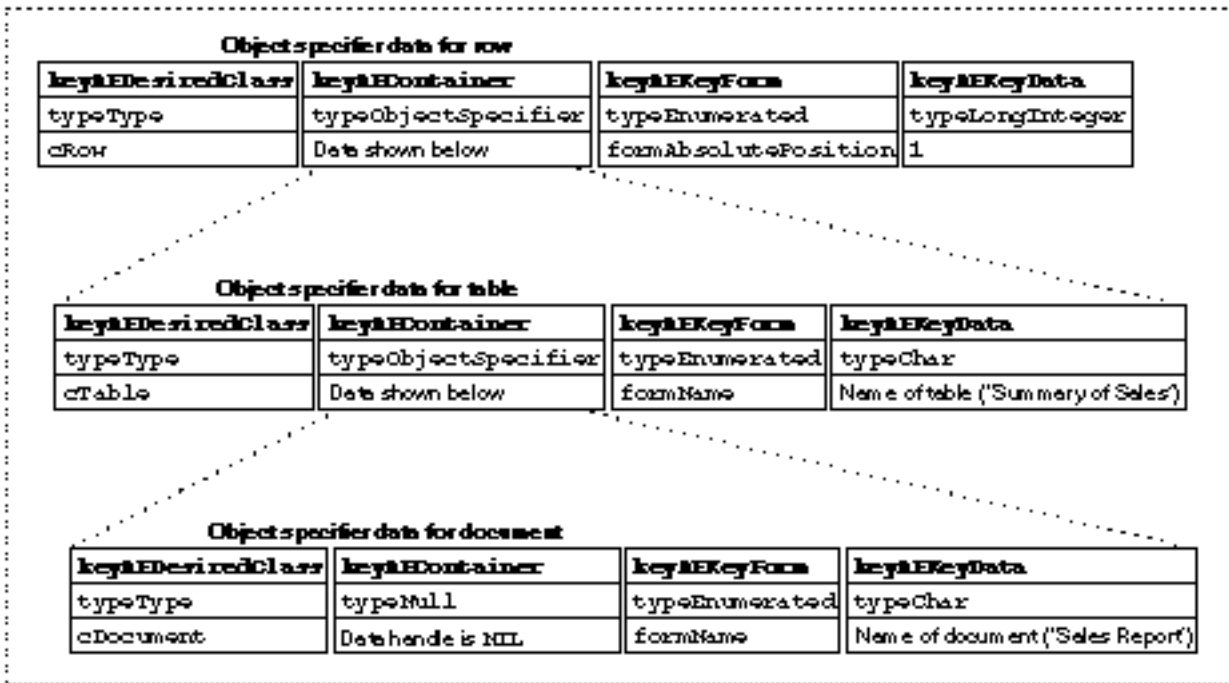
Note

The format used in Figure 6-2 and similar figures throughout this chapter does not show the structure of the nested object specifier records as they exist within an Apple event. Instead, these figures show what you would obtain after calling AEGetKeyDesc repeatedly to extract the object specifier records from an Apple event record.

When you call AEGetKeyDesc to extract a null descriptor record, the function returns a descriptor record of type AEDesc with a descriptor type of typeNull and a data handle whose value is 0. u

The object specifier data at the bottom of Figure 6-2 uses a null descriptor record to specify the document's container—that is, the default container for the application. The object specifier record for the document identifies the document named "Sales Report"; the object specifier record for the table identifies the table named "Summary of Sales" in the document "Sales Report"; and the object specifier record for the row identifies the first row of the table named "Summary of Sales" in the document "Sales Report."

An object specifier record in an Apple event parameter almost always includes nested object specifier records that specify the container hierarchy for the requested Apple event object. For the nested object specifier records shown in Figure 6-2, the relationship between each Apple event object and its container is always simple containment: it is located inside its container.

In other cases, the specified container may not actually contain the requested Apple event object. Instead, the relationship between a "container" and a specified object can be defined differently, depending on the key form. For example, the key form `formRelativePosition` indicates that the requested object is before or after its container.

Object specifier records that specify the key form `formTest` or `formRange` require key data that consists of several nested descriptor records, including additional object specifier records that identify either a group of elements to be tested or the boundary elements that demarcate a range. These object specifier records use two special descriptor types to specify containers: `typeObjectBeingExamined` (see page 6-19), which specifies a container that changes as a group of elements are tested one at a time, and `typeCurrentContainer` (see page 6-20), which specifies the container for a boundary element in a range. Both of these descriptor types require a data handle whose value is `NIL`, since they act much like variables whose value is supplied by the Apple Event Manager according to other information provided in the container hierarchy.

Key Form

The key form indicates how the key data should be interpreted. It can be specified by one of eight constants:

Keyword	Descriptor type	Data
keyAEKeyForm	typeEnumerated	formPropertyID
		formName
		formUniqueID
		formAbsolutePosition
		formRelativePosition
		formTest
		formWhose
		formRange

The next section describes the key data that corresponds to each key form.

Key Data

The nature of the information provided by the key data depends both on the specified key form and on the descriptor type of the descriptor record for the key data. Table 6-1 summarizes these relationships for the standard key forms.

Table 6-1   Standard descriptor types used with `keyAEKeyData`

Key form	Descriptor type	Data
formPropertyID	typeType	Property ID for an element's property
formName	typeChar or other text type	Element's name
formUniqueID	Any appropriate type	A value that uniquely identifies an object within its container or across an application
formAbsolutePosition	typeLongInteger	Offset from beginning (positive) or end (negative) of container
	typeAbsoluteOrdinal	kAEFirst kAEMiddle kAELast kAEAny kAEAll
formRelativePosition	typeEnumerated	kAENext kAEPrevious
formTest	typeCompDescriptor	(see Table 6-2 on page 6-16)
	typeLogicalDescriptor	(see Table 6-3 on page 6-17)
formRange	typeRangeDescriptor	(see Table 6-4 on page 6-20)
formWhose	typeWhoseDescriptor	(see Table 6-5 on page 6-42)

Most applications that resolve object specifier records need to support only the key forms `formPropertyID`, `formName`, `formUniqueID`, `formAbsolutePosition`, `formRelativePosition`, and `formRange` explicitly. You do not need to support these key forms for all object classes; for example, words usually do not have names, so most applications should return `errAEEventNotHandled` if they receive a request for a word by name.

If your application provides an object-counting function and an object-comparison function in addition to the appropriate object accessor functions, the Apple Event Manager can handle `formTest` automatically.

The Apple Event Manager uses the key form `formWhose` internally to optimize resolution of object specifier records that specify `formTest`. Applications that translate tests into their own query languages need to support `formWhose` explicitly. "Handling Whose Tests," which begins on page 6-41, describes `formWhose` in detail.

You can define custom key forms and the format for corresponding data for use by your own application if necessary. If you think you need to do this, check with the Apple Event Registrar first to find out whether existing key forms or others still under development can be adapted to the needs of your application.

One simple kind of key form involves identifying an object on the basis of a specified property. For example, the corresponding data for key form `formUniqueID` (defined in the *Apple Event Registry: Standard Suites*) always consists of a unique ID for the requested object. This ID is stored as a property identified by the constant `pID`. The four-character code that corresponds to both `formUniqueID` and `pID` is `'ID  '`.

If you discover that you do need to define a custom key form based on a property, use the same four-character code for both the key form and the associated property.

The rest of this section describes how the key data for the other key forms shown in Table 6-1 identifies Apple event objects.

Key Data for a Property ID

The key data for `formPropertyID` is specified by a descriptor record of descriptor type `typeType`. The *Apple Event Registry: Standard Suites* defines constants for the standard property IDs.

An object specifier record for a property specifies `cProperty` as the object class ID, an object specifier record for the object that contains the property as the container, `formPropertyID` as the key form, and a constant such as `pFont` as the key data. For example, if you were sending a Set Data event to change the font of a word to Palatino®, you could specify the data for the object specifier record in the direct parameter as follows:

Keyword	Descriptor type	Data
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeObjectSpecifier	**Object specifier record for word to which property belongs**
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeType	pFont

In this example, the Set Data Apple event parameter identified by the keyword `keyAETheData` would specify `Palatino` as the value to which to set the specified property. The reply Apple event for a subsequent Get Data event that included an object specifier record for the same property would return `Palatino` in the parameter identified by the keyword `keyAEResult`.

Key Data for an Object's Name

The key data for `formName` is specified by a descriptor record whose data consists of text, with a descriptor type such as `typeChar` or `typeIntlText`.

Figure 6-2 on page 6-10 includes two object specifier records that specify `formName`.

Key Data for a Unique ID

The key data for `formUniqueID` consists of a value that identifies an object. This ID must be unique either within the container, at a minimum, or unique across the application. A unique ID can be specified by a descriptor record of any appropriate type; for example, type `typeInteger`.

Key Data for Absolute Position

The key data for `formAbsolutePosition` consists of an integer that specifies either an offset or an ordinal position. For descriptor type `typeLongInteger`, the data is either a positive integer, indicating the offset of the requested element from the beginning of the container, or a negative integer, indicating its offset from the end of the container. The first object specifier record shown in Figure 6-2 on page 6-10 specifies `formAbsolutePosition` with key data that consists of the positive integer 1.

For descriptor type `typeAbsoluteOrdinal`, the data consists of one of these constants:

Constant	Meaning
kAEFirst	The first element in the specified container
kAEMiddle	The element in the middle of the specified container
kAELast	The last element in the specified container
kAEAny	A single element chosen at random from the specified container
kAEAll	All the elements in the specified container

If an object specifier record specifies `kAEMiddle` and the number of elements in the container is even, the Apple Event Manager rounds down; for example, the second word would be the "middle" word in a range of four words.

Key Data for Relative Position

The key data for `formRelativePosition` is specified by a descriptor record of type `typeEnumerated` whose data consists of one of these constants:

Constant	Meaning
kAENext	The Apple event object after the specified container
kAEPrevious	The Apple event object before the specified container

The "container" can be a single Apple event object or a group of Apple event objects; the requested elements are located immediately before or immediately after it, not inside it.

If your application can locate objects of the same class by absolute position, it can easily locate the same objects by relative position. For example, all applications that support `formAbsolutePosition` can easily locate the table immediately after a container specified as another table named "Summary of Sales."

Some applications may also be able to locate an object of one class before or after an object of another class. For example, a word processor might be able to locate the paragraph immediately after a container specified as a table named "Summary of Sales."

Key Data for a Test

The key data for `formTest` is specified by either a comparison descriptor record or a logical descriptor record. If your application provides an object-counting function and an object-comparison function in addition to the appropriate object accessor functions, the Apple Event Manager can handle `formTest` for you. Some applications may perform tests more efficiently by translating them into the application's own query language. For information about handling tests yourself, see "Handling Whose Tests," which begins on page 6-41.

The container for objects that pass a test can be one or more Apple event objects. The objects specified are those in the container that pass the test specified by the key data. For example, an object specifier record can describe "the first row in which the First Name column equals 'John' and the Last Name column equals 'Chapman' in the table 'MyAddresses' of the database 'SurfDB.'" To resolve such an object specifier record, the Apple Event Manager must evaluate a logical expression that applies the logical operator AND to two separate comparisons for each row: a comparison of the First Name column to the word "John" and a comparison of the Last Name column to the word "Chapman."

The Apple Event Manager evaluates comparisons and logical expressions on the basis of the information in comparison descriptor records and logical descriptor records. A **comparison descriptor record** is a coerced AE record of type `typeCompDescriptor` that specifies an Apple event object and either another Apple event object or data for the Apple Event Manager to compare to the first object. The Apple Event Manager can also use the information in a comparison descriptor record to compare elements in a container, one at a time, either to an Apple event object or to data. The data for a comparison descriptor record consists of three keyword-specified descriptor records with the descriptor types and data shown in Table 6-2.

Table 6-2   Keyword-specified descriptor records for `typeCompDescriptor`

Keyword	Descriptor type	Data
`keyAECompOperator`	`typeType`	`kAEGreaterThan` `kAEGreaterThanEquals` `kAEEquals` `kAELessThan` `kAELessThanEquals` `kAEBeginsWith` `kAEEndsWith` `kAEContains`
`keyAEObject1`	`typeObjectSpecifier`	Object specifier data
	`typeObjectBeingExamined`	Value of data handle is `NIL`
`keyAEObject2`	`typeObjectSpecifier`	Object specifier data for object to be compared
	`typeObjectBeingExamined`	Value of data handle is `NIL`
	any other type (`AEDesc`)	Data to be compared

The keyword `keyAEObject1` identifies a descriptor record for the element that is currently being compared to the object or data specified by the descriptor record for the keyword `keyAEObject2`. Either object can be described by a descriptor record of type `typeObjectSpecifier` or `typeObjectBeingExamined`. A descriptor record of `typeObjectBeingExamined` acts as a placeholder for each of the successive elements in a container when the Apple Event Manager tests those elements one at a time. The keyword `keyAEObject2` can also be used with a descriptor record of any other descriptor type whose data is to be compared to each element in a container.

You don't have to support all the available comparison operators for all Apple event objects; for example, the "begins with" operator probably doesn't make sense for objects of type `cRectangle`. It is up to you to decide which comparison operators are appropriate for your application to support, and how to interpret them.

If necessary, you can define your own custom comparison operators. If you think you need to do this, check with the Apple Event Registrar first to find out whether existing definitions of comparison operators or definitions still under development can be adapted to the needs of your application.

A **logical descriptor record** is a coerced AE record of type `typeLogicalDescriptor` that specifies a logical expression—that is, an expression that the Apple Event Manager evaluates to either `TRUE` or `FALSE`. The logical expression is constructed from a logical operator (one of the Boolean operators `AND`, `OR`, or `NOT`) and a list of logical terms to which the operator is applied. Each logical term in the list can be either another logical descriptor record or a comparison descriptor record. The Apple Event Manager short-circuits its evaluation of a logical expression as soon as one part of the expression fails a test. For example, if while testing a logical expression such as A `AND` B `AND` C the Apple Event Manager discovers that A `AND` B is not true, it will evaluate the expression to `FALSE` without testing C.

The data for a logical descriptor record consists of two keyword-specified descriptor records with the descriptor types and data shown in Table 6-3.

Table 6-3   Keyword-specified descriptor records for `typeLogicalDescriptor`

Keyword	Descriptor type	Data

`keyAELogicalOperator`	`typeEnumerated`	`kAEAND`
		`kAEOR`
		`kAENOT`

`keyAELogicalTerms`	`typeAEList`	One or more comparison or logical descriptor records

If the logical operator is `AND` or `OR`, the list can contain any number of logical terms, and the logical operator is applied to all the terms in the list. For example, the logical descriptor data shown in Figure 6-4 on page 6-19 consists of the logical operator `AND` and a list of logical terms that contains two comparison descriptor records. The entire logical descriptor record corresponds to the logical expression "the First Name column equals 'John' `AND` the Last Name column equals 'Chapman.'" If the logical operator is `NOT`, the list must contain a single term.

Figure 6-3 shows four object specifier records that specify the container hierarchy for the first row in the table "MyAddresses" of the database "SurfDB" that meets a test. The object specifier record at the top of Figure 6-3 specifies the first row contained in the set of rows that form its container. The container for the first row is specified by an object specifier record for a set of rows that meet a test. The two object specifier records at the bottom of Figure 6-3 specify the table named "MyAddresses," which contains the rows to be tested, in the database named "SurfDB."

Figure 6-3   The container hierarchy for the first row in a table that meets a test
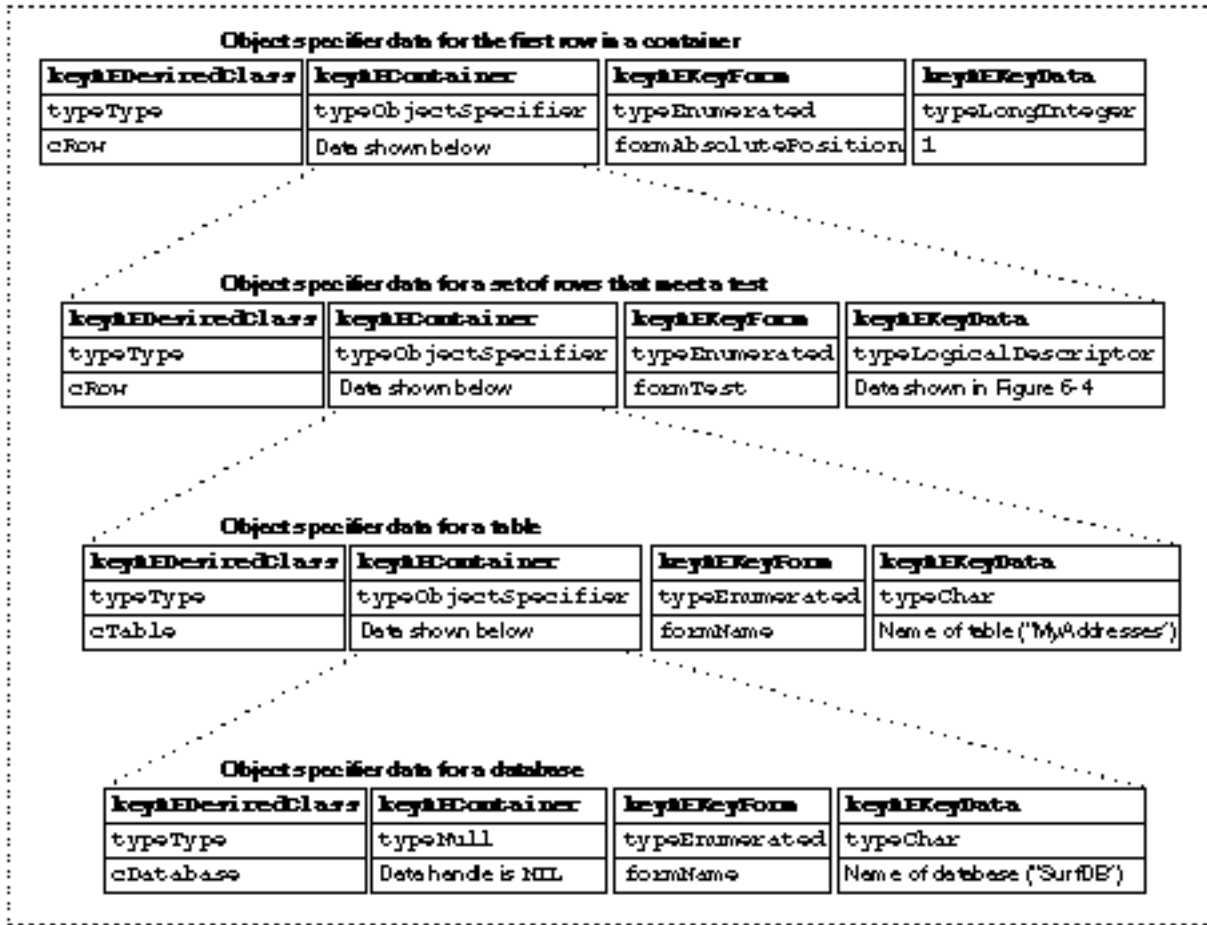
The object specifier record in Figure 6-3 for a set of rows that meet a test specifies
`formTest`. The corresponding key data consists of the logical descriptor record shown
in Figure 6-4, which applies the logical operator AND to two logical terms: a comparison
descriptor record that specifies all the rows in the container (the table "MyAddresses")
in which the column named "First Name" equals "John," and another comparison
descriptor record that specifies all the rows in which the column named "Last Name"
equals "Chapman." A row in the table "MyAddresses" passes the test only if both
comparison descriptor records evaluate as TRUE.

Figure 6-4   A logical descriptor record that specifies a test
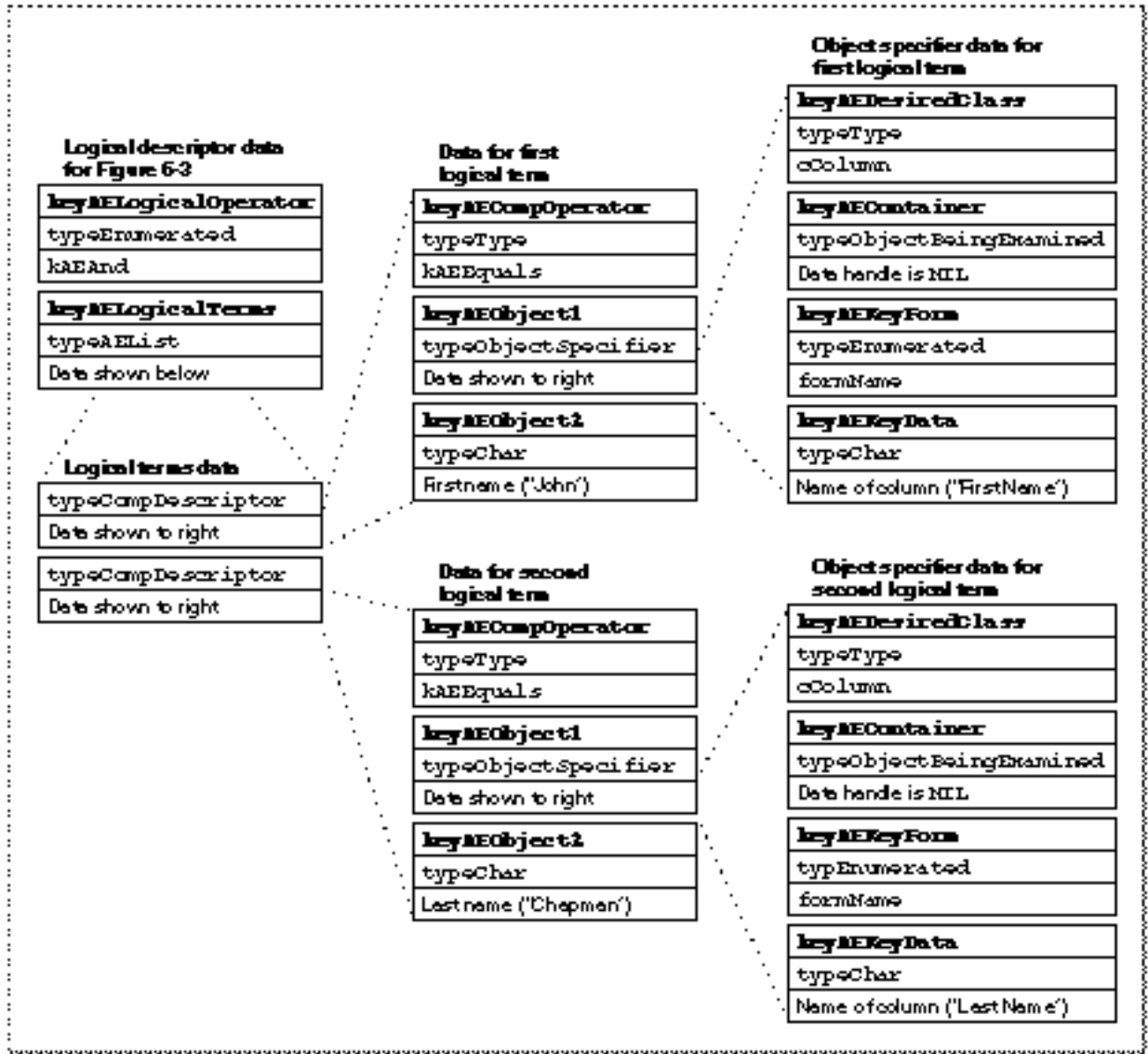
The keyword-specified descriptor records with the keyword `keyAEObject1` in Figure 6-4 each consist of an object specifier record that identifies a column by name. The row for each column is specified by a descriptor record of `typeObjectBeingExamined`, which acts as a placeholder for each row as the Apple Event Manager tests successive rows in the table. The Apple event object specified by each of these object specifier records consists of a column in the row. The Apple Event Manager (with the help of an object-comparison function) compares the contents of the column in successive rows to the string identified by the keyword `keyAEObject2` using the comparison operator identified by the keyword `keyAECompOperator`.

Key Data for a Range

The key data for `formRange` is specified by a **range descriptor record,** which is a coerced AE record of type `typeRangeDescriptor` that identifies two Apple event objects marking the beginning and end of a range of elements. The data for a range descriptor record consists of two keyword-specified descriptor records with the descriptor types and data shown in Table 6-4.

Table 6-4   Keyword-specified descriptor records in a descriptor record of type
                  `typeRangeDescriptor`

Keyword	Descriptor type	Data
keyAERangeStart	typeObjectSpecifier	An object specifier record for the first Apple event object in the desired range
keyAERangeStop	typeObjectSpecifier	An object specifier record for the last Apple event object in the desired range

The elements that identify the beginning and end of the range, which are known as **boundary objects,** do not have to belong to the same object class as the elements in the range itself. If the boundary objects belong to the same object class as the elements in the range, the boundary objects are included in the range. For example, the range of tables specified by boundary elements that are also tables would include the two boundary tables.

The container for boundary objects is usually the same as the container for the entire range, in which case the container for a boundary object can be specified by a placeholder—that is, a descriptor record of type `typeCurrentContainer` whose data handle has the value `NIL`.

When `AEResolve` calls an object accessor function to locate a range of objects, the Apple Event Manager replaces the descriptor record of type `typeCurrentContainer` with a token for the container of each boundary object. When using `AEResolve` to resolve the object specifier record, your application doesn't need to examine the contents of this token, because the Apple Event Manager keeps track of it. If your application attempts to resolve some or all of the object specifier record without calling `AEResolve`, the application may need to examine the token before it can locate the boundary objects. The token provided by the Apple Event Manager for a boundary object's container is a descriptor record of type `typeToken` whose data handle refers to a structure of type `ccntTokenRecord`.

```
TYPE ccntTokenRecord =
RECORD
    tokenClass:     DescType;       {class ID of container }
                                    { represented by token}
    token:          AEDesc;         {token for current container}
END;
```

This data type is of interest only if you attempt to resolve an object specifier record for a range without calling `AEResolve`. Otherwise, the Apple Event Manager keeps track of the container.

Installing Entries in the Object Accessor Dispatch Tables

If the direct parameter for an Apple event consists of an object specifier record, your handler for the event should call the `AEResolve` function to resolve the object specifier record: that is, to find the Apple event objects or properties it describes. The `AEResolve` function resolves the object specifier record with the help of object accessor functions provided by your application. Your application installs entries for its object accessor functions in an object accessor dispatch table, which is used by the Apple Event Manager to map requests for Apple event objects or their properties to the appropriate object accessor functions.

After being called by `AEResolve`, an object accessor function should return a token that identifies (in whatever manner is appropriate for your application) the specified Apple event object or property. An object accessor function also returns a result code that indicates whether it found the Apple event object or property. The token, which is a descriptor record of data type `AEDesc`, can be of any descriptor type, including descriptor types you define yourself. For an overview of the way `AEResolve` works with your application's object accessor functions to locate Apple event objects, see "Resolving Object Specifier Records," which begins on page 6-4.

Each object accessor function provided by your application should either find elements of a specified object class contained in an Apple event object identified by a token of a specified descriptor type, or find properties of an Apple event object identified by a token of a specified descriptor type. To determine which object accessor function to dispatch, the Apple Event Manager uses the object class ID specified in an object specifier record and the descriptor type of the token that identifies the requested object's container. For object accessor functions that find properties, you should specify the object class ID as the constant `cProperty`.

To install entries in your application's object accessor dispatch table, use the `AEInstallObjectAccessor` function. For each object class and property your application supports, you should install entries that specify

n   the object class of the requested Apple event object or property

n   the descriptor type of the token used to identify the container for the requested Apple event object or property

n   the address of the object accessor function that finds objects or properties of the specified object class in containers described by tokens of the specified descriptor type

n   a reference constant

You provide this information in the first four parameters to the `AEInstallObjectAccessor` function. The fifth parameter allows you to indicate whether the entry should be added to your application's object accessor dispatch table or the system object accessor dispatch table.

The **system object accessor dispatch table** is a table in the system heap that contains object accessor functions available to all processes running on the same computer. The object accessor functions in your application's object accessor dispatch table are available only to your application. If `AEResolve` cannot find an object accessor function for the Apple event object class in your application's object accessor dispatch table, it looks in the system object accessor dispatch table. If it doesn't find an object accessor function there either, it returns the result code `errAEAccessorNotFound`.

If `AEResolve` successfully calls the appropriate object accessor function in either the application object accessor dispatch table or the system object accessor dispatch table, the object accessor function returns a token and result code. The `AEResolve` function uses the token and result code to continue resolving the object specifier record. If, however, the token identifies the final Apple event object or property in the container hierarchy, `AEResolve` returns the token for the final resolution in the `theToken` parameter.

If the `AEResolve` function calls an object accessor function in the system object accessor dispatch table, your Apple event handler may not recognize the descriptor type of the token returned by the function. If this happens, your handler should attempt to coerce the token to an appropriate descriptor type. If coercion fails, return the result code `errAEUnknownObjectType`. When your handler returns this result code, the Apple Event Manager attempts to locate a system Apple event handler that can recognize the token.

It is up to you to decide how many object accessor functions you need to write and install for your application. You can install one object accessor function that locates Apple event objects of several different object classes, or you can write separate object accessor functions for certain object classes. Similarly, you may want to use only one descriptor type for all the tokens returned by your object accessor functions, or you may want to use several descriptor types. The sections that follow provide examples of alternative approaches.

For more information about object accessor functions, see "Writing Object Accessor Functions," which begins on page 6-28.

Installing Object Accessor Functions That Find Apple Event Objects

Listing 6-1 demonstrates how to add entries to your application's object accessor dispatch table for the object class `cText` and three of its element classes: the object classes `cWord`, `cItem`, and `cChar`. In this example, the container for each of these object classes is identified by a token that consists of a descriptor record of descriptor type `typeMyText`.

Listing 6-1   Installing object accessor functions that find elements of different classes for container tokens of the same type

```
myErr := AEInstallObjectAccessor(cText, typeMyText,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cWord, typeMyText,
                                 @MyFindWordObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cItem, typeMyText,
                                 @MyFindItemObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cChar, typeMyText,
                                 @MyFindCharObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

The first call to `AEInstallObjectAccessor` in Listing 6-1 adds an entry to the application's object accessor dispatch table. This entry indicates that the `AEResolve` function should call the `MyFindTextObjectAccessor` function when resolving any Apple event object with the `cText` object class and a container identified by a token of descriptor type `typeMyText`. The other calls to `AEInstallObjectAccessor` in Listing 6-1 add entries for Apple event objects of object classes `cWord`, `cItem`, and `cChar` in a container identified by a token of descriptor type `typeMyText`. For example, because all the entries created by the code in Listing 6-1 specify the descriptor type `typeMyText` for the token that identifies the container, the `AEResolve` function calls the `MyFindWordObjectAccessor` function to locate a requested word regardless of whether the container for the word is a run of text, another word, a paragraph, or an item.

The fourth parameter for the `AEInstallObjectAccessor` function specifies a reference constant passed to your handler by the Apple Event Manager each time `AEResolve` calls your object accessor function. Your application can use this reference constant for any purpose. If your application doesn't use the reference constant, you can use 0 as the value, as shown in Listing 6-1.

The last parameter for the `AEInstallObjectAccessor` function is a Boolean value that determines whether the entry is added to the system object accessor dispatch table (`TRUE`) or to your application's object accessor dispatch table (`FALSE`).

If you add an object accessor function to the system object accessor dispatch table, the function that you specify must reside in the system heap. If there was already an entry in the system object accessor dispatch table for the same object class and container descriptor type, that entry is replaced unless you chain it to your system handler. You can do this the same way you chain a previously installed system Apple event handler to your own system handler. See the description of `AEInstallEventHandler` on page 4-62 for details.

s **WARNING**

Before an application calls a system object accessor function, system software has set up the A5 register for the calling application. For this reason, if you provide a system object accessor function, it should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system object accessor function may crash. s

The code shown in Listing 6-1 installs a separate object accessor function for each object class, even though the code specifies the same descriptor type for tokens that identify the containers for Apple event objects of each class. Most word-processing applications can specify the same object accessor function as well as the same token descriptor type for Apple event objects of these four classes, in which case the code shown in Listing 6-1 can be altered as shown in Listing 6-2.

Listing 6-2   Installing one object accessor function that finds elements of different classes for container tokens of one type

```
myErr := AEInstallObjectAccessor(cText, typeMyText,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cWord, typeMyText,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cItem, typeMyText,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cChar, typeMyText,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

In some situations you may want to write different object accessor functions to locate Apple event objects of the same object class in containers identified by tokens of different descriptor types. For example, the code in Listing 6-3 installs two different object accessor functions: one that finds a word in a container identified by a token of type `typeMyTextToken`, and one that finds a word in a container identified by a token of `typeMyGraphicTextToken`.

Listing 6-3   Installing object accessor functions that find elements of the same class for container tokens of different types

```
myErr := AEInstallObjectAccessor(cWord, typeMyTextToken,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cWord, typeMyGraphicTextToken,
                                 @MyFindGrphcTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

Every application must provide one or more object accessor functions that can find
Apple event objects in the default container, which is always identified by a token of
descriptor type `typeNull`. Listing 6-4 demonstrates how to add entries to your
application's object accessor dispatch table for the object classes `cWindow` and
`cDocument`. The container for each of these classes is identified by a token of descriptor
type `typeNull`, which specifies an application's default container.

Listing 6-4   Installing object accessor functions that locate elements of different classes in the
default container

```
myErr := AEInstallObjectAccessor(cWindow, typeNull,
                                    @MyFindWindowObjectAccessor,
                                    0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cDocument, typeNull,
                                    @MyFindDocumentObjectAccessor,
                                    0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

For any entry in your object accessor dispatch table, you can specify a wildcard value for
the object class, for the descriptor type of the token used to identify the container, or for
both. You specify a wildcard by supplying the `typeWildCard` constant when installing
an entry into the object accessor dispatch table. A wildcard value matches all possible
values.

If an object accessor dispatch table contains one entry for a specific object class and a
specific token descriptor type, and another entry that is identical except that it specifies a
wildcard value for either the object class or the token descriptor type, the Apple Event
Manager dispatches the more specific entry. For example, if an object accessor dispatch
table includes one entry that specifies the object class as `cWord` and the token descriptor
type as `typeMyTextToken`, and another entry that specifies the object class as `cWord`
and the token descriptor type as `typeWildCard`, the Apple Event Manager dispatches
the object accessor function associated with the entry that specifies `typeMyTextToken`.

If you specify `typeWildCard` as the first parameter and `typeMyToken` as the second
parameter for the `AEInstallObjectAccessor` function and no other entry in the
dispatch table matches more exactly, the Apple Event Manager calls the object accessor
function that you specify in the third parameter when resolving Apple event objects of
any object class in containers identified by tokens of the `typeMyToken` descriptor type.

If you specify `cText` as the first parameter and `typeWildCard` as the second parameter for the `AEInstallObjectAccessor` function and no other entry in the dispatch table matches more exactly, the Apple Event Manager calls the object accessor function that you specify in the third parameter when resolving Apple event objects of the object class `cText` in containers identified by tokens of any descriptor type.

If you specify `typeWildCard` for both the first and second parameters of the `AEInstallObjectAccessor` function and no other entry in the dispatch table matches more exactly, the Apple Event Manager calls the object accessor function that you specify in the third parameter when resolving Apple event objects of any object class in containers identified by tokens of any descriptor type.

Once the Apple Event Manager finds a matching entry, whether exact or involving type `typeWildCard`, that is the only object accessor function it calls for that object class and token descriptor type. If that function fails, the Apple Event Manager won't look for another matching entry in the same table.

Installing Object Accessor Functions That Find Properties

The Apple event object to which a property belongs is that property's container. You should add entries to your application's object accessor dispatch table that specify object accessor functions for finding properties in containers identified by tokens of various descriptor types. Object specifier records do not specify a property's specific object class; instead, they specify the constant `cProperty` as the class ID for any property. Similarly, you should specify the constant `cProperty` as the object class for an object accessor function that can find any property of a container identified by a token of a given descriptor type. If you need to install different object accessor routines for finding properties of Apple event objects that belong to different object classes, you must use different descriptor types for the tokens that represent those Apple event objects.

For example, to specify an object accessor function that locates properties of Apple event objects identified by tokens of descriptor type `typeMyToken`, you can add a single entry to the object accessor dispatch table:

```
myErr := AEInstallObjectAccessor(cProperty, typeMyToken,
                                 @MyFindPropertyObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

The code in this example adds an object accessor function to the application's object accessor dispatch table that can find any property of any container identified by a token of descriptor type `typeMyToken`. If the second parameter were specified as `typeWildCard`, the `MyFindPropertyObjectAccessor` function would have to be capable of finding any property of any Apple event object in your application except for those found by handlers with more specific entries in the object accessor dispatch table.

Writing Object Accessor Functions

If the direct parameter for an Apple event consists of an object specifier record, your handler for the event should call the AEResolve function to resolve the object specifier record: that is, to find the Apple event objects or properties it describe. The AEResolve function resolves object specifier records with the help of object accessor functions provided by your application. For an overview of the way AEResolve works with your application's object accessor functions to locate Apple event objects, see "Resolving Object Specifier Records," which begins on page 6-4.

This section describes how to write object accessor functions. You need to read this section if your application supports the Core suite or any of the functional-area suites in the *Apple Event Registry: Standard Suites.*

Your application should provide object accessor functions that can find Apple event objects and their properties for all object classes supported by your application, including their corresponding properties and element classes. Because the Apple Event Manager dispatches object accessor functions according to the class ID of the requested Apple event object and the descriptor type of the token that identifies its container, you have a great deal of flexibility in deciding what object accessor functions you need to write for your application. The installation and dispatching of object accessor functions are described in "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

For example, if your application is a word processor, one object accessor function will probably work equally well for Apple event objects of object classes cParagraph, cItem, and cWord located in containers identified by tokens of descriptor type myTextToken. If you use a single descriptor type for tokens that identify any containers in which objects of these three object classes can be found, you can dispatch requests for all such elements to the same object accessor function. However, the same word processor might use one descriptor type for tokens identifying containers of class cCell and another descriptor type for tokens identifying containers of class cColumn—in which case it would need an object accessor function for each descriptor type.

For each object class that your application supports, your application should also provide one object accessor function that can find all the properties of that object class, or one object accessor function that can find all the properties of several object classes.

Here's the declaration for a sample object accessor function:

```
FUNCTION MyObjectAccessor (desiredClass: DescType;
                           containerToken: AEDesc;
                           containerClass: DescType;
                           keyForm: DescType; keyData: AEDesc;
                           VAR theToken: AEDesc;
                           theRefCon: LongInt): OSErr;
```

The `AEResolve` function passes the following information to your object accessor
function: the object class ID of the requested Apple event objects, the object class of their
container, a token that identifies the specific container in which to look for them, the key
form and key data that specify how to locate them, and the reference constant associated
with the object accessor function. Your object accessor function uses this information to
locate the requested objects.

Most applications that resolve object specifier records need to support only the
key forms `formPropertyID`, `formName`, `formUniqueID`, `formAbsolutePosition`,
`formRelativePosition`, and `formRange` explicitly. You do not need to support these
key forms for all object classes; for example, words usually do not have names, so most
applications should return `errAEEventNotHandled` if they receive a request for a
word by name.

If your application provides an object-counting function and an object-comparison
function in addition to the appropriate object accessor functions, the Apple Event
Manager can handle `formTest` automatically.

The Apple Event Manager uses the key form `formWhose` internally to optimize
resolution of object specifier records that specify `formTest`. Only applications that
translate tests into their own query languages need to support `formWhose` explicitly.
"Handling Whose Tests," which begins on page 6-41, describes `formWhose` in detail.

If your object accessor function successfully locates the requested Apple event objects,
your application should return the `noErr` result code and a token that identifies them.
The token can be of any descriptor type, as long as it is a descriptor record. For example,
to identify a file, your application might use a descriptor record of descriptor type
`typeAlias` or `typeFSS`. To identify an open document, your application might define
its own descriptor type, such as `typeMyDocToken`, for a descriptor record whose data
handle refers to a pointer to a document record. For more information about tokens, see
"Defining Tokens" on page 6-39.

IMPORTANT

Object accessor functions must not have side effects that change the
number or order of elements in a container while an object specifier
record is being resolved. If the number of elements in a container is
changed during the resolution of an object specifier record, the Apple
Event Manager may not be able to locate all the elements. s

Writing Object Accessor Functions That Find Apple Event Objects

The first three listings in this section demonstrate how to write three object accessor
functions that might be called in the following situation: An application receives a
Get Data event with a direct parameter that consists of an object specifier record for the
first word in the third paragraph of a document. The application's handler for the
Get Data event calls the `AEResolve` function to resolve the object specifier record.
The `AEResolve` function first calls the application's object accessor function for objects
of class `cDocument` in containers identified by a token of descriptor type `typeNull`.

The AEResolve function passes these values to the
MyFindDocumentObjectAccessor function shown in Listing 6-5:
in the desiredClass parameter, the constant cDocument; in the containerToken
parameter, a descriptor record of descriptor type typeNull with a data handle
whose value is NIL; in the containerClass parameter, the constant typeNull;
in the keyForm parameter, the constant formName; in the keyData parameter, a
descriptor record of descriptor type typeText whose data consists of the string
"MyDoc"; and the reference constant specified in the application's object accessor
dispatch table.

Listing 6-5   An object accessor function that locates Apple event objects of object class
cDocument

```
FUNCTION MyFindDocumentObjectAccessor
                            (desiredClass: DescType;
                             containerToken: AEDesc;
                             containerClass: DescType;
                             keyForm: DescType; keyData: AEDesc;
                             VAR token: AEDesc;
                             theRefCon: LongInt): OSErr;
VAR
    docName:        Str255;
    actSize:        Size;
    foundDoc:       Boolean;
    foundDocRecPtr: MyDocumentRecordPtr;
BEGIN
    IF keyform = formName THEN
    BEGIN
        {get the name of the document from the key data}
        MyGetStringFromDesc(keyData, docName, actSize);
        {look for a document with the given name by }
        { searching all document records}
        MySearchDocRecs(docName, foundDocRecPtr, foundDoc);
        IF NOT foundDoc THEN
            MyFindDocumentObjectAccessor := kObjectNotFound
        ELSE {create token that identifies the document}
            MyFindDocumentObjectAccessor :=
                    AECreateDesc(typeMyDocToken, @foundDocRecPtr,
                                SizeOf(foundDocRecPtr), token);
    END
    {handle the other key forms you support}
    ELSE
        MyFindDocumentObjectAccessor := kKeyFormNotSupported;
END;
```

The `MyFindDocumentObjectAccessor` function uses the information in the `keyForm` and `keyData` parameters to find the specified document. If it finds the Apple event object, `MyFindDocumentObjectAccessor` returns a token of descriptor type `typeMyDocToken` to `AEResolve`. The data handle for this token refers to a pointer to a document record (see Figure 6-5 on page 6-39). The `MyFindDocumentObjectAccessor` function returns this token and the `noErr` result code to the `AEResolve` function.

In the Get Data example, the token returned to `AEResolve` by the `MyFindDocumentObjectAccessor` function identifies the document "MyDoc." The `AEResolve` function then calls the application's object accessor function for objects of class `cParagraph` in containers identified by a token of descriptor type `typeMyDocToken`.

In this case, `AEResolve` passes these values to the `MyFindParaObjectAccessor` function shown in Listing 6-6: in the `desiredClass` parameter, the constant `cParagraph`; in the `containerToken` parameter, the token returned by the `MyFindDocumentObjectAccessor` function; in the `containerClass` parameter, the constant `cDocument`; in the `keyForm` parameter, the constant `formAbsolutePosition`; in the `keyData` parameter, a descriptor record with the `typeLongInteger` descriptor type and data that consists of the value 3 (indicating the third paragraph); and the reference constant specified in the application's object accessor dispatch table.

Listing 6-6   An object accessor function that locates Apple event objects of object class
`cParagraph`

```
FUNCTION MyFindParaObjectAccessor (desiredClass: DescType;
                                   containerToken: AEDesc;
                                   containerClass: DescType;
                                   keyForm: DescType;
                                   keyData: AEDesc;
                                   VAR token: AEDesc;
                                   theRefCon: LongInt): OSErr;
VAR
   index:           LongInt;
   {MyFoundTextRecord is an application-defined data type }
   { consisting of three fields: start, ending, and docPtr}
   foundParaRec:    MyFoundTextRecord;
   foundParaStart:  LongInt;
   foundParaEnd:    LongInt;
   foundDocRecPtr:  MyDocumentRecordPtr;
   success:         Boolean;
BEGIN
   IF keyForm = formAbsolutePosition THEN
   BEGIN
      {get the index of the paragraph from the key data}
      MyGetIndexFromDesc(keyData, index);
      {get the desired paragraph by index}
      success := MyGetPara(index, containerToken, foundParaStart,
                           foundParaEnd, foundDocRecPtr);
      IF NOT success THEN
         MyFindParaObjectAccessor := kObjectNotFound
      ELSE {create token that identifies the paragraph}
      BEGIN
         foundParaRec.start := foundParaStart;
         foundParaRec.ending := foundParaEnd;
         foundParaRec.docPtr := foundDocRecPtr;
         MyFindParaObjectAccessor :=
               AECreateDesc(typeMyTextToken, @foundParaRec,
                            SizeOf(foundParaRec), token);
      END;
   END
      {handle the other key forms you support}
   ELSE
      MyFindParaObjectAccessor := kKeyFormNotSupported;
END;
```

The `MyFindParaObjectAccessor` function uses another application-defined function, `MyGetPara`, to search the data structures associated with the document and find the desired paragraph. If it finds the paragraph, `MyGetPara` returns a value that identifies the beginning of the paragraph, a value that identifies the end of the paragraph, and a pointer to the document (which `MyGetPara` gets from the `containerToken` parameter). The `MyFindParaObjectAccessor` function returns an application-defined token that contains this information. This token is of descriptor type `typeMyTextToken`; it describes a range of characters that can be used to identify any range of text, including a paragraph or a word. The `MyFindParaObjectAccessor` function returns this token and the `noErr` result code to the `AEResolve` function.

In the Get Data example, the token returned to `AEResolve` by the `MyFindParaObjectAccessor` function identifies the third paragraph in the document "MyDoc." The `AEResolve` function then calls the application's object accessor function for objects of class `cWord` in containers identified by a token of descriptor type `typeMyTextToken`.

In this case, the `AEResolve` function passes these values to the `MyFindWordObjectAccessor` function shown in Listing 6-7: in the `desiredClass` parameter, the constant `cWord`; in the `containerToken` parameter, the token returned by the `MyFindParaObjectAccessor` function (a token of descriptor type `typeMyTextToken` that identifies a paragraph); in the `containerClass` parameter, the constant `cParagraph`; in the `keyForm` parameter, the constant `formAbsolutePosition`; in the `keyData` parameter, a descriptor record with the `typeLongInteger` descriptor type and data that consists of the value 1 (indicating the first word); and the reference constant specified in the application's object accessor dispatch table.

The `MyFindWordObjectAccessor` function uses another application-defined function, `MyGetWord`, to search the paragraph to find the desired word. If it finds the word, `MyGetWord` returns a value that identifies the beginning of the word, a value that identifies the end of the word, and a pointer to the document (which `MyGetWord` gets from the `containerToken` parameter). The `MyFindWordObjectAccessor` function returns a token that contains this information. This token is also of descriptor type `typeMyTextToken`; in this case, the token identifies a specific word. The `MyFindWordObjectAccessor` function returns this token and the `noErr` result code to the `AEResolve` function, which in turn returns the token to the Get Data event handler that originally called `AEResolve`.

Listing 6-7   An object accessor function that locates Apple event objects of object class
                    cWord

```
FUNCTION MyFindWordObjectAccessor
                          (desiredClass: DescType;
                           containerToken: AEDesc;
                           containerClass: DescType;
                           keyForm: DescType; keyData: AEDesc;
                           VAR token: AEDesc;
                           theRefCon: LongInt): OSErr;
VAR
    index:          LongInt;
    foundWordRec:   MyFoundTextRecord;
    foundWordStart: LongInt;
    foundWordEnd:   LongInt;
    foundDocRecPtr: MyDocumentRecPtr;
    success:        Boolean;
BEGIN
    IF keyForm = formAbsolutePosition THEN
    BEGIN
        {get the index of the word from the key data}
        MyGetIndexFromDesc(keyData, index);
        {get the desired word by index}
        success := MyGetWord(index, containerToken, foundWordStart,
                             foundWordEnd, foundDocRecPtr);
        IF NOT success THEN
            MyFindWordObjectAccessor := kObjectNotFound
        ELSE {create token that identifies the paragraph}
        BEGIN
            foundWordRec.start := foundWordStart;
            foundWordRec.ending := foundWordEnd;
            foundWordRec.docPtr := foundDocRecPtr;
            MyFindWordObjectAccessor :=
                    AECreateDesc(typeMyTextToken, @foundWordRec,
                                 SizeOf(foundWordRec), token);
        END;
    END
    {handle the other key forms you support}
    ELSE
        MyFindWordObjectAccessor := kKeyFormNotSupported;
END;
```

Listing 6-5 on page 6-30 shows an object accessor function that locates a document in the default container. Every application must provide one or more object accessor functions that can find Apple event objects in the default container, which is always identified by a descriptor record of descriptor type typeNull. Listing 6-8 provides another example of an object accessor function that locates an Apple event object in the default container. If the MyFindWindowObjectAccessor function shown in Listing 6-8 were installed in an application's object accessor dispatch table, the AEResolve function would call it as necessary to locate an object of class cWindow in a container identified by a token of descriptor type typeNull.

Listing 6-8   An object accessor function that locates Apple event objects of object class cWindow

```
FUNCTION MyFindWindowObjectAccessor (desiredClass: DescType;
                                     containerToken: AEDesc;
                                     containerClass: DescType;
                                     keyForm: DescType;
                                     keyData: AEDesc;
                                     VAR token: AEDesc;
                                     theRefCon: LongInt): OSErr;
VAR
   windowName:     Str255;
   actSize:        Size;
   windTitle:      Str255;
   window:         WindowPtr;
   index, iLoop:   Integer;
   found:          Boolean;
BEGIN
   IF keyForm = formName THEN
   BEGIN
      {get the name of the window to find from the keyData }
      { parameter. MyGetStringFromDesc gets data out of an }
      { AEDesc and returns a string and the string's size}
      MyGetStringFromDesc(keyData, windowName, actSize);
      {look for a window with the given name}
      window := FrontWindow;
      found := FALSE;
      WHILE ((window <> NIL) AND (found = FALSE)) DO
      BEGIN
         GetWTitle(window, windTitle);
         found := EqualString(windTitle, windowName, FALSE, TRUE);
         IF NOT found THEN
            window := WindowPtr(WindowPeek(window)^.nextWindow);
      END;      {of while}
```

```
      END {of formName}
      ELSE
      IF keyForm = formAbsolutePosition THEN
         {find the window given an index in key data}
         BEGIN    {get the index from the key data}
            MyGetIndexFromDesc(keyData, index);
            found := FALSE;
            iLoop := 0;
            window := FrontWindow;
            WHILE (window <> NIL) AND (found <> TRUE) DO
            BEGIN
               iLoop := iLoop +1;
               IF iLoop = index THEN
                  found := TRUE
               ELSE
               window := WindowPtr(WindowPeek(window)^.nextWindow);
            END; {of while}
         END {of formAbsolutePosition}
      {handle the other key forms you support}
      ELSE
      BEGIN
         MyFindWindowObjectAccessor := kKeyFormNotSupported;
         Exit(MyFindWindowObjectAccessor);
      END;
      IF window = NIL THEN
         MyFindWindowObjectAccessor := kObjectNotFound
      ELSE {create token that identifies the window}
         MyFindWindowObjectAccessor :=
                  AECreateDesc(typeMyWindow, @window,
                                   SizeOf(window), token);
   END;
```

The `keyForm` parameter of the `MyFindWindowObjectAccessor` function describes how the function should interpret the `keyData` parameter. If the key form is `formName`, then the key data contains the name of the window to locate. If the key form is `formAbsolutePosition`, the key data contains the position of the window to locate in the window list; for example, a value of 1 identifies the frontmost window.

The `MyFindWindowObjectAccessor` function supports only the `formName` and `formAbsolutePosition` key forms. Your object accessor functions should support all key forms that make sense for the kinds of objects the functions can locate.

For the `formName` keyword, the `MyFindWindowObjectAccessor` function starts with the frontmost window and compares the window's title to the name specified by the `keyData` parameter. It continues this search until it reaches either the end of the window list or finds a match. If the `MyFindWindowObjectAccessor` function finds a match, it uses the `AECreateDesc` function to create a descriptor record for the token, specifying the application-defined `typeMyWindow` descriptor type and the data for this descriptor type as a window pointer.

The `MyFindWindowObjectAccessor` function then sets its function result appropriately, and the `AEResolve` function either returns this function result and token, or uses the returned token to request the next Apple event object in the container hierarchy, such as a document in the window.

Writing Object Accessor Functions That Find Properties

The Apple event object to which a property belongs is that property's container. Your application should provide an object accessor function for finding properties in containers identified by tokens of various descriptor types. Your application does not need to be given a property's specific object class in order to find that property; instead, you can specify the object class ID for any property with the constant `cProperty`. Thus, you can write a single object accessor function that can find any property of an object identified by a token of a given descriptor type.

To install such an object accessor function, you can add a single entry to the object accessor dispatch table that specifies the desired object class as `cProperty` for a given token descriptor type. For example, Listing 6-9 shows an object accessor function that identifies any property of a window.

Listing 6-9   An object accessor function that identifies any property of a window

```
FUNCTION MyFindPropertyOfWindowObjectAccessor
                            (desiredClass: DescType;
                            containerToken: AEDesc;
                            containerClass: DescType;
                            keyForm: DescType; keyData: AEDesc;
                            VAR token: AEDesc;
                            theRefCon: LongInt): OSErr;
VAR
   theProperty: DescType;
BEGIN
   MyFindPropertyOfWindowObjectAccessor := noErr;
   MyGetPropFromKeyData(keyData, theProperty);
   IF keyForm = formPropertyID THEN
   BEGIN
      IF theProperty = pName THEN
         {create token that identifies name property of the }
         { window}
         MyCreateToken(typeMyWindowProp, containerToken, pName,
                       token)
      ELSE
      IF theProperty = pBounds THEN
         {create token that identifies bounds property of the }
         { window}
         MyCreateToken(typeMyWindowProp, containerToken, pBounds,
                       token)

      {create tokens for other properties as appropriate}

      ELSE
         MyFindPropertyOfWindowObjectAccessor :=
                                       kErrorPropNotFound;
   END
   ELSE
      MyFindPropertyOfWindowObjectAccessor :=
                                       kKeyFormNotSupported;
END;
```

The MyFindPropertyOfWindowObjectAccessor function takes a token that identifies a window and creates a token that identifies the requested property of that window. See Figure 6-6 on page 6-40 for an illustration of the logical organization of a token of descriptor type typeMyWindowProp.

This simplified example merely translates information about the requested property and the window to which it belongs into the form of a token of type typeMyWindowProp. This token can then be used by Apple event handlers to identify the corresponding window and its property, so that a handler can either retrieve the value of the property (for example, a Get Data handler) or change the value of the property (for example, a Set Data handler). Like other tokens, a token that identifies a property should always contain a reference to the corresponding property and the object to which it belongs—not a copy of the data for that object's property.

Defining Tokens

It is up to you to decide how many token descriptor types you need to define for your application. In many cases you may be able to define one token that can identify Apple event objects of several different object classes, such as a token of type typeMyTextToken that identifies Apple event objects of object classes cText, cWord, cItem, and cChar. In other cases you may need to define specific token descriptor types for specific object classes.

For example, the MyFindDocumentObjectAccessor routine shown in Listing 6-5 on page 6-30 returns a token of descriptor type typeMyDocToken, which identifies a document record.

```
CONST                                {application-defined token}
   typeMyDocToken       = 'docr'; {identifies a document record}
```

Figure 6-5 shows the logical arrangement of a descriptor record of descriptor type typeMyDocToken whose data is specified by a pointer to a document record.

Figure 6-5   Descriptor record for an application-defined token that identifies a document
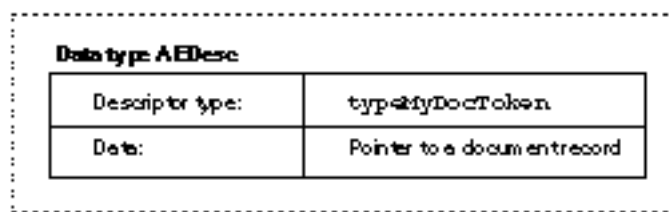
The `MyFindPropertyOfWindowObjectAccessor` routine shown in Listing 6-9 returns a token of descriptor type `typeMyWindowProp` for every property that it can locate.

```
CONST                                  {application-defined token}
   typeMyWindowProp     = 'wprp'; {a window pointer and a }
                                  { property ID}
```

Figure 6-6 shows the logical arrangement of a descriptor record of descriptor type `typeMyWindowProp` that identifies the bounds property of a window. Its data consists of a window pointer and the constant `pBounds`. The application can use this token either to return or to change the window's bounds setting, depending on the Apple event that specified the property. If the token specified `pName` instead, the application could use it either to return the window's name as a string or to change the window's name.

Figure 6-6   Descriptor record for an application-defined token that identifies the `pbounds` property of a window

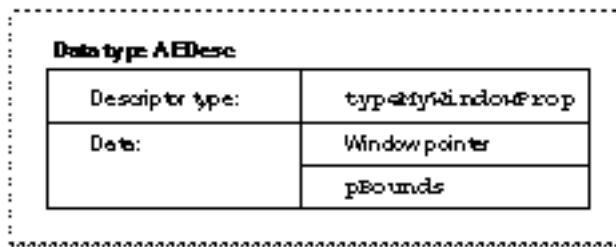A token's data should always contain a reference to the corresponding Apple event objects—not a copy of the data for those objects. This allows the same token to be used for both reading and writing tokens.

It's often possible to use the same token type for objects of several object classes, or for both an object of a given class and one of its properties. A token's data is private to your application and can be organized in any way that is convenient.

When an object accessor function that supports key form `formRange` locates a range of Apple event objects, it should normally return a descriptor list (`AEDescList`) of tokens for the individual objects. A typical exception is an object accessor function that returns a range of objects of class `cText`, which should return a single token representing the entire range. For example, an object accessor function that finds "all the characters from char 1 to char 1024" should return a token that consists of a list of 1024 objects, each of class `cChar`, whereas an object specifier function that finds "all the text from char 1 to char 1024" should return a single token for a single item of class `cText` that is 1024 characters long.

A token is valid only until the Apple Event Manager has located the requested element in the container the token represents and returned another token for the element. The Apple Event Manager disposes of intermediate tokens after it finishes resolving an object specifier record, but one token is always left over—the token that identifies the specified Apple event object or objects. Your application should dispose of this final token by calling the `AEDisposeToken` function, which in turn calls your application's token disposal function (if one exists), an optional object callback function that disposes of a token. See page 6-99 for the declaration of a token disposal function.

If your application does not provide a token disposal function, the Apple Event Manager uses the `AEDisposeDesc` function to dispose of tokens. This function does the job as long as disposing of tokens involves nothing more than simply disposing of a descriptor record. Otherwise, you need to provide a custom token disposal function. For example, suppose the data field of a token descriptor record contains a handle to a block that in turn contains references to memory for the Apple event object referred to by the token. In this case, the application should provide a token disposal function that performs the tasks required to dispose of the token and any associated structures.

Handling Whose Tests

If your application provides an object-counting function and an object-comparison function in addition to the appropriate object accessor functions, the Apple Event Manager can resolve object specifier records that specify `formTest` without any other assistance from your application. The Apple Event Manager translates object specifier records of key form `formTest` into object specifier records of key form `formWhose`. This involves collapsing the key form and key data from two object specifier records in a container hierarchy into one object specifier record with the key form `formWhose`.

Some applications may find it more efficient to translate whose tests into their own query languages rather than letting the Apple Event Manager handle the tests. This is useful only for applications that can make use of a test combined with either an absolute position or a range to locate objects. If you want the Apple Event Manager to let your application handle whose tests, set the `kAEIDoWhose` flag in the `callbackFlags` parameter of the `AEResolve` function. If for any reason one of your application's object accessor functions chooses not to handle a particular whose descriptor record, it should return `errAEEventNotHandled` as the result code, and the Apple Event Manager will try again using the original object specifier records, just as if the `kAEIDoWhose` flag were not set.

The key data for `formWhose` is specified by a **whose descriptor record,** which is a coerced AE record of descriptor type `typeWhoseDescriptor`. The data for a whose descriptor record consists of the two keyword-specified descriptor records shown in Table 6-5.

Table 6-5   Keyword-specified descriptor records for `typeWhoseDescriptor`

Keyword	Descriptor type	Data
keyAEIndex	typeLongInteger	Offset of requested element in group of elements that pass a test
	typeAbsoluteOrdinal	kAEFirst kAEMiddle kAELast kAEAny kAEAll
	typeWhoseRange	Whose range descriptor record
keyAETest	typeCompDescriptor	Comparison descriptor record
	typeLogicalDescriptor	Logical descriptor record

A whose descriptor record is never created directly by an application. The Apple Event Manager creates a whose descriptor record whenever an object specifier record of key form `formTest` is used to describe the container for elements described by an object specifier record of key form `formAbsolutePosition` or `formRange`, with some exceptions as noted in this section.

For example, Figure 6-3 on page 6-18 shows four object specifier records that show the container hierarchy for the first row that meets a test in the table "MyAddresses" of the database "SurfDB." The top two object specifier records in that figure use the key forms `formAbsolutePosition` and `formTest` to describe elements in a container. When it receives these two object specifier records, the Apple Event Manager collapses them into one, as shown in Figure 6-7. It then calls the application's object-counting function to find out how many objects of class `cRow` the table contains and the object-comparison function to test the rows in the table until it finds the first row that passes the test.

Figure 6-7   A container hierarchy created by the Apple Event Manager using a whose descriptor record
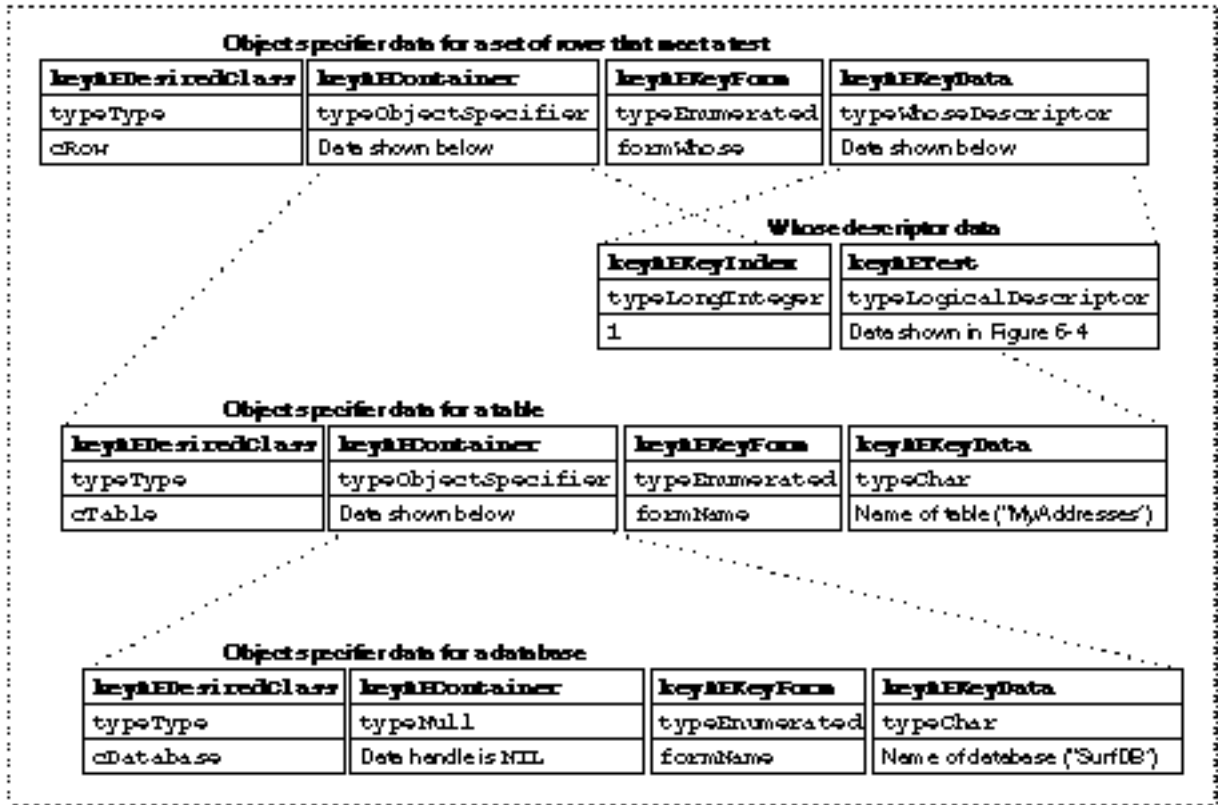
If the elements to be tested are described by an object specifier record of key form `formAbsolutePosition` or `formRange` but are not of the same object class as their container, the Apple Event Manager cannot collapse the existing object specifier records into a whose descriptor record. Instead, the Apple Event Manager creates a whose descriptor record as if a third object specifier record of key form `formAbsolutePosition` and `kAEAll` were inserted between the object specifier record for the container and that for the tested elements. For example, the Apple Event Manager would interpret a request for "character 1 of word whose first letter = 'a'" as "character 1 of every word whose first letter = 'a'".

When an object specifier record of key form `formTest` is used to describe the container for elements described by an object specifier record of key form `formRange`, the Apple Event Manager will, under certain conditions, coerce the corresponding range descriptor record to a **whose range descriptor record,** which is a coerced AE record of `typeWhoseRange`. **The data for a whose range descriptor record consists of two keyword-specified descriptor records with the descriptor types and data shown in Table 6-6.**

Table 6-6   Keyword-specified descriptor records for `typeWhoseRange`

Keyword	Descriptor type	Data
`keyAEWhoseRangeStart`	`typeLongInteger`	Offset of beginning of range
	`typeAbsoluteOrdinal`	`kAEFirst` `kAEMiddle` `kAELast` `kAEAny` `kAEAll`
`keyAEWhoseRangeStop`	`typeLongInteger`	Offset of end of range
	`typeAbsoluteOrdinal`	`kAEFirst` `kAEMiddle` `kAELast` `kAEAny` `kAEAll`

A whose range descriptor record describes the absolute position of the boundary elements, within the set of all elements that pass a test, that identify the beginning and end of the desired range.

The Apple Event Manager coerces a range descriptor record to a whose range descriptor record if the specified container and its elements are of the same class, if the container for the specified range of elements is a group of Apple event objects that pass a test, and if the boundary objects in the original range descriptor record meet these conditions:

n   Both boundary objects are of the same object class as the Apple event objects in the range they specify.

n   The object specifier record for each boundary object specifies its container with a descriptor record of descriptor type `typeCurrentContainer`.

n   The object specifier record for each boundary object specifies a key form of `formAbsolutePosition`.

If these conditions are not met, the Apple Event Manager doesn't create a whose range descriptor record. Instead, as described earlier in this section, the Apple Event Manager creates a whose descriptor record as if the original request specified every element that passed the test.

If your application sets the `kAEIDoWhose` flag in the `callbackFlags` parameter of `AEResolve`, you should provide object accessor functions that can handle `formWhose`. These functions should coerce the whose descriptor record specified as key data for an object specifier record to an AE record and extract the data from the AE record by calling the `AEGetKeyPtr` and `AEGetKeyDesc` functions. If the keyword-specified descriptor record with the keyword `keyAEIndex` specifies descriptor type `typeWhoseRange`, your object accessor function must also coerce that descriptor record to an AE record and extract the data. Your object accessor function should then attempt to locate the requested objects and, if successful, return a token that identifies them.

If your application sets the `kAEIDoWhose` flag and attempts to resolve every whose descriptor record it receives, the Apple Event Manager does not attempt to resolve object specifier records of any key form. The object-counting and object-comparison functions are never called, and your application is solely responsible for determining the formats and types of all tokens.

Writing Object Callback Functions

If an Apple event parameter consists of an object specifier record, your handler for the Apple event typically calls `AEResolve` to begin the process of locating the requested Apple event object or objects. In turn, `AEResolve` calls object accessor functions and, if necessary, object callback functions provided by your application.

Every application that supports Apple event objects should provide object accessor functions that can locate Apple event objects belonging to any of the supported object classes. For an overview of the way `AEResolve` calls object accessor functions to locate Apple event objects described by object specifier records, see "Resolving Object Specifier Records," which begins on page 6-4.

In addition to object accessor functions, your application can provide up to seven object callback functions:

n   An *object-counting function* counts the number of elements of a specified class in a specified container, so that the Apple Event Manager can determine how many elements it must examine to find the element or elements that pass a test. Your application must provide one object-counting function to handle object specifier records that specify tests. (See "Writing an Object-Counting Function," which begins on page 6-48.)

n   An *object-comparison function* compares one element either to another element or to a descriptor record and returns either `TRUE` or `FALSE`. Your application must provide one object-comparison function to handle object specifier records that specify tests. (See "Writing an Object-Comparison Function" on page 6-50.)

n   A *token disposal function* disposes of a token after your application calls the `AEDisposeToken` function. If your application doesn't provide a token disposal function, the Apple Event Manager uses the `AEDisposeDesc` function instead. Your application must provide a token disposal function if it requires more than a call to `AEDisposeDesc` to dispose of one of its tokens. This is true, for example, if your application supports marking by modifying its own data structures. (See page 6-99 for the declaration of a token disposal function.)

n   An *error callback function* gives the Apple Event Manager an address to which to write the descriptor record it is currently working with if an error occurs while `AEResolve` is attempting to resolve an object specifier record. Your application is not required to provide an error callback function. (See page 6-100 for the declaration of an error callback function.)

n   Three *marking callback functions* are used by the Apple Event Manager to get a mark token from your application, to mark specific Apple event objects, and to pare down a group of marked Apple event objects. Your application must provide all three marking functions if it supports marking. (See "Writing Marking Callback Functions" on page 6-53.)

To make your object callback functions available to the Apple Event Manager, use the `AESetObjectCallbacks` function:

```
myErr := AESetObjectCallbacks (@MyCompareObjects,
                              @MyCountObjects, @MyDisposeToken,
                              @MyGetMarkToken, @MyMark,
                              @MyAdjustMarks, @MyGetErrDesc);
```

Each parameter to the `AESetObjectCallbacks` function consists of either a pointer to the corresponding application-defined function or `NIL` if no function is provided. The `AESetObjectCallbacks` function sets object callback functions that are available only to your application. To set system object callback functions, which are available to all applications and processes running on the same computer, use the `AEInstallSpecialHandler` function as described on page 4-100.

To handle object specifier records that specify tests, your application must provide an object-counting function and an object-comparison function. The Apple Event Manager calls your application's object-counting function to determine the number of Apple event objects in a specified container that need to be tested. The Apple Event Manager calls your application's object-comparison function when it needs to compare one Apple event object to either another Apple event object or to a value in a descriptor record.

If your application does not provide a token disposal function, the Apple Event Manager uses the AEDisposeDesc function to dispose of tokens. This function does the job as long as disposing of tokens involves nothing more than simply disposing of a descriptor record. Otherwise, you need to provide custom token disposal function. For example, suppose the data field of a token descriptor record contains a handle to a block that in turn contains references to storage for the Apple event object referred to by the token. In this case, the application can provide a token disposal function that performs the tasks required to dispose of the token and any associated structures.

Whenever more than one Apple event object passes a test, AEResolve can either return a list of tokens or make use of a target application's ability to mark its own objects. Sometimes a list of tokens can become unmanageably large. For example, if a Get Data event asks for the names and addresses of all customers with a specified zip code who have purchased a specified product, the object accessor function that locates all the customers with the specified zip code might return a list of many thousands of tokens; the elements identified by those tokens would then have to be tested for the specified product. However, if your application uses some method of marking objects, you can choose simply to mark the requested objects rather than returning a list of tokens. "Writing Marking Callback Functions" on page 6-53 describes how to do this. If your application supports marking by modifying its own data structures, you must provide a token disposal function.

When one of your application's Apple event handlers calls the AEResolve function, the handler should pass a value in the callbackFlags parameter that specifies whether your application supports whose descriptor records or provides marking callback functions. You can add the following constants, as appropriate, to provide a value for the callbackFlags parameter:

```
CONST kAEIDoMinimum  =  $0000; {does not handle whose tests or }
                                  { provide marking callbacks}
      kAEIDoWhose    =  $0001; {supports key form formWhose}
      kAEIDoMarking  =  $0004; {provides marking functions}
```

For example, this code instructs the Apple Event Manager to call any marking functions previously set with the AESetObjectCallbacks function while resolving the object specifier record in the objectSpecifier parameter:

```
VAR
   objectSpecifier:    AEDesc;
   resultToken:        AEDesc;
   myErr:              OSErr;


myErr := AEResolve(objectSpecifier, kAEIDoMarking, resultToken);
```

If any of the marking callback functions are not installed, `AEResolve` returns the error `errAEHandlerNotFound`.

IMPORTANT

If your application doesn't specify `kAEIDoWhose`, the Apple Event Manager attempts to resolve all object specifier records of key form `formTest`. To do so, the Apple Event Manager uses your application's object-counting and object-comparison functions, and returns a token of type `typeAEList`.

If your application does specify `kAEIDoWhose`, the Apple Event Manager does not attempt to resolve object specifier records of any key form. In this case, the object-counting and object-comparison functions are never called; your application determines the formats and types of all tokens; and your application must interpret whose descriptor records created by the Apple Event Manager during the resolution of object specifier records. For more information, see "Handling Whose Tests," which begins on page 6-41. s

Writing an Object-Counting Function

To handle object specifier records that specify tests, your application should provide an object-counting function (unless it specifies `kAEIDoWhose` as just described). Your object-counting function should be able to count the number of elements of a given object class in a given container. For example, if your application supports Apple event objects that belong to the object class `cText` in the Text suite, your application should provide an object-counting function that can count Apple event objects of each element class listed in the definition of `cText` in the *Apple Event Registry: Standard Suites.* In this case, your application should provide an object-counting function that can count the number of words, items, or characters in a text object.

You specify your object-counting function with the `AESetObjectCallbacks` function. Whenever it is resolving an object specifier record and it requires a count of the number of elements in a given container, the Apple Event Manager calls your object-counting function.

Here's the declaration for a sample object-counting function:

```
FUNCTION MyCountObjects (desiredClass: DescType;
                         containerClass: DescType;
                         containerToken: AEDesc;
                         VAR result: LongInt): OSErr;
```

The Apple Event Manager passes the following information to your object-counting function: the object class ID of the Apple event objects to count, the object class of their container, and a token identifying their container. (The container class can be useful if you want to use one token type for several object classes.) Your object-counting function uses this information to count the number of Apple event objects of the specified object

class in the specified container. After counting the Apple event objects, your application should return the `noErr` result code and, in the `result` parameter, the number of Apple event objects counted.

Listing 6-10 shows an application-defined function, `MyCountObjects`, that counts the number of objects for any object class supported by the application.

Listing 6-10   An object-counting function

```
FUNCTION MyCountObjects (desiredClass: DescType; containerClass: DescType;
                          containerToken: AEDesc; VAR result: LongInt): OSErr;
VAR
   window: WindowPtr;
BEGIN
   result := 0;
   IF desiredClass = cWindow THEN
   BEGIN
      IF containerClass = typeNull THEN
      BEGIN
         {count the number of windows}
         window := FrontWindow;
         WHILE window <> NIL DO
            BEGIN
               result := result + 1;
               window := WindowPtr(WindowPeek(window)^.nextWindow);
            END; {of while}
      END;
      MyCountObjects := noErr;
   END {of cWindow}
   ELSE
      IF desiredClass = cWord THEN
         {count the number of words in the container}
         MyCountObjects := MyCountWords(containerClass, containerToken,
                                          result)
   ELSE
      IF desiredClass = cParagraph THEN
         {count the number of paragraphs in the container}
         MyCountObjects := MyCountParas(containerClass, containerToken,
                                          result)
   ELSE
      {this app does not support any other object classes}
      MyCountObjects := kObjectClassNotFound;
END;
```

Writing an Object-Comparison Function

To handle object specifier records that specify tests, your application should provide an object-comparison function (unless it specifies kAEIDoWhose as described on page 6-48). Your object-comparison function should be able to compare one Apple event object to another Apple event object or to another descriptor record.

You specify your object-comparison function with the AESetObjectCallbacks function. Whenever it is resolving object specifier records and needs to compare the value of an Apple event object with another object or with data, the Apple Event Manager calls your object-comparison function.

Here's the declaration for a sample object-comparison function:

```
FUNCTION MyCompareObjects (comparisonOperator: DescType;
                           object: AEDesc;
                           objectOrDescToCompare: AEDesc;
                           VAR result: Boolean): OSErr;
```

The Apple Event Manager passes the following information to your object-comparison function: a comparison operator that specifies how the two objects should be compared, a token for the first Apple event object, and either a token that describes the Apple event object to compare or a descriptor record.

It is up to your application to interpret the comparison operators it receives. The meaning of comparison operators differs according to the Apple event objects being compared, and not all comparison operators apply to all object classes. After successfully comparing the Apple event objects, your object-comparison function should return the noErr result code and, in the result parameter, a Boolean value specifying TRUE if the result of the comparison is true and FALSE otherwise. If for any reason your comparison function is unable to compare the specified Apple event objects, it should return the result code errAEEventNotHandled; then the Apple Event Manager will try an alternative method of comparing the Apple event objects, such as calling the equivalent system object-comparison function, if one exists.

Your object-comparison function should be able to compare an Apple event object belonging to any object class with another Apple event object. Your function should also be able to compare two Apple event objects with different object classes, if appropriate. For example, an object-comparison function for a word-processing application might be asked to compare the First Name column of a specified row in a table with the first word on a specified page—that is, to compare an Apple event object of object class cColumn with an Apple event object of object class cWord. You must decide what kinds of comparisons make sense for your application.

The *Apple Event Registry: Standard Suites* defines standard comparison operators. Here is a list of the constants that correspond to these comparison operators:

```
CONST
    kAEGreaterThan       = '>   ';
    kAEGreaterThanEquals = '>=  ';
    kAEEquals            = '=   ';
    kAELessThan          = '<   ';
    kAELessThanEquals    = '<=  ';
    kAEBeginsWith        = 'bgwt';
    kAEEndsWith          = 'ends';
    kAEContains          = 'cont';
```

The comparison operators always relate the first operand to the second. For example, the constant kAEGreaterThan means that the object-comparison function should determine whether or not the value of the first operand is greater than the value of the second operand. For more information, see page 6-90.

Listing 6-11 shows an application-defined function, MyCompareObjects, that compares two Apple event objects of any object class supported by the application.

Listing 6-11   Object-comparison function that compares two Apple event objects

```
FUNCTION MyCompareObjects (comparisonOperator: DescType;
                           theObject: AEDesc;
                           objectOrDescToCompare: AEDesc;
                           VAR result: Boolean): OSErr;
BEGIN
   result := FALSE;
   {compare two objects for equivalence}
   IF comparisonOperator = kAEEquals THEN
      MyCompareObjects := MyCompEquals(theObject,
                                       objectOrDescToCompare,
                                       result)
   ELSE
      {compare two objects for greater than}
      IF comparisonOperator = kAEGreaterThan THEN
         MyCompareObjects := MyCompGreaterThan(theObject,
                                       objectOrDescToCompare,
                                       result)
   ELSE
      {compare two objects for less than}
      IF comparisonOperator = kAELessThan THEN
         MyCompareObjects := MyCompLessThan(theObject,
                                       objectOrDescToCompare,
                                       result)
   ELSE
   {this app does not support any other comparison operators}
      MyCompareObjects := errAEEventNotHandled;
END;
```

The `MyCompareObjects` function calls a separate application-defined routine for each comparison operator. In each case, the application-defined routine that actually performs the comparison can compare an Apple event object with either another Apple event object or with a descriptor record's data. If for any reason the comparison cannot be performed, the `MyCompareObjects` function returns the result code `errAEEventNotHandled`.

Writing Marking Callback Functions

Marking callback functions allow applications such as databases that can mark their own objects to take advantage of that capability when resolving object specifier records. Instead of returning a list of tokens for a group of Apple event objects that pass a test, your application can simply mark the Apple event objects and return a token that identifies how they have been marked. In this way, you can speed the resolution of complex object specifier records and reduce the amount of memory you need to allocate for tokens.

The use of marking callback functions is optional and usually makes sense if (a) you can reasonably expect that the tokens created in the process of resolving some object specifier records might not all fit in memory at once or (b) your application already uses a marking mechanism. If you want the Apple Event Manager to use marking callback functions provided by your application, you must add the kAEIDoMarking constant to the value of the callbackFlags parameter for the AEResolve function. If for any reason your application cannot mark a requested set of Apple event objects, it should return errAEEventNotHandled as the result code, and the Apple Event Manager will attempt to continue resolving the object specifier record by some other method, such as using a system marking function, if one exists.

If your application supports marking callback functions, it must provide three functions with declarations that match these examples:

```
FUNCTION MyGetMarkToken (containerToken: AEDesc;
                         containerClass: DescType;
                         VAR Result: AEDesc): OSErr;

FUNCTION MyMark (theToken: AEDesc; markToken: AEDesc;
                markCount: LongInt): OSErr;

FUNCTION MyAdjustMarks (newStart, newStop: LongInt;
                        markToken: AEDesc): OSErr;
```

For more detailed information about these sample declarations, see "Object Callback Functions," which begins on page 6-96.

To resolve a given object specifier record with the aid of the marking callback functions provided by your application, the Apple Event Manager first calls your application's **mark token function** (MyGetMarkToken), passing a token that identifies the container of the elements to be marked in the containerToken parameter and the container's object class in the containerClass parameter. The mark token function returns a mark token. A **mark token,** like other tokens, can be a descriptor record of any type; however, unlike other tokens, it identifies the way your application marks Apple event objects during the current session while resolving a single test. A mark token does not identify a specific Apple event object; rather, it allows your application to associate a group of objects with a marked set.

After it receives the mark token, the Apple Event Manager can call your application's **object-marking function** (`MyMark`) repeatedly to mark specific Apple event objects. The Apple Event Manager passes the following information to your marking function: in the `theToken` parameter, a token for the object to be marked (obtained from the appropriate object accessor function); in the `markToken` parameter, the current mark token; and in the `markCount` parameter, the mark count. The **mark count** indicates the number of times the Apple Event Manager has called the marking function for the current mark token. Your application should associate the mark count with each Apple event object it marks.

When the Apple Event Manager needs to identify either a range of elements or the absolute position of an element in a group of Apple event objects that pass a test, it can use your application's **mark-adjusting function** (`MyAdjustMarks`) to unmark objects that it has previously marked. For example, suppose an object specifier record specifies "any row in the table 'MyCustomers' for which the City column is 'San Francisco.'" The Apple Event Manager first uses the appropriate object accessor routine to locate all the rows in the table for which the City column is "San Francisco" and calls the application's marking function repeatedly to mark them. It then generates a random number between 1 and the number of rows it found that passed the test and calls the application's mark-adjusting function to unmark all the rows whose mark count does not match the randomly generated number. If the randomly chosen row has a mark count value of 5, the Apple Event Manager passes the mark-adjusting function 5 in both the `newStart` parameter and the `newStop` parameter, and the current mark token in the `markToken` parameter. The `newStart` and `newStop` parameters identify the beginning and end of the new set of marked objects that the mark-adjusting function will create by unmarking those previously marked objects not included in the new set.

When the Apple Event Manager calls your mark-adjusting function, your application must dispose of any data structures that it may have created to mark the previously marked objects. The Apple Event Manager calls your mark-adjusting function only once for a given mark token.

A mark token is valid until the Apple Event Manager either disposes of it (by calling `AEDisposeToken`) or returns it as the result of the `AEResolve` function. If the final result of a call to the `AEResolve` function is a mark token, the Apple event objects currently marked for that mark token are those specified by the object specifier record passed to `AEResolve`, and your application can proceed to do whatever the Apple event has requested. Note that your application is responsible for disposing of a final mark token with a call to `AEDisposeToken`, just as for any other final token.

If your application supports marking, it should also provide a token disposal function. When the Apple Event Manager calls `AEDisposeToken` to dispose of a mark token that is not the final result of a call to `AEResolve`, the subsequent call to your token disposal function lets you know that you can unmark the Apple event objects marked with that mark token. A call to `AEDisposeDesc` to dispose of a mark token (which would occur if you did not provide a token disposal function) would leave the objects marked.

Creating Object Specifier Records

If your application creates and sends Apple events that require the target application to locate Apple event objects, your application must create object specifier records for those events. This section describes how to use the four keyword-specified descriptor records described in "Descriptor Records Used in Object Specifier Records," which begins on page 6-8, to specify the object class ID, container, key form, and key data for an object specifier record.

Because the internal structure of an object specifier record is nearly identical to the internal structure of an AE record, it is possible to use `AECreateList`, `AEPutPtr`, and `AEPutKeyDesc` to add the four keyword-specified descriptor records to an AE record, then use `AECoerceDesc` to coerce the AE record to a descriptor record of type `typeObjectSpecifier`. However, it is usually preferable to use the `CreateObjSpecifier` function to accomplish the same goal. The `CreateObjSpecifier` function adds the keyword-specified descriptor records directly to an object specifier record, thus eliminating several steps that are required if you create an AE record first. The instructions that follow make use of `CreateObjSpecifier`.

To specify the class ID for an object specifier record, your application can specify the appropriate class ID value as the `desiredClass` parameter for the `CreateObjSpecifier` function, which uses it to create a keyword-specified descriptor record with the keyword `keyAEDesiredClass` as part of an object specifier record.

To specify the container for an object specifier record, your application must create a keyword-specified descriptor record with the keyword `keyAEContainer` that fully describes the container of the Apple event object. Because this container is usually another Apple event object, the container is usually specified by another object specifier record.

To specify the complete container hierarchy of an Apple event object, your application must create a series of nested object specifier records, starting with the object specifier record for the Apple event object whose container is outermost. With the exception of this first object specifier record, each object specifier record specifies another object specifier record in the chain as a container.

For example, Figure 6-2 on page 6-10 shows a series of nested object specifier records that specify the first row of a table named "Summary of Sales" in a document named "Sales Report." The logical organization of the same object specifier records is summarized in Table 6-7.

Table 6-7   Nested object specifier records that describe a container hierarchy

Keyword	Descriptor type	Data
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cTable
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cDocument
keyAEContainer	typeNull	**Data handle is** NIL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"Sales Report"
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"Summary of Sales"
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	1

Note

The format used in Table 6-7 and similar tables throughout this chapter does not show the structure of nested object specifier records as they exist within an Apple event. Instead, this format shows what you would obtain after calling AEGetKeyDesc repeatedly to extract the object specifier records from an Apple event record.

When you call AEGetKeyDesc to extract a null descriptor record, AEGetKeyDesc returns a descriptor record of type AEDesc with a descriptor type of typeNull and a data handle whose value is 0. u

To specify the default container for an object specifier record (such as the container for the document in Table 6-7), you can use AECreateDesc to create a null descriptor record, which you can then pass in the theContainer parameter of the CreateObjSpecifier function. The CreateObjSpecifier function uses the null descriptor record to create a keyword-specified descriptor record with the keyword keyAEContainer as part of an object specifier record.

The object specifier record that specifies the default container is always the first record you create in a series of nested object specifier records that specifies the complete container hierarchy for an Apple event object. Each one in the series uses the previously created object specifier record to specify its container. As with the null descriptor record, you can pass an object specifier record as the second parameter to the `CreateObjSpecifier` function, which uses it to create a keyword-specified descriptor record with the keyword `keyAEContainer`.

To specify the key form for an object specifier record, your application can specify a key form constant as the third parameter to the `CreateObjSpecifier` function, which uses it to create a keyword-specified descriptor record with the keyword `keyAEKeyForm` as part of an object specifier record. The standard key forms for object specifier records are summarized in Table 6-1 on page 6-12.

For example, the key form for the object specifier records in Table 6-7 that specify the document and the table is `formName`. In other words, the key data identifies the document and the table by their names. Similarly, the key form for the object specifier record in Table 6-7 that specifies the first row in the table is `formAbsolutePosition`. In other words, the key data identifies the position of the row compared to other rows in the same container.

To specify the key data for an object specifier record, your application must create a keyword-specified descriptor record with the keyword `keyAEKeyData` whose data handle refers to the appropriate data for the specified key form. You can use `AECreateDesc`, `CreateCompDescriptor`, `CreateLogicalDescriptor`, and related functions to create the descriptor record, which you can then pass in the fourth parameter of the `CreateObjSpecifier` function. The `CreateObjSpecifier` function uses this descriptor record to create a keyword-specified descriptor record with the keyword `keyAEKeyData` as part of an object specifier record.

Creating a Simple Object Specifier Record

This section shows how to use the `CreateObjSpecifier` function to create the object specifier record shown in Table 6-7. The `CreateObjSpecifier` function creates the necessary keyword-specified descriptor records for the class ID, container, key form, and key data and returns the resulting object specifier record as a descriptor record of type `typeObjectSpecifier`.

Listing 6-12 shows how the `CreateObjSpecifier` function creates an object specifier record from parameters that an application specifies.

Listing 6-12   Creating an object specifier record using `CreateObjSpecifier`

```
VAR
    desiredClass:        DescType;
    myObjectContainer:   AEDesc;
    myKeyForm:           DescType;
    myKeyDataDesc:       AEDesc;
    disposeInputs:       Boolean;
    myObjSpecRec:        AEDesc;
    myErr:               OSErr;

desiredClass := cRow;
myObjectContainer := MyGetContainer;
myKeyForm := formAbsolutePosition;
myKeyDataDesc := MyGetKeyData;
disposeInputs := TRUE;
{create an object specifier record}
myErr := CreateObjSpecifier(desiredClass, myObjectContainer,
                                myKeyForm, myKeyDataDesc,
                                disposeInputs, myObjSpecRec);
```

The code shown in Listing 6-12 demonstrates how an application might use the `CreateObjSpecifier` function to create four keyword-specified descriptor records as part of a descriptor record of type `typeObjectSpecifier`. The `CreateObjSpecifier` function returns a result code of `noErr` if the object specifier record was successfully created. The object specifier record returned in the `myObjSpecRec` parameter describes an Apple event object of the class specified by the `desiredClass` parameter, located in the container specified by the `myObjectContainer` parameter, with the key form specified by the `myKeyForm` parameter and key data specified by the `myKeyDataDesc` parameter.

You can specify `TRUE` in the `disposeInputs` parameter if you want the `CreateObjSpecifier` function to dispose of the descriptor records you created for the `myObjectContainer` and `myKeyDataDesc` parameters. If you specify `FALSE`, then your application is responsible for disposing of these leftover descriptor records.

Listing 6-13 shows an application-defined function that uses `CreateObjSpecifier` to create an object specifier record for the first row in the table named "Summary of Sales" in the document "Sales Report," then uses the object specifier record returned in the `myObjSpecRec` parameter as the direct parameter for a Get Data event.

Listing 6-13   Using `CreateObjSpecifier` in an application-defined function

```
FUNCTION MyRequestRowFromTarget (targetAddress: AEAddressDesc;
                                 VAR reply: AppleEvent): OSErr;
VAR
   desiredClass:          DescType;
   myKeyForm:             DescType;
   myObjectContainer:     AEDesc;
   myObjSpecRec:          AEDesc;
   myKeyDataDesc:         AEDesc;
   keyData:               LongInt;
   theAppleEvent:         AppleEvent;
   myErr:                 OSErr;
   ignoreErr:             OSErr;

BEGIN
   {initialize (set to null descriptor records) the two descriptor records }
   { that must eventually be disposed of}
   MyInit2DescRecs(myObjSpecRec, theAppleEvent);

   desiredClass := cRow;                        {specify the class}
                                                {specify container for the row}
   myErr := MyCreateTableContainer(myObjectContainer,
                                   'Summary of Sales', 'Sales Report');
   IF myErr = noErr THEN
   BEGIN
      myKeyForm := formAbsolutePosition;    {specify the key form}
      keyData := 1;                         {specify the key data for row}
      myErr := AECreateDesc(typeLongInteger, @keyData, Sizeof(keyData),
                            myKeyDataDesc);
      IF myErr = noErr THEN
         {create the object specifier record}
         myErr := CreateObjSpecifier(desiredClass, myObjectContainer,
                                     myKeyForm, myKeyDataDesc,
                                     TRUE, myObjSpecRec);
      IF myErr = noErr THEN
         {myObjSpecRec now describes an Apple event object, and will become }
         { direct parameter of a Get Data event; first create Get Data event}
         myErr := AECreateAppleEvent(kAECoreSuite, kAEGetData, targetAddress,
                                     kAutoGenerateReturnID,
                                     kAnyTransactionID, theAppleEvent);
```

```
        IF myErr = noErr THEN
            {add myObjSpecRec as the direct parameter of the Get Data event}
            myErr := AEPutParamDesc(theAppleEvent, keyDirectObject,
                                    myObjSpecRec);
        IF myErr = noErr THEN
            myErr := AESend(theAppleEvent, reply, kAEWaitReply +
                            kAENeverInteract, kAENormalPriority, 120,
                            @MyIdleFunction, NIL);
    END;
    ignoreErr := AEDisposeDesc(myObjSpecRec);
    ignoreErr := AEDisposeDesc(theAppleEvent);
    MyRequestRowFromTarget := myErr;
END;
```

The `MyRequestRowFromTarget` function shown in Listing 6-13 specifies the class ID as `cRow`, indicating that the desired Apple event object is a row in a table. It uses the application-defined function `MyCreateTableContainer` to create an object specifier record for the table that contains the row, passing "Summary of Sales" and "Sales Report" as the second and third parameters to identify the name of the table and the name of the document that contains the table. (The next section, "Specifying the Container Hierarchy," explains how to construct the `MyCreateTableContainer` function.) It then specifies the key form as the constant `formAbsolutePosition`, which indicates that the key data specifies the position of the row within its container; sets the `keyData` variable to 1, indicating the first row, and uses `AECreateDesc` to create a descriptor record for the key data; and uses `CreateObjSpecifier` to create the object specifier record that describes the desired word.

The desired row is now fully described by the `myObjSpecRec` variable, which contains a descriptor record of type `typeObjectSpecifier` that contains the three nested object specifier records shown in Table 6-7 on page 6-56. After using `AECreateAppleEvent` to create a Get Data event, the `MyRequestRowFromTarget` function uses the `AEPutParamDesc` function to add the `myObjSpecRec` variable to the Get Data event as a direct parameter, then uses `AESend` to send the Get Data event.

Note that the `MyRequestRowFromTarget` function begins by using the application-defined function `MyInit2DescRecs` to set `myObjSpecRec` and `theAppleEvent` to null descriptor records. These two functions must be disposed of whether the function is successful or not. By setting them to null descriptor records, the function can dispose of them at the end regardless of where an error may have occurred.

Specifying the Container Hierarchy

Because the container for an object specifier record usually consists of a chain of other object specifier records that specify the container hierarchy, your application must create all the object specifier records in the chain, starting with the record for the outermost container. Listing 6-14 and Listing 6-15 demonstrate how to use the `CreateObjSpecifier` function to create the first two object specifier records in such a chain: the records for a document and a table.

Listing 6-14   Specifying a document container

```
FUNCTION MyCreateDocContainer (VAR myDocContainer: AEDesc;
                                    docName: Str255): OSErr;
VAR
   myDocDescRec:   AEDesc;
   nullDescRec:    AEDesc;
   myErr:          OSErr;
BEGIN
   {create a descriptor record for the name of the document}
   myErr := AECreateDesc(typeChar, @docName[1],
                            Length(docName), myDocDescRec);
   IF myErr = noErr THEN
      {create a null descriptor record}
      myErr := AECreateDesc(typeNull, NIL, 0, nullDescRec);
   IF myErr = noErr THEN
      {create an object specifier record to specify the }
      { document object}
      myErr := CreateObjSpecifier(cDocument, nullDescRec,
                                    formName, myDocDescRec, TRUE,
                                    myDocContainer);
   MyCreateDocContainer := myErr;
END;
```

The function `MyCreateDocContainer` in Listing 6-14 creates an object specifier record that identifies a document by name. It starts by using the `AECreateDesc` function to create two descriptor records: one of type `typeChar` for the name of the document, and one of type `typeNull` for the null descriptor record that specifies the default container (because the document is not contained in any other Apple event object). These two descriptor records can then be used as parameters for the `CreateObjSpecifier` function, which returns an object specifier record (that is, a descriptor record of type `typeObjectSpecifier`) in the `myDocContainer` variable. The object specifier record specifies an Apple event object of the object class `cDocument` in the container specified by the `nullDescRec` variable with a key form of `formName` and the key data specified by the `myDocDescRec` variable. This object specifier can be used by itself to specify a document, or it can be used to specify the container for another Apple event object.

Listing 6-15 shows an application-defined function, `MyCreateTableContainer`, that creates an object specifier record describing a table contained in a document.

Listing 6-15   Specifying a table container

```
FUNCTION MyCreateTableContainer (VAR myTableContainer: AEDesc;
                                 tableName: Str255;
                                 docName: Str255): OSErr;
VAR
   myDocDescRec:        AEDesc;
   myTableDescRec:      AEDesc;
   myErr:               OSErr;
BEGIN
   {create a container for the document}
   myErr := MyCreateDocContainer(myDocDescRec, docName);
   IF myErr = noErr THEN
   BEGIN
      {create the table container, }
      { first specify the descriptor record for the key data}
      myErr := AECreateDesc(typeChar, @tableName[1],
                            Length(tableName), myTableDescRec);
      IF myErr = noErr THEN
         myErr := CreateObjSpecifier(cTable, myDocDescRec,
                                     formName, myTableDescRec,
                                     TRUE, myTableContainer);
   END;
   MyCreateTableContainer := myErr;
END;
```

The function `MyCreateTableContainer` in Listing 6-15 starts by using the function `MyCreateDocContainer` from Listing 6-14 to create an object specifier record that identifies the table's container—the document in which the table is located. Then it uses the `AECreateDesc` function to create a descriptor record for the key data—a name that, when combined with the key form `formName`, will identify the table in the document. The object specifier record for the document and the descriptor record specifying the table's name are passed to the function `CreateObjSpecifier`. It returns an object specifier record in the `myTableContainer` parameter that specifies an Apple event object of the object class `cTable` in the container specified by the `MyDocDescRec` variable with a key form of `formName` and the key data specified by the `myTableDescRec` variable. This object specifier record can be used by itself to specify a table, or it can be used to specify the container for another Apple event object.

Listing 6-13 uses the `MyCreateTableContainer` function shown in Listing 6-15 to specify the container hierarchy illustrated in Table 6-7 on page 6-56. The nested object specifier records shown in Table 6-7 use the key forms `formName` and `formRelativePosition`. You can create key data for the key forms `formPropertyID`, `formUniqueID`, and `formRelativePosition` using similar techniques.

Specifying a Property

The key form `formPropertyID` allows your application to specify key data identifying a property of the object specified as a container. For example, an object specifier record that identifies the font property of a word specifies `cProperty` as the class ID, an object specifier record for the word as the property's container, `formPropertyID` as the key form, and the constant `pFont` as the key data.

Note that an object specifier record that identifies a property does not include a value for the property, such as `Palatino`. The value of a property is returned or set as a parameter of an Apple event. For example, an application that sends a Get Data event to get the `pFont` property of a word receives a value such as `Palatino` in the `keyAEResult` parameter of the reply event, and an application that sends a Set Data event to change the `pFont` property of a word specifies a font in the `keyAEData` parameter of the Set Data event.

To specify the key data for a key form of `formPropertyID`, your application must create a descriptor record of `typeType` whose data consists of a constant specifying a property. You can use `AECreateDesc` to create a descriptor record that specifies the constant for a property, then use `CreateObjSpecifier` to add the descriptor record to an object specifier record as a keyword-specified descriptor record with the keyword `keyAEKeyData`.

For more information about object specifier records that specify a property, see "Key Data for a Property ID" on page 6-13.

Specifying a Relative Position

The key form `formRelativePosition` allows your application to specify key data identifying an element or a set of elements that are immediately before or after the specified container. For example, if the container is a table, you could use a key form of `formRelativePosition` to specify the paragraph before or after the table.

To specify the key data for a key form of `formRelativePosition`, your application must create a descriptor record of `typeEnumerated` whose data consists of a constant specifying either the element after (`kAENext`) or the element before (`kAEPrevious`) the specified container.

You can use `AECreateDesc` to create a descriptor record that specifies one of these constants, then use `CreateObjectSpecifier` to add it to an object specifier record as a keyword-specified descriptor record with the keyword `keyAEKeyData`.

For more information about object specifier records that specify a relative position, see "Key Data for Relative Position" on page 6-15.

Creating a Complex Object Specifier Record

This section describes how to create object specifier records that specify a test or a range. You can specify the object class ID for these object specifier records the same way you would for any other object specifier record. When you create the other three keyword-specified descriptor records, however, you can use additional Apple Event Manager routines and descriptor types to specify any combination of Apple event objects.

Specifying a Test

The key form `formTest` allows your application to specify key data that identifies one or more elements in the specified container that pass a test. To do so, your application must construct several interconnected descriptor records that specify comparisons and, if necessary, logical expressions.

For example, to specify "the first row in which the First Name column equals 'John' and the Last Name column equals 'Chapman' in the table 'MyAddresses' of the database 'SurfDB,'" your application must construct an object specifier record whose key data describes a logical expression that applies the logical operator AND to two separate comparisons for each row: a comparison of the First Name column to the word "John" and a comparison of the Last Name column to the word "Chapman."

The logical organization of the data for the object specifier record that specifies this test is summarized in Table 6-8 and Table 6-9. (It is also illustrated in Figure 6-3 and Figure 6-4, beginning on page 6-18.) The listings in the remainder of this section demonstrate how to create this object specifier record. For general information about the organization of key data for a test, see "Key Data for a Test," which begins on page 6-15.

Table 6-8   Object specifier record for the first row that meets a test in the table named "MyAddresses"

Keyword	Descriptor type	Data
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cTable
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cDatabase
keyAEContainer	typeNull	Data handle is NIL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"SurfDB"
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"MyAddresses"
keyAEKeyForm	typeEnumerated	formTest
keyAEKeyData	typeLogicalDescriptor	(see Table 6-9)
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	1

Table 6-9   Logical descriptor record that specifies a test

Keyword	Descriptor type	Data
keyAELogicalOperator	typeEnumerated	kAEAnd
keyAELogicalTerms	typeAEList	(see indented records)
	typeCompDescriptor	(see indented record)
keyAECompOperator	typeType	kAEEquals
keyAEObject1	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cColumn
keyAEContainer	typeObjectBeingExamined	Data handle is NIL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"First Name"
keyAEObject2	typeChar	"John"
	typeCompDescriptor	(see indented record)
keyAECompOperator	typeType	kAEEquals
keyAEObject1	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cColumn
keyAEContainer	typeObjectBeingExamined	Data handle is NIL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"Last Name"
keyAEObject2	typeChar	"Chapman"

Because both the database and the table shown in Table 6-8 are specified by name, it would be convenient to have an application-defined routine that creates an object specifier record that uses the key form `formName`. The `MyCreateFormNameObjSpecifier` function shown in Listing 6-16 can be used for this purpose.

Listing 6-16   Creating an object specifier record with the key form `formName`

```
FUNCTION MyCreateFormNameObjSpecifier
                        (class: DescType; container: AEDesc;
                         keyDataName: str255;
                         VAR resultObjSpecRec: AEDesc): OSErr;
VAR
   keyDataDescRec: AEDesc;
   myErr:          OSErr;
BEGIN
   myErr := AECreateDesc(typeChar, @keyDataName[1],
                            Length(keyDataName), keyDataDescRec);
   IF myErr = noErr THEN
      myErr := CreateObjSpecifier(class, container, formName,
                                     keyDataDescRec, TRUE,
                                     resultObjSpecRec);
   MyCreateFormNameObjSpecifier := myErr;
END;
```

The `MyCreateFormNameObjSpecifier` function shown in Listing 6-16 returns, in the `resultObjSpecRec` parameter, an object specifier record that describes an Apple event object of the class specified by the `class` parameter, located in the container specified by the `container` parameter, with the key form `formName` and key data specified by the `keyDataName` parameter. This function is used in Listing 6-19 on page 6-70 to create object specifier records that use the key form `formName` for the database and the table.

The nested object specifier records shown in Table 6-9 specify "the rows in which the First Name column equals 'John' and the Last Name column equals 'Chapman.'" To identify the rows that pass this test, the Apple Event Manager needs to evaluate two comparisons: the comparison of each row of the First Name column to the word "John," and the comparison of each row of the Last Name column to the word "Chapman."

The Apple Event Manager uses the information in comparison descriptor records to compare the specified elements in a container, one at a time, either to another Apple event object or to the data associated with a descriptor record. The two comparison descriptor records you need to create for this example are summarized in Table 6-9 on page 6-66.

You can use the CreateCompDescriptor function to create a comparison descriptor
record, or you can create an AE record and use AECoerceDesc to coerce it to a
comparison descriptor record. Listing 6-17 shows an example of an application-defined
routine that creates an object specifier record and a descriptor record of typeChar, then
uses the CreateCompDescriptor function to add them to a comparison descriptor
record.

Listing 6-17 Creating a comparison descriptor record

```
FUNCTION MyCreateComparisonDescRec (VAR compDesc: AEDesc;
                                    colName: str255;
                                    name: str255): OSErr;
VAR
   logicalContainer, colNameDesc, nameDesc:  AEDesc;
   myObjectExaminedContainer:                AEDesc;
   myErr:                                    OSErr;
BEGIN
   {create the object specifier record for keyAEObject1; }
   { first create container}
   myErr := AECreateDesc(typeObjectBeingExamined, NIL, 0,
                         myObjectExaminedContainer);
   {create key data}
   IF myErr = noErr THEN
      myErr := AECreateDesc(typeChar, @colName[1],
                            Length(colName), colNameDesc);
   {now create the object specifier record}
   IF myErr = noErr THEN
      myErr := CreateObjSpecifier(cColumn,
                                  myObjectExaminedContainer,
                                  formName, colNameDesc, TRUE,
                                  logicalContainer);
   {create the descriptor record for keyAEObject2}
   IF myErr = noErr THEN
      myErr := AECreateDesc(typeChar, @name[1], Length(name),
                            nameDesc);
   {create the first logical term (comp descriptor record)}
   IF myErr = noErr THEN
      myErr := CreateCompDescriptor(kAEEquals, logicalContainer,
                                    nameDesc, TRUE, compDesc);
   MyCreateComparisonDescRec := myErr;
END;
```

The `MyCreateComparisonDescRec` function takes two strings and uses them to create a comparison descriptor record. The string passed in the second parameter specifies the name of the column whose contents should be compared to the string passed in the third parameter. First, the `MyCreateComparisonDescRec` function uses `AECreateDesc` to create a descriptor record of `typeObjectBeingExamined`, which is returned in the variable `myObjectExaminedContainer`. Next, `AECreateDesc` creates a descriptor record of descriptor type `typeChar`, whose data consists of the string in the variable `colName`, and which is returned in the variable `colNameDesc`. The code then passes the variables `myObjectExaminedContainer` and `colNameDesc` to the `CreateObjSpecifier` function, which uses them to create an object specifier record, returned in the `logicalContainer` variable, that becomes the keyword-specified descriptor record with the keyword `keyAEObject1`.

Next, the `MyCreateComparisonDescRec` function uses `AECreateDesc` and the `name` parameter to create the descriptor record for `keyAEObject2`, which `AECreateDesc` returns in the `nameDesc` variable. Finally, the code passes the constant `kAEEquals`, the variable `logicalContainer`, and the variable `nameDesc` to the `CreateCompDescriptor` function, which creates a comparison descriptor record that allows the Apple Event Manager (with the help of object-comparison functions provided by the server application) to determine whether the specified column in the row currently being checked equals the specified string.

You can use the `MyCreateComparisonDescRec` function to create both the descriptor records of type `typeCompDescriptor` shown in Table 6-9 on page 6-66. These descriptor records provide two logical terms for a logical descriptor record. The entire logical descriptor record corresponds to the logical expression "the First Name column equals 'John' `AND` the Last Name column equals 'Chapman.'"

You can use the `CreateLogicalDescriptor` function to create a logical descriptor record, or you can create an AE record and use the `AECoerceDesc` function to coerce it to a comparison descriptor record. Listing 6-18 shows an application-defined function that adds two comparison descriptor records to a descriptor list, then uses the `CreateLogicalDescriptor` function to create a logical descriptor record whose logical terms are the two comparison descriptor records.

Listing 6-18   Creating a logical descriptor record

```
FUNCTION MyCreateLogicalDescRec (VAR compDesc1, compDesc2: AEDesc;
                                logicalOperator: DescType;
                                VAR logicalDesc: AEDesc): OSErr;
VAR
   logicalTermsList: AEDescList;
   myErr:            OSErr;
BEGIN
   {create a logical descriptor record that contains two }
   { comparison descriptor records}
   {first create a list}
   myErr := AECreateList(NIL, 0, FALSE, logicalTermsList);
   IF myErr = noErr THEN
      myErr := AEPutDesc(logicalTermsList, 1, compDesc1);
   IF myErr = noErr THEN
      myErr := AEPutDesc(logicalTermsList, 2, compDesc2);
   IF myErr = noErr THEN
      myErr := AEDisposeDesc(compDesc1);
   IF myErr = noErr THEN
      myErr := AEDisposeDesc(compDesc2);
   IF myErr = noErr THEN
      myErr := CreateLogicalDescriptor(logicalTermsList,
                                       logicalOperator, TRUE,
                                       logicalDesc);
   MyCreateLogicalDescRec := myErr;
END;
```

Listing 6-19 uses the application-defined functions shown in Listing 6-16, Listing 6-17, and Listing 6-18 to build the object specifier record illustrated in Table 6-8 and Table 6-9.

Listing 6-19   Creating a complex object specifier record

```
FUNCTION MyCreateObjSpecRec (VAR theResultObj: AEDesc): OSErr;
VAR
   nullContainer, databaseContainer, tableContainer:  AEDesc;
   compDesc1, compDesc2:                              AEDesc;
   logicalTestDesc, rowTestContainer, rowOffset:      AEDesc;
   myErr:                                             OSErr;
```

```
BEGIN
   {create a null container}
   myErr := AECreateDesc(typeNull, NIL, 0, nullContainer);
   {create a container for the database}
   IF myErr = noErr THEN
      myErr := MyCreateFormNameObjSpecifier(cDatabase, nullContainer,
                                            'SurfDB', databaseContainer);
   {create a container for the table}
   IF myErr = noErr THEN
      myErr := MyCreateFormNameObjSpecifier(cTable, databaseContainer,
                                            'MyAddresses', tableContainer);
   {create a container for the row--an object specifier record that }
   { specifies a test (the row whose First Name column = 'John' and }
   { Last Name column = 'Chapman')}

   {create the first comparison descriptor record}
   IF myErr = noErr THEN
      myErr := MyCreateComparisonDescRec(compDesc1, 'First Name', 'John');
   {create the second comparison descriptor record}
   IF myErr = noErr THEN
      myErr := MyCreateComparisonDescRec(compDesc2, 'Last Name', 'Chapman');

   {create the logical descriptor record}
   IF myErr = noErr THEN
      myErr := MyCreateLogicalDescRec(compDesc1, compDesc2, kAEAND,
                                      logicalTestDesc);

   {now create the object specifier record that specifies the test}
   IF myErr = noErr THEN
      myErr := CreateObjSpecifier(cRow, tableContainer, formTest,
                                  logicalTestDesc, TRUE, rowTestContainer);

   {create the object specifier record for the Apple event object}
   {first, create the descriptor record for the key data}
   IF myErr = noErr THEN
      myErr := CreateOffsetDescriptor (1, rowOffset);
   {now create the object specifier record}
   IF myErr = noErr THEN
      myErr := CreateObjSpecifier (cRow, rowTestContainer,
                                   formAbsolutePosition, rowOffset,
                                   TRUE, theResultObj);
   MyCreateObjSpecRec := myErr;
END;
```

The `MyCreateObjSpecRec` function shown in Listing 6-19 begins by using `AECreateDesc` to create a null descriptor record, then uses the `MyCreateFormNameObjSpecifier` function (shown in Listing 6-16) to specify the default container for the database named "SurfDB." The code then calls the `MyCreateFormNameObjSpecifier` function again, this time passing the object specifier record for SurfDB to specify the container for the table "MyAddresses." The next two calls are both to the `MyCreateComparisonDescRec` function (shown in Listing 6-17), which creates the comparison descriptor records that allow the Apple Event Manager to compare the First Name column and Last Name column to the names "John" and "Chapman," respectively. The next call passes these two comparison descriptor records to the `MyCreateLogicalDescRec` function (shown in Listing 6-18) in the `compDesc1` and `compDesc2` variables.

Now all the components of the logical descriptor record are ready to assemble. The next call, to `CreateObjSpecifier`, specifies the logical descriptor record in the `logicalTestDesc` variable as the key data for the object specifier record that specifies the test. A call to the Apple Event Manager routine `CreateOffsetDescriptor` then creates an offset descriptor record that contains the integer 1. Finally, the code passes the offset descriptor record to the `CreateObjSpecifier` function in the `rowOffset` variable to create the final object specifier record, which describes the requested row as the first row that passes the test.

The `CreateOffsetDescriptor` function creates a descriptor record of type `typeLongInteger` that can be used as the key data with a key form of `formAbsolutePosition` to indicate an element's offset within its container. A positive integer indicates an offset from the beginning of the container (the first element has an offset of 1), and a negative integer indicates an offset from the end of the container (the last element has an offset of –1). Using `CreateOffsetDescriptor` accomplishes the same thing as setting a variable to an integer and passing the variable to `AECreateDesc` to create a descriptor record of type `typeLongInteger`.

Specifying a Range

The key form `formRange` allows your application to specify key data that identifies a range of elements in the specified container. To do so, your application must first create a range descriptor record. The Apple Event Manager uses a range descriptor record to identify the two Apple event objects that specify the beginning and end of a range of elements.

For example, an object specifier record for a range of text in a document could specify the table named "Summary of Sales" as the first boundary object and the figure named "Best-Selling Widgets for 1991" as the second boundary object for a range that consists of all the text between the table and the figure. Any word processor that keeps track of the relative positions of text, tables, and figures should be capable of supporting such a request.

Table 6-10 summarizes the logical organization of the data for the object specifier record that specifies this range. For general information about the organization of data within a range descriptor record, see "Key Data for a Range" on page 6-20.

Table 6-10   A range descriptor record

Keyword	Descriptor type	Data
keyAERangeStart	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cTable
keyAEContainer	typeCurrentContainer	Data handle is NIL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"Summary of Sales"
keyAERangeStop	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cFigure
keyAEContainer	typeCurrentContainer	Data handle is NIL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"Best-Selling Widgets for 1991"

You can use the CreateRangeDescriptor function to create a range descriptor record, or you can create an AE record and use AECoerceDesc to coerce it to a range descriptor record. Listing 6-20 provides an example of an application-defined routine that creates two object specifier records, then uses the CreateRangeDescriptor function to add them to a range descriptor record.

The container for the boundary objects in the range descriptor record created by Listing 6-20 is the same as the container for the range itself. The object specifier record for the range's container is added to an object specifier record of key form formRange at the same time that the range descriptor record is added as key data. The container for the two boundary objects can therefore be specified in the range descriptor record by a descriptor record of type typeCurrentContainer whose data handle has the value NIL. The Apple Event Manager interprets this as a placeholder for the range's container when it is resolving the range descriptor record.

Listing 6-20   Creating a range descriptor record

```
FUNCTION MyCreateRangeDescriptor (VAR rangeDescRec: AEDesc): OSErr;
VAR
   rangeStart:          AEDesc;
   rangeEnd:            AEDesc;
   currentContainer:    AEDesc;
   tableNameDescRec:    AEDesc;
   figureNameDescRec:   AEDesc;
   myErr:               OSErr;
BEGIN
{create the object specifier record for the start of the range }
{ (the table named 'Summary of Sales' in 'MyDoc' document)}

   {create a descriptor record of type typeCurrentContainer}
   myErr := AECreateDesc(typeCurrentContainer, NIL, 0, currentContainer);

   {create the object specifier record}
   IF myErr = noErr THEN
      myErr := MyCreateNameDescRec(tableNameDescRec,
                                   'Summary of Sales');
   IF myErr = noErr THEN
   myErr := CreateObjSpecifier(cTable, currentContainer, formName,
                               tableNameDescRec, FALSE, rangeStart);

   myErr := AEDisposeDesc(tableNameDescRec);
   {create the object specifier record for the end of the range }
   { (the figure named 'Best-Selling Widgets...' in 'MyDoc') }
   IF myErr = noErr THEN
      myErr := MyCreateNameDescRec(figureNameDescRec,
                                   'Best-Selling Widgets for 1991');

   IF myErr = noErr THEN
      myErr := CreateObjSpecifier(cFigure, currentContainer, formName,
                                  figureNameDescRec, TRUE, rangeEnd);

   {now create the range descriptor record}
   IF myErr = noErr THEN
      myErr := CreateRangeDescriptor(rangeStart, rangeEnd, TRUE,
                                     rangeDescRec);
   MyCreateRangeDescriptor := myErr;
END;
```

After creating a descriptor record of type `typeCurrentContainer` and a descriptor record for the first table's name, Listing 6-20 uses the `CreateObjSpecifier` function to create an object specifier record that identifies the beginning of the range. The parameters to `CreateObjSpecifier` specify that the beginning of the range is an Apple event object of the object class `cTable` in the current container, with a key form of `formName` and key data that identifies the table by name. A second call to `CreateObjSpecifier` creates the object specifier record that identifies the end of the range—an Apple event object of the `cFigure` object class in the current container, with a key form of `formName` and key data that identifies the figure by name. Finally, the code in Listing 6-20 uses the `CreateRangeDescriptor` function to create the range descriptor record, using the two previously created object specifier records to specify the beginning and end of the range.

Reference to Resolving and Creating Object Specifier Records

This section describes the Apple Event Manager routines your application can use to resolve and create object specifier records. It also describes application-defined object accessor functions and object callback functions that your application can provide for use by the Apple Event Manager in resolving object specifier records.

The first section, "Data Structures Used in Object Specifier Records," summarizes the descriptor types and associated data that can be used in an object specifier record. "Routines for Resolving and Creating Object Specifier Records," which begins on page 6-77, describes the Apple Event Manager routines you use to initialize the Object Support Library, resolve object specifier records, set and manipulate object accessor functions, deallocate memory for tokens, and create object specifier records. "Application-Defined Routines," which begins on page 6-94, describes the object accessor functions and object callback functions that a server application can provide.

Data Structures Used in Object Specifier Records

The data for object specifier records can be specified using a variety of descriptor records and descriptor types. These are described in detail in "Descriptor Records Used in Object Specifier Records," which begins on page 6-8, and summarized in Table 6-11.

Table 6-11   Keyword-specified descriptor records for `typeObjectSpecifier`

Keyword	Descriptor type	Data
`keyAEDesiredClass`	`typeType`	Object class ID
`keyAEContainer`	`typeObjectSpecifier`	Object specifier record
	`typeNull`	Data handle is NIL. Specifies the default container at the top of the container hierarchy.
	`typeObjectBeingExamined`	Data handle is NIL. Specifies the container for elements that are tested one at a time; used only with `formTest`.
	`typeCurrentContainer`	Data handle is NIL. Specifies a container for an element that demarcates one boundary in a range. Used only with `formRange`.
`keyAEKeyForm`	`typeEnumerated`	`formPropertyID` `formName` `formUniqueID` `formAbsolutePosition` `formRelativePosition` `formTest` `formRange` `formWhose`
`keyAEKeyData`		(See indented key forms)
for `formPropertyID`	`typeType`	Property ID for an element's property
for `formName`	`typeChar` **or other text type**	Element's name
for `formUniqueID`	**Any appropriate type**	Element's unique ID
for `formAbsolutePosition`	`typeLongInteger`	Offset from beginning (positive) or end (negative) of container
	`typeAbsoluteOrdinal`	`kAEFirst` `kAEMiddle` `kAELast` `kAEAny` `kAEAll`
for `formRelativePosition`	`typeEnumerated`	`kAENext` `kAEPrevious`
for `formTest`	`typeCompDescriptor`	(See Table 6-2 on page 6-16)
	`typeLogicalDescriptor`	(See Table 6-3 on page 6-17)
for `formRange`	`typeRangeDescriptor`	(See Table 6-4 on page 6-20)
for `formWhose`	`typeWhoseDescriptor`	(See Table 6-5 on page 6-42)

Routines for Resolving and Creating Object Specifier Records

This section describes routines for initializing the Object Support Library, resolving object specifier records, setting and manipulating object accessor functions, deallocating memory for tokens, and creating object specifier records.

Initializing the Object Support Library

You should call the `AEObjectInit` function to initialize the Apple Event Manager routines that handle object specifier records and Apple event objects. To make these routines available to your application with version 1.01 and earlier versions of the Apple Event Manager, you must also link the Apple Event Object Support Library with your application when you build it.

AEObjectInit

You use the `AEObjectInit` function to initialize the Object Support Library.

```
FUNCTION AEObjectInit: OSErr;
```

DESCRIPTION

You must call this function before calling any of the Apple Event Manager routines that describe or manipulate Apple event objects.

RESULT CODES

noErr	0	No error occurred
memFullErr	–108	Not enough room in heap zone
errAENewerVersion	–1706	Need a newer version of the Apple Event Manager

Setting Object Accessor Functions and Object Callback Functions

The Apple Event Manager provides two routines that allow you to specify the object accessor functions and object callback functions provided by your application. The `AEInstallObjectAccessor` function adds an entry for an object accessor function to either the application's object accessor dispatch table or the system object accessor dispatch table. The `AESetObjectCallbacks` function allows you to specify the object callback functions to be called for your application.

AEInstallObjectAccessor

You can use the `AEInstallObjectAccessor` function to add an entry for an object accessor function to either the application's object accessor dispatch table or the system object accessor dispatch table.

```
FUNCTION AEInstallObjectAccessor (desiredClass: DescType;
                                  containerType: DescType;
                                  theAccessor: AccessorProcPtr;
                                  accessorRefcon: LongInt;
                                  isSysHandler: Boolean): OSErr;
```

`desiredClass`
: The object class of the Apple event objects to be located by the object accessor function for this table entry.

`containerType`
: The descriptor type of the token used to specify the container for the desired objects. The object accessor function finds objects in containers specified by tokens of this type.

`theAccessor`
: A pointer to the object accessor function for this table entry. Note that an object accessor function listed in the system dispatch table must reside in the system heap; thus, if the value of the `isSysHandler` parameter is `TRUE`, the `theAccessor` parameter should point to a location in the system heap. Otherwise, if you put your system object accessor function in your application heap, you must call `AERemoveObjectAccessor` to remove the function before your application terminates.

`accessorRefcon`
: A reference constant passed by the Apple Event Manager to the object accessor function whenever the function is called. If your object accessor function doesn't use a reference constant, use 0 as the value of this parameter. To change the value of the reference constant, you must call `AEInstallObjectAccessor` again.

`isSysHandler`
: A value that specifies the object accessor dispatch table to which the entry is added. If the value of `isSysHandler` is `TRUE`, the Apple Event Manager adds the entry to the system object accessor dispatch table. Entries in the system object accessor dispatch table are available to all applications running on the same computer. If the value is `FALSE`, the Apple Event Manager adds the entry to your application's object accessor table. When searching for object accessor functions, the Apple Event Manager searches the application's object accessor dispatch table first; it searches the system object accessor dispatch table only if the necessary function is not found in your application's object accessor dispatch table.

DESCRIPTION

The `AEInstallObjectAccessor` function adds an entry to either the application or system object accessor dispatch table. You must supply parameters that specify the object class of the Apple event objects that the object accessor function can locate, the descriptor type of tokens for containers in which the object accessor function can locate objects, the address of the object accessor function for which you are adding an entry, and whether the entry is to be added to the system object accessor dispatch table or your application's object accessor dispatch table. You can also specify a reference constant that the Apple Event Manager passes to your object accessor function each time the Apple Event Manager calls the function.

RESULT CODES

noErr	0	No error occurred
paramErr	–50	The handler pointer is `NIL` or odd, or `AEObjectInit` was not called before this function

SEE ALSO

For more information about installing object accessor functions, see "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

For a description of the `AERemoveObjectAccessor` function, see page 6-84.

AESetObjectCallbacks

You can use the `AESetObjectCallbacks` function to specify the object callback functions to be called for your application.

```
FUNCTION AESetObjectCallbacks (myCompareProc, myCountProc,
                              myDisposeTokenProc,
                              myGetMarkTokenProc, myMarkProc,
                              myAdjustMarksProc,
                              myGetErrDescProc: ProcPtr): OSErr;
```

`myCompareProc`
> Either a pointer to the object-comparison function provided by your application or `NIL` if no function is provided.

`myCountProc`
> Either a pointer to the object-counting function provided by your application or `NIL` if no function is provided.

`myDisposeTokenProc`
> Either a pointer to the token disposal function provided by your application or `NIL` if no function is provided.

myGetMarkTokenProc
Either a pointer to the function for returning a mark token provided by your application or `NIL` if no function is provided.

myMarkProc Either a pointer to the object-marking function provided by your application or `NIL` if no function is provided.

myAdjustMarksProc
Either a pointer to the mark-adjusting function provided by your application or `NIL` if no function is provided.

myGetErrDescProc
Either a pointer to the error callback function provided by your application or `NIL` if no function is provided.

DESCRIPTION

Your application can provide only one each of the object callback functions specified by `AESetObjectCallbacks`: one object-comparison function, one object-counting function, and so on. As a result, each of these callback functions must perform the requested task (comparing, counting, and so on) for all the object classes that your application supports. In contrast, your application may provide many different object accessor functions if necessary, depending on the object classes and token types your application supports.

To replace object callback routines that have been previously installed, you can make another call to `AESetObjectCallbacks`. Each additional call to `AESetObjectCallbacks` replaces any object callback functions installed by previous calls to `AESetObjectCallbacks`. You cannot use `AESetObjectCallbacks` to replace system object callback routines or object accessor functions. Only those routines you specify are replaced; to avoid replacing existing callback functions, specify a value of `NIL` for the functions you don't want to replace.

RESULT CODES

noErr	0	No error occurred
paramErr	–50	The handler pointer is `NIL` or odd, or `AEObjectInit` was not called before this function
memFullErr	–108	There is not enough room in heap zone
errAENotASpecialFunction	–1714	The keyword is not valid for a special function

SEE ALSO

For information about writing object callback functions, see "Application-Defined Routines," which begins on page 6-94.

To install system object callback functions, use the `AEInstallSpecialHandler` function described on page 4-100.

Getting, Calling, and Removing Object Accessor Functions

The Apple Event Manager provides three functions that allow you to get, call, and remove object accessor functions that you have installed in either the system or application object accessor dispatch table with the `AEInstallObjectAccessor` function. The `AEGetObjectAccessor` and `AECallObjectAccessor` functions get and call object accessor functions installed in the dispatch table you specify, and `AERemoveObjectAccessor` removes an installed function.

AEGetObjectAccessor

You can use the `AEGetObjectAccessor` function to get a pointer to an object accessor function and the value of its reference constant.

```
FUNCTION AEGetObjectAccessor (desiredClass: DescType;
                              containerType: DescType;
                              VAR theAccessor: AccessorProcPtr;
                              VAR accessorRefcon: LongInt;
                              isSysHandler: Boolean): OSErr;
```

desiredClass
: The object class of the Apple event objects located by the requested object accessor function. This parameter can also contain the constant `typeWildCard` or the constant `cProperty`.

containerType
: The descriptor type of the token that identifies the container for the objects located by the requested object accessor function. This parameter can also contain the constant `typeWildCard`.

theAccessor
: The `AEGetObjectAccessor` function returns a pointer to the requested object accessor function in this parameter.

accessorRefcon
: The `AEGetObjectAccessor` function returns the reference constant from the object accessor dispatch table entry for the specified object accessor function in this parameter.

isSysHandler
: A value that specifies the object accessor table from which to get the object accessor function and its reference constant. If the value of `isSysHandler` is `TRUE`, `AEGetObjectAccessor` gets the function from the system object accessor dispatch table. If the value of `isSysHandler` is `FALSE`, `AEGetObjectAccessor` gets the function from the application's object accessor dispatch table.

DESCRIPTION

The `AEGetObjectAccessor` function returns a pointer to the object accessor function installed for the object class specified in the `desiredClass` parameter and the descriptor type specified in the `containerType` parameter. It also returns the reference constant associated with the specified function. You must supply a value in the `isSysHandler` parameter that specifies which object accessor dispatch table you want to get the function from.

Calling `AEGetObjectAccessor` does not remove the object accessor function from an object accessor dispatch table.

To get an object accessor function whose entry in an object accessor dispatch table specifies `typeWildCard` as the object class, you must specify `typeWildCard` as the value of the `desiredClass` parameter. Similarly, to get an object accessor function whose entry in an object accessor dispatch table specifies `typeWildCard` as the descriptor type of the token used to specify the container, you must specify `typeWildCard` as the value of the `containerType` parameter.

To get an object accessor function whose entry in an object accessor dispatch table specifies `cProperty` (a constant used to specify a property of any object class), you must specify `cProperty` as the `desiredClass` parameter.

RESULT CODES

`noErr`	0	No error occurred
`paramErr`	−50	`AEObjectInit` was not called before this function was called
`errAEAccessorNotFound`	−1723	There is no object accessor function for the specified object class and container type

AECallObjectAccessor

You can use the `AECallObjectAccessor` function to invoke one of your application's object accessor functions.

```
FUNCTION AECallObjectAccessor (desiredClass: DescType;
                                containerToken: AEDesc;
                                containerClass: DescType;
                                keyForm: DescType;
                                keyData: AEDesc;
                                VAR theToken: AEDesc): OSErr;
```

`desiredClass`
    The object class of the desired Apple event objects.

containerToken
            The token that identifies the container for the desired objects.

containerClass
            The object class of the container for the desired objects.

keyForm   The key form specified by the object specifier record for the object or
            objects to be located.

keyData   The key data specified by the object specifier record for the object or
            objects to be located.

theToken   The object accessor function that is invoked returns a token specifying the
            desired object or objects in this parameter.

DESCRIPTION

If you want your application to do some of the Apple event object resolution normally
performed by the AEResolve function, you can use AECallObjectAccessor to
invoke an object accessor function. This might be useful, for example, if you
have installed an object accessor function using typeWildCard for the
AEInstallObjectAccessor function's desiredClass parameter and typeAEList
for the containerType parameter. To return a list of tokens for a request like "every
line that ends in a period," the object accessor function can create an empty list, then call
AECallObjectAccessor for each requested element, adding tokens for each element
to the list one at a time.

The parameters of AECallObjectAccessor are identical to the parameters of an object
accessor function, with one exception: the parameter that specifies the reference constant
passed to the object accessor function whenever it is called is added by the Apple Event
Manager when it calls the object accessor function.

To call an object accessor function whose entry in an object accessor dispatch table
specifies typeWildCard as the object class, you must specify typeWildCard as the
value of the desiredClass parameter.

To call an object accessor function whose entry in an object accessor dispatch table
specifies cProperty, you must specify cProperty as the desiredClass parameter.

RESULT CODES

In addition to the following result codes, AECallObjectAccessor returns any other
result codes returned by the object accessor function that is called.

noErr	0	No error occurred
paramErr	–50	AEObjectInit was not called before this function was called
errAEAccessorNotFound	–1723	No object accessor was found

AERemoveObjectAccessor

You can use the `AERemoveObjectAccessor` function to remove an object accessor function from an object accessor dispatch table.

```
FUNCTION AERemoveObjectAccessor (desiredClass: DescType;
                                 containerType: DescType;
                                 theAccessor: AccessorProcPtr;
                                 isSysHandler: Boolean): OSErr;
```

desiredClass
:   The object class of the Apple event objects located by the object accessor function. The `desiredClass` parameter can also contain the constant `typeWildCard` or the constant `cProperty`.

containerType
:   The descriptor type of the token that identifies the container for the objects located by the object accessor function. The `containerType` parameter can also contain the constant `typeWildCard`.

theAccessor
:   A pointer to the object accessor function you want to remove. Although the parameters `desiredClass` and `containerType` would be sufficient to identify the function to be removed, providing the parameter `theAccessor` guarantees that you remove the correct function. If this parameter does not contain a pointer to the object accessor function you want to remove, its value should be `NIL`.

isSysHandler
:   A value that specifies the object accessor dispatch table from which to remove the object accessor function. If the value of `isSysHandler` is `TRUE`, `AEGetObjectAccessor` removes the routine from the system object accessor dispatch table. If the value is `FALSE`, `AEGetObjectAccessor` removes the routine from the application object accessor dispatch table.

DESCRIPTION

The `AERemoveObjectAccessor` function removes the object accessor function you have installed for the object class specified in the `desiredClass` parameter and the descriptor type specified in the `containerType` parameter.

To remove an object accessor function whose entry in an object accessor dispatch table specifies `typeWildCard` as the object class, you must specify `typeWildCard` as the value of the `desiredClass` parameter. Similarly, to remove an object accessor function whose entry in an object accessor dispatch table specifies `typeWildCard` as the descriptor type of the token used to specify the container for the desired objects, you must specify `typeWildCard` as the value of the `containerType` parameter.

To remove an object accessor function whose entry in an object accessor dispatch table specifies `cProperty` (a constant used to specify a property of any object class), you must specify `cProperty` as the `desiredClass` parameter.

RESULT CODES

`noErr`	0	No error occurred
`paramErr`	–50	`AEObjectInit` was not called before this function was called
`errAEAccessorNotFound`	–1723	There is no object accessor function for the specified object class and container type

Resolving Object Specifier Records

If an Apple event parameter consists of an object specifier record, your handler for the event typically calls the `AEResolve` function to begin the process of resolving the object specifier record.

AEResolve

You can use the `AEResolve` function to resolve an object specifier record in an Apple event parameter.

```
FUNCTION AEResolve (objectSpecifier: AEDesc;
                    callbackFlags: Integer;
                    VAR theToken: AEDesc): OSErr;
```

`objectSpecifier`

The object specifier record to be resolved.

`callbackFlags`

A value that determines what additional assistance, if any, your application can give the Apple Event Manager when it parses the object specifier record. The value is specified by adding the following constants, as appropriate:

```
CONST  kAEIDoMinimum  = $0000;    {supports minimum }
                                  { callbacks only}
       kAEIDoWhose    = $0001;    {supports formWhose}
       kAEIDoMarking  = $0004;    {provides marking }
                                  { functions}
```

theToken   The AEResolve function returns, in this parameter, a token that identifies
            the Apple event objects specified by the objectSpecifier parameter.
            Your object accessor functions may need to create many tokens to resolve
            a single object specifier record; this parameter contains only the final
            token that identifies the requested Apple event object. If an error occurs,
            AEResolve returns a null descriptor record.

DESCRIPTION

The AEResolve function resolves the object specifier record passed in the
objectSpecifier parameter with the help of the object accessor functions and object
callback functions provided by your application.

RESULT CODES

noErr	**0**	No error occurred
paramErr	–50	AEObjectInit was not called before this function was called
errAEHandlerNotFound	–1717	The necessary object callback function was not found (this result is returned only for object callback functions; errAEAccessorNotFound [–1723] is returned when an object accessor function is not found)
errAEImpossibleRange	–1720	The range is not valid because it is impossible for a range to include the first and last objects that were specified; an example is a range in which the offset of the first object is greater than the offset of the last object
errAEWrongNumberArgs	–1721	The number of operands provided for the kAENOT logical operator is not 1
errAEAccessorNotFound	–1723	There is no object accessor function for the specified object class and token descriptor type
errAENoSuchLogical	–1725	The logical operator in a logical descriptor record is not kAEAND, kAEOR, or kAENOT
errAEBadTestKey	–1726	The descriptor record in a test key is neither a comparison descriptor record nor a logical descriptor record
errAENotAnObjectSpec	–1727	The objSpecifier parameter of AEResolve is not an object specifier record
errAENegativeCount	–1729	An object-counting function returned a negative result
errAEEmptyListContainer	–1730	The container for an Apple event object is specified by an empty list

In addition to the result codes listed here, AEResolve also returns any result code
returned by one of your application's object accessor functions or object callback
functions. For example, an object accessor function can return errAENoSuchObject
(–1728) when it can't find an Apple event object, or it can return more specific result
codes.

If any object accessor function or object callback function returns a result code other than `noErr` or `errAEEventNotHandled`, `AEResolve` immediately disposes of any existing tokens and returns. The result code it returns in this case is the result code returned by the object accessor function or the object callback function.

SEE ALSO

For an overview of the way `AEResolve` works with object accessor functions, see "Resolving Object Specifier Records," which begins on page 6-4.

Deallocating Memory for Tokens

Whenever the `AEResolve` function returns a final token to your event handler as the result of the resolution of an object specifier record passed to `AEResolve`, your application can call the `AEDisposeToken` function to deallocate the memory used by the token.

AEDisposeToken

You can use the `AEDisposeToken` function to deallocate the memory used by a token.

```
FUNCTION AEDisposeToken (VAR theToken: AEDesc): OSErr;
```

theToken   The token to be disposed of.

DESCRIPTION

When your application calls the `AEDisposeToken` function, the Apple Event Manager first calls your application's token disposal function, if you have provided one. If you haven't provided a token disposal function, or if your application's token disposal function returns `errAEEventNotHandled` as the function result, the Apple Event Manager calls the system token disposal function if one is available. If there is no system token disposal function or the function returns `errAEEventNotHandled` as the function result, the Apple Event Manager calls the `AEDisposeDesc` function to dispose of the token.

RESULT CODES

In addition to the following result codes, `AEDisposeToken` also returns result codes returned by the token disposal function that disposed of the token.

noErr	0	No error occurred
paramErr	–50	`AEObjectInit` was not called before this function was called
notASpecialFunction	–1714	The keyword is not valid for a special function

Creating Object Specifier Records

The Apple Event Manager provides five functions that you can use to create some of the components of an object specifier record or to assemble an object specifier record:

n The `CreateOffsetDescriptor` function creates an offset descriptor record, which specifies the position of an element in relation to the beginning or end of its container.

n The `CreateCompDescriptor` function creates a comparison descriptor record, which specifies how to compare one or more Apple event objects with either another Apple event object or a descriptor record.

n The `CreateLogicalDescriptor` function creates a logical descriptor record, which specifies a logical operator and one or more logical terms for the Apple Event Manager to evaluate.

n The `CreateRangeDescriptor` function creates a range descriptor record, which specifies a series of consecutive elements in the same container.

n The `CreateObjSpecifier` function assembles an object specifier record, which identifies one or more Apple event objects, from other descriptor records.

Instead of using these functions, you can create the corresponding descriptor records yourself using the `AECreateDesc` function, add them to an AE record using other Apple Event Manager routines, and coerce the AE record to a descriptor record of type `typeObjectSpecifier`. However, in most cases it is easier to use the functions listed in this section.

All of these functions except for `CreateOffsetDescriptor` include a `disposeInputs` parameter. If the value of this parameter is `TRUE`, the function automatically disposes of any descriptor records you have provided as parameters to the function. If the value is `FALSE`, the application must dispose of the records itself. A value of `FALSE` may be more efficient for some applications because it allows them to reuse descriptor records.

For more information about these functions and examples of their use, see "Creating Object Specifier Records," which begins on page 6-55.

CreateOffsetDescriptor

You can use the `CreateOffsetDescriptor` function to create an offset descriptor record.

```
FUNCTION CreateOffsetDescriptor (theOffset: LongInt;
                                 VAR theDescriptor: AEDesc)
                                 : OSErr;
```

theOffset   A positive integer that specifies the offset from the beginning of the
             container (the first element has an offset of 1), or a negative integer that
             specifies the offset from the end (the last element has an offset of –1).

theDescriptor
             The offset descriptor record created by CreateOffsetDescriptor.

DESCRIPTION

The CreateOffsetDescriptor function creates an offset descriptor record that
specifies the position of an element in relation to the beginning or end of its container.

RESULT CODES

noErr   **0**   No error occurred
memFullErr   **–108**   Not enough room in heap zone

CreateCompDescriptor

You can use the CreateCompDescriptor function to create a comparison descriptor
record.

```
FUNCTION CreateCompDescriptor (comparisonOperator: DescType;
                               VAR operand1: AEDesc;
                               VAR operand2: AEDesc;
                               disposeInputs: Boolean;
                               VAR theDescriptor: AEDesc)
                               : OSErr;
```

comparisonOperator
             The comparison operator for comparing the descriptor records in the
             operand1 and operand2 parameters. The operator is specified by the
             constants listed in the description that follows.

operand1   An object specifier record.

operand2   A descriptor record (which can be an object specifier record or any other
             descriptor record) whose value is to be compared to the value of
             operand1.

disposeInputs
             A Boolean value indicating whether the function (TRUE) or your
             application (FALSE) should dispose of the descriptor records for the two
             operands.

theDescriptor
             The comparison descriptor record created by CreateCompDescriptor.

DESCRIPTION

The `CreateCompDescriptor` function creates a comparison descriptor record, which specifies how to compare one or more Apple event objects with either another Apple event object or a descriptor record.

The actual comparison of the two operands is performed by the object-comparison function provided by the client application. The way a comparison operator is interpreted is up to each application.

These are the currently defined standard comparison operators:

Constant	Meaning
kAEGreaterThan	The value of operand1 is greater than the value of operand2.
kAEGreaterThanEquals	The value of operand1 is greater than or equal to the value of operand2.
kAEEquals	The value of operand1 is equal to the value of operand2.
kAELessThan	The value of operand1 is less than the value of operand2.
kAELessThanEquals	The value of operand1 is less than or equal to the value of operand2.
kAEBeginsWith	The value of operand1 begins with the value of operand2 (for example, the string "operand" begins with the string "opera").
kAEEndsWith	The value of operand1 ends with the value of operand2 (for example, the string "operand" ends with the string "and").
kAEContains	The value of operand1 contains the value of operand2 (for example, the string "operand" contains the string "era").

RESULT CODES

noErr	0	No error occurred
paramErr	–50	Error in parameter list
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to the requested Apple event data type
errAEWrongDataType	–1703	Wrong Apple event data type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed

SEE ALSO

For an example of how to use the `CreateCompDescriptor` function to create a comparison descriptor record, see "Specifying a Test," which begins on page 6-64.

CreateLogicalDescriptor

You can use the `CreateLogicalDescriptor` function to create a logical descriptor record.

```
FUNCTION CreateLogicalDescriptor
                          (VAR theLogicalTerms: AEDescList;
                           theLogicOperator: DescType;
                           disposeInputs: Boolean;
                           VAR theDescriptor: AEDesc): OSErr;
```

theLogicalTerms
          A list containing comparison descriptor records, logical descriptor records, or both. If the value of the parameter `theLogicOperator` is `kAEAND` or `kAEOR`, the list can contain any number of descriptors. If the value of the parameter `theLogicOperator` is `kAENOT`, logically this list should contain a single descriptor record. However, the function will not return an error if the list contains more than one descriptor record for a logical operator of `kAENOT`.

theLogicOperator
          A logical operator represented by one of the following constants:

```
CONST   kAEAND = 'AND ';
        kAEOR  = 'OR  ';
        kAENOT = 'NOT ';
```

disposeInputs
          A Boolean value indicating whether the function (`TRUE`) or your application (`FALSE`) should dispose of the descriptor records in the other parameters.

theDescriptor
          The logical descriptor record created by `CreateLogicalDescriptor`.

DESCRIPTION

The `CreateLogicalDescriptor` function creates a logical descriptor record, which specifies a logical operator and one or more logical terms for the Apple Event Manager to evaluate.

RESULT CODES

noErr	0	No error occurred
paramErr	–50	Error in parameter list
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to requested Apple event data type
errAEWrongDataType	–1703	Wrong Apple event data type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed

SEE ALSO

For an example of how to use the `CreateLogicalDescriptor` function to create a logical descriptor record, see "Specifying a Test," which begins on page 6-64.

CreateRangeDescriptor

You can use the `CreateRangeDescriptor` function to create a range descriptor record.

```
FUNCTION CreateRangeDescriptor (VAR rangeStart: AEDesc;
                                VAR rangeStop: AEDesc;
                                disposeInputs: Boolean;
                                VAR theDescriptor: AEDesc): OSErr;
```

rangeStart
: An object specifier record that identifies the first Apple event object in the range.

rangeStop
: An object specifier record that identifies the last Apple event object in the range.

disposeInputs
: A Boolean value indicating whether the function (`TRUE`) or your application (`FALSE`) should dispose of the descriptor records for the `rangeStart` and `rangeStop` parameters.

theDescriptor
: The range descriptor record created by `CreateRangeDescriptor`.

DESCRIPTION

The `CreateRangeDescriptor` function creates a range descriptor record, which specifies a series of consecutive elements in the same container. Although the `rangeStart` and `rangeStop` parameters can be any object specifier records—including object specifier records that specify more than one Apple event object—most applications expect these parameters to specify single Apple event objects.

RESULT CODES

noErr	0	No error occurred
paramErr	–50	Error in parameter list
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to the requested Apple event data type
errAEWrongDataType	–1703	Wrong Apple event data type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed

SEE ALSO

For an example of how to use the `CreateRangeDescriptor` function to create a range descriptor record, see "Specifying a Range" on page 6-72.

CreateObjSpecifier

You can use the `CreateObjSpecifier` function to create an object specifier record.

```
FUNCTION CreateObjSpecifier (desiredClass: DescType;
                             VAR theContainer: AEDesc;
                             keyForm: DescType;
                             VAR keyData: AEDesc;
                             disposeInputs: Boolean;
                             VAR objSpecifier: AEDesc): OSErr;
```

desiredClass
: The object class of the desired Apple event objects.

theContainer
: A description of the container for the requested object, usually in the form of another object specifier record.

keyForm
: The key form for the object specifier record.

keyData
: The key data for the object specifier record.

disposeInputs
: A Boolean value indicating whether the function (`TRUE`) or your application (`FALSE`) should dispose of the descriptor records for the other parameters.

objSpecifier
: The object specifier record created by the `CreateObjSpecifier` function.

DESCRIPTION

The `CreateObjSpecifier` function assembles an object specifier record from the specified constants and other descriptor records.

RESULT CODES

noErr	0	No error occurred
paramErr	–50	Error in parameter list
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to the requested Apple event data type
errAEWrongDataType	–1703	Wrong Apple event data type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed

SEE ALSO

For information about how to assemble the components of an object specifier record with the `CreateObjSpecifier` function, see "Creating Object Specifier Records," which begins on page 6-55.

Application-Defined Routines

The `AEResolve` function performs tasks that are required to resolve any object specifier record, such as parsing its contents, keeping track of the results of tests, and handling memory management. When necessary, `AEResolve` calls application-defined functions to perform tasks that are unique to the application, such as locating a specific Apple event object in the application's data structures or counting the number of Apple event objects in a container.

`AEResolve` can call two kinds of application-defined functions:

n *Object accessor functions* locate Apple event objects. Every application that supports simple object specifier records must provide one or more object accessor functions.

n *Object callback functions* perform other tasks that only an application can perform, such as counting, comparing, or marking Apple event objects. You can provide up to seven object callback functions, depending on the needs of your application.

This section provides model declarations for the object accessor functions and object callback functions that your application can provide.

Object Accessor Functions

You must provide one or more object accessor functions that can locate all the element classes and properties listed in the *Apple Event Registry: Standard Suites* for the object classes supported by your application. This section provides the routine declaration for an object accessor function.

MyObjectAccessor

Object accessor functions locate Apple event objects of a specified object class in a container identified by a token of a specified descriptor type.

```
FUNCTION MyObjectAccessor (desiredClass: DescType;
                           containerToken: AEDesc;
                           containerClass: DescType;
                           keyForm: DescType; keyData: AEDesc;
                           VAR theToken: AEDesc;
                           theRefcon: LongInt): OSErr;
```

desiredClass
                The object class of the desired Apple event objects.

containerToken
                A token that specifies the container of the desired Apple event objects.

containerClass
                The object class of the container.

keyForm   The key form specified by the object specifier record being resolved.

keyData   The key data specified by the object specifier record being resolved.

theToken   The token returned by the `MyObjectAccessor` function.

theRefcon   A reference constant that the Apple Event Manager passes to the object accessor function each time it is called.

DESCRIPTION

Each object accessor function provided by your application should either find elements of a specified object class or find properties of an Apple event object. The `AEResolve` function uses the object class ID of the specified Apple event object and the descriptor type of the token that identifies the object's container to determine which object accessor function to call. To install an object accessor function either in your application's object accessor dispatch table or in the system object accessor dispatch table, use the `AEInstallObjectAccessor` function, which is described on page 6-78.

SPECIAL CONSIDERATIONS

If the Apple Event Manager receives the result code `errAEEventNotHandled` after calling an object accessor function, it attempts to use other methods of locating the requested objects, such as calling an equivalent system object accessor function. Thus, an object accessor function that can't locate a requested object should return `errAEEventNotHandled`. This allows the Apple Event Manager to try other object accessor functions that may be available.

RESULT CODES

noErr	**0**	No error occurred
errAEEventNotHandled	**−1708**	The object accessor function is unable to locate the requested Apple event object or objects

SEE ALSO

For information about installing object accessor functions, see "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

For information about writing object accessor functions, see "Writing Object Accessor Functions," which begins on page 6-28.

Object Callback Functions

If an Apple event parameter consists of an object specifier record, your handler for the Apple event typically calls AEResolve to begin the process of locating the requested Apple event objects. The AEResolve function in turn calls object accessor functions and, if necessary, object callback functions provided by your application when it needs the information they can provide.

This section provides declarations for the seven object callback functions that your application can provide: the object-counting function (MyCountObjects), object-comparison function (MyCompareObjects), token disposal function (MyDisposeToken), error callback function (MyGetErrorDesc), mark token function (MyGetMarkToken), object-marking function (MyMark), and mark-adjusting function (MyAdjustMarks).

For information about writing and installing object callback functions, see "Writing Object Callback Functions," which begins on page 6-45.

MyCountObjects

If you want the Apple Event Manager to help your application resolve object specifier records of key form formTest (and if your application doesn't specify kAEIDoWhose as described on page 6-48), you should provide an object-counting function and an object-comparison function. An object-counting function counts the number of Apple event objects of a specified class in a specified container.

```
FUNCTION MyCountObjects (desiredClass: DescType;
                          containerClass: DescType;
                          theContainer: AEDesc;
                          VAR result: LongInt): OSErr;
```

desiredClass
            The object class of the Apple event objects to be counted.

containerClass
>   The object class of the container for the Apple event objects to be counted.

theContainer
>   A token that identifies the container for the Apple event objects to be counted.

result   Your object-counting function should return in this parameter the number of Apple objects of the specified class in the specified container.

DESCRIPTION

The Apple Event Manager calls your object-counting function when, in the course of resolving an object specifier record, the manager requires a count of the number of Apple event objects of a given class in a given container.

SPECIAL CONSIDERATIONS

If the Apple Event Manager receives the result code `errAEEventNotHandled` after calling an object-counting function, it attempts to use other methods of counting the specified objects, such as calling an equivalent system object-counting function. Thus, an object-counting function that can't count the specified objects should return `errAEEventNotHandled`. This allows the Apple Event Manager to try other object-counting functions that may be available.

RESULT CODES

noErr	0	No error occurred
errAEEventNotHandled	−1708	The object-counting function is unable to count the specified Apple event objects

SEE ALSO

For more information, see "Writing an Object-Counting Function" on page 6-48.

MyCompareObjects

If you want the Apple Event Manager to help your application resolve object specifier records of key form `formTest` (and if your application doesn't specify `kAEIDoWhose` as described on page 6-48), you should provide an object-counting function and an object-comparison function. After comparing one Apple event object to another or to the data for a descriptor record, an object-comparison function should return `TRUE` or `FALSE` in the `result` parameter.

```
FUNCTION MyCompareObjects (comparisonOperator: DescType;
                           object: AEDesc;
                           objectOrDescToCompare: AEDesc;
                           VAR result: Boolean): OSErr;
```

comparisonOperator
: The comparison operator. See the description of `CreateCompDescriptor` on page 6-89 for standard comparison operators at the time of publication of this book. The current version of the *Apple Event Registry: Standard Suites* lists all the constants for comparison operators.

object
: A token.

objectOrDescToCompare
: A token or some other descriptor record that specifies either an Apple event object or a value to compare to the Apple event object specified by the `object` parameter.

result
: Your object-comparison function should return, in this parameter, a Boolean value that indicates whether the values of the `object` and `objectOrDescToCompare` parameters have the relationship specified by the `comparisonOperator` parameter (TRUE) or not (FALSE).

DESCRIPTION

The Apple Event Manager calls your object-comparison function when, in the course of resolving an object specifier record, the manager needs to compare an Apple event object with another or with a value.

It is up to your application to interpret the comparison operators it receives. The meaning of comparison operators differs according to the Apple event objects being compared, and not all comparison operators apply to all object classes.

SPECIAL CONSIDERATIONS

If the Apple Event Manager receives the result code `errAEEventNotHandled` after calling an object-comparison function, it attempts to use other methods of comparison, such as calling an equivalent system object-comparison function. Thus, an object-comparison function that can't perform a requested comparison should return `errAEEventNotHandled`. This allows the Apple Event Manager to try other object-comparison functions that may be available.

RESULT CODES

noErr	0	No error occurred
errAEEventNotHandled	–1708	The object-comparison function is unable to compare the specified Apple event objects

SEE ALSO

For more information, see "Writing an Object-Comparison Function" on page 6-50.

MyDisposeToken

If your application requires more than a call to the `AEDisposeDesc` function to dispose of a token, or if it supports marking callback functions, you must provide one token disposal function. A token disposal function disposes of a specified token.

```
FUNCTION MyDisposeToken (VAR unneededToken: AEDesc): OSErr;
```

`unneededToken`
    The token to dispose of.

DESCRIPTION

The Apple Event Manager calls your token disposal function whenever it needs to dispose of a token. It also calls your disposal function when your application calls the `AEDisposeToken` function. If your application does not provide a token disposal function, the Apple Event Manager calls `AEDisposeDesc` instead.

Your token disposal function must be able to dispose of all of the token types used by your application.

If your application supports marking, a call to `MyDisposeToken` to dispose of a mark token lets your application know that it can unmark the objects marked with that mark token.

SPECIAL CONSIDERATIONS

If the Apple Event Manager receives the result code `errAEEventNotHandled` after calling a token disposal function, it attempts to dispose of the token by some other method, such as calling an equivalent system token disposal function if one is available or, if that fails, by calling `AEDisposeDesc`. Thus, a token disposal function that can't dispose of a token should return `errAEEventNotHandled`. This allows the Apple Event Manager to try other token disposal functions that may be available.

RESULT CODES

noErr	0	No error occurred
errAEEventNotHandled	–1708	The token disposal function is unable to dispose of the token

MyGetErrorDesc

If you want to find out which descriptor record is responsible for an error that occurs during a call to the AEResolve function, you can provide an error callback function. An error callback function returns a pointer to an address. The Apple Event Manager uses this address to store the descriptor record it is currently working with if an error occurs during a call to AEResolve.

```
FUNCTION MyGetErrorDesc (VAR errDescPtr: DescPtr): OSErr;
```

errDescPtr
          A pointer to an address.

DESCRIPTION

Your error callback function simply returns an address. Shortly after your application calls AEResolve, the Apple Event Manager calls your error callback function and writes a null descriptor record to the address returned, overwriting whatever was there previously. If an error occurs during the resolution of the object specifier record, the Apple Event Manager calls your error callback function again and writes the descriptor record—often an object specifier record—to the address returned. If AEResolve returns an error during the resolution of an object specifier record, this address contains the descriptor record responsible for the error.

Normally you should maintain a single global variable of type AEDesc whose address your error callback function returns no matter how many times it is called. Be careful if you use any other method. When recovering from an error, the Apple Event Manager never writes to the address you provide unless it already contains a null descriptor record. Thus, if you don't maintain a single global variable as just described, you should write null descriptor records to any addresses passed by your error callback function that are different from the addresses returned the first time your function is called after a given call to AEResolve.

If the result code returned by the MyGetErrorDesc function has a nonzero value, the Apple Event Manager continues to resolve the object specifier record as if it had never called the error callback function.

RESULT CODE

noErr   **0**   No error occurred

MyGetMarkToken

If your application supports marking, you must provide one mark token function. A mark token function returns a mark token.

```
FUNCTION MyGetMarkToken (containerToken: AEDesc;
                         containerClass: DescType;
                         VAR result: AEDesc): OSErr;
```

containerToken
: The Apple event object that contains the elements to be marked with the mark token.

containerClass
: The object class of the container that contains the objects to be marked.

result
: Your mark token function should return a mark token in this parameter.

DESCRIPTION

To get a mark token, the Apple Event Manager calls your mark token function. Like other tokens, the mark token returned can be a descriptor record of any type; however, unlike other tokens, a mark token identifies the way your application will mark Apple event objects during the current session while resolving a single object specifier record that specifies the key form `formTest`.

A mark token is valid until the Apple Event Manager either disposes of it (by calling `AEDisposeToken`) or returns it as the result of the `AEResolve` function. If the final result of a call to `AEResolve` is a mark token, the Apple event objects currently marked for that mark token are those specified by the object specifier record passed to `AEResolve`, and your application can proceed to do whatever the Apple event has requested. Note that your application is responsible for disposing of a final mark token with a call to `AEDisposeToken`, just as for any other final token.

If your application supports marking, it should also provide a token disposal function modeled after the token disposal function described on page 6-99. When the Apple Event Manager calls `AEDisposeToken` to dispose of a mark token that is not the final result of a call to `AEResolve`, the subsequent call to your token disposal function lets you know that you can unmark the Apple event objects marked with that mark token. A call to `AEDisposeDesc` to dispose of a mark token (which would occur if you did not provide a token disposal function) would go unnoticed.

RESULT CODES

noErr	0	No error occurred
errAEEventNotHandled	–1708	The mark token function is unable to return a mark token; if the Apple Event Manager gets this result, it attempts to get a mark token by calling the equivalent system marking callback function

MyMark

If your application supports marking, you must provide one object-marking function. An object-marking function marks a specific Apple event object.

```
FUNCTION MyMark (theToken: AEDesc; markToken: AEDesc;
                markCount: LongInt): OSErr;
```

theToken   The token for the Apple event object to be marked.

markToken   The mark token used to mark the Apple event object.

markCount   The number of times MyMark has been called for the current mark token (that is, the number of Apple event objects that have so far passed the test, including the element to be marked).

DESCRIPTION

To mark an Apple event object using the current mark token, the Apple Event Manager calls the object-marking function provided by your application. In addition to marking the specified object, your MyMark function should record the mark count for each object that it marks. The mark count recorded for each marked object allows your application to determine which of a set of marked tokens pass a test, as described in the next section for the MyAdjustMarks function.

RESULT CODES

noErr	0	No error occurred
errAEEventNotHandled	–1708	The MyMark function is unable to mark the specified Apple event object; if the Apple Event Manager gets this result, it attempts to mark the object by calling the equivalent system object-marking function

MyAdjustMarks

If your application supports marking, you must provide one mark-adjusting function. A mark-adjusting function adjusts the marks made with the current mark token.

```
FUNCTION MyAdjustMarks (newStart, newStop: LongInt;
                        markToken: AEDesc): OSErr;
```

newStart   The mark count value (provided when the `MyMark` callback routine was called to mark the object) for the first object in the new set of marked objects.

newStop   The mark count value (provided when the `MyMark` callback routine was called to mark the object) for the last object in the new set of marked objects.

markToken   The mark token for the marked objects.

DESCRIPTION

When the Apple Event Manager needs to identify either a range of elements or the absolute position of an element in a group of Apple event objects that pass a test, it can use your application's mark-adjusting function to unmark objects previously marked by a call to your marking function. For example, suppose an object specifier record specifies "any row in the table 'MyCustomers' for which the City column is 'San Francisco.'" The Apple Event Manager first uses the appropriate object accessor function to locate all the rows in the table for which the City column is "San Francisco" and calls the application's marking function repeatedly to mark them. It then generates a random number between 1 and the number of rows it found that passed the test and calls the application's mark-adjusting function to unmark all the rows whose mark count does not match the randomly generated number. If the randomly chosen row has a mark count value of 5, the Apple Event Manager passes the value 5 to the mark-adjusting function in both the `newStart` parameter and the `newStop` parameter, and passes the current mark token in the `markToken` parameter.

When the Apple Event Manager calls your `MyAdjustMarks` function, your application must dispose of any data structures that it created to mark the previously marked objects.

RESULT CODES

noErr	0	No error occurred
errAEEventNotHandled	–1708	The `MyAdjustMarks` function is unable to adjust the marks as requested; if the Apple Event Manager gets this result, it attempts to adjust the marks by calling the equivalent system mark-adjusting function

SEE ALSO

For more information, see "Writing Marking Callback Functions" on page 6-53.

Summary of Resolving and Creating Object Specifier Records

Pascal Summary

Constants

```
CONST
   gestaltAppleEventsAttr     = 'evnt';    {selector for Apple events}
   gestaltAppleEventsPresent  = 0;         {if this bit is set, Apple }
                                           { Event Manager is available}


   {logical operators for descriptor records with keyword }
   { keyAELogicalOperator}
   kAEAND                 = 'AND ';
   kAEOR                  = 'OR  ';
   kAENOT                 = 'NOT ';


   {absolute ordinals used as key data in an object specifier }
   { record with key form formAbsolutePosition}
   kAEFirst               = 'firs';
   kAELast                = 'last';
   kAEMiddle              = 'midd';
   kAEAny                 = 'any ';
   kAEAll                 = 'all ';


   {relative ordinals used as key data in an object specifier record }
   { with key form formRelativePosition}
   kAENext                = 'next';
   kAEPrevious            = 'prev';


   {keywords for object specifier records}
   keyAEDesiredClass      = 'want'; {object class ID}
   keyAEContainer         = 'from'; {description of container}
   keyAEKeyForm           = 'form'; {key form}
   keyAEKeyData           = 'seld'; {key data for specified key form}


   {keywords for range descriptor records}
   keyAERangeStart        = 'star';        {beginning of range}
```

```
keyAERangeStop          =  'stop';       {end of range}


{values for the keyAEKeyForm field of an object specifier record}
formAbsolutePosition    =  'indx';       {for example, 1 = first }
                                         { element in container, -2 = }
                                         { second from end of container}
formRelativePosition    =  'rele';       {key data specifies element }
                                         { before or after container}
formTest                =  'test';       {key data specifies a test}
formRange               =  'rang';       {key data specifies a range}
formPropertyID          =  'prop';       {key data is property ID}
formName                =  'name';       {key data is element's name}


{descriptor types used to identify Apple event objects}
typeObjectSpecifier     =  'obj ';       {object specifier record, often }
                                         { used as keyAEContainer}
typeObjectBeingExamined =  'exmn';       {used as keyAEContainer}
typeCurrentContainer    =  'ccnt';       {used as keyAEContainer}
typeToken               =  'toke';       {substituted for 'ccnt' }
                                         { before accessor called}
typeAbsoluteOrdinal     =  'abso';       {formAbsolutePosition}
typeRangeDescriptor     =  'rang';       {formRange}
typeLogicalDescriptor   =  'logi';       {formTest}
typeCompDescriptor      =  'cmpd';       {formTest}


{various relevant keywords}
keyAECompOperator       =  'relo';       {operator for comparison: }
                                         { '=', '<=', etc.}
keyAELogicalTerms       =  'term';       {an AEList of terms to be }
                                         { related by 'logc' below}
keyAELogicalOperator    =  'logc';       {kAEAND, kAEOR, or kAENOT}
keyAEObject1            =  'obj1';       {first of two objects being }
                                         { compared; must be object }
                                         { specifier record}
keyAEObject2            =  'obj2';       {the other object; may be }
                                         { simple descriptor record }
                                         { or object specifier record}


{special handler selectors used with AESetObjectCallbacks}
keyDisposeTokenProc     =  'xtok';
keyAECompareProc        =  'cmpr';
keyAECountProc          =  'cont';
keyAEMarkTokenProc      =  'mkid';
```

```
keyAEMarkProc            =  'mark';
keyAEAdjustMarksProc     =  'adjm';
keyAEGetErrDescProc      =  'indc';


{additive values for callbackFlags parameter to AEResolve}
kAEIDoMinimum            =  $0000;   {server does not support whose }
                                     { descriptor records or marking}
kAEIDoWhose              =  $0001;   {server supports whose }
                                     { descriptor records}
kAEIDoMarking            =  $0004;   {server supports marking}


{constants for whose descriptor records}
typeWhoseDescriptor      =  'whos';  {whose descriptor record}
formWhose                =  'whos';  {key form for key data of descriptor }
                                     { type typeWhoseDescriptor}
typeWhoseRange           =  'wrng';  {whose range descriptor record}
keyAEWhoseRangeStart     =  'wstr';  {beginning of range}
keyAEWhoseRangeStop      =  'wstp';  {end of range}
keyAEIndex               =  'kidx';  {index for whose descriptor record}
keyAETest                =  'ktst';  {test for whose descriptor record}
```

Data Types

```
TYPE
  ccntTokenRecord =                      {used for rewriting tokens in }
  RECORD                                 { place of 'ccnt' descriptor }
    tokenClass:   DescType;              { records; only of interest to }
    token:        AEDesc;                { those who, when they get ranges }
  END;                                   { as key data in their object }
                                         { accessor functions, resolve }
  ccntTokenRecPtr = ^ccntTokenRecord;    { the object specifier records }
  ccntTokenRecHandle = ^ccntTokenRecPtr; { for the end points manually}

  DescPtr = ^AEDesc;
  DescHandle = ^DescPtr;

  AccessorProcPtr = ProcPtr;
```

Routines for Resolving and Creating Object Specifier Records

Initializing the Object Support Library

```
FUNCTION AEObjectInit : OSErr;
```

Setting Object Accessor Functions and Object Callback Functions

```
FUNCTION AEInstallObjectAccessor
                            (desiredClass: DescType;
                             containerType: DescType;
                             theAccessor: AccessorProcPtr;
                             accessorRefcon: LongInt;
                             isSysHandler: Boolean): OSErr;
FUNCTION AESetObjectCallbacks
                            (myCompareProc, myCountProc,
                             myDisposeTokenProc,
                             myGetMarkTokenProc, myMarkProc,
                             myAdjustMarksProc, myGetErrDescProc:
                             ProcPtr): OSErr;
```

Getting, Calling, and Removing Object Accessor Functions

```
FUNCTION AEGetObjectAccessor
                            (desiredClass: DescType;
                             containerType: DescType;
                             VAR theAccessor: AccessorProcPtr;
                             VAR accessorRefcon: LongInt;
                             isSysHandler: Boolean): OSErr;
FUNCTION AECallObjectAccessor
                            (desiredClass: DescType;
                             containerToken: AEDesc;
                             containerClass: DescType;
                             keyForm: DescType;
                             keyData: AEDesc;
                             VAR theToken: AEDesc): OSErr;
FUNCTION AERemoveObjectAccessor
                            (desiredClass: DescType;
                             containerType: DescType;
                             theAccessor: AccessorProcPtr;
                             isSysHandler: Boolean): OSErr;
```

Resolving Object Specifier Records

```
FUNCTION AEResolve          (objectSpecifier: AEDesc;
                             callbackFlags: Integer;
                             VAR theToken: AEDesc): OSErr;
```

Deallocating Memory for Tokens

```
FUNCTION AEDisposeToken     (VAR theToken: AEDesc): OSErr;
```

Creating Object Specifier Records

```
FUNCTION CreateOffsetDescriptor
                        (theOffset: LongInt;
                         VAR theDescriptor: AEDesc):
                         OSErr;
FUNCTION CreateCompDescriptor
                        (comparisonOperator: DescType;
                         VAR operand1: AEDesc;
                         VAR operand2: AEDesc;
                         disposeInputs: Boolean;
                         VAR theDescriptor: AEDesc): OSErr;
FUNCTION CreateLogicalDescriptor
                        (VAR theLogicalTerms: AEDescList;
                         theLogicOperator: DescType;
                         disposeInputs: Boolean;
                         VAR theDescriptor: AEDesc): OSErr;
FUNCTION CreateRangeDescriptor
                        (VAR rangeStart: AEDesc;
                         VAR rangeStop: AEDesc;
                         disposeInputs: Boolean;
                         VAR theDescriptor: AEDesc): OSErr;
FUNCTION CreateObjSpecifier (desiredClass: DescType;
                         VAR theContainer: AEDesc;
                         keyForm: DescType;
                         VAR keyData: AEDesc;
                         disposeInputs: Boolean;
                         VAR objSpecifier: AEDesc): OSErr;
```

Application-Defined Routines

Object Accessor Functions

```
FUNCTION MyObjectAccessor   (desiredClass: DescType;
                             containerToken: AEDesc;
                             containerClass: DescType;
                             keyForm: DescType; keyData: AEDesc;
                             VAR theToken: AEDesc;
                             theRefcon: LongInt): OSErr;
```

Object Callback Functions

```
FUNCTION MyCountObjects     (desiredClass: DescType;
                             containerClass: DescType;
                             theContainer: AEDesc;
                             VAR result: LongInt): OSErr;
```

```
FUNCTION MyCompareObjects    (comparisonOperator: DescType;
                              theobject: AEDesc;
                              objectOrDescToCompare: AEDesc;
                              VAR result: Boolean): OSErr;
FUNCTION MyDisposeToken      (VAR unneededToken: AEDesc): OSErr;
FUNCTION MyGetErrorDesc      (VAR errDescPtr: DescPtr): OSErr;
FUNCTION MyGetMarkToken      (containerToken: AEDesc;
                              containerClass: DescType;
                              VAR result: AEDesc): OSErr;
FUNCTION MyMark              (theToken: AEDesc; markToken: AEDesc;
                              markCount: LongInt): OSErr;
FUNCTION MyAdjustMarks       (newStart, newStop: LongInt;
                              markToken: AEDesc): OSErr;
```

C Summary

Constants

```
enum {
      #define gestaltAppleEventsAttr    'evnt' /*selector for Apple events*/
      gestaltAppleEventsPresent        = 0   /*if this bit is set, then */
                                             /* Apple Event Manager is */
};                                           /* available*/

/*logical operators for descriptor records with keyword */
/* keyAELogicalOperator*/
#define kAEAND                   'AND '
#define kAEOR                    'OR  '
#define kAENOT                   'NOT '

/*absolute ordinals used as key data in an object specifier */
/* record with key form formAbsolutePosition*/
#define kAEFirst                 'firs'
#define kAELast                  'last'
#define kAEMiddle                'midd'
#define kAEAny                   'any '
#define kAEAll                   'all '

/*relative ordinals used as key data in an object specifier record */
/* with key form formRelativePosition*/
#define kAENext                  'next'
```

```
#define kAEPrevious               'prev'

/*keywords for object specifier records*/
#define keyAEDesiredClass         'want'   /*object class ID*/
#define keyAEContainer            'from'   /*description of container*/
#define keyAEKeyForm              'form'   /*key form*/
#define keyAEKeyData              'seld'   /*key data for specified key */
                                           /* form*/


/*keywords for range descriptor records*/
#define keyAERangeStart           'star'   /*beginning of range*/
#define keyAERangeStop            'stop'   /*end of range*/

/*values for the keyAEKeyForm field of an object specifier record*/
#define formAbsolutePosition      'indx'   /*for example, 1 = first */
                                           /* element in container, -2 = */
                                           /* second from end of */
                                           /* container*/
#define formRelativePosition      'rele'   /*key data specifies element */
                                           /* before or after container*/
#define formTest                  'test'   /*key data specifies a test*/
#define formRange                 'rang'   /*key data specifies a range*/
#define formPropertyID            'prop'   /*key data is property ID*/
#define formName                  'name'   /*key data is element's name*/

/* descriptor types used to identify Apple event objects*/
#define typeObjectSpecifier       'obj '   /*object specifier record, */
                                           /* often used as */
                                           /* keyAEContainer*/
#define typeObjectBeingExamined   'exmn'   /*used as keyAEContainer*/
#define typeCurrentContainer      'ccnt'   /*used as keyAEContainer*/
#define typeToken                 'toke'   /*substituted for 'ccnt' */
                                           /* before accessor called*/
#define typeAbsoluteOrdinal       'abso'   /*formAbsolutePosition*/
#define typeRangeDescriptor       'rang'   /*formRange*/
#define typeLogicalDescriptor     'logi'   /*formTest*/
#define typeCompDescriptor        'cmpd'   /*formTest*/
/*various relevant keywords*/
#define keyAECompOperator         'relo'   /*operator for comparison: */
                                           /* '=', '<=', etc.*/
#define keyAELogicalTerms         'term'   /*an AEList of terms to be */
                                           /* related by 'logc' below*/
#define keyAELogicalOperator      'logc'   /*kAEAND, kAEOR, or kAENOT*/
```

```
#define keyAEObject1                    'obj1'  /*first of two objects being */
                                                /* compared; must be object */
                                                /* specifier record*/
#define keyAEObject2                    'obj2'  /*the other object; may be */
                                                /* simple descriptor record */
                                                /* or object specifier record*/
/*special handler selectors used with AESetObjectCallbacks*/
#define keyDisposeTokenProc             'xtok'
#define keyAECompareProc                'cmpr'
#define keyAECountProc                  'cont'
#define keyAEMarkTokenProc              'mkid'
#define keyAEMarkProc                   'mark'
#define keyAEAdjustMarksProc            'adjm'
#define keyAEGetErrDescProc             'indc'


/*additive values for callbackFlags parameter to AEResolve*/
#define kAEIDoMinimum                   0x0000  /*server does not support */
                                                /* whose descriptor records */
                                                /* or marking*/
#define kAEIDoWhose                     0x0001  /*server supports whose */
                                                /* descriptor records*/
#define kAEIDoMarking                   0x0004  /*server supports marking*/


/*constants for whose descriptor records*/
#define typeWhoseDescriptor             'whos'  /*whose descriptor record*/
#define formWhose                       'whos'  /*key form for key data of */
                                                /* descriptor type */
                                                /* typeWhoseDescriptor*/
#define typeWhoseRange                  'wrng'  /*whose range descriptor */
                                                /* record*/
#define keyAEWhoseRangeStart            'wstr'  /*beginning of range*/
#define keyAEWhoseRangeStop             'wstp'  /*end of range*/
#define keyAEIndex                      'kidx'  /*index for whose descriptor */
                                                /* record*/
#define keyAETest                       'ktst'  /*test for whose descriptor */
                                                /* record*/
```

Data Types

```
struct ccntTokenRecord {                        /*used for rewriting tokens */
   DescType tokenClass;                         /* in place of 'ccnt' */
   AEDesc token;                                /* descriptor records; only */
};                                              /* of interest to those who, */
```

```
                                                /* when they get ranges as */
typedef struct ccntTokenRecord ccntTokenRecord, /* key data in their object */
*ccntTokenRecPtr, **ccntTokenRecHandle;          /* accessor functions, */
                                                /* resolve them manually*/


typedef AEDesc *DescPtr, **DescHandle;


/*typedefs providing type checking for procedure pointers*/
typedef pascal OSErr (*accessorProcPtr) (DescType desiredClass,
                                         const AEDesc *container,
                                         DescType containerClass,
                                         DescType form,
                                         const AEDesc *selectionData,
                                         AEDesc *value, long LongInt);
typedef pascal OSErr (*compareProcPtr)(DescType oper, const AEDesc *obj1,
                                       const AEDesc *obj2,
                                       Boolean *result);
typedef pascal OSErr (*countProcPtr)(DescType desiredClass,
                                     DescType containerClass,
                                     const AEDesc *container,
                                     long *result);
typedef pascal OSErr (*disposeTokenProcPtr)(AEDesc *unneededToken);
typedef pascal OSErr (*getMarkTokenProcPtr)(const AEDesc *ContainerToken,
                                            DescType containerClass,
                                            AEDesc *result);


typedef pascal OSErr (*getErrDescProcPtr)(DescPtr *appDescPtr);
```

Routines for Resolving and Creating Object Specifier Records

Initializing the Object Support Library

```
pascal OSErr AEObjectInit   ();
```

Setting Object Accessor Functions and Object Callback Functions

```
pascal OSErr AEInstallObjectAccessor
                            (DescType desiredClass, DescType containerType,
                             accessorProcPtr theAccessor,
                             long accessorRefcon, Boolean isSysHandler);
pascal OSErr AESetObjectCallbacks
                            (compareProcPtr myCompareProc,
                             countProcPtr myCountProc,
                             disposeTokenProcPtr myDisposeTokenProc,
```

```
                                    getMarkTokenProcPtr myGetMarkTokenProc,
                                    markProcPtr myMarkProc,
                                    adjustMarksProcPtr myAdjustMarksProc,
                                    getErrDescProcPtr myGetErrDescProc);
```

Getting, Calling, and Removing Object Accessor Functions

```
pascal OSErr AEGetObjectAccessor
                                (DescType desiredClass, DescType containerType,
                                 accessorProcPtr *theAccessor,
                                 long *accessorRefcon, Boolean isSysHandler);
pascal OSErr AECallObjectAccessor
                                (DescType desiredClass,
                                 const AEDesc *containerToken,
                                 DescType containerClass, DescType keyForm,
                                 const AEDesc *keyData, AEDesc *theToken);
pascal OSErr AERemoveObjectAccessor
                                (DescType desiredClass, DescType containerType,
                                 accessorProcPtr theAccessor,
                                 Boolean isSysHandler);
```

Resolving Object Specifier Records

```
pascal OSErr AEResolve      (const AEDesc *objectSpecifier,
                             short callbackFlags, AEDesc *theToken);
```

Deallocating Memory for Tokens

```
pascal OSErr AEDisposeToken (AEDesc *theToken);
```

Creating Object Specifier Records

```
pascal OSErr CreateOffsetDescriptor
                                (long theOffset, AEDesc *theDescriptor);
pascal OSErr CreateCompDescriptor
                                (DescType comparisonOperator, AEDesc* operand1,
                                 AEDesc* operand2, Boolean disposeInputs,
                                 AEDesc* theDescriptor);
pascal OSErr CreateLogicalDescriptor
                                (AEDescList *theLogicalTerms,
                                 DescType theLogicOperator,
                                 Boolean disposeInputs, AEDesc *theDescriptor);
pascal OSErr CreateRangeDescriptor
                                (AEDesc *rangeStart, AEDesc *rangeStop,
                                 Boolean disposeInputs, AEDesc *theDescriptor);
```

```
pascal OSErr CreateObjSpecifier
                           (DescType desiredClass, AEDesc *theContainer,
                            DescType keyForm, AEDesc *keyData,
                            Boolean disposeInputs, AEDesc *objSpecifier);
```

Application-Defined Routines

Object Accessor Functions

```
pascal OSErr MyObjectAccessor
                           (DescType desiredClass,
                            const AEDesc *containerToken,
                            DescType containerClass,
                            DescType keyForm, const AEDesc *keyData,
                            AEDesc *theToken, long *theRefcon);
```

Object Callback Functions

```
pascal OSErr MyCountObjects (DescType desiredClass, DescType containerClass,
                            const AEDesc *theContainer, long *result);
pascal OSErr MyCompareObjects
                           (DescType comparisonOperator,
                            const AEDesc *theObject,
                            const AEDesc *objectOrDescToCompare,
                            Boolean *result);
pascal OSErr MyDisposeToken (AEDesc *unneededToken);
pascal OSErr MyGetErrorDesc (DescPtr *errDescPtr);
pascal OSErr MyGetMarkToken (const AEDesc *containerToken,
                            DescType containerClass, AEDesc *result);
pascal OSErr MyMark        (const AEDesc *theToken,
                            const AEDesc *markToken, long markCount);
pascal OSErr MyAdjustMarks  (long newStart, long newStop,
                            const AEDesc *markToken);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

_Pack8

Selector	Routine
$023A	AEDisposeToken
$0536	AEResolve
$0738	AERemoveObjectAccessor
$0937	AEInstallObjectAccessor
$0939	AEGetObjectAccessor
$0C3B	AECallObjectAccessor
$0E35	AESetObjectCallbacks

Result Codes

noErr	0	No error
paramErr	−50	Parameter error (for example, value of handler pointer is NIL or odd)
eLenErr	−92	Buffer too big to send
memFullErr	−108	Not enough room in heap zone
userCanceledErr	−128	User canceled an operation
procNotFound	−600	No eligible process with specified process serial number
bufferIsSmall	−607	Buffer is too small
noOutstandingHLE	−608	No outstanding high-level event
connectionInvalid	−609	Nonexistent signature or session ID
noUserInteractionAllowed	−610	Background application sends event requiring authentication
noPortErr	−903	Client hasn't set 'SIZE' resource to indicate awareness of high-level events
destPortErr	−906	Server hasn't set 'SIZE' resource to indicate awareness of high-level events, or else is not present
sessClosedErr	−917	The kAEDontReconnect flag in the sendMode parameter was set, and the server quit and then restarted
errAECoercionFail	−1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	−1701	Descriptor record was not found
errAECorruptData	−1702	Data in an Apple event could not be read
errAEWrongDataType	−1703	Wrong descriptor type
errAENotAEDesc	−1704	Not a valid descriptor record
errAEBadListItem	−1705	Operation involving a list item failed

errAENewerVersion	–1706	Need a newer version of the Apple Event Manager
errAENotAppleEvent	–1707	Event is not an Apple event
errAEEventNotHandled	–1708	Event wasn't handled by an Apple event handler
errAEReplyNotValid	–1709	`AEResetTimer` was passed an invalid reply
errAEUnknownSendMode	–1710	Invalid sending mode was passed
errAEWaitCanceled	–1711	User canceled out of wait loop for reply or receipt
errAETimeout	–1712	Apple event timed out
errAENoUserInteraction	–1713	No user interaction allowed
errAENotASpecialFunction	–1714	The keyword is not valid for a special function
errAEParamMissed	–1715	Handler cannot understand a parameter the client considers required
errAEUnknownAddressType	–1716	Unknown Apple event address type
errAEHandlerNotFound	–1717	No handler found for an Apple event or a coercion, or no object callback function found
errAEReplyNotArrived	–1718	Reply has not yet arrived
errAEIllegalIndex	–1719	Not a valid list index
errAEImpossibleRange	–1720	The range is not valid because it is impossible for a range to include the first and last objects that were specified; an example is a range in which the offset of the first object is greater than the offset of the last object
errAEWrongNumberArgs	–1721	The number of operands provided for the `kAENOT` logical operator is not 1
errAEAccessorNotFound	–1723	There is no object accessor function for the specified object class and token descriptor type
errAENoSuchLogical	–1725	The logical operator in a logical descriptor record is not `kAEAND`, `kAEOR`, or `kAENOT`
errAEBadTestKey	–1726	The descriptor record in a test key is neither a comparison descriptor record nor a logical descriptor record
errAENotAnObjectSpec	–1727	The `objSpecifier` parameter of `AEResolve` is not an object specifier record
errAENoSuchObject	–1728	A run-time resolution error, for example: object specifier record asked for the third element, but there are only 2.
errAENegativeCount	–1729	Object-counting function returned negative value
errAEEmptyListContainer	–1730	The container for an Apple event object is specified by an empty list
errAEUnknownObjectType	–1731	Descriptor type of token returned by `AEResolve` is not known to server application
errAERecordingIsAlreadyOn	–1732	Attempt to turn recording on when it is already on

Introduction to Scripting

Contents

This chapter provides an overview of the tasks involved in making your application scriptable and recordable. This chapter also introduces some of the ways your application can use the Component Manager and scripting components to manipulate and execute scripts. The three chapters that follow provide detailed information, including sample code, about the topics introduced in this chapter.

The chapter "Introduction to Interapplication Communication" in this book describes the Open Scripting Architecture (OSA) and its relationship to the Apple Event Manager and other parts of the IAC architecture. If your application supports the appropriate core and functional-area events defined in the *Apple Event Registry: Standard Suites,* you can make it scriptable (that is, capable of responding to Apple events sent by scripting components) by providing an Apple event terminology extension (`'aete'`) resource. This chapter describes some of the tasks involved in making your application scriptable and introduces the `'aete'` resource. The next chapter, "Apple Event Terminology Resources," describes in detail how to create an `'aete'` resource.

This chapter also introduces Apple event recording and the use of the standard scripting component routines to manipulate and execute scripts. The chapter "Recording Apple Events" describes in detail how to make your application recordable, and the chapter "Scripting Components" describes how to use the standard scripting component routines.

To use this chapter or any of the chapters that follow, you should be familiar with the chapters "Introduction to Apple Events" and "Responding to Apple Events" in this book. If you plan to make your application recordable, you should also read the chapters "Creating and Sending Apple Events" and "Resolving and Creating Object Specifier Records."

The *AppleScript Software Developers' Kit* (available from APDA) provides development tools, sample applications, and information about the AppleScript language that you will find useful when you begin to apply the information in this chapter to your application.

If you are developing a scripting component, you should provide support for the standard scripting component routines described in the chapter "Scripting Components," and you should read the instructions for creating components in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

This chapter begins with an overview of scripts and scripting components. The rest of the chapter describes how the OSA makes it possible to

n   make your application scriptable

n   make your application recordable

n   have your application manipulate and execute scripts

About Scripts and Scripting Components

A **script** is any collection of data that, when executed by the appropriate program, causes a corresponding action or series of actions. The Open Scripting Architecture (OSA) provides a standard mechanism that allows users to control multiple applications with scripts written in a variety of scripting languages. Each scripting language has a corresponding **scripting component.** Each scripting component supports the standard scripting component routines described in the chapter "Scripting Components" in this book.

When a scripting component executes a script, it performs the actions described in the script, including sending Apple events to applications if necessary. Like other components that use the Component Manager, scripting components can provide their own routines in addition to the standard routines that must be supported by all components of the same type.

Scripting components typically implement a text-based scripting language based on Apple events. For example, the **AppleScript component** implements **AppleScript,** the standard user scripting language defined by Apple Computer, Inc. This book uses AppleScript examples to demonstrate how applications can interact with scripting components.

Other scripting components may support the standard scripting component routines in different ways. Scripting components need not implement a text-based scripting language, or even one that is based on Apple events. For example, specialized scripting components can play sounds, execute XCMDs, or perform almost any other action when they execute scripts.

This chapter describes three ways that you can take advantage of the OSA:

n   You can make your application **scriptable,** or capable of responding to Apple events sent to it by a scripting component. An application is scriptable if it

   n   Responds to the appropriate standard Apple events as described in the chapter "Responding to Apple Events" in this book.

   n   Provides an Apple event terminology extension (`'aete'`) resource describing the Apple events that your application supports and the user terminology that corresponds to those events. The `'aete'` resource allows scripting components to interpret scripts correctly and send the appropriate Apple events to your application during script execution.

n   You can make your application **recordable**—that is, capable of sending Apple events
    to itself to report user actions to the Apple Event Manager for recording purposes.
    After a user has turned on recording for a particular scripting component, the
    scripting component receives a copy of every subsequent Apple event that any
    application on the local computer sends to itself. The scripting component records
    such events in the form of a script.

n   You can have your application manipulate and execute scripts with the aid of a
    scripting component. To do so, your application must

    n   Use the Component Manager to open a connection with the appropriate
        component.

    n   Use the standard scripting component routines described in the chapter "Scripting
        Components" to record, edit, compile, save, load, or execute scripts.

Users of scriptable applications can execute scripts to perform tasks that might otherwise
be difficult to accomplish, especially repetitive or conditional tasks that involve multiple
applications. For example, a user can execute an AppleScript script to locate database
records with specific characteristics, create a series of graphs based on those records,
import the graphs into a page-layout document, and send the document to a remote
computer on the network via electronic mail. When a user executes such a script, the
AppleScript component attempts to perform the actions the script describes, including
sending Apple events when necessary.

To respond appropriately to the Apple events sent to it by the AppleScript component,
the database application in this example must be able to locate records with specific
characteristics so that it can identify and return the requested data. These characteristics
are described by an object specifier record that is part of an Apple event supported by
the application. Also, the other applications involved must support Apple events that
manipulate the data in the ways described in the script. Each application in this example
must also provide an `aete` resource describing the Apple events that the application
supports and the user terminology that corresponds to those events, so that the
AppleScript component can interpret the script correctly.

Even with little or no knowledge of a particular scripting language, users of applications
that are recordable as well as scriptable can record simple scripts. More knowledgeable
users may also wish to record their actions as scripts with recordable applications and
then edit or combine scripts as needed.

An application that uses scripting components to manipulate and execute scripts need
not be scriptable; however, if it is scriptable, it can execute scripts that control its own
behavior. In other words, it can perform tasks by means of scripts and allow users to
modify those scripts to suit their own needs.

The next three sections provide an overview of the way scripting components can
interact with applications.

Script Editors and Script Files

A **script editor** is an application that allows users to record, edit, save, and execute scripts. For example, the AppleScript component uses the services of the Script Editor application.

Figure 7-1 shows an AppleScript script displayed in a Script Editor window. The Record, Stop, and Run buttons control a script in much the same way that the equivalent buttons on a cassette recorder control an audio tape. A **script comment** at the top of the window describes what the script does. Users with some knowledge of a text-based scripting language such as AppleScript can use Script Editor to modify recorded scripts or write their own scripts.

Figure 7-1   A script window in the Script Editor application
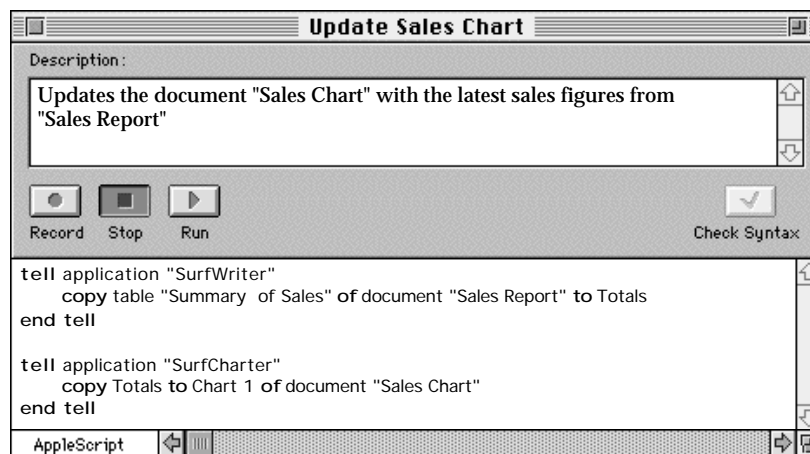
Script Editor provides entry-level scripting capabilities, but it is not intended for intensive script development. Users who wish to write complex scripts may replace Script Editor with more sophisticated editors that provide specialized debugging and development tools.

A script like the one in Figure 7-1 can be stored in a **script file** represented by an icon in the Finder, or it can be stored within an application or one of its documents. Figure 7-2 shows the four icons representing the files in which Script Editor stores scripts.

Figure 7-2   Script file icons in the Finder and corresponding user actions
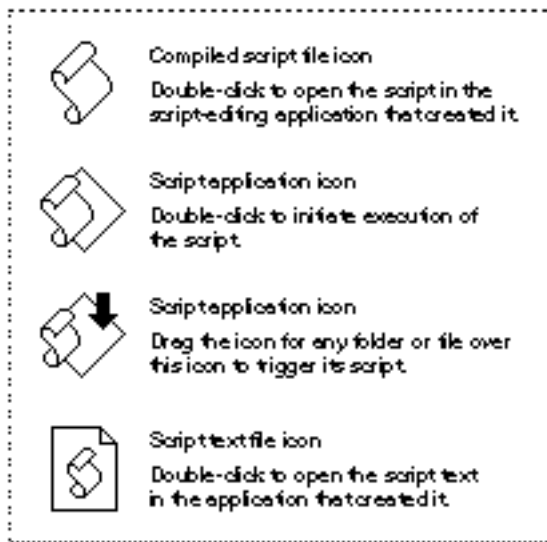
Script Editor and similar script-editing applications allow users to store scripts using three file types:

n A **compiled script file** has the file type `'osas'` and contains the script data as a resource of type `'scpt'`. Before executing the script in a compiled script file, a user must first open the script from the Finder or from an application such as Script Editor. After opening a compiled script in an application that supports script editing, the user can view the script, modify it if necessary, and execute it.

n A **script application** has the file type `'APPL'` and contains the script data as a resource of type `'scpt'`. Its kind is "application." A script application takes one of two forms, each with its own icon:

n A script application with the creator signature `'aplt'`. A user double-clicks the icon to trigger the script.

n A script application with the creator signature `'dplt'`. A user can drag the icon for another file or a folder over this script application's icon to trigger a script that acts on that object.

By default, when a user triggers the script in either kind of script application, a splash screen appears that allows the user either to quit or to run the script. Users can also save a script application in a form that bypasses the splash screen, running the script immediately after the user double-clicks its icon.

n   A **script text file** contains only a plain-text version of uncompiled scripting-language statements. This format is useful primarily as a last resort for saving a script that can't be compiled because of syntax errors or other problems. It is also useful for exchanging unstyled text with other text-based applications. A user must open a script text file in a script editor and successfully compile it before it will execute.

Like sound resources, scripts can be stored within applications and documents as well as in distinct files that can be manipulated from the Finder. Your application can use the standard scripting component routines to manipulate and execute both its own internally stored scripts and scripts stored as separate files whose icons appear in the Finder. For more information about script storage formats, see "Saving Script Data" on page 10-12.

The next two sections describe how scripting components interact with scriptable applications and with applications that execute scripts.

Scripting Components and Scriptable Applications

Scripting components control the behavior of scriptable applications by means of Apple events. For example, when the AppleScript component executes the AppleScript script shown in Figure 7-1, it sends the Apple events shown in Figure 7-3 to trigger the actions described by the script. The client application in this example would most commonly be a script editor but could also be any other application that uses standard scripting component routines to manipulate and execute scripts.

Figure 7-3   How the AppleScript component executes a script
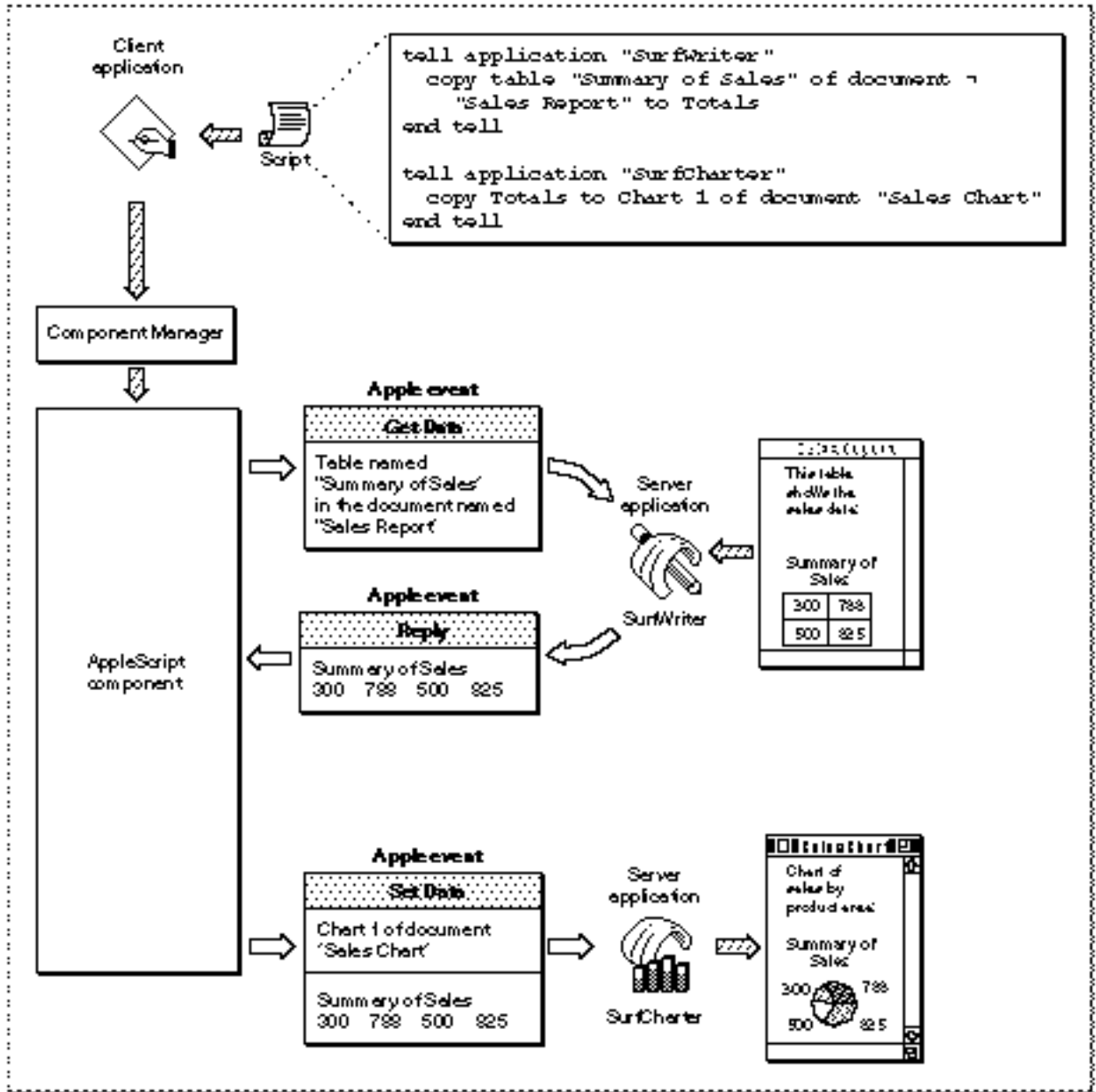
As described in the chapter "Introduction to Apple Events" in this book, a client application is any application that uses Apple events to request a service or information. A client application that executes a script does not send the corresponding Apple events itself; instead, it uses scripting component routines to manipulate and execute the script. The scripting component sends Apple events when necessary to trigger the actions described in the script. Similarly, a scriptable application that responds to the Apple events sent by a scripting component can be considered the server application for those Apple events.

When a scripting component evaluates a script, it attempts to perform all the actions described in the script, including sending Apple events when necessary. In the example shown in Figure 7-3, the AppleScript component first performs the action described in the first `tell` statement:

```
tell application "SurfWriter"
   copy table "Summary of Sales" of document¬
      "Sales Report" to Totals
end tell
```

To perform this action, the AppleScript component sends a Get Data event to the SurfWriter application requesting the data from the specified table. The SurfWriter application returns the data to the AppleScript component in a standard reply Apple event, and the AppleScript component sets the value of the variable `Totals` to the data returned by SurfWriter.

Then the AppleScript component performs the action described in the second `tell` statement:

```
tell application "SurfCharter"
   copy Totals to Chart 1 of document "Sales Chart"
end tell
```

In this case, the AppleScript component sends a Set Data event to the SurfCharter application that sets the specified chart to the value of the variable `Totals`.

Both SurfWriter and SurfCharter are server applications for the Apple events sent by the AppleScript component, because they are performing services in response to requests made by the client application via the script.

To send the appropriate Apple events to a scriptable application while executing a script, a scripting component must obtain information about the nature of that application's support for Apple events and the human-language terminology to associate with those events. A scriptable application provides this information in the form of an Apple event terminology extension (`'aete'`) resource. A scripting component uses both the `'aete'` resource provided by a scriptable application and the Apple event user terminology (`'aeut'`) resource provided by the scripting component itself to obtain the information it needs to execute a script that controls that application.

See "Making Your Application Scriptable," which begins on page 7-14, for an overview of the tasks you should perform to make your application scriptable and a more detailed description of the `'aete'` and `'aeut'` resources. See "Making Your Application Recordable" on page 7-20 for an overview of the tasks you should perform if you want your application to be recordable as well as scriptable.

Scripting Components and Applications That Execute Scripts

To store and execute scripts as a client application, your application must first establish a connection with a scripting component registered with the Component Manager on the same computer. Each scripting component can manipulate and execute scripts written in the corresponding scripting language when your application calls the standard scripting component routines.

Your application can use scripting component routines to

n   obtain a handle to a script in a form that can be saved, and load the script again when necessary

n   allow users to modify scripts that have been previously saved

n   compile and execute scripts

n   redirect Apple events to script contexts

n   supply application-defined functions for use by scripting components

n   control the recording process directly, turning recording off and on and saving the recorded script for use by your application

Your application can perform these tasks as a client application regardless of whether it is scriptable or recordable. If your application is scriptable, however, it can execute scripts that control its own behavior, thus acting as both the client application and the server application for the corresponding Apple events. For example, your application can allow users to associate a script with a custom menu command that performs a series of routine actions on a selected object, sets preferences, or automates other actions within your application.

You can also use scripting component routines to execute scripts that perform tasks for your application with the aid of other applications. For example, a user of a word-processing application might be able to attach a script to a specific word so that the application executes the script whenever that word is double-clicked. Such a script could trigger Apple events that cause other applications to look up and display related information, run a QuickTime movie, perform a calculation, play a voice annotation, and so on.

Your application can associate a script with either Apple event objects or application-defined objects. Almost any user action can be used to trigger such a script: choosing a menu command, clicking a button, tabbing from one table cell to another, and so on. The script can be executed directly by the application when it detects a triggering action; or, if the script is associated with an Apple event object in the form of a script context, it can be executed automatically when a specified Apple event performs an action on that object.

The rest of this section describes one way that an application could execute such a script. Suppose a forms application allows users to create custom forms that can include scripts associated with specific fields on the form. These scripts are executed when the user presses Enter or Tab in the appropriate field. For the purposes of this example, it doesn't matter whether a field with which a script is associated is an Apple event object (which can be described in an object specifier record) or some other application-defined object (which can't be described in an object specifier record).

A company could use the forms application to create a custom order form for taking telephone orders. If the customer has ordered from the company before, the user can quickly retrieve the customer's address from the company database by typing the customer's name in a field and pressing the Tab key. In response, the application executes the script associated with the field. The script might look like this in AppleScript:

```
set custName to field "Customer Name"

tell application SurfDB
    copy the first record in the table MyAddresses ¬
        whose cell "Customer Name" = custName to Address
end tell

set field "Street" to item 2 of Address
set field "City" to item 3 of Address
set field "Zip" to item 4 of Address
```

To execute such a script (or to manipulate it any other way, such as when the form is first created), the forms application must previously have established a connection with the appropriate scripting component—in this case, the AppleScript component. When the user enters a customer name and presses Tab, the forms application calls scripting component routines to execute the script. As shown in Figure 7-4, the AppleScript component first sends the forms application a Get Data event that requests the contents of the "Customer Name" field and sets the variable custName to that value. It then sends SurfDB a Get Data event that requests the appropriate address information and copies it to the variable Address. (The replies to the Get Data events are not shown in Figure 7-4.) Finally, the AppleScript component sends the forms application a Set Data event that copies the address information from the variable Address to the appropriate fields.

The AppleScript component needs to maintain the binding of the variables custName and Address throughout execution of the script. Scripting components bind variables with the aid of a **script context,** which is a script that maintains context information for the execution of other scripts. An application specifies a script context when it executes a script. The forms application in Figure 7-4 provides a context for the scripting component to use whenever it executes a script associated with a button.

Figure 7-4   How an application uses the AppleScript component to execute a script
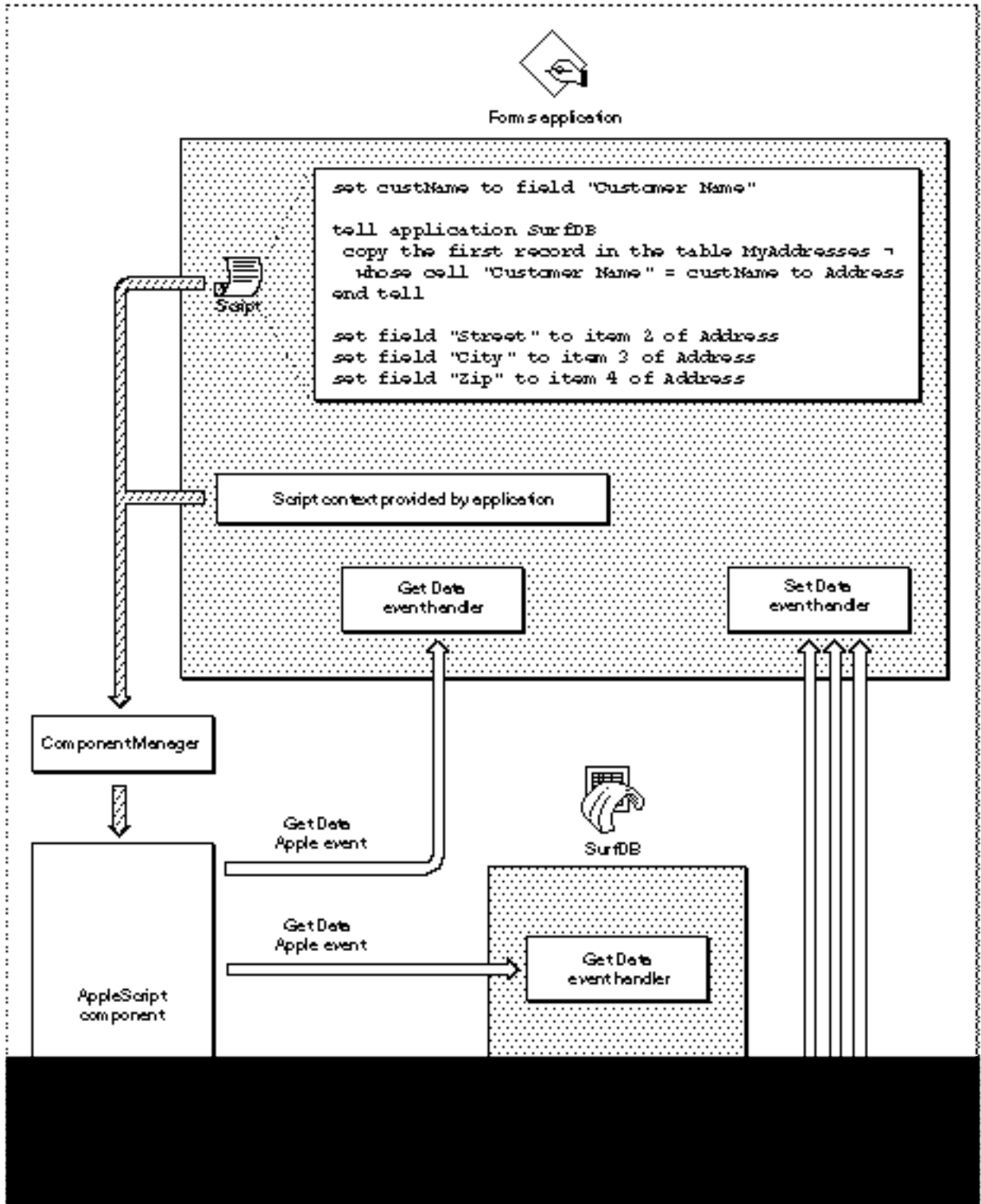
In the example shown in Figure 7-4, the application executes the script directly when the cursor is in the appropriate field and the user presses Tab or Enter. Your application can also associate such a script with an object in the form of a script context, so that the script context is executed whenever a specified Apple event acts on the field. The section "Using a Script Context to Handle an Apple Event," which begins on page 7-25, describes this approach in more detail.

See "Manipulating and Executing Scripts," which begins on page 7-22, for an overview of methods your application can use to save and load script data, compile source data, and perform other useful tasks with scripting component routines. The chapter "Scripting Components" in this book provides full implementation details, including sample code and human interface guidelines for associating scripts with objects.

Making Your Application Scriptable

To make your application scriptable, you need to

n define a hierarchy of Apple event objects within your application that you want client applications to be able to identify—that is, which objects can be contained by other Apple event objects in your application, which properties each kind of object can have, and so on

n write Apple event handlers, object accessor functions, and other routines required to implement the Apple events and related object classes that you want to support

n create an `'aete'` resource

The chapters "Introduction to Apple Events," "Responding to Apple Events," and "Resolving and Creating Object Specifier Records" in this book describe how to perform the first two tasks. The extent to which scripts can control your application depends mainly on the extent of your application's support for Apple events. For example, if your application does not provide the Apple event handlers and object accessor functions required to locate and manipulate windows, users will not be able to use scripts to control your application's windows. Although you should use the definitions in the *Apple Event Registry: Standard Suites* whenever possible, you have considerable freedom to extend or limit your implementation of the standard Apple events according to the needs of your application.

The OSA makes it possible to design new kinds of applications that always operate in the background and can be controlled only by means of scripts. For example, it is possible to design a simple telecommunications program that can log on to a network, send and receive text files created by another application, and perform other basic operations in response to scripts without providing any other form of user interface. Such an application would not need to support Apple events that control window movement, the File menu, or the Edit menu; instead, it would need to support only those Apple events that execute its basic telecommunications operations.

At the other extreme, some applications allow users to arrange windows, palettes, and dialog boxes on their screen in many different ways, or to customize menus or other aspects of the presentation of information. If such an application can respond to scripts that control windows, dialog boxes, specialized preferences, and other aspects of the presentation of information, it can allow users who might not otherwise explore those capabilities to take advantage of them. For example, a naive user could execute a script that sets up a powerful word processor with the appropriate menus, window and palette arrangement, and formatting templates for a particular task, such as producing a company newsletter.

Scripting components use `'aeut'` and `'aete'` resources to associate Apple event codes supported by your application with corresponding human-language terms used in scripts that control your application. Each scripting component supplies an `'aeut'` resource, and each scriptable application provides an `'aete'` resource. The next section introduces the `'aeut'` and `'aete'` resources.

About Apple Event Terminology Resources

As explained in the chapter "Introduction to Apple Events" in this book, applications can support different combinations of the standard suites of Apple events. Applications can also extend the definitions of individual Apple events and object classes, or define custom Apple events and object classes. Scripting components use the **Apple event user terminology resources,** `'aeut'` and `'aete'`, to associate the IDs, keywords, and other codes used in Apple events with the corresponding human-language terms used in scripts that control your application.

The Apple event user terminology (`'aeut'`) resource contains terminology information for all the standard suites of Apple events defined in the *Apple Event Registry: Standard Suites.* The resource consists of a sequence of concatenated arrays that map human-language names to each of the following:

n the ID defined for each suite

n the Apple events defined for each suite

n the parameters defined for each Apple event

n the Apple event object classes defined for each suite

n the properties defined for each object class

n the elements defined for each object class

n the key forms defined for each element class

n the comparison operators defined for each suite

n the values for enumerators defined for each suite

Each scripting component provides its own `'aeut'` resource. A scripting component can also provide different versions of the `'aeut'` resource; for example, the user terminology provided by the `'aeut'` resource for the AppleScript Japanese dialect component is in Japanese. The IDs, keywords, and other codes listed in the `'aeut'` resource are based on the *Apple Event Registry: Standard Suites* and do not vary from one version to another.

An `'aete'` resource has the same format as the `'aeut'` resource but serves a different purpose. Each scriptable application must include its own `'aete'` resource describing which of the standard suites listed in the `'aeut'` resource it supports and providing other application-specific information. Since the human-language equivalents for the standard suites are defined in the `'aeut'` resource, applications that support standard suites without any modifications do not have to define such equivalents; instead, they can simply list, in the `'aete'` resource, the suites they support. The scripting component associates the standard suites listed in the `'aete'` resource with the corresponding Apple event descriptions in its `'aeut'` resource.

Applications can also use the `'aete'` resource to describe extensions to the standard suites, such as additional parameters for standard Apple events, additional properties and element classes for the standard Apple event object classes, and additional key forms for each element class. Information about such extensions must be included in the appropriate arrays of the `'aete'` resource, along with the equivalent human-language terms. Similarly, an application can use the `'aete'` resource to describe the parts of each standard suite it supports (if it doesn't support the entire suite) and any custom Apple events or Apple event object classes defined by the application.

The human language in which your Apple event extensions or custom Apple events are displayed in scripts depends on the corresponding user terminology you specify in your application's `'aete'` resource. Therefore, if your application implements such extensions or custom Apple events, you must provide a separate version of this resource for each localized version of your application.

Scripting components can use the information in the `'aete'` and `'aeut'` resources in a variety of ways. The next section, "How AppleScript Uses Terminology Information," describes how the AppleScript component uses these resources when it executes or records a script. The next chapter, "Apple Event Terminology Resources," describes how to create an `'aete'` resource for your application.

If you want users to be able to control your application with scripts written in the AppleScript scripting language, you also need to know how the AppleScript component interprets AppleScript commands that trigger Apple events. In this way, you can make sure you support Apple events and specify the user terminology for your `'aete'` resource in a way that translates easily into AppleScript statements. The section "Defining Terminology for Use by the AppleScript Component," which begins on page 8-3, discusses these issues. If you implement Apple events so that they translate into logical and useful AppleScript scripts, your implementation will probably work well with other scripting components that resemble AppleScript.

How AppleScript Uses Terminology Information

The manner in which the AppleScript component uses the information in `'aete'` resources depends on specific characteristics of the AppleScript scripting language. An AppleScript expression consists of an internal compiled form and corresponding expressions in **dialects,** or versions of the AppleScript scripting language that resemble different human languages. Users can select the dialect they want to use from within the Script Editor application. If a script is displayed in a window and the user selects a different dialect, the AppleScript component converts the script to the new dialect. Users can install additional dialects as necessary.

This section describes how the AppleScript component uses the information in the `'aeut'` and `'aete'` resources, not how it obtains that information. For a description of the methods available to scripting components for loading information from terminology resources, see "Dynamic Loading of Terminology Information" on page 7-20.

Figure 7-5 shows how the AppleScript component uses information from its `'aeut'` resource and an application's `'aete'` resource to execute a script that consists of AppleScript statements displayed in a script editor window. When a user executes the script from the script editor (for example, by pressing the Run button in the Script Editor application), the AppleScript component first compiles the script into the equivalent compiled expressions, using information from its `'aeut'` resource and the application's `'aete'` resource to map application-specific terms in the script with the equivalent Apple events and Apple event parameters. The AppleScript component then evaluates each expression and performs actions or sends Apple events as appropriate.

For example, the AppleScript component evaluates the expression

```
2*3
```

as the value 6. The AppleScript component can then decompile and display this value in the script editor window, assign it to a variable, or otherwise manipulate it according to the rest of the script. However, to compile the statement

```
print Chart 1 of document "Sales Report"
```

the AppleScript component uses its `'aeut'` resource and the SurfWriter application's `'aete'` resource to associate the terms used in the script with the Print Apple event, the object class for charts, and the object class for documents, so that it can describe the event accurately in the form of a compiled expression. When the AppleScript component evaluates the compiled expression, it creates and sends a Print event whose direct parameter is an object specifier record that the SurfWriter application can resolve as the specified chart. The SurfWriter application then handles the Apple event by printing the chart as requested.

Figure 7-5   Role of the `'aete'` and `'aeut'` resources when the AppleScript component
compiles and executes a script
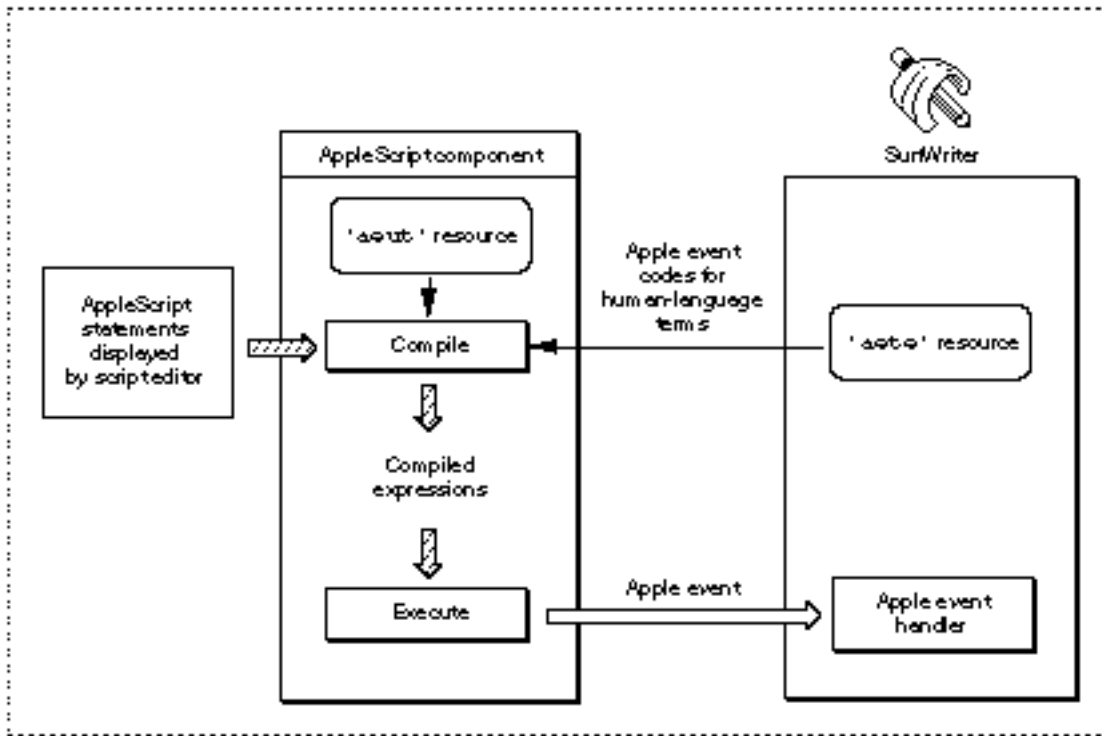
Note that although Figure 7-5 shows only one Apple event generated as a result of
executing a script, the AppleScript component could also send a series of Apple events to
several different applications, depending on the content of the script.

A recordable application generally needs to be able to send itself a subset of the
Apple events that it can handle as a scriptable application. A **recordable event** is
any Apple event that any recordable application sends to itself while recording is turned
on for the local computer (with the exception of events that the application explicitly
identifies as not for recording purposes). After a user turns on recording from the
Script Editor application, the Apple Event Manager sends copies of all recordable events
to Script Editor. A scripting component previously selected by the user handles each
copied event for Script Editor by translating the event and recording the translation as
part of a Script Editor script. When a scripting component executes a recorded script, it
sends the corresponding Apple events to the applications in which they were recorded.

Every scripting component must be able to handle copies of recordable events sent to a
recording process (such as Script Editor) by recording them in an appropriate form.
For example, as shown in Figure 7-6, the AppleScript component records copies of
recordable events in the form of compiled expressions. The AppleScript component can
then use information from its `'aeut'` resource and the application's `'aete'` resource to

translate the compiled expressions into the appropriate human-language terms and display them as AppleScript statements in the script editor window. When the user opens a recorded script in Script Editor and presses Run, the AppleScript component recompiles the script if necessary and sends the Apple events described by the compiled expressions to the SurfWriter application, just as in Figure 7-5.

Figure 7-6   Role of the `'aete'` and `'aeut'` resources when the AppleScript component records and decompiles a script
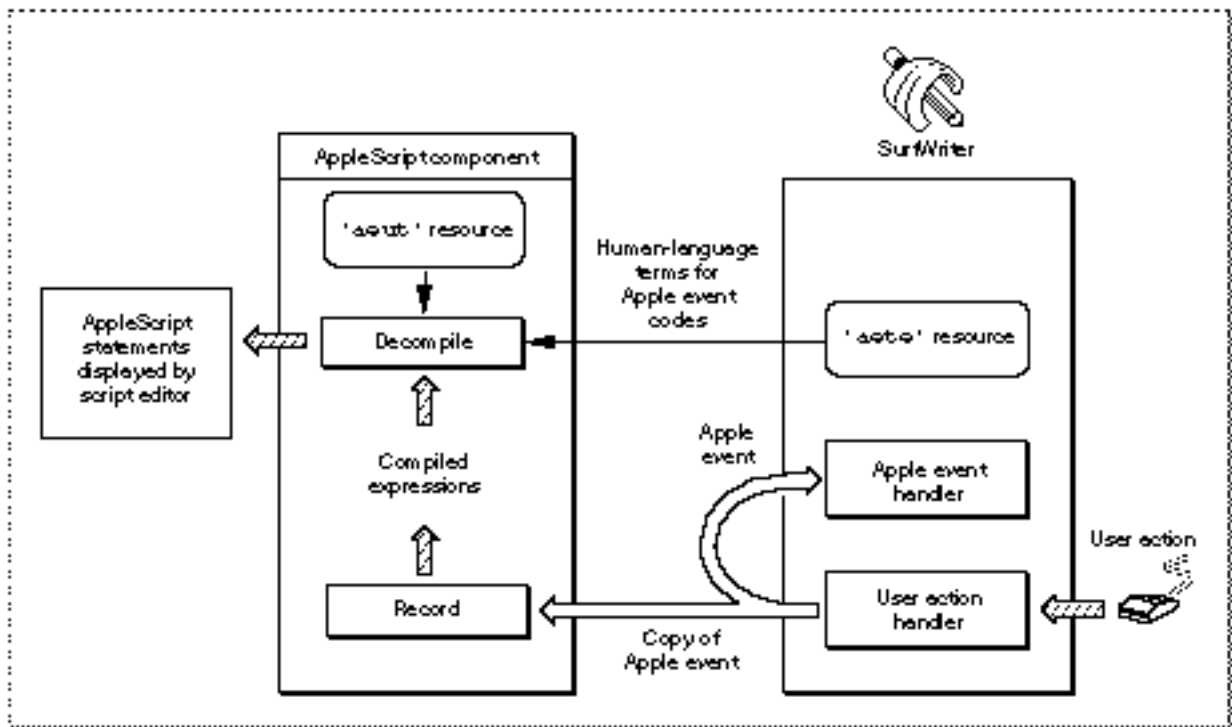
If the user copies a chart from one document to another document and the SurfWriter application performs this task by sending itself Apple events, the equivalent statements in the recorded script might look something like this:

```
tell application "SurfWriter"
    select Chart 1 of document "Sales Chart"
    copy
    select paragraph 3 of document "Monthly Report"
    paste
end tell
```

To display these statements in the script editor window, the AppleScript component first translates the Set Data, Copy, and Paste Apple events sent by the recordable application into compiled expressions. It then uses information from its `'aeut'` resource and the application's `'aete'` resource to decompile the compiled expressions and pass the equivalent source data to the script editor for display to the user. After completing a recording session, the user can edit and save the resulting script and execute it again at any time.

As shown in Figure 7-5 and Figure 7-6, the AppleScript component uses information it obtains from the `'aeut'` and `'aete'` resources when it is compiling and decompiling scripts. Other scripting components might use the same information during execution or recording, or in other ways that are specific to each component.

Dynamic Loading of Terminology Information

When a scripting component needs information about the user terminology defined in your application's `'aete'` resource, it sends a Get AETE event to your application. If your application does not handle the Get AETE event, the scripting component reads the terminology information it needs directly from your application's `'aete'` resource.

Your application does not need to handle the Get AETE event unless it provides separate `'aete'` resources for plug-in components. If your application does provide separate plug-in components, the Get AETE event allows it to gather terminology information from the `'aete'` resources for the components that are currently running and add that information to the reply event.

If your application handles the Get AETE event, you must also provide a scripting size resource. A scripting size resource is a resource of type `'scsz'` that provides information about an application's capabilities and preferences for use by scripting components.

To take advantage of dynamic loading, your application must be running. Note that if your application does not provide a handler for the Get AETE event, the scripting component can obtain terminology information directly from your application's `'aete'` resource even if your application is not running.

Making Your Application Recordable

If you decide to make your application scriptable, you can also make it recordable. A **recordable application** is an application that uses Apple events to report user actions to the Apple Event Manager for recording purposes. A recordable event is any Apple event that a recordable application sends to itself while recording is turned on for the local computer (with the exception of events sent with the `kAEDontRecord` flag set in the `sendMode` parameter of `AESend`).

When a user turns on recording by clicking the Record button in the Script Editor application, the Apple Event Manager sends copies of all subsequent recordable events to Script Editor. The AppleScript component handles each copied event for Script Editor by translating it into compiled expressions and recording the compiled expressions as part of a script. (Figure 7-6 on page 7-19 shows how the AppleScript component uses the `'aete'` and `'aeut'` resources when it records a script.) The user can view the equivalent decompiled source data in Script Editor while the script is being recorded. When a user executes a recorded script, the AppleScript component sends the corresponding Apple events to the applications in which they were recorded.

Applications generally have two parts: the code that implements the application's user interface and the code that actually performs the work of the application when the user manipulates the interface. One way to make your application recordable is to separate these two parts of your application, using Apple events to connect user actions with the work your application performs. This is called **factoring** your application. In a fully factored application, almost all tasks are carried out in response to Apple events. The application translates low-level events that result in significant actions into recordable Apple events and then sends them to itself.

Factoring your application is the recommended method of making your application recordable. However, it is also possible for your application to report user actions by means of Apple events even though it actually performs those actions by some means other than Apple events. You can indicate that you want the Apple Event Manager to record events in this manner, without executing them, by adding the constant `kAEDontExecute` to the `sendMode` parameter of `AESend`.

Before you decide how to map the user's potential actions to recordable Apple events supported by your application, you need to answer these questions:

n   What are the significant (that is, undoable) actions a user can perform with your application that you want to record?

n   Which actions can you execute by means of Apple events, and which actions should cause Apple events to be sent but not executed?

n   How do you want to record actions that can be described in a scripting language in several different ways?

For example, if your application is a word processor, the user's selection of a range of text should probably not generate an Apple event, because users often select various different pieces of text before deciding to do something to the selection. However, if a user changes the font of a selection, a recordable word processor should generate a corresponding Apple event so that the scripting component can record the change.

In general, a recordable application should generate Apple events for any user action that the user could reverse by choosing Undo. A recordable application can usually handle a greater variety of Apple events than it can record, because it must record the same action the same way every time even though Apple events might be able to trigger that action in several different ways.

For more information about recordable applications, factoring, and the Apple Event Manager's recording mechanism, see the chapter "Recording Apple Events" in this book. For a description of the role of the `'aete'` and `'aeut'` resources when the AppleScript component records a script, see "How AppleScript Uses Terminology Information," which begins on page 7-17.

Manipulating and Executing Scripts

Your application can use scripting component routines to manipulate and execute scripts written in any scripting language based on the OSA. This section describes how scripting components use script data and summarizes some of the tasks your application can perform by calling the standard scripting component routines.

Your application can manipulate and execute scripts regardless of whether it is scriptable or recordable. However, if your application is scriptable, you can easily make it capable of manipulating and executing scripts that control its own behavior. For example, the forms application shown in Figure 7-4 on page 7-13 uses standard scripting component routines to execute a script whenever the cursor is in the appropriate field and the user presses Enter or Tab. Applications can also use scripting component routines to allow users to edit, recompile, save, and load such scripts in order to adapt them to their own purposes.

Before using any scripting component routines, your application must open a connection with at least one scripting component. After opening a connection with a component, your application receives a component instance that it can use as the first parameter for any scripting component routine. You can use the Component Manager to establish a connection with the generic scripting component or to establish an explicit connection with any other scripting component. Your application can open connections with different scripting components under different circumstances and, if necessary, simultaneously.

To manipulate or execute scripts written in any scripting language based on the OSA, your application can open a connection with the **generic scripting component.** The generic scripting component in turn attempts to open connections dynamically with the appropriate scripting component for a given script. If your application opens a connection with the generic scripting component, it can load and execute scripts created by any scripting component that is registered with the Component Manager on the current computer. The generic scripting component also provides routines that allow you to determine which scripting component created a particular script and to perform other useful tasks when you are using multiple scripting components.

To manipulate and execute scripts written in a single scripting language only, your application can open an explicit connection with the scripting component for that language. In this case your application can load and execute only those scripts that were created by that component; however, your application can also take advantage of additional routines and other special capabilities provided by the component.

After your application has established a connection with the appropriate scripting component for an existing script, it can use the standard scripting component routines to execute scripts. A script that has not yet been compiled consists of **source data,** or statements in a scripting language. Before executing source data, your application must use scripting component routines to compile it so that the scripting component can keep track of it in memory and execute it.

Scripting components can refer to at least three kinds of **script data** in memory:

n   A **compiled script** consists of compiled code that an application can decompile into source data or execute using the standard scripting component routines.

n   A **script value** consists of an integer, a string, a Boolean value, constants, PICT data, or any other fixed data returned or used by a scripting component in the course of executing a script.

n   A **script context** maintains context information for the execution of other scripts. A script context can also contain executable statements in a scripting language. Like a compiled script, a script context can be decompiled as source data.

For example, a script context can contain user-defined handlers for specific Apple events. In AppleScript, a script context that contains such handlers or other executable statements is called a **script object.** Handlers in a script object resemble HyperTalk message handlers. They consist of AppleScript statements and have no corresponding entry in Apple event dispatch tables.

Scripting components keep track of script data in memory by means of **script IDs** of type OSAID.

```
TYPE OSAID = LongInt;
```

A scripting component assigns a script ID to a compiled script or script context whenever the component creates or loads the corresponding script data. The scripting component routines that compile, load, and execute scripts all return script IDs, and you must pass valid script IDs to many of the other routines that manipulate scripts.

Applications most commonly use scripting component routines to

n   compile source data and execute the resulting compiled script, so that a user can create a new script and execute it immediately from within the application

n   get a handle to script data in a form that can be saved, and load and execute the script data again when necessary

n   allow users to modify a script, then recompile and save the script

n   redirect Apple events to script contexts

The remainder of this section provides an overview of the scripting component routines you can use to perform these tasks.

Your application can also use scripting component routines to

n   get information about scripts

n   get information about scripting components

n   coerce script values to descriptor records and vice versa

n   set a resume dispatch function and alternative send, create, and active functions for
    use by a scripting component

n   control the recording process directly, turning recording off and on and saving the
    recorded script for use by your application

The chapter "Scripting Components" in this book provides detailed information about
using all the standard scripting component routines as well as additional routines
provided by the AppleScript component and the generic scripting component.

Compiling, Saving, Modifying, and Executing Scripts

This section introduces some of the scripting component functions your application can
use to compile, save, modify, and execute scripts.

To create and execute a script using the Script Editor application, a user can type the
script, then press the Run button to execute it. Your application can provide similar
capabilities by using these functions to compile source data and execute the resulting
compiled script:

n   The `OSACompile` function takes a descriptor record with a handle to source data
    (usually text) and a script ID. If you specify `kOSANullScript` instead of an existing
    script ID, `OSACompile` returns a script ID for the new compiled script, which you can
    then pass to the `OSAExecute` function.

n   The `OSAExecute` function takes a script ID for a compiled script and a script ID for a
    script context, executes the script, and returns a script ID for the resulting script value.

The binding of any global variables in the compiled script is determined by the
script context whose script ID you pass to `OSAExecute`. If you pass `kOSANullScript`
instead of the script ID for a script context, the scripting component provides its own
default context. If you want to provide your own script context rather than using the
scripting component default context, you can use either `OSACompile` or
`OSAMakeContext` to create a script context, which you can load and store just like a
compiled script.

After creating a script and trying it out, a user may want to save it for future use. Your
application should normally save its scripts as script data rather than source data, so that
it can reload and execute the data without recompiling it. Before saving script data, you
must first call the `OSAStore` function to get a handle to the data in the form of a
descriptor record. You can then save the data to disk as a resource or write it to the data
fork of a document.

To allow a user to reload and execute a previously compiled and saved script, your
application can call these functions:

n   The `OSALoad` function takes a descriptor record that contains a handle to the saved
    script data and returns a script ID for the compiled script.

n   The `OSAExecute` function takes a script ID for a compiled script and a script ID for a
    script context, executes the script, and returns a script ID for the resulting script value.

In most cases you will want to allow users to modify saved scripts and save them again. To allow a user to modify and save a compiled script, your application can call these functions:

n The OSAGetSource function takes a script ID and returns a descriptor record with a handle to the equivalent source data.

n The OSACompile function takes a descriptor record with a handle to source data and a script ID, and returns the same script ID updated so that it refers to the modified and recompiled script.

n The OSAStore function takes a script ID and returns a copy of the corresponding script data in the form of a storage descriptor record.

You can pass the script ID for the compiled script to be modified to the OSAGetSource function, which returns a descriptor record with a handle to the equivalent source data. Your application can then present the source data to the user for editing. When the user has finished editing the source data, you can pass the modified source data and the original script ID to the OSACompile function to update the script ID so that it refers to the modified and recompiled script. Finally, to obtain a handle to the modified script data so you can save it in a resource or write it to the data fork of a document, you can pass the script ID for the modified compiled script to the OSAStore function.

If your application has no further use for a compiled script or a resulting script value after successfully loading, saving, compiling, or executing a script, you can use the OSADispose function to release the memory assigned to them. The OSADispose function takes a script ID and releases the memory assigned to the corresponding script data. A script ID is no longer valid after the memory associated with it has been released. This means, for example, that a scripting component may assign a different script ID to the same compiled script each time you load it, and that a scripting component may reuse a script ID that is no longer associated with a specific script.

"Using Scripting Component Routines," which begins on page 10-7, provides more information about the standard scripting component routines described in this section.

Using a Script Context to Handle an Apple Event

One way to associate a script with an object is to associate a script context with a specific Apple event object—that is, with any object in your application that can be identified by an object specifier record. When an Apple event acts on an Apple event object with which a script context is associated, your application attempts to use the script context to handle the Apple event. This approach can be useful if you want to associate many different scripts with many different kinds of objects.

Figure 7-7 illustrates one way that an application can use a script context to handle an Apple event. This example shows how you can use a general Apple event handler to provide initial processing for all Apple events received by your application. If an Apple event acts on an object with which a script context is associated, the general handler attempts to use the script context to handle the event.

The SurfWriter application in Figure 7-7 associates script contexts (called script objects in AppleScript) with geometric shapes such as circles or squares. These script contexts can contain one or more user-defined handlers for specific Apple events. For example, the script context shown in Figure 7-7 is associated with a circle and contains this handler:

```
on move to {x, y}
    continue move to {x, item 2 of position of this}
end move
```

This handler exists only as AppleScript statements in the script context and doesn't have an entry in SurfWriter's Apple event dispatch table. SurfWriter does have its own standard Apple event handlers installed in its Apple event dispatch table. When SurfWriter receives a Move event that acts on the circle with which this script context is associated, SurfWriter uses the handler in the script context to modify its own standard handling of the event. The rest of this section describes how this works.

Figure 7-7   Using a handler in a script context to handle an Apple event
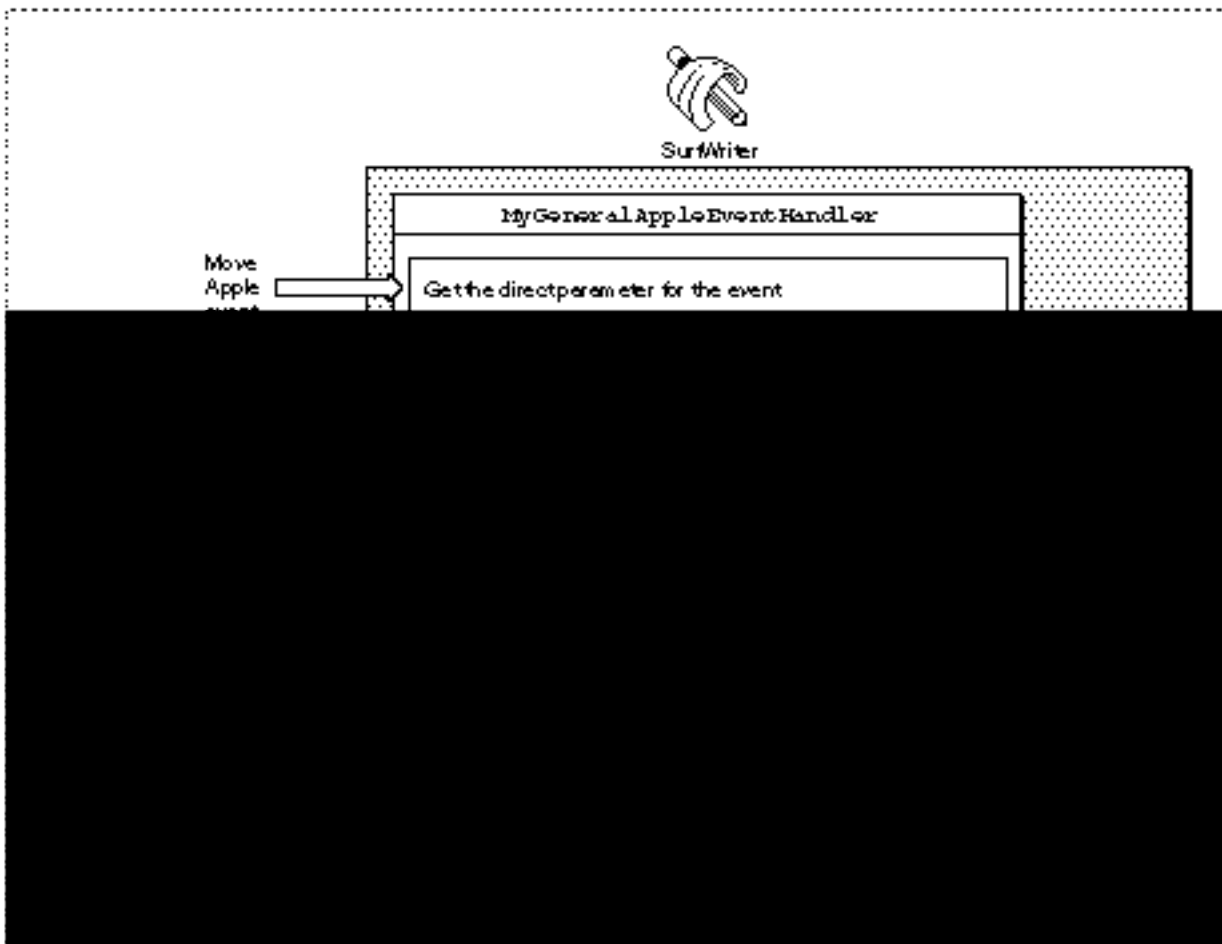
The `MyGeneralAppleEventHandler` function in Figure 7-7 is installed in SurfWriter's special handler dispatch table. Thus, `MyGeneralAppleEventHandler` provides initial processing for all Apple events received by SurfWriter. When it receives an Apple event, `MyGeneralAppleEventHandler` checks whether a script context is associated with the object on which the event acts. If so, `MyGeneralAppleEventHandler` passes the event and a script ID for the script context to the `OSADoEvent` function. If not, `MyGeneralAppleEventHandler` returns `errAEEventNotHandled`, which causes the Apple Event Manager to look for the appropriate handler in SurfWriter's Apple event dispatch table.

The `OSADoEvent` function looks for a handler in the specified script context that can handle the specified event. If the script context doesn't include an appropriate handler, `OSADoEvent` returns `errAEEventNotHandled`. If the script context includes an appropriate handler (in this example, a handler that begins `on move`), `OSADoEvent` attempts to use the handler to handle the event.

When it encounters the `continue` statement during execution of the `on move` handler shown in Figure 7-7, the AppleScript component calls SurfWriter's resume dispatch function. A **resume dispatch function** takes an Apple event and invokes the application's default handler for that event directly, bypassing the application's special handler dispatch table and the `MyGeneralAppleEventHandler` handler (or its equivalent). In this case, the AppleScript component uses SurfWriter's default Move handler to move the circle to a different location than the one specified in the original Move event. The location specified by `{x, item 2 of position of this}` has the same horizontal coordinate as the location specified by the original event, but specifies the circle's original vertical coordinate (item 2 of the circle's original position), thus constraining motion to a horizontal direction.

The AppleScript component calls the resume dispatch function as soon as it encounters a `continue` statement during script execution. For example, if the handler in Figure 7-7 contained additional indented statements after the `continue` statement, the AppleScript component would proceed with the execution of those statements after calling the resume dispatch function successfully.

A script context can modify the event and use the default Apple event handler to execute the modified event, as in this example; or it can override the default handler completely, performing some completely different action; or it can perform some action and then pass the original event to the application's default handler to be handled in the usual way. Script contexts associated with Apple event objects thus provide a way for users to modify or override the way an application responds to a particular Apple event that manipulates those objects.

A general Apple event handler can use the `OSAExecuteEvent` function instead of `OSADoEvent` to execute a script context. The main difference between these functions is is that `OSAExecuteEvent` returns the script ID for the resulting script value, whereas `OSADoEvent` returns a reply event.

To create a script context, pass the source data for the scripting-language statements you want the script context to contain to `OSACompile` with the `modeFlags` parameter set to `kOSACompileIntoContext`. The resulting script context is identical to a script context returned by the `OSAMakeContext` function, except that it contains compiled statements.

"Using a Script Context to Handle an Apple Event," which begins on page 10-19, describes this method of executing a script in more detail.

Apple Event Terminology Resources

Contents

This chapter describes the resource structure used by both the `'aeut'` and `'aete'` resources and explains how to create an `'aete'` resource for your application. It also explains how applications that support additional plug-in modules, each with its own `'aete'` resource, can write a handler for the Get AETE event that collects the `'aete'` resources from the modules that are currently running.

Before you read this chapter, you should read the chapter "Introduction to Scripting" in this book and the chapters about the Apple Event Manager that are relevant to your application.

The first section in this chapter describes how the AppleScript component interprets AppleScript statements that trigger Apple events. The first section also explains how to define both Apple events and the corresponding user terminology for your application in a way that translates easily into AppleScript statements. If you implement Apple events so that they translate into logical and useful AppleScript scripts, your implementation will probably work well with other scripting components that resemble AppleScript.

The next two sections describe how to

n create an `'aete'` resource

n handle the Get AETE event

For details about the structure of the data in an `'aeut'` resource and an `'scsz'` resource, see "Reference to Apple Event Terminology Resources," which begins on page 8-26.

Defining Terminology for Use by the AppleScript Component

You should keep two principles in mind when you are defining the Apple event object hierarchy and corresponding terminology for your application:

n Avoid defining new Apple events unless absolutely necessary. For example, instead of defining a custom Find event, use the Get Data event with whose tests. (For more information about whose tests, see the chapter "Resolving and Creating Object Specifier Records" in this book.)

n Use existing object classes, or if you must define your own, define them in a general fashion.

This section describes how the terms you specify in your application's `'aete'` resource are used in AppleScript statements that control your application. Before you implement the Apple event object hierarchy for your application, try out your proposed user terminology in AppleScript statements that use the standard syntax forms described here. This will help you discover some of the advantages and disadvantages of both your proposed object hierarchy and the human-language terminology you are planning to use.

Some AppleScript commands, such as `if`, `repeat`, and `tell`, are executed directly by the AppleScript component and do not correspond to Apple events. Other commands trigger Apple events when the AppleScript component evaluates them.

AppleScript command	Corresponding Apple event
`open`	Open
`close`	Close
`save`	Save
`move`	Move
`delete`	Delete
`set`	Set Data

The AppleScript component interprets the terms used in scripts according to rules defined by the AppleScript language. For example, the `open` command must be followed by an argument that specifies the objects to open, and the `save` command must be followed by an argument that specifies the objects to save. The AppleScript component uses the information in an application's `'aete'` resource to map the human-language terms used in these arguments to specific Apple event keywords and codes, so that it can construct object specifier records that describe the objects on which the Open and Save events act.

In general, the syntax for AppleScript commands that trigger Apple events follows this pattern:

event name *expression* *parameter name* *expression . . . parameter name* *expression*

The underlined terms are supplied by the AppleScript component's `'aeut'` resource, by the application's `'aete'` resource, or by the `'aeut'` resource available on the current computer. The argument that follows *event name* corresponds to the direct parameter for the event, if there is one. Each subsequent argument corresponds to an additional parameter.

An argument that corresponds to a direct parameter can use any of the syntax forms shown in Table 8-1. These forms correspond to the key forms that can be used to identify the key data in an object specifier record.

Table 8-1   Syntax for AppleScript arguments that correspond to direct parameters

Syntax of argument	AppleScript example	Key form
property name	`the font`	`formPropertyID`
class name expression	`table "Fred"` `table 4`	`formName` `formAbsolutePosition`
class name `before` | `after` expression	`word after table 2`	`formRelativePosition`
class name expression `thru` expression	`words 1 thru 30`	`formRange`
`every` _class name_ `whose` expression	`every word whose¬` `    font = "Palatino"`	`formWhose`
expression **of** _expression_	`first row of¬` `    table "Fred"`	**Any key form; sets container for elements or properties**

If the Apple event object hierarchy for your application requires you to specify terms in your `'aete'` resource that are not included in the `'aeut'` resource, make sure those terms read naturally when they appear in AppleScript statements that use the syntax shown in Table 8-1. Any of the underlined terms in the table may be supplied by your application's `'aete'` resource.

For example, in the AppleScript statement

`copy` _name_ `to` _expression_

the argument _name_ corresponds to a direct parameter that can use any of the syntax variations shown in Table 8-1. The word `to` and the expression that follows it correspond to an additional parameter that describes the location to which to copy the objects described by the direct parameter.

Many AppleScript commands, including the `copy` command, take additional arguments that correspond to **insertion location descriptor records**, which are descriptor records of type `typeInsertionLoc` defined as part of the Core suite. An insertion location descriptor record is a coerced AE record that consists of two keyword-specified descriptor records with the following keywords:

Keyword	Description
`keyAEObject`	An object specifier record that identifies a single container
`keyAEPosition`	A constant that specifies where to put the Apple event object described in an Apple event's direct parameter in relation to the container specified in the descriptor record with the keyword `keyAEObject`

You can specify one of these constants for the data in a descriptor record identified by the keyword keyAEPosition:

Constant	Meaning
kAEBefore	Before the container
kAEAfter	After the container
kAEBeginning	In the container and before all other elements of the same class as the object being inserted
kAEEnd	In the container and after all other objects of the same class as the object being inserted
kAEReplace	Replace the container

The syntax that corresponds to an insertion descriptor record can take any of the forms shown in Table 8-2.

Table 8-2   Syntax for AppleScript arguments that correspond to insertion location descriptor records

Syntax of argument	AppleScript example	keyAEPosition constant
before | after *expression*	before Figure 1	kAEBefore | kAEAfter
beginning of | end of *expression*	end of window 2	kAEBeginning | kAEEnd
expression	Figure 1	kAEReplace

For example, in the AppleScript statement

```
copy Chart 1 of document "Sales Chart" to before Figure 1
```

the term copy corresponds to a Clone event, and Chart 1 of document "Sales Chart" corresponds to the direct parameter for the Clone event. The term to is the human-language name specified by the 'aeut' resource for the additional parameter identified by the keyword kAEInsertHere, which always consists of an insertion location descriptor record. The term before corresponds to the constant kAEBefore in the descriptor record identified by the keyword keyAEPosition, and Figure 1 corresponds to the object specifier record identified by the keyword keyAEObject.

The AppleScript component handles statements that describe the replacement of one object with another differently from statements that specify an insertion location before, after, at the beginning of, or at the end of an object.

For example, in the statement

```
copy Chart 1 of document "Sales Chart" to Figure 1
```

the term `to` is the human-language name for the additional parameter identified by the keyword `kAEInsertHere`. When `to` is followed immediately by an element expression like `Figure 1`, the Clone Apple event sent by the AppleScript component includes an additional parameter that consists of an object specifier record for Figure 1. When your application requests the parameter as an insertion location descriptor record, a system coercion handler installed by the AppleScript component converts the object specifier record to an insertion location descriptor record that specifies `kAEReplace` in the descriptor record identified by the keyword `keyAEPosition`.

If your application defines any extensions to the standard Apple events or object classes that require the use of insertion locations, use standard insertion location descriptor records to specify them, and make sure your Apple event object hierarchy and the corresponding human terminology in your `'aete'` resource allow the AppleScript component to translate insertion location descriptor records into meaningful statements in an AppleScript dialect.

Unlike most other AppleScript commands, the `copy` command causes the AppleScript component to send different Apple events under different circumstances. In the examples just discussed, the `copy` command corresponds to a Clone event. However, after evaluating the statements

```
tell application "SurfWriter"
     copy table "Summary of Sales" of document¬
        "Sales Report" to Totals
end tell
```

the AppleScript component sends a Get Data event and sets the variable `Sales91` to the value of the returned data; and the statements

```
tell application "SurfCharter"
     copy Totals to Chart 1 of document "Sales Chart"
end tell
```

cause the AppleScript component to send a Set Data event that sets the data in the specified chart to the value of the variable `Totals`.

All scriptable applications should support the Get Data, Set Data, and Clone events for all Apple event objects that a user might want to manipulate from a script with the `copy` command. Scriptable applications should also support the other core events and any appropriate functional-area events.

If you find it difficult to come up with meaningful AppleScript statements based on your proposed implementation of Apple events, you may need to rethink your implementation.

Structure of Apple Event Terminology Resources

Table 8-3 summarizes the resource structure used by both the `'aeut'` and `'aete'` resources. Each asterisk (*) in the table indicates the beginning of an array. Each array can contain any number of items, including both additional arrays and specific definitions (n).

Table 8-3   Structure of the `'aeut'` and `'aete'` resources

n Template version

n Language code

* Array of suites:

    n Suite information

    * Array of events:

        n Event information (including information about the direct parameter)

        * Array of other parameters:

            n Parameter information

    * Array of classes:

        n Class description

        * Array of properties:

            n Property information

        * Array of elements:

            n Element information

            * Array of key forms:

                n Key form information

    * Array of comparison operators:

        n Comparison operator information

    * Array of enumerations:

        n Enumeration information

        * Array of enumerators:

            n Enumerator information

Listing 8-1 shows the resource type declaration in Rez format for the 'aeut' resource, which can also serve as a template for an 'aete' resource. (Rez is a resource compiler available with the MPW programming environment.) For complete descriptions of all the fields shown in Listing 8-1, see "Reference to Apple Event Terminology Resources," beginning on page 8-26.

Listing 8-1   Resource type declaration for the 'aeut' resource

```
type 'aeut' {
   hex byte;                            /*major version in binary-coded */
                                        /* decimal (BCD)*/
   hex byte;                            /*minor version in BCD*/
   integer  Language, english = 0, japanese = 11;  /*language code*/
   integer  Script, roman = 0;       /*script code*/
   integer = $$Countof(Suites);
   array Suites {
      pstring;                         /*human-language name of suite*/
      pstring;                         /*suite description*/
      align word;                      /*alignment*/
      literal longint;                 /*suite ID*/
      integer;                         /*suite level*/
      integer;                         /*suite version*/
      integer = $$Countof(Events);
      array Events {
         pstring;                      /*human-language name of event*/
         pstring;                      /*event description*/
         align word;                   /*alignment*/
         literal longint;              /*event class*/
         literal longint;              /*event ID*/
         literal longint   noReply = 'null';    /*reply type*/
         pstring;                      /*reply description*/
         align word;                   /*alignment*/
         boolean  replyRequired,       /*if the reply is */
                  replyOptional;       /* required*/
         boolean  singleItem,          /*if the reply must be a list*/
                  listOfItems;
         boolean  notEnumerated,       /*if the type is enumerated*/
                  enumerated;
         boolean  reserved;            /*these 13 bits are reserved; */
         boolean  reserved;            /* set them to "reserved"*/
         boolean  reserved;
         boolean  reserved;
         boolean  reserved;
```

```
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved,            /*if event is verb event or nonverb */
         nonVerbEvent;        /* event; used by Japanese dialect*/
literal longint   noParams = 'null'; /*direct param type*/
pstring;                      /*direct param description*/
align word;                   /*alignment*/
boolean  directParamRequired, /*if the direct param is required*/
         directParamOptional;
boolean  singleItem,          /*if the param must be a list*/
         listOfItems;
boolean  notEnumerated,       /*if the type is enumerated*/
         enumerated;
boolean  doesntChangeState,   /*if the event changes server's state*/
         changesState;
boolean  reserved;            /*these 12 bits are reserved; */
boolean  reserved;            /* set them to "reserved"*/
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
integer = $$Countof(OtherParams);
array OtherParams {
   pstring;                   /*human-language name for parameter*/
   align word;                /*alignment*/
   literal longint;           /*parameter keyword*/
   literal longint;           /*parameter type*/
   pstring;                   /*parameter description*/
   align word;                /*alignment*/
   boolean  required,         /*if param is required*/
            optional;
```

```
   boolean  singleItem,    /*if the param must be a list*/
            listOfItems;
   boolean  notEnumerated, /*if the type is enumerated*/
            enumerated;
   boolean  isNamed,       /*indicates if this should be the */
            isUnnamed;     /* unnamed parameter; only one */
                           /* parameter can be so marked; set to */
                           /* reserved if not required*/
   boolean  reserved;      /*these 9 bits are reserved; */
   boolean  reserved;      /* set them to "reserved"*/
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  notFeminine,   /*feminine; set to reserved if not  */
            feminine;      /* required*/
   boolean  notMasculine,  /*masculine; set to reserved if not */
            masculine;     /* required*/
   boolean  singular,
            plural;        /*plural*/
   };
};
integer = $$Countof(Classes);
array Classes {
   pstring;                    /*human-language name for class*/
   align word;                /*alignment*/
   literal longint;           /*class ID*/
   pstring;                    /*class description*/
   align word;                /*alignment*/
   integer = $$Countof(Properties);
   array Properties {
      pstring;                 /*human-language name for property*/
      align word;             /*alignment*/
      literal longint;        /*property ID*/
      literal longint;        /*property class*/
      pstring;                 /*property description*/
      align word;             /*alignment*/
      boolean  reserved;      /*reserved*/
      boolean  singleItem,    /*if the property must be a list*/
               listOfItems;
```

```
        boolean  notEnumerated, /*if the type is enumerated*/
                 enumerated;
        boolean  readOnly,       /*can only read it*/
                 readWrite;      /*can read or write it*/
        boolean  reserved;       /*these 9 bits are reserved; */
        boolean  reserved;       /* set them to "reserved"*/
        boolean  reserved;
        boolean  reserved;
        boolean  reserved;
        boolean  reserved;
        boolean  reserved;
        boolean  reserved;
        boolean  reserved;
        boolean  reserved;
        boolean  notFeminine,    /*feminine; set to reserved if not  */
                 feminine;       /* required*/
        boolean  notMasculine,   /*masculine; set to reserved if not */
                 masculine;      /* required*/
        boolean  singular,
                 plural;         /*plural*/
    };
    integer = $$Countof(Elements);
    array Elements {
        literal longint;         /*element class*/
        integer = $$Countof(KeyForms);
        array KeyForms {         /*list of key forms*/
           literal longint
           formAbsolutePosition = 'indx',
           formName = 'name';    /*key form ID*/
        };
    };
};
integer = $$Countof(ComparisonOps);
array ComparisonOps {
   pstring;                      /*human-language name for */
                                 /* comparison operator*/
   align word;                   /*alignment*/
   literal longint;              /*comparison operator ID*/
   pstring;                      /*comparison operator description*/
   align word;                   /*alignment*/
};
integer = $$Countof(Enumerations);
array Enumerations {             /*list of enumerations*/
   literal longint;              /*enumeration ID*/
```

```
        integer = $$Countof(Enumerators);
        array Enumerators {            /*list of enumerators*/
          pstring;                     /*human-language name for enumerator*/
          align word;                  /*alignment*/
          literal longint;             /*enumerator ID*/
          pstring;                     /*enumerator description*/
          align word;                  /*alignment*/
        };
      };
    };
};
```

Creating an Apple Event Terminology Extension Resource

Scriptable applications must include an Apple event terminology extension (`'aete'`) resource. You use an `'aete'` resource to inform scripting components about the extent of your application's support for the standard Apple event suites, any custom Apple events or Apple event objects defined by your application, and the corresponding human-language terms for use in scripts that control your application.

The format of the `'aete'` resource is identical to that of the `'aeut'` resource, although it serves a different purpose. The `'aeut'` resource maps human-language names to IDs, keywords, and other codes used in the Apple events described by the current edition of the *Apple Event Registry: Standard Suites*. The `'aete'` resource for an application uses the same format to accomplish the following:

n Indicate when a set of definitions for a particular suite included in the `'aeut'` resource is supported in its entirety by the application. For example, an application can indicate that it supports all of the `'aeut'` resource definitions for the Required and Core suites simply by identifying the suite as a whole in its `'aete'` resource; the detailed information for each standard suite does not need to be repeated.

n Describe extensions, if any, to the definitions included in the `'aeut'` resource, such as additional parameters for standard Apple events, additional properties and element classes for standard object classes, and additional key forms for each element class. For example, an application can indicate that it supports all of the definitions for the Required and Core suites included in the `'aeut'` resource, an additional parameter for one of the core events defined in the `'aeut'` resource, and an additional property for one of the core object classes defined in the `'aeut'` resource.

n Describe the standard Apple events and object classes that belong to suites the application does not support in their entirety.

n Describe the application's custom suite—that is, the application's custom Apple events and object classes, if any.

By specifying a suite ID, suite level, and suite version, your application can indicate that it supports an entire suite. Because the `'aeut'` resource provided by each scripting component lists the human-language terms for all the standard suites, you do not have to repeat this information if you support a suite in its entirety. If you support a subset of a standard suite, you must list all the Apple events, Apple event parameters, object classes, and so on and equivalent human-language terms for the parts of the suite your application does support.

You can include at most one `'aete'` resource per application or per module. The language code for this resource must match the language code of the language for which you are developing your application. Applications that support additional modules with their own `'aete'` resources must provide an `'scsz'` resource and handle the Get AETE event as described in "Handling the Get AETE Event," which begins on page 8-23.

IMPORTANT

Each human-language term supported by an application should correspond to a unique Apple event ID, keyword, or other code in either the application's `'aete'` resource or the `'aeut'` resource. For example, since the `'aeut'` resource defines "size" as the human-language equivalent for the property identified by the four-character code `'ptsz'` (the `pPointSize` property of text objects), an application's `'aete'` resource must not define "size" as the human-language equivalent for some other part of an Apple event or object class. However, more than one human-language term can correspond to the same Apple event ID or code. For example, an application's `'aete'` resource can define a second human-language term, "point size," that corresponds to the Apple event identifier `'ptsz'`. s

The *AppleScript Software Developers' Kit* (available from APDA) includes a tool that allows you to specify your application's support for Apple events and creates the equivalent `'aete'` resource. The previous section, "Structure of Apple Event Terminology Resources," describes the basic format used by both the `'aeut'` and `'aete'` resources.

The sections that follow provide examples of `'aete'` resources that can be generated with the tools in the *AppleScript Software Developers' Kit*.

Supporting Standard Suites Without Extensions

To indicate that your application supports a standard suite in its entirety, without any extensions, your `'aete'` resource needs to provide only the information that identifies the suite. For example, Listing 8-2 shows the Rez input for an `'aete'` resource provided by an application that supports the entire Required and Core suites with no omissions or extensions.

Every `'aete'` resource must provide the major and minor version numbers for the content of the resource (1 and 0 in Listing 8-2) and the language code (English in Listing 8-2). For each suite that an application supports in its entirety, without extensions, the `'aete'` resource provides only the name, suite description, suite ID, suite level (1 for all current suites), and suite version (1 for all current suites). If the `'aete'` resource provides an empty string as the human-language name for such a

suite, a scripting component uses the name provided for the corresponding suite by the `'aeut'` resource. If an application does not extend or omit any of the definitions in a standard suite, a scripting component can get the rest of the information about the suite—its event and object class definitions, comparison operators, and enumerated groups—from the `'aeut'` resource. The corresponding arrays in the `'aete'` resource can therefore be left empty.

Note that the Rez input for resources does not include the `align word` fields shown in the `'aeut'` resource type declaration in Listing 8-1. Rez takes care of word alignment automatically.

Listing 8-2   Rez input for an `'aete'` resource for an application that supports the Required and Core suites in their entirety

```
resource 'aete' (0, "JustTwoSuites") {
   1,                                        /*major version in BCD*/
   0,                                        /*minor version in BCD*/
   english,                                  /*language code*/
   roman,                                    /*script code*/
   {  /*array Suites: 2 elements*/
      /*[1]*/
      "",                                    /*human-language name for suite; */
                                             /* 'aeut' supplies "Required Suite"*/
      "Events that every application should support", /*suite description*/
      kAERequiredSuite,                      /*suite code*/
      1,                                     /*suite level*/
      1,                                     /*suite version*/
      {  /*array Events: 0 elements*/
      },
      {  /*array Classes: 0 elements*/
      },
      {  /*array ComparisonOps: 0 elements*/
      },
      {  /*array Enumerations: 0 elements*/
      },
      /*[2]*/
      "",                                    /*human-language name for suite; */
                                             /* 'aeut' supplies "Core Suite"*/
      "Suite that applies to all applications", /*suite description*/
      kAECoreSuite,                          /*suite code*/
      1,                                     /*suite level*/
      1,                                     /*suite version*/
      {  /*array Events: 0 element*/
      },
```

```
    {   /*array Classes: 0 elements*/
    },
    {   /*array ComparisonOps: 0 elements*/
    },
    {   /*array Enumerations: 0 elements*/
    }
  }
};
```

Extending the Standard Suites

If, like the `'aete'` resource shown in Listing 8-2, your application's `'aete'` resource indicates that you support an entire standard suite, the scripting component automatically makes use of all the terminology for that suite provided by its `'aeut'` resource. For this reason, you can easily extend the definitions for a suite that your application supports in its entirety: just provide arrays in the `'aete'` resource for the definitions not already included in the `'aeut'` resource. For example, if you're extending the definition of an event to support a single additional parameter, you should provide an array of parameters that includes one item: the description of the new parameter. Similarly, if you're not extending the contents of a particular array, you don't need to include the array in the `'aete'` resource.

Although an array of descriptions in an `'aete'` resource need not include descriptions that are already provided by the `'aeut'` resource, you must include information that defines the position of the array in relation to the other information in the `'aete'` resource. As Table 8-3 on page 8-8 shows, you can nest the arrays in an `'aete'` resource within other arrays: for example, an array of parameters is part of the description of an event, and the event description is, in turn, part of the array of event descriptions for a suite.

To add a description of a single new parameter to the definition of an Apple event in a suite that your application supports in its entirety, you need to provide

n   an array of parameters containing one element: the description of the new parameter

n   information that identifies the event definition to which you're adding the parameter

n   information that identifies the suite containing the event

Listing 8-3 illustrates how this works. This Rez input adds two new parameters ("number of copies" and "print quality") to the Print Documents event in the Required suite, one enumeration (three values for the "print quality" parameter of the Print Documents event) to the Required suite, and a new property ("first indent") to the cParagraph class in the Text suite. It also adds a plural synonym for the cParagraph class: the word "paragraphs."

Listing 8-3 Rez input for an `'aete'` resource that extends the definitions of the Required, Core, and Text suites

```
#define keyMyNumberOfCopies    'numc'
#define keyMyPrintQuality      'prtq'
#define typePrintQuality       'pqen'
#define kFast                  'fast'
#define kNormal                'nrml'
#define kHighQuality           'hiql'
#define pFirstIndent           'indt'

resource 'aete' (0, "SuiteExtensions") {
   1,                                   /*major version in BCD*/
   0,                                   /*minor version in BCD*/
   english,                             /*language code*/
   roman,                               /*script code*/
   {  /*array Suites: 3 elements*/
      /*[1]*/
      "",                               /*human-language name for suite; */
                                        /* 'aeut' supplies "Required Suite"*/
      "Events that every application should support", /*suite description*/
      kAERequiredSuite,                 /*suite code*/
      1,                                /*suite level*/
      1,                                /*suite version*/
      {  /*array Events: 1 element*/
         /*[1]*/
         "",                            /*human-language name for event; */
                                        /* 'aeut' supplies "Print Documents"*/
         "Print the specified list of documents",    /*event description*/
         kCoreEventClass,               /*event class*/
         kAEPrintDocuments,             /*event ID*/
         noReply,                       /*reply type*/
         "",                            /*reply description*/
         replyOptional,                 /*reply is optional*/
         singleItem,                    /*reply must be single item*/
         notEnumerated,                 /*type is not enumerated*/
         reserved,                      /*these 13 bits are reserved*/
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
```

```
reserved,
reserved,
reserved,
reserved,
reserved,
typeAlias,                          /*direct parameter type*/
"List of documents to print",      /*direct parameter description*/
directParamRequired,               /*direct parameter is required*/
listOfItems,                       /*direct parameter must be list*/
notEnumerated,                     /*type is not enumerated*/
doesntChangeState,                 /*event does not change */
                                   /* server's state*/
reserved,                          /*these 12 bits are reserved*/
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
{  /*array OtherParams: these fields describe the additional */
   /* parameters Number of Copies and Print Quality to the */
   /* standard Print Documents event*/
   /*[1]*/
   "number of copies",         /*human-language name for parameter*/
   keyMyNumberOfCopies,        /*parameter keyword*/
   typeShortInteger,           /*parameter type*/
   "Number of copies to print", /*parameter description*/
   optional,                   /*parameter is optional*/
   singleItem,                 /*parameter must be single item*/
   notEnumerated,              /*type is not enumerated*/
   reserved,                   /*these 13 bits are reserved*/
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
```

```
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         /*[2]*/
         "print quality",            /*human-language name for parameter*/
         keyMyPrintQuality,          /*parameter keyword*/
         typePrintQuality,           /*parameter type*/
         "The quality of the printing",/*parameter description*/
         optional,                   /*parameter is optional*/
         singleItem,                 /*parameter must be single item*/
         enumerated,                 /*type is enumerated*/
         reserved,                   /*these 13 bits are reserved*/
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved
      }
   },
   {  /*array Classes: 0 elements*/
   },
   {  /*array ComparisonOps: 0 elements*/
   },
   {  /*array Enumerations: 1 element*/
      /*these fields add the 'pqen' enumeration to the Required suite*/
      /*[1]*/
      typePrintQuality,                 /*enumeration ID*/
      {  /*array Enumerators: 3 elements*/
         /*[1]*/
         "Fast",                        /*enumerator name*/
         kFast,                         /*enumerator ID*/
         "Print as quickly as possible",/*enumerator description*/
         /*[2]*/
         "Normal",                      /*enumerator name*/
```

```
      kNormal,                        /*enumerator ID*/
      "Print at normal speed",        /*enumerator description*/
      /*[3]*/
      "High-Quality",                 /*enumerator name*/
      kHighQuality,                   /*enumerator ID*/
      "Print at highest quality possible" /*enumerator description*/
   }
},
/*[2]*/
"",                                   /*human-language name for suite; */
                                      /* 'aeut' supplies "Core Suite"*/
"Suite that applies to all applications",       /*suite description*/
kAECoreSuite,                         /*suite code*/
1,                                    /*suite level*/
1,                                    /*suite version*/
{  /*array Events: 0 elements*/
},
{  /*array Classes: 0 elements*/
},
{  /*array ComparisonOps: 0 elements*/
},
{  /*array Enumerations: 0 elements*/
},
/*[3]*/
"",                                   /*human-language name for suite; */
                                      /* 'aeut' supplies "Text Suite"*/
"A set of basic classes for text processing",   /*suite description*/
kAETextSuite,                         /*suite code*/
1,                                    /*suite level*/
1,                                    /*suite version*/
{  /*array Events: 0 elements*/
},
{  /*array Classes: 1 element*/
   /*[1]*/
   "paragraph",                       /*human-language name for class*/
   cParagraph,                        /*class ID*/
   "A paragraph",                     /*class description*/
   {  /*array Properties: 1 element*/
      /*[1]*/
      "first indent",             /*human-language name for property*/
      pFirstIndent,               /*property ID*/
      cLongInteger,               /*property class*/
      "First indent of paragraph in points",/*property description*/
```

```
   reserved,                            /*reserved*/
   singleItem,                          /*property is single item*/
   notEnumerated,                       /*type is not enumerated*/
   readWrite,                           /*property can be modified*/
   reserved,                            /*these 12 bits are reserved*/
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved
},
{  /*array Elements: 0 elements*/
},
"paragraphs",                      /*human-language name for class*/
cParagraph,                        /*class ID*/
"Every paragraph",                 /*class description*/
{  /*array Special Properties: 1 element*/
   /*[1]*/
   "",                                  /*human-language name for property*/
   kAESpecialClassProperties,    /*property ID*/
   cType,                               /*property class*/
   "",                                  /*property description*/
   reserved,                       /*reserved*/
   singleItem,                     /*property is single item*/
   notEnumerated,                  /*type is not enumerated*/
   readOnly,                       /*property cannot be modified*/
   reserved,                       /*these 11 bits are reserved*/
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
```

```
            plural                              /*human-language name is */
                                                /* plural form*/
        },
        {  /*array Elements: 0 elements*/
        },
    },
    {  /*array ComparisonOps: 0 elements*/
    },
    {  /*array Enumerations: 0 elements*/
    }
  }
};
```

In Listing 8-3, the possible values for the "print quality" parameter belong to an enumeration. This is indicated by the term enumerated in the parameter description. For this reason, the parameter type field contains the ID for the enumeration—typePrintQuality.

Listing 8-3 also adds a plural synonym for "paragraph" to the array of classes: the word "paragraphs." Note that this is listed as if it were an additional class, except that it also specifies cParagraph as the class ID. The first property listed for the synonym has property ID kAESpecialClassProperties. This property describes characteristics of the class as a whole; the last flag bit for this property is set to plural, indicating that the term paragraphs is a plural term for the specified class. This property must always be the first property listed for a class. For more information about the kAESpecialClassProperties property, see "Property Data," which begins on page 8-38.

An enumeration is described only by its ID; its declaration does not include a name or description field. However, a name, value, and description must be provided for each of the enumerators in an enumeration.

You can use the method illustrated in Listing 8-3 only to add to the definitions of Apple events and Apple event object classes, not to support subsets of them. For example, to support only a subset of the parameters of an Apple event or only some of the elements or properties of an existing object class, you must list all the definitions from that suite that you do support. The next section, "Supporting Subsets of Suites," provides more information about how to do this.

Human-language names for Apple events, object classes, and so on (including extensions) can include both uppercase and lowercase letters and spaces. For comparison purposes, case doesn't matter. However, note that the human-language names defined in Listing 8-3 are all lowercase. This convention ensures that scripts in which these terms appear won't have capital letters in unexpected places.

Scripting components that get identifiers or strings from user terminology resources are free to change the identifiers or strings as necessary (eliminating spaces, converting identifiers to all uppercase or lowercase, or changing the identifiers altogether) to meet the requirements of a particular task.

Supporting Subsets of Suites

Your application is not required to support all the definitions in a suite. If you wish to support a subset of the definitions in one or more standard suites, you can collect individual definitions from any number of suites in a placeholder suite whose suite ID is the application's signature or `typeWildCard` (`'****'`). When you support a subset of a suite, you must provide all the definitions you want to support in your `'aete'` resource.

Supporting New Suites

If your application defines its own custom Apple events or other Apple event constructs, you should include a separate suite section for the suite in the `'aete'` resource. You should use your application's signature for both the suite ID and the class ID of all events in the suite.

Handling the Get AETE Event

A scripting component sends the Get AETE event to an application when it needs information about the user terminology specified by the application. For example, the AppleScript component sends the Get AETE event when it first attempts to compile a `tell` statement that specifies a particular application. If your application does not handle the Get AETE event, the scripting component reads the terminology information it needs directly from your application's `'aete'` resource. Applications that support additional plug-in modules, each with its own `'aete'` resource, must provide an `'scsz'` resource and a handler for the Get AETE event that collects the `'aete'` resources from the modules that are currently running.

If your application does provide separate plug-in modules, the Get AETE event allows it to gather information from the `'aete'` resources for the modules that are currently running and return the terminology information along with your application's built-in terminology information to the scripting component in the reply event.

Here is a summary of the structure of a Get AETE event:

Get AETE—Get an application's `'aete'` resource

Event class	`kASAppleScriptClass`
Event ID	`kGetAETE`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeInteger`
Data:	**Language code**
Required reply parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeAEList` or `typeAETE`
Data:	**The application's terminologies**
Description	**Sent by a scripting component to an application when the scripting component needs information about the application's user terminology**

Your application can't handle the Get AETE event unless it is running. If your application doesn't provide a handler for the Get AETE event, the scripting component can obtain terminology information directly from your application's `'aete'` resource even if your application is not running.

If your application handles the Get AETE event, it must also provide a scripting size resource. A scripting size resource is a resource of type `'scsz'` that provides information about an application's capabilities for use by scripting components. It allows your application to declare whether it needs the Get AETE event and to specify preferences for the sizes of the portion of your application's heap used by a scripting component. For information about the `'scsz'` resource, see "The Scripting Size Resource" on page 8-45.

A handler for the Get AETE event should perform the following tasks:

n Obtain the language code specified by the event.

n Create a descriptor list to hold the `'aete'` resources.

n Collect the `'aete'` resources from all the application's plug-in modules that are currently running, including the application itself, and add them to the list.

n Add the list to the reply Apple event.

Listing 8-4 provides an example of a handler for the Get AETE event.

Listing 8-4   A handler for the Get AETE event

```
FUNCTION MyGetAETE (theAE: AppleEvent; theReply: AppleEvent;
                       refCon: LongInt): OSErr;
VAR
   theList:       AEDescList;
   returnedType:  DescType;
   actualSize:    Size;
   languageCode:  Integer;
   myErr:         OSErr;
BEGIN
   MyGetAETE := errAEEventNotHandled;
   languageCode := 0;
   {if a reply was not requested, then don't handle}
   IF theReply.dataHandle = NIL THEN
      Exit(MyGetAETE);
   {get the language code that AppleScript is requesting so that }
   { this function can return the aete of a specified language}
   myErr := AEGetParamPtr(theAE, keyDirectObject,
                          typeLongInteger, returnedType,
                          @languageCode, sizeOf(LongInt),
                          actualSize);
   IF myErr <> noErr THEN
      Exit(MyGetAETE);
   {create a list}
   myErr := AECreateList(NIL, 0, FALSE, theList);
   IF myErr <> noErr THEN
      Exit(MyGetAETE);
   {get the requested 'aete' resources and put in the list--the }
   { MyGrabAETE application-defined function does this}
   {your code should iterate all of your installed code }
   { extensions and add the aete for each that matches the }
   { language code requested}
   myErr := MyGrabAETE(languageCode, theList);
   IF myErr <> noErr THEN
   BEGIN
      myErr := AEDisposeDesc(theList);
      Exit(MyGetAETE);
   END;
   {add list to reply Apple event}
   myErr := AEPutParamDesc(theReply, keyDirectObject, theList);
   myErr := AEDisposeDesc(theList);
   myGetAETE := myErr;
END;
```

The `MyGetAETE` handler in Listing 8-4 begins by setting the function result to `errAEEventNotHandled`. The function is set to this result if for any reason the handler doesn't successfully handle the event, so that a system handler provided by the scripting component can at least read the terminology information directly from the application's own `'aete'` resource. The handler in Listing 8-4 then checks the language code specified by the event. After checking to make sure the reply exists, the handler creates a list and uses the application-defined function `MyGrabAETE` to collect all the appropriate terminology information and append it to the list. The `MyGetAETE` handler then adds the list to the reply event.

Reference to Apple Event Terminology Resources

Listing 8-1 on page 8-9 shows the complete resource type declaration in Rez format for the `'aeut'` resource. The same resource structure is used by both the `'aeut'` and `'aete'` resources. Figure 8-1 shows the format of a compiled `'aeut'` or `'aete'` resource.

Figure 8-1   Structure of an `'aeut'` or `'aete'` resource
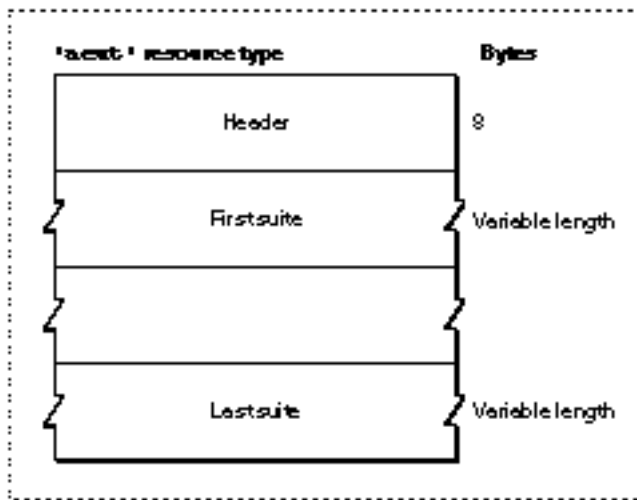
An `'aeut'` or `'aete'` resource contains the following:

n   a header containing the version and language code of the template and a count of the number of suites the resource describes

n   a variable number of suite descriptions

The sections that follow describe the content of the header and each suite description in detail.

Header Data for an Apple Event Terminology Resource

The header for an 'aeut' or 'aete' resource specifies the version of its contents, the language of the human-language equivalents contained in the resource, a script code, and a count of the number of suites the resource describes. Figure 8-2 shows the header format.

Figure 8-2   Structure of the header data in an 'aeut' or 'aete' resource

The header contains the following items:

n   The major version number of the content of the resource in binary-coded decimal (the major version number for the first release of the 'aeut' resource is 1). The major and minor versions describe the content of the resource, not its template. You can use these fields to provide version numbers for the content of your application's 'aete' resource.

n   The minor version number of the template in binary-coded decimal (the minor version number for the first release of the 'aeut' resource is 0).

n   The language code for the resource. *Inside Macintosh: Text* provides a list of language codes. This code must be the same as the resource ID for the resource.

n   The script code for the resource, taken from the list of script codes provided in *Inside Macintosh: Text.*

n   A count of the number of suites described by the resource.

Suite Data for an Apple Event Terminology Resource

Each item in the array of suites for an 'aeut' or 'aete' resource includes information about the suite ID, level, and version and four arrays that specify the events, object classes, comparison operators, and enumerations for that suite. Figure 8-3 shows the format of this suite data.

Figure 8-3   Structure of suite data in an `'aeut'` or `'aete'` resource
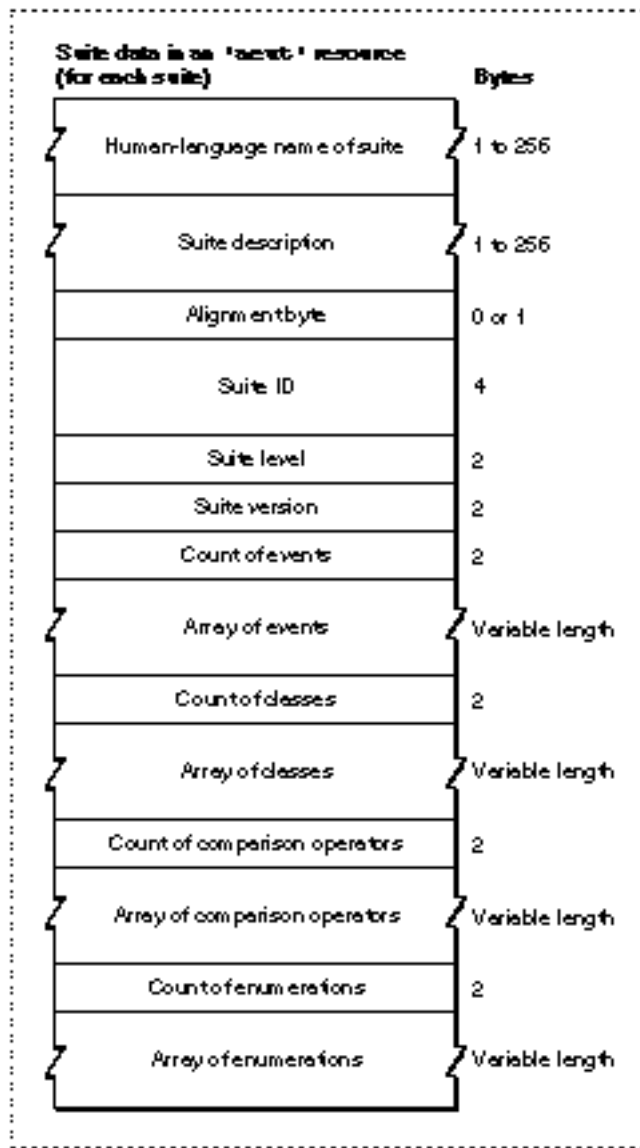
The data for each suite consists of the following items:

n The human-language name of the suite. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. If the `'aete'` resource specifies the name as an empty string, the scripting component looks up, in its `'aeut'` resource, the suite name and other suite data that correspond to the specified suite ID, suite level, and suite version. This strategy simplifies specification of an entire suite and facilitates localization, since the human-language name is provided by the `'aeut'` resource.

If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same suite ID, suite level, and suite version, the scripting component uses the new name with the same suite data from the `'aeut'` resource. Unless you are defining a custom suite, you should specify an empty string for the name of a suite.

n A human-language description of the suite. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

n A four-character ID that distinguishes the suite from all other suites defined in either the `'aeut'` or `'aete'` resources. This value is normally the same as the event class for the Apple events in the suite.

If the `'aete'` resource specifies a standard suite name but a suite ID that is different from the suite ID for the standard suite of that name described in the `'aeut'` resource, the scripting component uses the new suite ID with the standard suite data for the specified name. In general, you should use the standard suite ID for any standard suite that you support.

If your application uses a custom suite, you should use your application's signature as the event class for the events in the suite and, in addition, as its suite ID. When you register your application's signature with Developer Technical Support, the corresponding event class is automatically registered for your application, and only you can register events that belong to that event class. For information about registering Apple events, contact the Apple Event Registrar.

n The level and version of the suite. For the first version of any suite, the level is usually 1 (indicating that it is the suite that contains the most basic definitions) and the version is 1 (the version of this suite level). More advanced suites (such as a suite for performing more sophisticated text manipulation than the current Text suite allows) will have level numbers greater than 1. All currently defined suites have a level of 1 and a version of 1.

n A count of the events defined for this suite and an array of event definitions.

n A count of the object classes defined for this suite and an array of class definitions.

n A count of the comparison operators defined for this suite and an array of comparison operator definitions.

n A count of the enumerations defined for this suite and an array of enumeration definitions.

Event Data

Each item in the array of events for a suite specified in an `'aeut'` or `'aete'` resource includes information about the event, the reply, and the direct parameter, and an array that specifies the additional parameters for the event. Figure 8-4 shows the format of this event data.

Figure 8-4 Structure of event data in an `'aeut'` or `'aete'` resource
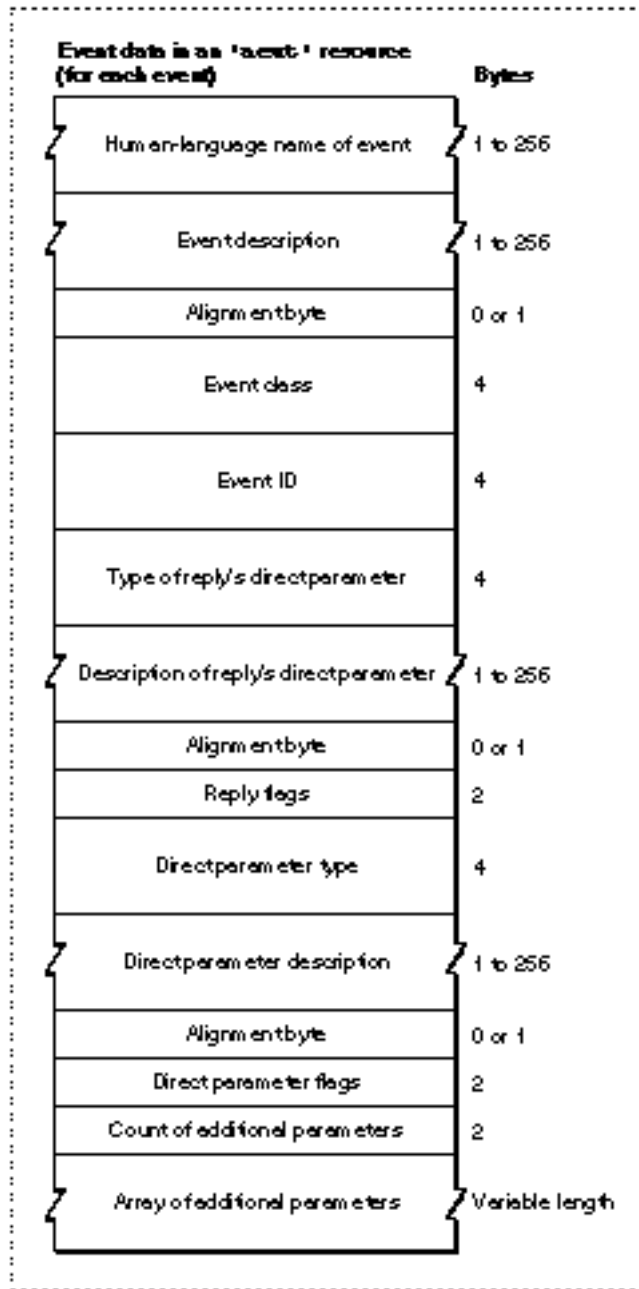
The data for each event consists of the following items:

n   The human-language name of the event. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. If the 'aete' resource specifies the name as an empty string, the scripting component looks up, in its 'aeut' resource, the event name and other event data that correspond to the specified event class and event ID. This strategy facilitates localization, since the human-language name is provided by the 'aeut' resource. In this case the scripting component will use the standard data from the 'aeut' resource for the event plus the data provided by the 'aete' resource for any additional parameters.

If the 'aete' resource specifies a name other than the name provided by the 'aeut' resource for the same event class and event ID, the scripting component uses the new name with the same suite data from the 'aeut' resource. You should specify an empty string for the name of any standard event that your application lists explicitly in its 'aete' resource.

n   A human-language description of the event. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

n   The four-character event class for the event. If the 'aete' resource specifies a standard event name and an event class other than the event class for the equivalent standard event, the scripting component uses the new event class with the standard event data for the specified name. You should specify the standard event class for any standard event that your application lists explicitly in its 'aete' resource.

n   The four-character event ID for the event. If the 'aete' resource specifies a standard event name and an event ID other than the event ID for the equivalent standard event, the scripting component uses the new event ID with the standard event data for the specified name. You should specify the standard event ID for any standard event that your application lists explicitly in its 'aete' resource.

n   A four-character descriptor type for the direct parameter of the reply. If the event never needs a reply, or if the reply does not include a direct parameter, this value must be typeNull. Otherwise, the meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the parameter is a list (singleItem or listOfItems), and the other specifies whether the values for the parameter are enumerated (enumerated or notEnumerated):

    n   If the parameter is not a list and its values are not enumerated, this value is the descriptor type for the direct parameter.

    n   If the parameter is a list and its values are not enumerated, this value is the descriptor type for each of the items in the list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value singleItem, and the value of this field must be typeAEList.)

n If the parameter is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the `'aete'` or `'aeut'` resource that contains the allowable values for the parameter. (If the values are enumerated but the enumeration is not defined in either the `'aete'` or `'aeut'` resource, the flag specifying whether the parameter's values are enumerated must have the value `notEnumerated`, and the value of this field must be `typeEnumerated`.)

n If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. All items in the list must have one of these enumerated values.

n A human-language description of the direct parameter of the reply. This is a Pascal string that can include any characters. Although the reply may include other parameters, only the direct parameter of the reply is described here. When the resource description is compiled, the resource compiler aligns the string on a word boundary.

n Flags that specify the following as Boolean values:

n Whether the direct parameter of the reply is required (`replyRequired`) or optional (`replyOptional`).

n Whether the direct parameter of the reply is a single item (`singleItem`) or a list of items (`listOfItems`). (See the earlier description of the reply event's four-character descriptor type for information about how this value changes the meaning of the reply type.)

n Whether named constants, called enumerators, are specified as the only valid values for the direct parameter of the reply (`enumerated` or `notEnumerated`). (See the earlier description of the four-character descriptor type for the reply event's direct parameter for information about how this value changes the meaning of the direct parameter type.) For information about specifying enumerators, see "Enumeration and Enumerator Data" on page 8-43.

n Following 5 bits are reserved for future use. The values of these bits must be set to `reserved`.

n Following 7 bits are reserved for future use as dialect-specific flags. The values of these bits must be set to `reserved`.

n Whether the event is a nonverb event (`nonVerbEvent`). This bit is used by dialects such as the AppleScript Japanese dialect that make this distinction. For all other dialects, set the value of this bit to `reserved`.

n A four-character descriptor type for the direct parameter of the event. If the event never has a direct parameter, this value must be `typeNull`. Otherwise, the meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the parameter is a list (`singleItem` or `listOfItems`), and the other specifies whether the values for the parameter are enumerated (`enumerated` or `notEnumerated`):

- n If the parameter is not a list and its values are not enumerated, this value is the descriptor type for the direct parameter.
- n If the parameter is a list and its values are not enumerated, this value is the descriptor type for each of the items in the list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value `singleItem`, and the value of this field must be `typeAEList`.)
- n If the parameter is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the `'aete'` or `'aeut'` resource that contains the allowable values for the parameter. (If the values are enumerated but the enumeration is not defined in either the `'aete'` or `'aeut'` resource, the flag specifying whether the parameter's values are enumerated must have the value `notEnumerated`, and the value of this field must be `typeEnumerated`.)
- n If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. The values of the items in the list must all be one of these enumerated values.

n A human-language description of the direct parameter. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

n Flags that specify the following as Boolean values:
  - n Whether the direct parameter of the event is required (`directParamRequired`) or optional (`directParamOptional`).
  - n Whether the direct parameter of the event is a single item (`singleItem`) or a list of items (`listOfItems`). (See the earlier description of the direct parameter's four-character descriptor type for information about how this value changes the meaning of the direct parameter type.)
  - n Whether named constants, called enumerators, are specified as the only valid values for the direct parameter (`enumerated` or `notEnumerated`). (See the earlier description of the direct parameter's four-character descriptor type for information about how this value changes the meaning of the direct parameter type.) For information about specifying enumerators, see "Enumeration and Enumerator Data" on page 8-43.
  - n Whether receiving this event changes (`changesState`) or doesn't change (`doesntChangeState`) the internal state of the receiving application. Events that only get information do not change the state of the application, whereas events such as Cut and Move do.
  - n Following 4 bits are reserved for future use. The values of these bits must be set to `reserved`.
  - n Following 8 bits are reserved for future use as dialect-specific flags. The values of these bits must be set to `reserved`.

n A count of the additional parameters described for this event and an array of additional parameter definitions.

Additional Parameter Data

Each item in the array of additional parameters for an event specified in an `'aeut'` resource includes information about a single additional parameter. Figure 8-5 shows the format of additional parameter data in an `'aeut'` or `'aete'` resource.

Figure 8-5   Structure of additional parameter data in an `'aeut'` or `'aete'` resource
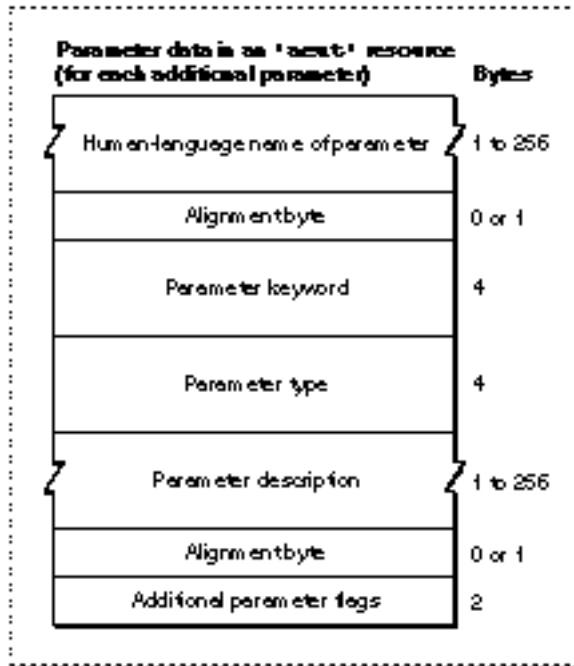
The data for each additional parameter consists of the following items:

n   The human-language name of the parameter. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

If the `'aete'` resource specifies the name of an additional parameter as an empty string, the scripting component looks up, in its `'aeut'` resource, the parameter name and other parameter data that correspond to the specified parameter keyword. If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same parameter keyword, the scripting component uses the new name with the same parameter data from the `'aeut'` resource. You should specify an empty string for the name of any standard additional parameter that you list explicitly in an `'aete'` resource.

n The four-character keyword for the parameter. If the `'aete'` resource specifies a standard parameter name and a parameter keyword other than the keyword for the equivalent standard parameter, the scripting component uses the new parameter keyword with the standard parameter data for the specified name. You should specify the standard parameter keyword for any standard additional parameter that you list explicitly in an `'aete'` resource.

n A four-character descriptor type for the parameter. The meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the parameter is a list (`singleItem` or `listOfItems`), and the other specifies whether the values for the parameter are enumerated (`enumerated` or `notEnumerated`):

   n If the parameter is not a list and its values are not enumerated, this value is the descriptor type for the direct parameter.

   n If the parameter is a list and its values are not enumerated, this value is the descriptor type for each of the items in the list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value `singleItem`, and the value of this field must be `typeAEList`.)

   n If the parameter is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the `'aete'` or `'aeut'` resource that contains the allowable values for the parameter. (If the values are enumerated but the enumeration is not defined in either the `'aete'` or `'aeut'` resource, the flag specifying whether the parameter's values are enumerated must have the value `notEnumerated`, and the value of this field must be `typeEnumerated`.)

   n If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. The values of the items in the list must all be one of these enumerated values.

n A human-language description of the parameter. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

n Flags that specify the following as Boolean values:

   n Whether the parameter is required (`required`) or optional (`optional`).

   n Whether the parameter is a single item (`singleItem`) or a list of items (`listOfItems`). (See the earlier description of the additional parameter's four-character descriptor type for information about how this value changes the meaning of the parameter type.)

   n Whether named constants, called enumerators, are specified as the only valid values for the parameter (`enumerated` or `notEnumerated`). (See the earlier description of the parameter's four-character descriptor type for information about how this value changes the meaning of the parameter type.) For information about specifying enumerators, see "Enumeration and Enumerator Data" on page 8-43.

n   Whether the parameter is the event's only unnamed parameter (isUnNamed) or is named (isNamed). This bit is used by dialects such as AppleScript Japanese that make this distinction. For all other dialects, set the value of this bit to reserved.

n   Following 4 bits are reserved for future use. The values of these bits must be set to reserved.

n   Following 8 bits are reserved for future use as dialect-specific flags. The values of these bits must be set to reserved.

"Extending the Standard Suites," which begins on page 8-16, includes sample Rez input for an 'aete' resource that adds new parameters to a standard Apple event.

Object Class Data

Each item in the array of object classes for a suite includes information about the class and arrays that specify the properties and elements for that class. Figure 8-6 shows the format of the object class data in an 'aeut' or 'aete' resource.

Figure 8-6   Structure of object class data in an 'aeut' or 'aete' resource
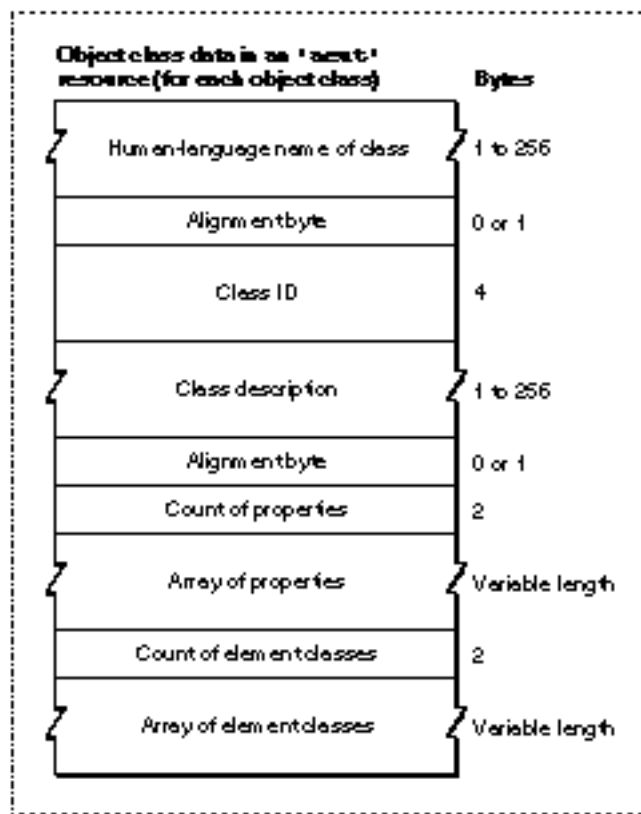
The data for each object class consists of the following items:

n   The human-language name of the object class. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

If the `'aete'` resource specifies the name of an object class as an empty string, the scripting component looks up, in its `'aeut'` resource, the class name and other object class data that correspond to the specified class ID. If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same class ID, the scripting component uses the new name with the same object class data from the `'aeut'` resource. You should specify an empty string for the name of any standard object class that you list explicitly in an `'aete'` resource.

n   The four-character class ID for the object class. If the `'aete'` resource specifies a standard object class name and a class ID other than the class ID for the equivalent standard object class, the scripting component uses the new class ID with the standard object class data for the specified name. You should specify the standard class ID for any standard object class that you list explicitly in an `'aete'` resource.

n   A human-language description of the class. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

n   A count of the properties described for this class and an array of property definitions.

n   A count of the element classes described for this class and an array of element class definitions.

To define characteristics of an object class (for instance, whether an object of that class is a single item or a list of items, whether it is singular or plural, and so on), your application's `'aete'` resource must define a special property of property ID `kAESpecialClassProperties` as the first property in the array of properties. Because object class data does not include flag bits, the flag bits of this property are used to specify attributes for the class to which the property belongs. The next section describes how this property is defined and used.

Property Data

Each item in the array of properties for an object class includes information about a single property. Figure 8-7 shows the format of the property data in an `'aeut'` or `'aete'` resource.

Figure 8-7   Structure of property data in an `'aeut'` or `'aete'` resource
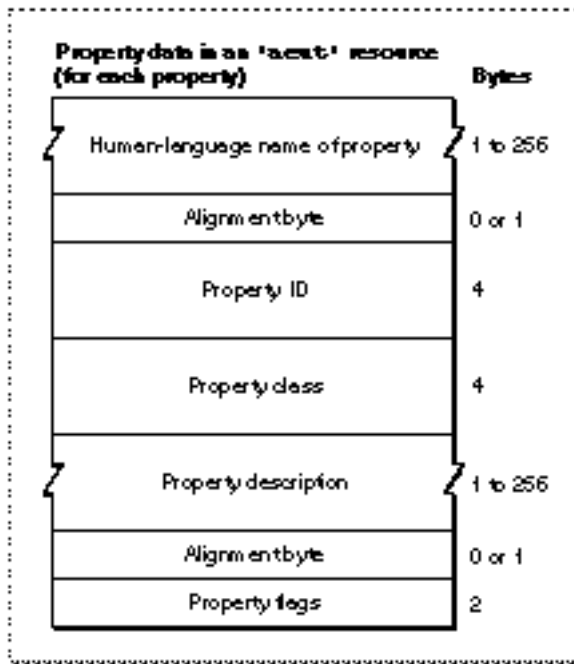
The data for each property consists of the following items:

n   The human-language name of the property. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

If the `'aete'` resource specifies the name of a property as an empty string, the scripting component looks up, in its `'aeut'` resource, the property name and other property data that correspond to the specified property ID. If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same property ID, the scripting component uses the new name with the same property data from the `'aeut'` resource. You should specify an empty string for the name of any standard property that you list explicitly in an `'aete'` resource.

n   The four-character property ID for the property. If the `'aete'` resource specifies a standard property name and a property ID other than the property ID for the equivalent standard property, the scripting component uses the new property ID with the standard property data for the specified name. You should specify the standard property ID for any standard property that you list explicitly in an `'aete'` resource.

n   A four-character class ID for the object class to which the property belongs. The meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the property is a list (`singleItem` or `listOfItems`), and the other specifies whether the values for the parameter are enumerated (`enumerated` or `notEnumerated`):

   n   If the property is not a list and its values are not enumerated, this value is the class ID for the property.

   n   If the property is a list and its values are not enumerated, this value is the class ID for each of the items in the list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value `singleItem`, and the value of this field must be `cAEList`.)

   n   If the property is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the `'aete'` or `'aeut'` resource that contains the allowable values for the property. (If the values are enumerated but the enumeration is not defined in either the `'aete'` or `'aeut'` resource, the flag specifying whether the property's values are enumerated must have the value `notEnumerated`, and the value of this field must be `typeEnumerated`.)

   n   If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. The values of the items in the list must all be one of these enumerated values.

n   A human-language description of the property. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

n   Flags that specify the following as Boolean values:

   n   The first bit is reserved for future use. Its value must be set to `reserved`.

   n   Whether the property is a single item (`singleItem`) or a list of items (`listOfItems`). (See the earlier description of the property's four-character class ID for information about how this value changes the meaning of the class ID.)

   n   Whether named constants, called enumerators, are specified as the only valid values for the property (`enumerated` or `notEnumerated`). (See the earlier description of the property's four-character class ID for information about how this value changes the meaning of the class ID.) For information about specifying enumerators, see "Enumeration and Enumerator Data" on page 8-43.

- Whether the property's value can (`readWrite`) or cannot (`readOnly`) be set by the Set Data Apple event.
- Following 4 bits are reserved for future use.
- Following 5 bits are reserved for future use as dialect-specific flags.
- Whether the human-language name of the property is feminine (`feminine`) or not (`notFeminine`). This bit is used by dialects such as the AppleScript French dialect that make this distinction. For all other dialects, set the value of this bit to `reserved`.
- Whether the human-language name of the property is masculine (`masculine`) or not (`notMasculine`). This bit is used by dialects such as AppleScript French that make this distinction. For all other dialects, set the value of this bit to `reserved`.
- Whether the human-language name of the property is singular (`singular`) or plural (`plural`). This bit is used by dialects such as AppleScript French that make this distinction. If you set this bit to `reserved`, the scripting component will assign it the value `singular`.

"Extending the Standard Suites," which begins on page 8-16, includes sample Rez input for an `'aete'` resource that adds a new property to a standard object class.

The array of properties in an `'aeut'` resource begins with a definition of a special property that describes characteristics of the class as a whole using the flags in the definition of that property. A property used in this way to define characteristics of a class must be defined first in the array of properties for that class and must specify `kAESpecialClassProperties` (`'c@#!'`) as the property ID, `cType` as the property class, and an empty string for the property name and property description. If you don't define such a property for a class in your application's `'aete'` resource, the scripting component will assign that class the default values specified by the first constant for each flag bit in the Rez declaration for the `'aeut'` resource. (See Listing 8-1, which begins on page 8-9, for the `'aeut'` resource type declaration.)

Element Class Data

Each item in the array of elements for an object class includes information about a single element class and an array of key forms for that element class. Figure 8-8 shows the format of the object class data in an `'aeut'` or `'aete'` resource.

Figure 8-8   Structure of element class data in an `'aeut'` or `'aete'` resource
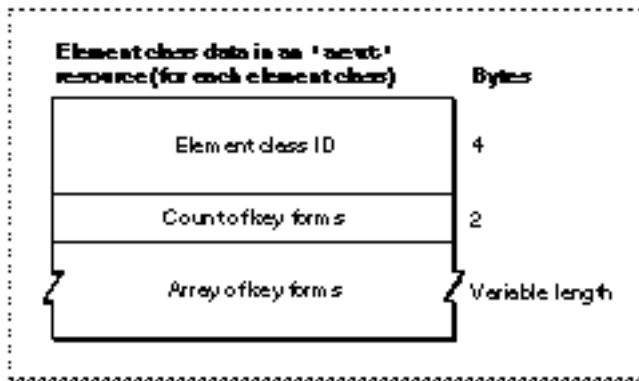
The following statements are included for each element class in the array of element classes for an object class:

n The four-character class ID for the element's object class.

n A count of key forms that apply to elements of this class within objects of the class for which these element classes are defined, followed by an array of key forms. Each item in the array must be a value from a special `'kfrm'` enumeration. (The `'aeut'` resource includes enumerators for the standard key forms defined in the *Apple Event Registry: Standard Suites;* an `'aete'` resource can contain `'kfrm'` enumerators for additional key forms that are specific to an application. For information about defining enumerators and enumerations, see "Enumeration and Enumerator Data" on page 8-43.) The enumerators for a `'kfrm'` enumeration can include `'indx'` (for the key form `formAbsolutePosition`), `'name'` (for the key form `formName`), `'ID  '` (for the key form `formUniqueID`), `'prop'` (for the key form `formPropertyID`), `'rang'` (for the key form `formRange`), `'rele'` (for the key form `formRelativePosition`), and `'test'` (for the key form `formTest`).

No names or descriptions are provided for element classes, because elements are specified by their object classes, and the declaration of each object class includes the name and description of the class.

Comparison Operator Data

Each item in the array of comparison operators for a suite includes information about a single comparison operator. Figure 8-9 shows the format of the comparison operator data in an `'aeut'` or `'aete'` resource.

Note

The AppleScript component currently doesn't use information about comparison operators. Other scripting components may use this information. u

Figure 8-9   Structure of comparison operator data in an `'aeut'` or `'aete'` resource
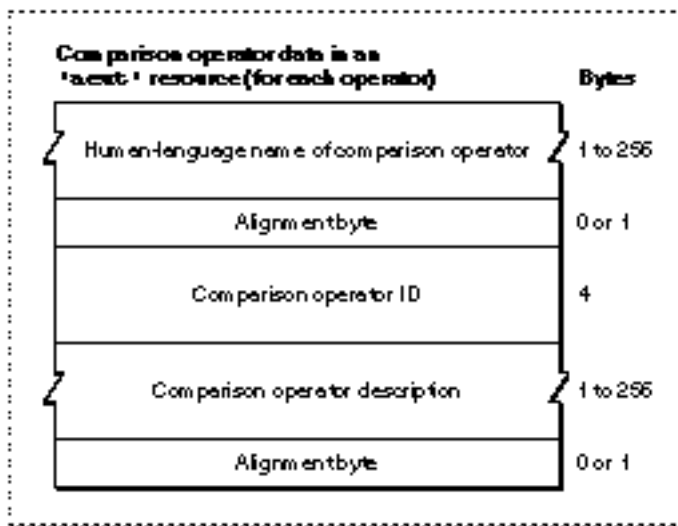
The data for each comparison operator consists of the following items:

n The human-language name of the comparison operator. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

If the `'aete'` resource specifies the name of a comparison operator as an empty string, the scripting component looks up, in its `'aeut'` resource, the comparison operator name and other comparison operator data that correspond to the specified comparison operator ID. If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same comparison operator ID, the scripting component uses the new name with the same comparison operator data from the `'aeut'` resource. You should specify an empty string for the name of any standard comparison operator that you list explicitly in an `'aete'` resource.

n   The four-character comparison operator ID for the property. If the `'aete'` resource specifies a standard comparison operator name and a comparison operator ID other than the comparison operator ID for the equivalent standard comparison operator, the scripting component uses the new comparison operator ID with the standard comparison operator data for the specified name. You should specify the standard comparison operator ID for any standard comparison operator that you list explicitly in an `'aete'` resource.

n   A human-language description of the comparison operator. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

"Extending the Standard Suites," which begins on page 8-16, includes sample Rez input for an `'aete'` resource that adds a comparison operator to a standard suite.

Enumeration and Enumerator Data

Each item in the array of enumerations for a suite includes information about a single enumeration and an array of enumerators for that enumeration.

Figure 8-10 shows the format of the enumeration data in an `'aeut'` or `'aete'` resource.

Figure 8-10   Structure of enumeration data in an `'aeut'` or `'aete'` resource
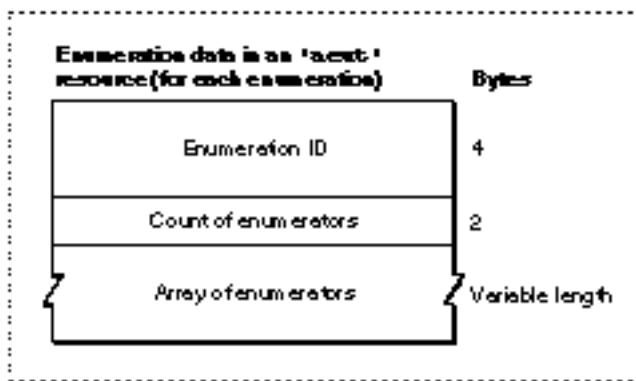
The data for each enumeration consists of the following items:

n   a four-character enumeration ID

n   a count of constants, known as enumerators, that specify the allowable values for the enumeration, and an array of enumerators

Figure 8-11 shows the format of the enumerator data.

Figure 8-11 Structure of enumerator data in an `'aeut'` or `'aete'` resource
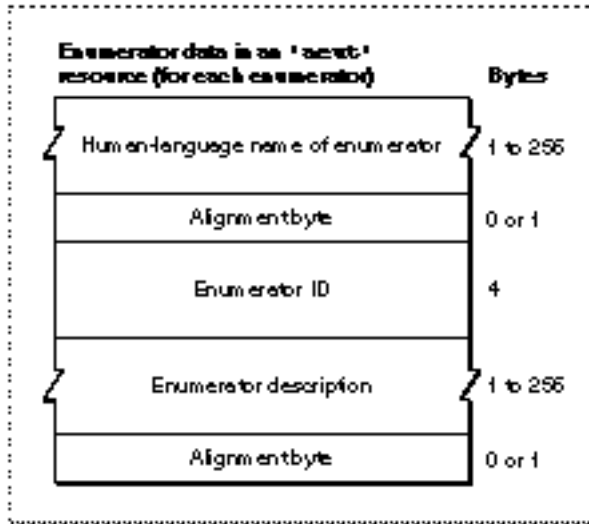
The data for each enumerator consists of the following items:

n The human-language name of the enumerator. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

If the `'aete'` resource specifies the name of an enumerator as an empty string, the scripting component looks up, in its `'aeut'` resource, the enumerator name and other enumerator data that correspond to the specified enumerator ID. If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same enumerator ID, the scripting component uses the new name with the same enumerator data from the `'aeut'` resource. You should specify an empty string for the name of any standard enumerator that you list explicitly in an `'aete'` resource.

n The four-character enumerator ID for the enumerator. If the `'aete'` resource specifies a standard enumerator name and an enumerator ID other than the enumerator ID for the equivalent standard enumerator, the scripting component uses the new enumerator ID with the standard enumerator data for the specified name. You should specify the standard enumerator ID for any standard enumerator that you list explicitly in an `'aete'` resource.

n A human-language description of the enumerator. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

"Extending the Standard Suites," which begins on page 8-16, includes sample Rez input for an `'aete'` resource that specifies an enumeration and an array of enumerators.

The Scripting Size Resource

If your application handles the Get AETE event, you must provide a scripting size resource. A scripting size resource is a resource of type `'scsz'` that provides information about an application's capabilities for use by scripting components. It also allows your application to specify preferences for the sizes of the portion of your application's heap used by a scripting component for its application-specific heap and stack.

Listing 8-5 shows the resource type declaration in Rez format for the `'scsz'` resource.

Listing 8-5   Resource type declaration for the `'scsz'` resource

```
type 'scsz' {
   boolean  dontReadExtensionTerms, /*if application needs */
            readExtensionTerms;      /* Get AETE event*/
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;

/*memory sizes are in bytes; 0 means use default*/
   unsigned longint  minStackSize;       /*minimum stack size*/
   unsigned longint  preferredStackSize; /*preferred stack size*/
   unsigned longint  maxStackSize;       /*maximum stack size*/
   unsigned longint  minHeapSize;        /*minimum heap size*/
   unsigned longint  preferredHeapSize;  /*preferred stack size*/
   unsigned longint  maxHeapSize;        /*maximum heap size*/
};
```

The data for an 'scsz' resource consists of the following items:

n Flags that specify Boolean values:

m Whether the scripting component should (readExtensionTerms) or shouldn't (dontReadExtensionTerms) read the application's terminology information directly from its 'aete' resource. If the application is not running, this flag allows a scripting component to determine whether it should read the application's terminology information without sending it a Get AETE event.

m The following 15 bits are reserved for future use. Their values must be set to reserved.

n The minimum size for the portion of the application's heap used by the scripting component's application-specific stack

n The preferred size for the portion of the application's heap used by the scripting component's application-specific stack

n The maximum size for the portion of the application's heap used by the scripting component's application-specific stack

n The minimum size for the portion of the application's heap used by the scripting component's application-specific heap

n The preferred size for the portion of the application's heap used by the scripting component's application-specific heap

n The maximum size for the portion of the application's heap used by the scripting component's application-specific heap

If you specify 0 for any of the fields that specify memory size or number of script IDs, the scripting component uses its own default values for those fields.

The AppleScript component provides a function, ASInit, that allows your application to initialize the component with desired values for memory sizes or number of script IDs. If your application doesn't call ASInit, the AppleScript component initializes itself using either the values specified in the application's 'scsz' resource or, for those values not provided by the 'scsz' resource, default values provided by the AppleScript component. For more information about ASInit, see "Initializing AppleScript" on page 10-80.

Recording Apple Events

Contents

This chapter describes the general characteristics of a recordable application and provides some examples of how to factor your application for recording. It also provides guidelines to help you decide which user actions to record and how to record them.

Before you read this chapter, you should read the chapter "Introduction to Scripting" in this book. To factor your application, you must know how to respond to Apple events, create and send Apple events, and resolve and create object specifier records. For comprehensive information about implementing Apple events, see the chapters "Introduction to Apple Events," "Responding to Apple Events," "Creating and Sending Apple Events," and "Resolving and Creating Object Specifier Records" in this book.

The first three sections in this chapter provide

n a description of the basic requirements for recordable applications

n examples of how to begin factoring your application

n guidelines for what to record

The fourth section describes how Apple event recording works. You need to read it only if you are developing a script editor, an application that can initiate recording, or a scripting component.

About Recordable Applications

A **recordable application** is an application that uses Apple events to report user actions to the Apple Event Manager for recording purposes. One way to do this is to separate the code that implements your application's user interface from the code that actually performs work when the user manipulates the interface. This is called **factoring** your application. A factored application translates low-level events generated by the user into recordable Apple events that the application sends to itself to perform tasks.

A **recordable event** is any Apple event that any recordable application sends to itself while recording is turned on for the local computer, with the exception of events that are sent with the kAEDontRecord flag set in the sendMode parameter of AESend. A **recording process** is any process (for example, the Script Editor application) that can turn recording on and off and can receive and record recordable Apple events.

After Apple event recording has been turned on by a recording process, the Apple Event Manager sends that process copies of all recordable Apple events on the local computer. For example, when a user presses the Record button in the Script Editor application, it calls a scripting component routine to turn on recording for the AppleScript component (or any other scripting component). While recording is on, the Apple Event Manager sends Script Editor copies of all subsequent recordable Apple events, which Script Editor records (with the aid of the scripting component) in the form of a compiled script. After turning off recording from Script Editor, the user can edit or execute the recorded script.

Although factoring your application is the recommended method of making your application recordable, it is also possible to report user actions by means of Apple events only when Apple event recording is turned on, even though the application may respond to those actions by some means other than Apple events. In effect, the application uses Apple events to describe user actions without actually using the events to perform the action. To indicate that you want the Apple Event Manager to send a copy of a recordable event to the recording process without actually sending the event to your application, add the constant `kAEDontExecute` to the `sendMode` parameter of the `AESend` function.

Even in a factored application, it may not always be possible to send an Apple event that actually executes the task initiated by the user. For example, if the user types some text, it is more practical to use standard TextEdit or QuickDraw routines to draw the text than to send a separate Apple event each time the user presses a key. In this case, the application can draw the text as it is typed in the most convenient manner available; then, when the user finishes typing a sequence of characters—by clicking the mouse button while the cursor is somewhere else in the document or performing some other action—the application can create an Apple event that corresponds to the typing and add the constant `kAEDontExecute` to the `sendMode` parameter when it sends the Apple event.

If your application needs to know when Apple event recording is turned on and off, it should install handlers for the Recording On and Recording Off events.

Recording On—perform actions associated with beginning of recording session

Event class	`kCoreEventClass`
Event ID	`kAENotifyStartRecording`
Parameters	None
Description	Sent by the Apple Event Manager to all running processes on the local computer to inform them that recording has been turned on

Recording Off—perform actions associated with end of recording session

Event class	`kCoreEventClass`
Event ID	`kAENotifyStopRecording`
Parameters	None
Description	Sent by the Apple Event Manager to all running processes on the local computer to inform them that recording has been turned off

When a recording process turns on recording, the Apple Event Manager sends all running processes on the local computer a Recording On event. When a user turns off recording, the Apple Event Manager sends all running processes the Recording Off event with the `kAEWaitReply` flag set. If an application has stored some data (for instance, keystrokes) that needs to be recorded as an Apple event, this is the last chance to send an event for recording purposes. If your application needs to know which recording process has turned recording on or off, it can check the `keyOriginalAddressAttr` attribute of the Recording On or Recording Off event for the address of the recording process.

Factoring your application is the recommended method of making your application recordable because it guarantees that any action a user can perform via your application's user interface can also be accomplished via Apple events. Factoring also allows you to avoid duplicating code within your application. Instead of using one piece of code to respond to some user action within your application, and another piece of code to respond to the equivalent Apple event, you can use the same code to respond to the Apple event, whether it is sent by your application in response to a user action, by some other application, or by a scripting component in the course of executing a script.

The next section, "Factoring Your Application for Recording," provides some examples of how to go about factoring your application. Regardless of how you factor your application, making it recordable requires you to make decisions about the most useful methods of recording user actions that can be described in several different ways in scripts. If a user moves a window, for example, the window can be described in the corresponding recorded script as `window 1`, or `the window named Fred`, or `the first window`. Although the OSA permits recording at a high level and thus avoids many of the problems users encounter with applications that record low-level events such as keystrokes and mouse clicks, scripting components cannot predict what information a user cares about in a given situation. Therefore, a recordable application should send Apple events that correspond to the simplest possible statements in a scripting language. "What to Record," which begins on page 9-14, provides some general guidelines for making these kinds of decisions.

"How Apple Event Recording Works," which begins on page 9-35, describes the Apple Event Manager's recording mechanism in more detail, including the role of the Recording On and Recording Off events.

Factoring Your Application for Recording

The recommended way to make your application recordable, or capable of sending Apple events to itself whenever a user performs a significant action, is to factor the code that controls your application's user interface from the code that responds to the user's manipulation of the interface. A fully factored application translates user actions into Apple events that the application sends to itself to initiate tasks.

The examples that follow demonstrate how to factor code that responds to relatively simple user actions such as creating a new document or moving a window. They are intended only to illustrate the general approach you should take; many of the decisions you will need to make while factoring will be unique to your application. "What to Record," which begins on page 9-14, provides guidelines for deciding which user actions to record and how to record them. For examples of factored applications, see the *AppleScript Software Developers' Kit.*

If you are factoring an existing application, it's usually a good idea to begin with the required Apple events and any other Apple events that you plan to send in order to execute commands in the File menu. You can then proceed to other menu commands and mouse actions. If you are designing a new application and want to make it recordable, you should build factoring into every aspect of your application design.

Factoring the Quit Command and the New Command

This section demonstrates how to factor two File menu commands: Quit and New.

When the user chooses a menu command, an application first determines which one was chosen and then performs the action associated with that command. For example, when a user chooses Quit from the File menu, an application that is not factored simply calls an application-defined `DoQuit` routine. Because Quit Application is one of the required Apple events, it is relatively easy for most applications that support Apple events to factor the code that responds to the Quit command.

After a factored application has determined that the user has chosen the Quit command, it sends the Quit Application event to itself by calling its `MyDoMenuQuit` routine.

```
PROCEDURE MyDoMenuQuit;
VAR
    myErr: OSErr;
BEGIN
    myErr := MySendAEQuit(kAEAskUser);
    {handle any errors}
END;
```

The `MyDoMenuQuit` routine in turn calls the `MySendAEQuit` routine shown in
Listing 9-1, which creates the Quit Application event and sends it.

Listing 9-1   A function used by a factored application to send itself a Quit Application event

```
FUNCTION MySendAEQuit (saveOpt: DescType): OSErr;
VAR
   myAppleEvent, defReply: AppleEvent;
   myErr, ignoreErr:       OSErr;
BEGIN
   {create Quit event}
   myErr := AECreateAppleEvent(kCoreEventClass,
                               kAEQuitApplication,
                               gSelfAddrDesc,
                               kAutoGenerateReturnID,
                               kAnyTransactionID, myAppleEvent);
   IF myErr = noErr THEN
      {add optional parameter that specifies whether this app }
      { should prompt user if window is dirty}
      myErr := AEPutParamPtr(myAppleEvent, keyAESaveOptions,
                             typeEnumerated, @saveOpt,
                             SizeOf(saveOpt));
   IF myErr = noErr THEN
      {send event}
      myErr := AESend(myAppleEvent, defReply,
                      kAENoReply+kAEAlwaysInteract,
                      kAENormalPriority, kAEDefaultTimeOut,
                      NIL, NIL);
   MySendAEQuit := myErr;
   ignoreErr := AEDisposeDesc(myAppleEvent);
END;
```

The input to the `MySendAEQuit` routine is a constant that indicates whether to save
dirty windows without asking the user (`kAEYes`), quit without saving dirty windows
(`kAENo`), or ask the user whether each dirty window should be saved (`kAEAskUser`).
In this example, the constant `kAEAskUser` passed to the `MySendAEQuit` routine
indicates that the user will be asked whether each dirty window should be saved.

After the application receives the Quit Application event, the MyHandleQuit handler shown in Listing 9-2 performs all the actions associated with that event, such as saving any open documents. (Note that your application should call the ExitToShell procedure from the main event loop, not from your handler for the Quit Application event.)

Listing 9-2   A routine used by a factored application to handle a Quit Application event

```
FUNCTION MyHandleQuit (theAppleEvent, reply: AppleEvent;
                       handlerRefcon: LongInt): OSErr;
VAR
   userCanceled:        Boolean;
   saveOpt, returnedType:  DescType;
   actSize:             Size;
   myErr:               OSErr;
BEGIN
   {check for missing required parameters}
   myErr := MyGotRequiredParams(theAppleEvent);
   IF myErr = noErr THEN
   BEGIN
      {pick up optional save parameter}
      saveOpt := kAEAskUser;  {the default}
      myErr := AEGetParamPtr(theAppleEvent, keyAESaveOptions,
                             typeEnumerated, returnedType,
                             @saveOpt, SizeOf(saveOpt), actSize);
      IF myErr = errAEDescNotFound THEN
         myErr := noErr;
      MyHandleQuit := myErr;
      IF myErr = noErr THEN
      BEGIN
         userCanceled := MyPrepareToTerminate(saveOpt);
         IF userCanceled THEN
            MyHandleQuit := kUserCanceled;
      END;
   END
   ELSE
      MyHandleQuit := myErr;
END;
```

The handler in Listing 9-2 calls another function supplied by the application, the
`MyPrepareToTerminate` function. When the value of the optional parameter
that specifies how to deal with dirty windows equals `kAEAskUser`, this function
asks the user whether to save each dirty window and returns a Boolean value that
indicates whether the user canceled the Quit request. It also responds appropriately to
the other possible values of the optional parameter.

If recording has been turned on for a scripting component (for example, after a user
clicks the Record button in the Script Editor application) and the user quits the
application, the Apple Event Manager automatically sends the scripting component a
copy of the Quit Application event sent by the `MySendAEQuit` routine. The scripting
component records the event in a compiled script. When a user executes the recorded
script, the scripting component sends the same Quit Application event to the application,
which calls the `MyHandleQuit` function and responds to the event just as if the user had
chosen Quit from the File menu.

After you have factored the commands associated with required Apple events for an
existing application, you can move on to the other commands in the File menu, such as
New. After a factored application has determined that the user has chosen New, it calls
its `MyDoMenuNew` routine, which sends the Create Element event to the application.

```
PROCEDURE MyDoMenuNew;
VAR
   myErr := OSErr;
BEGIN
   myErr := MySendAECreateElement(gNullDesc, cDocument);
   {handle any errors}
END;
```

The container for the new element is the application's default container, specified by a
null descriptor record, and the desired class is `cDocument`. The `MyDoMenuNew` routine
in turn calls the `MySendAECreateElement` routine shown in Listing 9-3, which creates
the Apple event and sends it.

Listing 9-3   A routine used by a factored application to send itself a Create Element event

```
FUNCTION MySendAECreateElement (cont: AEDesc;
                                elemClass: DescType): OSErr;
VAR
   myAppleEvent, defReply: AppleEvent;
   myErr, ignoreErr:       OSErr;
BEGIN
   {create Create Element event}
   myErr := AECreateAppleEvent(kCoreEventClass, kAECreateElement,
                               gSelfAddrDesc,
                               kAutoGenerateReturnID,
                               kAnyTransactionID, myAppleEvent);
   IF myErr = noErr THEN
      {add parameter that specifies insertion location for the }
      { new element}
      myErr :=  AEPutParamDesc(myAppleEvent,keyAEInsertHere,cont);
   IF myErr = noErr THEN
      {add parameter that specifies new element's object class}
      myErr := AEPutParamPtr(myAppleEvent, keyAEObjectClass,
                             typeType, @elemClass,
                             SizeOf(elemClass));
   IF myErr = noErr THEN
      {send the event}
      myErr := AESend(myAppleEvent, defReply,
                      kAENoReply+kAECanInteract,
                      kAENormalPriority, kAEDefaultTimeOut, NIL,
                      NIL);
   MySendAECreateElement := myErr;
   ignoreErr := AEDisposeDesc(myAppleEvent); {must dispose of }
                                             { event}
END;
```

For the purposes of this example, the routine shown in Listing 9-3 sends only the required parameters and can only create a new active window with the default name. After the application receives the Create Element event, its `MyHandleCreateElement` handler performs the requested action, as shown in Listing 9-4. In this case, it creates a new active window with a default title.

Listing 9-4   The Create Element event handler for a factored application

```
FUNCTION MyHandleCreateElement (theAppleEvent: AppleEvent;
                               reply: AppleEvent;
                               handlerRefCon: LongInt): OSErr;
VAR
   myCont:                  AEDesc;
   returnedType, newElemClass: DescType;
   actSize:                 Size;
   contClass:               DescType;
   window:                  WindowPtr;
   myErr:                   OSErr;
BEGIN
   {get the parameters out of the event}
   {first get the direct parameter, which specifies insertion }
   { location for new window--that is, frontmost window}
   myCont.dataHandle := NIL;
   myErr := AEGetParamDesc(theAppleEvent, keyAEInsertHere,
                           typeWildCard, myCont);
   IF myErr = noErr THEN
      {get the other required parameter, which specifies class }
      { cDocument when MyHandleCreateElement creates a new window}
      myErr := AEGetParamPtr(theAppleEvent, keyAEObjectClass,
                              typeType, returnedType,
                              @newElemClass,
                              SizeOf(newElemClass), actSize);
   IF myErr = noErr THEN
      myErr := MyGotRequiredParams(theAppleEvent);
   MyHandleCreateElement := myErr;
   IF myErr = noErr THEN
   BEGIN
      {check container and class, just to make sure}
      IF (myCont.descriptorType <> typeNull) OR (newElemClass <>
            cDocument) THEN
         MyHandleCreateElement := kWrongContainerOrElement
      ELSE
         {MyNewWindow creates a new window with a default name }
         { and returns a pointer to it in the window parameter}
         MyHandleCreateElement := MyNewWindow(window);
   END;
   myErr := AEDisposeDesc(myCont);
   {if your app sends a reply in response to the Create Element }
   { event, then set up the reply event as appropriate}
END;
```

If recording has been turned on for a scripting component (for example, after a user clicks the Record button in the Script Editor application), the Apple Event Manager automatically sends the scripting component a copy of the Create Element event sent by the `MySendAECreateElement` routine. The scripting component records the Apple event as a statement in a compiled script. When a user executes the recorded script, the scripting component sends the same Create Element event to the application, which calls its `MyHandleCreateElement` handler and responds to the event just as if the user had chosen New from the File menu.

Sending Apple Events Without Executing Them

If an application is fully factored, it carries out almost all the tasks a user can perform by sending itself Apple events in the manner illustrated by the listings in the preceding sections. However, in some cases it may not be practical to send an Apple event that actually executes a task performed by the user.

For example, if the user drags a window by its title bar from one position to another, it is inefficient to send a series of Apple events that move the window through a series of positions until the user releases the mouse button. Instead, your application can call the Window Manager routine `DragWindow` to allow the user to drag the window to a new position. Until the user releases the mouse button, it's not possible to send a single Apple event that drags the window to the new position, because the new position is not yet known. When `DragWindow` returns, the window has already been dragged to its new position, and its window record has been updated.

At this point your application can send itself the Set Data event that performs the same action; but to avoid repeating the action that was just performed with `DragWindow`, you should add the `kAEDontExecute` constant to the `sendMode` parameter of the `AESend` function when you send the event. The Apple Event Manager then sends the Set Data event to the recording process, if any, but does not send it to the application.

Listing 9-5 shows an application-defined routine, `MyDoDragWindow`, that illustrates this approach. The `MyDoDragWindow` routine calls `DragWindow` in the usual way, then uses another application-defined routine, `MyCreateAESetWindowPos`, and the `AESend` function to create and send a Set Data Apple event that sets the window position to the new location. However, because the window has already been moved, there is no need to execute the Set Data event. To send the event for recording purposes without actually executing it, the `MyDoDragWindow` routine adds the `kAEDontExecute` constant to the `sendMode` parameter of the `AESend` function when it sends the Set Data event.

Listing 9-5   A routine used by a factored application to handle window movement

```
PROCEDURE MyDoDragWindow (theWindow: WindowPtr; startPt: Point;
                         boundsRect: Rect);
VAR
   newPos:       Point;
   index:        Integer;
   theAppleEvent: AppleEvent;
   reply:        AppleEvent;
   myErr:        OSErr;
BEGIN
   DragWindow(theWindow, startPt, boundsRect);
   newPos := WindowPeek(theWindow)^.contRgn^^.rgnBBox.topLeft;
   index := MyIndexFromWndwPtr(theWindow);
   MyCreateAESetWindowPos(index, newPos, theAppleEvent);
   myErr := AESend(theAppleEvent, reply, kAENoReply +
                   kAECanInteract + kAEDontExecute,
                   kAENormalPriority, kAEDefaultTimeout, NIL,
                   NIL);
END;
```

If recording has been turned on and the user moves a window, the Apple Event Manager automatically sends the scripting component a copy of the Set Data event sent by the MyDoDragWindow routine but does not send the event to the application. The scripting component records the event as a statement in a compiled script. When a user executes the recorded script, the scripting component sends the same Set Data event to the application. The application's handler for the Set Data event then changes the position of the window.

What to Record

Factoring an application involves making decisions about which user actions generate Apple events, about the content of those events, and about when to send events for recording purposes. For example, the preceding section, "Sending Apple Events Without Executing Them," describes how an application should generate an Apple event that corresponds to a change in the position of a window. Other actions can be more complicated to define in terms of Apple events. This section provides general guidelines for deciding which user actions should generate Apple events and how those events should be defined.

When the user records a series of actions as a script, playing the recorded script back later in exactly the same circumstances must produce exactly the same result. If the circumstances at execution time are similar but not exactly the same as when the script was recorded, the script should also work correctly. However, certain differences will always lead to unexpected results or cause execution to fail.

The goal of these guidelines is to help you create scripts that will work correctly in the largest number of circumstances with the fewest post-recording changes by the user. To accomplish this goal, a recordable application should send itself Apple events that describe as specifically as possible the user's actions in the application's domain without making guesses about the user's intentions.

The way your application uses Apple events to record a user's actions depends in part on the kind of script being recorded. From the user's perspective, there are at least three kinds of scripts:

n A script application. The icons for these files appear in the Finder, for example, in the Apple Menu Items folder or the Startup Items folder.

n A script that functions like a menu command, usually acting on the current selection in the current application, and stored either as a compiled script file that appears in the Finder or as a script stored within an application or one of its documents.

n A script that is "embedded" in an application—that is, explicitly associated with something in a document, such as a field in a form, a cell or row of a spreadsheet, or a button.

The recording guidelines in the sections that follow apply to the recording of scripts that function like menu commands and scripts that are embedded in an application. Because such scripts are executed under a user's direct control, the user expects their execution to cause something to happen, possibly changing the current selection, the Clipboard, or the active window.

The execution of a script application, however, may cause a scripting component to send events to one or more applications intermittently without the user's knowledge. If the script in a script application refers to the current selection, the Clipboard, or the active window, its execution may interfere with other tasks being performed by the user or tasks performed during the execution of other scripts. To create a script application and ensure that it works correctly when executed, a scripter may need to modify the script after it has been recorded.

For example, to eliminate references to the Clipboard, a scripter can use a script variable as a user-defined Clipboard and convert Cut, Copy, and Paste statements to appropriate combinations of Move, Copy, New, and Delete statements, while supplying the previously defined selection as the argument. It may also be necessary to convert a description such as "the front document" to a specific filename or a variable.

Recording User Actions

Two general guidelines apply to the recording of all user actions:

n Send Apple events that correspond to simple statements in a script rather than compound statements.

n Don't record superfluous actions.

In most cases, if the user performs several related actions, your application should send Apple events for each action rather than saving the actions and creating an event that combines them.

For example, if the user selects some text, cuts it, and then pastes it somewhere else, your application should send itself four events that correspond to these actions:

1. Select the text

2. Cut

3. Set the insertion point

4. Paste

Thus, if the user selects characters 5 through 20 of the frontmost document, chooses the Cut command from the Edit menu, places the insertion point after character 72, and chooses the Paste command, your application should send the following events.

n   A Select event (event class `kAEMiscStandards`, event ID `kAESelect`) with this direct parameter:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cChar
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cDocument
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	1
keyAEKeyForm	typeEnumerated	formRange
keyAEKeyData	typeRangeDescriptor	(see indented record)
keyAERangeStart	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cChar
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	5
keyAERangeStop	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cChar
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	20

n   A Cut event (event class `kAEMiscStandards`, event ID `kAECut`)

n A Select event with this direct parameter:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cInsertionLoc
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cChar
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cDocument
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	1
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	72
keyAEKeyForm	typeEnumerated	formRelativePosition
keyAEKeyData	typeEnumerated	kAEAfter

n A Paste event

Note

The format used for the direct parameters in this example and throughout this chapter does not show the structure of the direct parameters as they exist within the Apple events. Instead, this format shows what you would obtain after calling AEGetKeyDesc repeatedly to extract the nested descriptor records from the Apple events.

When you call AEGetKeyDesc to extract the descriptor record that specifies an application's default container, AEGetKeyDesc returns a descriptor record of type AEDesc with a descriptor type of typeNull and a data handle whose value is 0. u

The first Select event in this example sets the application's pSelection property (that is, the current selection) to the objects identified by the object specifier record in the direct parameter—characters 5 through 20. The second Select event places the insertion point after the object identified by the object specifier in the direct parameter—after character 72.

You could also interpret these four actions as a single Move event that simply moves characters 5 through 20 to after character 72. A user could write such a statement in a script, but for recording purposes four separate events correspond more precisely to the user's actions. For example, if the user performed another paste operation after the first four actions, a Move event would not produce the correct results.

It is equally important for a recordable application not to send superfluous events. For example, your application should not send an event every time the user makes a selection. Instead, it should keep track of the most recent selection made. When the user performs some action on the selection, the application should send an event that sets the selection followed by the event that corresponds to the action taken by the user. However, if the user doesn't perform an action on the selection, the application should not send an event.

IMPORTANT

If something is already selected when recording begins, your application should not record that selection. Subsequent user actions should be recorded assuming that there is a selection. By not recording the current selection, you allow the user to record scripts that work, without further modification, much like menu commands that operate on the current selection. s

The example just discussed assumes that the application has multiple documents. In such an application, document 1 is always the document in the frontmost window. The examples that follow are simplified, as if they were generated by an application like TeachText that can have only one document open at a time and can therefore locate objects such as characters in the default container. For more complex applications that locate text in cells, documents, and other containers, you must specify additional containers as appropriate.

Recording the Selection of Text Objects

When your application needs to record a selection that the user has made by dragging through a range of text, it should send itself a Select event that selects a range of characters. For example, a Select event with this direct parameter selects characters 80 through 764:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cChar
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formRange
keyAEKeyData	typeRangeDescriptor	(see indented record)
keyAERangeStart	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cChar
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	80
keyAERangeStop	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cChar
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	764

It is sufficient to record such a text selection as a range of characters. However, recording selections in other units can make the corresponding scripts easier to read. If you decide to record text selections in other units, keep these guidelines in mind:

n Use the largest whole unit that completely describes the selection.

n Do not mix units.

n Use units appropriate to the method of selection.

n Use logical units rather than units that vary with reformatting.

n Don't try to guess the user's intentions.

The rest of this section provides examples of how to apply these guidelines.

If you do record text selections in units other than characters, record each selection in terms of the largest whole unit that completely describes the selection. For example, suppose the user selects characters 115 through 170 by dragging. Further, suppose the selected characters are exactly the same as words 33 through 50 and also the same as paragraph 2. In this case your application should send itself a Select event with this direct parameter:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cParagraph
keyAEContainer	typeNull	**No data**
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	2

However, if the selected characters don't match a larger unit exactly—for example, if paragraph 2 is larger than the selection or the selection is a portion of two paragraphs—use the largest unit available, in this case words.

For example, a Select event with this direct parameter selects word 33 through word 45:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cText
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formRange
keyAEKeyData	typeRangeDescriptor	(see indented record)
keyAERangeStart	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cWord
keyAEContainer	typeCurrentContainer	**No data**
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	33
keyAERangeStop	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cWord
keyAEContainer	typeCurrentContainer	**No data**
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	45

Do not mix units. You should not send Apple events that define selections like character 2 of word 3 of line 5 of paragraph 2 in document "MyDocument." Instead, define selections as simply as possible; for example, character 45 in the document "MyDocument."

When the user selects text by double-clicking it, your application should send a Select event that specifies words. For example, your application should send a Select event with this direct parameter when the user double-clicks word 5:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cWord
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	5

If the user double-clicks word 5 and then extends the selection through word 9, your application should send a Select event with this direct parameter:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cText
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formRange
keyAEKeyData	typeRangeDescriptor	(see indented record)
keyAERangeStart	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cWord
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	5
keyAERangeStop	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cWord
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	9

If your application supports selection of a paragraph, for example by clicking the left margin, triple-clicking, or some other action, your application should send a Select event that selects the paragraph. For example, a Select event with this direct parameter selects paragraph 2:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cParagraph
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	2

If your application supports the selection of other units (for instance, cells, rows, and columns in a spreadsheet; embedded graphics in a word processor; or buttons) and if users can select a range of such units, your application should record using those units when appropriate. For example, a Select event with this direct parameter selects row 5 through row 23:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formRange
keyAEKeyData	typeRangeDescriptor	(see indented record)
keyAERangeStart	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	5
keyAERangeStop	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	23

A Select event with this direct parameter selects the second `'PICT'` image:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cPICT
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	2

When the user chooses a Select All command, your application should send a Select event with this direct parameter to select the contents of the document:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeLongInteger	pContents

Units that vary with reformatting, such as lines and pages in a text document, are not as useful as logical units that describe the data more precisely. Whenever possible, use logical units such as character, word, paragraph, section, and so on.

Don't try to guess the user's intentions. For example, if a selection can be described as either "word 14" or as "the third bold word in paragraph 3," use the simpler description. If you guess the user's intentions, you will be wrong often enough to cause the user to distrust the recording process.

Recording Insertion Points

The insertion point and a selection are synonymous in the Macintosh Operating System. However, scripting languages need a way of specifying a zero-width selection. Sometimes the best way to specify an insertion location is in relation to another object; for example, "after word 5." This section describes recommended methods of specifying an insertion point in a recordable event.

The insertion point can be specified in Apple events by either an insertion location descriptor record (`typeInsertionLocation`) or an object specifier record (`typeObjectSpecifier`) that specifies the class `cInsertionLoc` and the key form `formRelativePosition`. The Move, Clone, and Create events accept an insertion location descriptor record; other events, including Select and Set Data, require an object specifier record.

Five constants can be used to describe an insertion point in relation to an object or container:

Constant	Corresponding insertion point
kAEReplace	The specified object will be replaced if not qualified by one of the other phrases
kAEBefore	Just before the specified object (either type typeObjectSpecifier or type typeInsertionLocation)
kAEAfter	Just after the specified object (either type typeObjectSpecifier or type typeInsertionLocation)
kAEBeginning	In the specified container and before all other elements of the same class in that container (type typeInsertionLocation only)
kAEEnd	In the specified container and after all other elements of the same class in that container

For more information about the way AppleScript uses insertion location descriptor records, see "Defining Terminology for Use by the AppleScript Component," which begins on page 8-3, and the *Apple Event Registry: Standard Suites.* The rest of this section provides examples of object specifier records used to specify insertion points.

Users usually insert objects after some other object. So, unless the insertion point is clearly at the beginning or end of a container or identifies an object to be replaced, use the constant kAEAfter to record the location.

For example, if the user places the insertion point after character 2, your application should send a Select event with this direct parameter:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cInsertionLoc
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cChar
keyAEContainer	typeNull	**No data**
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	2
keyAEKeyForm	typeEnumerated	formRelativePosition
keyAEKeyData	typeEnumerated	kAEAfter

If the selection is not 0 characters wide, the user is replacing the selection with another object, so you can specify the location simply as the object specifier record for the object to be replaced.

If the user clicks the white space after a paragraph somewhere in the middle of the document, defining the insertion point becomes more complex because different applications deal with this situation in different ways. Some place the insertion point at the end of the current paragraph, while others place the insertion point at the beginning of the next paragraph. Depending on the way your application handles this situation, you should use an object specifier record that specifies either kAEBeginning or kAEEnd.

Remember that the Select event requires an object specifier record. Thus, if you want to place the insertion point at the beginning of a paragraph, use an object specifier record that specifies a location just before the first item of the paragraph, rather than an insertion location descriptor record.

For example, a Select event with this direct parameter places the insertion point just before the first item of paragraph 3:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cInsertionLoc
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cItem
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cParagraph
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	3
keyAEKeyForm	typeType	formAbsolutePosition
keyAEKeyData	typeLongInteger	1
keyAEKeyForm	typeEnumerated	formRelativePosition
keyAEKeyData	typeEnumerated	kAEPrevious

If the user clicks the left edge of the first line in a paragraph, thus setting the insertion point before the beginning of the paragraph, you should use a similar strategy. However, this is the only situation in which you should use kAEPrevious.

When the insertion point is at the end of a document record, use an object specifier record that specifies the location after the last item in the document.

For example, a Select event with this direct parameter places the insertion point just after the last item in a document:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cInsertionLoc
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cItem
keyAEContainer	typeNull	**No data**
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	-1
keyAEKeyForm	typeEnumerated	formRelativePosition
keyAEKeyData	typeEnumerated	kAENext

A Select event with this direct parameter places the insertion point just after the last item in paragraph 3:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cInsertionLoc
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cParagraph
keyAEContainer	typeNull	**No data**
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	3
keyAEKeyForm	typeEnumerated	formRelativePosition
keyAEKeyData	typeEnumerated	kAENext

Recording Typing

In general, to record typing your application should send itself a Set Data event that sets the contents of the selection. The data should be unstyled text. When your application handles the Set Data event, it should apply the styles that prevail at the insertion point. If your application supports styled text, you need to decide how to apply styles to new text and how to record style changes to selected text. Follow these general guidelines for recording typing:

n   When the user sets an insertion point and types new text, use the styles defined for the text just before the insertion location.

n   When a user selects text and changes its style, apply the changes to the selection.

n   If a user types or pastes new text into a selection, place the insertion point after the new text.

The rest of this section provides examples of how to apply these guidelines.

Suppose the user sets an insertion point and then types something. Your application should use the style, font, size, and other characteristics of the text just before the insertion point for the new text, and it should record only the new characters inserted. For example, to place the insertion point after word 30 and insert the text "This is the new text," your application can send a Select event followed by a Set Data event:

n   A Select event with this direct parameter places the insertion point after word 30:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cInsertionLoc
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cWord
keyAEContainer	typeNull	**No data**
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	30
keyAEKeyForm	typeEnumerated	formRelativePosition
keyAEKeyData	typeEnumerated	kAENext

n A Set Data event (event class `kAECoreSuite`, event ID `kAESetData`) with these parameters (`keyDirectObject` and `keyAEData`) sets the selection to the new text:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeNull	**No data**
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeLongInteger	pSelection
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeType	pContents
keyAEData	typeChar	"This is the new text"

Notice that the Select event in this example causes your application to set its `pSelection` property (the current selection) to the location specified by the object specifier record in the direct parameter—that is, after word 30. The Set Data event then sets the contents of the selection to a text string. The `pContents` property specified by the object specifier record in the direct parameter of the Set Data event represents the contents of the selection, and the text string in the `keyAEData` parameter is the text to which the selection's contents is to be set.

At this stage, the insertion point is after word 35—the last word added by typing. If the user now selects one of the new words, say word 34, and changes the style to boldface and the font to Helvetica®, send a Select event and two Set Data events to record the action:

n A Select event with this direct parameter selects word 34:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cWord
keyAEContainer	typeNull	**No data**
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	34

n   A Set Data event with these parameters sets the style of the selection to boldface:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeLongInteger	pSelection
keyAEKeyForm	typeType	formPropertyID
keyAEKeyData	typeType	pTextStyles
keyAEData	typeEnumerated	kAEBold

n   A Set Data event with these parameters sets the font of the selection to Helvetica:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeLongInteger	pSelection
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeType	pFont
keyAEData	typeChar	"Helvetica"

After these three events are sent, word 34 remains selected. Thus, subsequent user actions upon the same selection do not require your application to send an additional event to set the selection. Your application should maintain the selection as long as the selected text is not replaced. If the user types or pastes new text into the selection, your application should place the insertion point after the new text.

Such a strategy might result in a series of events like these:

n A Set Data event with these parameters sets the contents of a selection to "More new text":

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeLongInteger	pSelection
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeType	pContents
keyAEData	typeChar	"More new text"

n Two Paste events paste the contents of the Clipboard twice after the new text.

Recording the Selection of Nontext Objects

The selection of nontext objects differs from the selection of text objects mainly in the way a recordable application specifies the objects. For example, if the user is working in a table or spreadsheet and selects row 5, column 3, your application can send a Select event with this direct parameter:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cColumn
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	3
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	5

When recording a range of cells, use a range of rows through a range of columns.

For example, if the user selects row 5 column 3 through row 6 column 4, specify columns 3 through 4 of rows 5 through 6 by sending a Select event with this direct parameter:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cColumn
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formRange
keyAEKeyData	typeRangeDescriptor	(see indented record)
keyAERangeStart	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cColumn
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	3
keyAERangeStop	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cColumn
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	4
keyAEKeyForm	typeEnumerated	formRange
keyAEKeyData	typeRangeDescriptor	(see indented record)
keyAERangeStart	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	5
keyAERangeStop	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	6

In some drawing and layout applications, users are used to dealing with insertion points at specific locations rather than relative to other objects. For example, setting an insertion point in a recordable drawing application might cause the application to send itself a Select event that places the insertion location at (235, 330)—that is, the location defined by a vertical coordinate of 235 and a horizontal coordinate of 330. A Select event that does this could have this direct parameter:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cInsertionLocation`
`keyAEContainer`	`typeNull`	**No data**
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeAEList`	(see indented record)
	`typeLongInteger`	235
	`typeLongInteger`	330

Notice that the key data corresponds to an application's extension of the standard interpretation of key form `formAbsolutePosition`.

To set a selection that consists of noncontiguous objects, an application should send events that correspond to statements like these:

```
select {¬
   row 5 thru 6 of column 3 thru 4, ¬
   row 22 of column 6}

select {circle 2, rectangle 12, text frame 2}

select {file "Guidelines", file "Test Results"}
```

Identifying Objects

The way a recordable application identifies objects can involve assumptions about the user's criteria for selecting those objects. In general, such assumptions should be avoided. Follow these guidelines for identifying objects:

n   If you aren't absolutely certain of the user's criteria for selecting an object, identify the object by name.

n   If the object doesn't have a name, identify it by index.

n   Determine the index based on the order in which a user would see the objects when reading a document.

n   Identify windows and open documents on which actions are taken as the frontmost window or document.

The rest of this section provides examples of how to apply these guidelines.

Suppose a user is working with an electronic mail application that permits a variety of sorting methods for messages received. If the user is currently looking at messages sorted by date and then deletes the second message in the list, that message should be identified by name rather than by date. Use an object's name in any situation where it is not completely clear which identifying criteria the user had in mind.

Suppose a user has used the application's Find command to locate all messages created on a certain date. In this case it might be appropriate to identify "every message whose creation date . . ." in the corresponding Apple event. However, if the user did not ask for all messages created on that date, you can't be sure whether the user really wanted every message or only a particular one. For instance, perhaps the user couldn't remember the name, but only an approximate date. In this case a recordable application should identify the message by name.

Just as names are more specific and usually more desirable than whose tests, names are usually more specific and more readable than identifiers or indices. However, some objects may not have a name, only some other identifier or an index. Even though an identifier is more specific than an index, a logically defined index of position is more readable and is therefore recommended. For example, if a document contains unnamed illustrations, the user is more likely to identify a figure by index (order from the beginning of the file) than identifier (such as order created).

Suppose a document contains two figures that appear at first glance to be side by side, except that the right one is slightly taller and therefore begins higher on the page than the left one. In cases such as this, your application should determine the index based on the order in which the user would see the objects when reading a document. For Roman script systems, this means reading from left to right and from top to bottom. In the example just described, the leftmost, shorter figure would have a lower figure number than the rightmost, taller one.

When your application needs to refer to a window or a document, it should identify the object with an object specifier record that corresponds to the first, or front, window:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cWindow
keyAEContainer	typeNull	**No data**
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	1

This strategy allows users to record scripts that will work on any window, regardless of its name. Similarly, events that act on an open document should identify it as "document 1."

It is usually possible to describe objects in several different ways. If an object has a unique name, use that. For example, instead of an object specifier record that describes "column number 7," use one that describes "the column named 'March'":

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cColumn
keyAEContainer	typeNull	**No data**
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeLongInteger	"March"

It may be that such an object could also be described in a more complex manner, such as picture 1 of paragraph 302 of chapter 2. But complex descriptions like this should be used only as a last resort if no simpler name is available.

In general, be as specific as possible when you identify a selection in a recordable event. The user can generalize as necessary by editing the recorded script.

Moving the Selection During Recording

If recording is turned on and the user makes a selection, performs some action, and then makes a different selection, your application must make a decision: should it record the second selection in absolute terms or relative to the first selection? That is, should the corresponding AppleScript statement be

```
select insertion location before paragraph 5
```

or

```
select insertion location before paragraph after selection
```

Both statements may be appropriate under different conditions. But suppose that the user had selected paragraph 3 and now selects paragraph 12 or picture 3. Relative addressing doesn't make sense in these situations because the distance involved is too great or the unit is different. When you can't be sure of the user's intent, you should use absolute addressing. You can safely use relative addressing only when the user moves the selection or insertion point by only one unit, as with the arrow keys.

Even the use of the arrow keys does not guarantee that you can use relative addressing. For example, suppose that the user has selected cell 5 of row 2 in a spreadsheet and then presses the Left Arrow key three times. In this case, it is best to send Apple events equivalent to the statement

```
select cell 3 of row 2
```

rather than the statements

```
select the cell before selection
select the cell before selection
select the cell before selection
```

Using relative addressing in certain circumstances may minimize the amount of editing that the user must do after recording a script. However, recordable applications are not required to use relative addressing.

Recording Interactions With Dialog Boxes

When executing scripts, users normally do not want to see dialog boxes. Therefore, your application should record information specified by the user in dialog boxes rather than sending events that would cause the dialog boxes to appear during script execution.

For example, suppose a user chooses the Close command and the standard save changes dialog box appears. If the user then clicks Save, your application should send a Close event that corresponds to a statement like this:

```
close document "MyDoc" saving Yes
```

Any settings in a dialog box that the user does not change (such as the range of pages to print in a Print dialog box) should not be recorded.

How Apple Event Recording Works

Scripting components use the Apple Event Manager's recording mechanism to allow a recording process such as the Script Editor application to control recording into scripts. Script editors and applications that provide their own recording capabilities can take advantage of the recording mechanism via standard scripting component routines.

This section describes how scripting components use Apple event recording. You need to read this section if you are developing a scripting component or a script-editing application, or if you want your application to initiate and control Apple event recording. For information about using the standard scripting component routines to turn recording off and on, see "Recording Scripts" on page 10-26.

When a user turns on recording for a recording process (for example, by clicking the Record button in Script Editor), the recording process calls a scripting component routine (OSAStartRecording) to turn recording on. The scripting component responds by sending a Start Recording event to the recording process (or any running process on the local computer).

Start Recording—begin sending copies of recordable events to recording process

Event class	kCoreEventClass
Event ID	kAEStartRecording
Parameters	None
Description	Sent by a scripting component to the recording process (or to any running process on the local computer), but handled by the Apple Event Manager. The Apple Event Manager responds by turning on recording and sending a Recording On event to all running processes on the local computer.
	This event must be addressed using a process serial number (PSN); it should never be sent to an address specified as kCurrentProcess.

The recording process should not handle the Start Recording event. Instead, the Apple Event Manager handles it by sending a Recording On event to all running processes on the local computer and sending copies of all subsequent recordable events to the recording process. (The Recording On event is described on page 9-4.)

If an application that supports Apple events is launched on a computer for which recording is turned on, the Apple Event Manager will also send it a Recording On event for each recording process that is currently recording.

The recording process receives recordable events by means of a Receive Recordable Event handler—that is, a handler installed in the Apple event dispatch table for event class kCoreEventClass and event ID kAENotifyRecording. Scripting components install this handler on behalf of a recording process when recording is first turned on and remove the handler when recording is turned off. Much like a handler for event class typeWildCard and event ID typeWildCard, the Receive Recordable Event handler handles all recordable events sent to the recording process by the Apple Event Manager. Any other Apple events received by the recording process are dispatched in the usual manner. The Receive Recordable Event handler handles recordable events by recording them in the script specified by the recording process's call to OSAStartRecording.

Receive Recordable Event—receive and record a copy of a recordable event

Event class	kCoreEventClass
Event ID	kAENotifyRecording
Parameters	Same as Apple event being recorded
Description	Wildcard event class and event ID handled by a recording process in order to receive and record copies of recordable events sent to it by the Apple Event Manager. Scripting components install a handler for this event on behalf of a recording process when recording is turned on and remove the handler when recording is turned off.

Whenever the Receive Recordable Event handler receives a recordable event, the scripting component sends your application a Recorded Text event. The Recorded Text event contains the decompiled source data for the recorded event in the form of styled text. For a description of the Recorded Text event, see "Recording Scripts" on page 10-26.

When a user turns off recording (for example, by clicking Script Editor's Stop button), the recording process calls a scripting component routine (OSAStopRecording) to turn recording off. The scripting component responds by sending a Stop Recording event to the recording process (or any running process on the local computer).

Stop Recording—stop sending copies of recordable events to recording process

Event class	kCoreEventClass
Event ID	kAEStopRecording
Parameters	None
Description	Sent by a scripting component to the recording process (or to any running process on the local computer), but handled by the Apple Event Manager. The Apple Event Manager responds by sending a Recording Off event to all running processes on the local computer.
	This event must be addressed using a process serial number (PSN); it should never be sent to an address specified as kCurrentProcess.

Like the Start Recording event, the Stop Recording event is handled by the Apple Event Manager. The Apple Event Manager responds by sending a Recording Off event to all running processes on the local computer. (The Recording Off event is described on page 9-4.)

Recording continues, and the recording process may continue to receive recordable events, until the Apple Event Manager has notified all running processes that recording has been turned off for that recording process. The Apple Event Manager sends all running processes the Recording Off event with the kAEWaitReply flag set. If an application has stored some data (for instance, keystrokes) that needs to be recorded as an Apple event, this is the last chance for the application to send the event for recording purposes. Recording stops only after the Apple Event Manager returns a reply for the Stop Recording event.

The Apple Event Manager supports multiple simultaneous recording processes. A Stop Recording event sent for one of them does not affect the others. If your application needs to know which of several recording processes has turned recording on or off, it can check the keyOriginalAddressAttr attribute of the Recording On or Recording Off event for the address of the recording process.

If the Apple Event Manager does not receive a Stop Recording event for a recording process that quits unexpectedly, the applications being recorded don't find out immediately. When it attempts to send a copy of a recordable event to a recording process that is no longer active, the Apple Event Manager sends a Recording Off event to all running processes on behalf of that recording process and specifies the address for that process in the keyOriginalAddressAttr attribute. If a recording process that quits is the only actively recording process, recording stops completely after the Apple Event Manager has informed all running processes that recording has been turned off.

Scripting Components

Contents

This chapter describes how your application can use the Component Manager and scripting components to manipulate and execute scripts.

Before you read this chapter, you should read the chapter "Introduction to Scripting" in this book and the chapters about the Apple Event Manager that are relevant to your application.

Your application can use the standard scripting component data structures and routines described in this chapter to manipulate scripts written in any scripting language based on the Open Scripting Architecture (OSA). Your application need not be scriptable or recordable to use these routines. However, if your application is scriptable, you can easily make it capable of manipulating and executing scripts that control its own behavior.

The first section in this chapter describes how to establish a connection with a scripting component. The next two sections provide

n   examples of how to use the standard scripting component routines

n   information for developers of scripting components

The section "Scripting Components Reference" describes, in addition to the standard scripting component routines, routines provided by the AppleScript component, routines provided by the generic scripting component, and routines called by scripting components.

If you are developing a scripting component, you should also read the instructions for creating components in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

Connecting to a Scripting Component

To manipulate and execute scripts written in different scripting languages, your application can use Component Manager routines either to open a connection with each corresponding scripting component individually or to open a single connection with the generic scripting component. The generic scripting component, in turn, attempts to open connections dynamically with the appropriate scripting component for each script. By opening a connection with the generic scripting component, your application can load and execute scripts created by any scripting component that is registered with the Component Manager on the current computer.

In general, you should use the generic scripting component to execute and manipulate existing scripts and a specific scripting component when you create new scripts. When you call `OSACompile` or `OSAStartRecording`, the generic scripting component examines the script ID to determine which scripting component to use. If instead of a script ID you pass the constant `kOSANullScript` to these routines, the generic scripting component uses its current **default scripting component.** Each instance of the generic scripting component has its own default scripting component. From the user's point of view, the default scripting component corresponds to the scripting language selected in the Script Editor application when the user first creates a new script.

The generic scripting component provides routines you can use to get and set the default scripting component, determine which scripting component created a particular script, and perform other useful tasks when you are using multiple scripting components. See the section "Generic Scripting Component Routines," which begins on page 10-84, for descriptions of these routines.

You can use the Component Manager function `OpenComponent` to open a connection to a scripting component you specify with the component identifier returned by the `FindNextComponent` function. You can also use the `OpenDefaultComponent` function to open a scripting component without calling the `FindNextComponent` function.

The `OpenComponent` and `OpenDefaultComponent` functions return a component instance. This value identifies your application's connection to a component. You must supply this value whenever you call a standard scripting component routine.

Note
Your application may maintain several connections to a single component, or it may have connections to several components at the same time. Because some scripting components (including the current version of AppleScript) can execute only one script at a time per component instance, a multithreaded application must provide a separate component instance for each script that it compiles or executes while it is simultaneously executing other scripts. u

The Component Manager type code for scripting components that support the routines described in this chapter is `'osa '`, and the subtype code for the generic scripting component is `'scpt'`.

```
CONST
    kOSAComponentType                     = 'osa ';
    kOSAGenericscriptingComponentSubtype  = 'scpt';
```

You can open a connection to a scripting component by calling the `OpenDefaultComponent` function, which returns a component instance. For example, this code opens a connection with the generic scripting component and stores the returned value in an application-defined variable:

```
VAR
    gScriptingComponent: ComponentInstance;
{open connection to generic scripting component}
gScriptingComponent := OpenDefaultComponent(kOSAComponentType,
                        kOSAGenericscriptingComponentSubtype);
```

The generic scripting component in turn opens connections with other scripting components as necessary. The generic scripting component provides routines you can use to get instances of other scripting components when you want to use component-specific routines.

It is also possible to open an explicit connection directly with a specific scripting component such as AppleScript:

```
VAR
    gScriptingComponent: ComponentInstance;
{open connection to AppleScript component}
gScriptingComponent := OpenDefaultComponent(kOSAComponentType,
                                            kAppleScriptSubtype);
```

The scripting component routines described in this chapter include eight groups of optional routines that scripting components can support. If necessary, you can use the FindNextComponent function and other Component Manager routines to find a scripting component that supports a specific group of routines or to determine whether a particular scripting component supports a specific group of routines.

When you call FindNextComponent, you can provide, in a component description record (a data structure of type ComponentDescription), information about the scripting component you wish to find. The flag bits in the componentFlags field of a component description record provide this information. To find a scripting component that supports a specific group of optional routines, you can specify one or more of these constants in the componentFlags field:

```
CONST
    kOSASupportsCompiling      = $0002;
    kOSASupportsGetSource      = $0004;
    kOSASupportsAECoercion     = $0008;
    kOSASupportsAESending      = $0010;
    kOSASupportsRecording      = $0020;
    kOSASupportsConvenience    = $0040;
    kOSASupportsDialects       = $0080;
    kOSASupportsEventHandling  = $0100;
```

The routines that correspond to these constants are described in "Optional Scripting Component Routines," which begins on page 10-46.

Note

Although the generic scripting component supports all the scripting component routines represented by these flags, the support it can actually provide depends on the individual components with which it opens connections. u

Listing 10-1 shows how you can use these flags and the FindNextComponent function to locate a scripting component with specific characteristics. The componentFlags field of the component description record passed to FindNextComponent specifies the flags kOSASupportsCompiling and kOSASupportsGetSource. Because the componentFlagsMask field also specifies these flags, the FindNextComponent function locates a scripting component that supports these routines, regardless of whether or not it supports any others. The FindNextComponent function returns a

component identifier that you can then use to get more information about
the component or to open it.

Listing 10-1   Locating a scripting component that supports specific optional routines

```
FUNCTION MyConnectToScripting (VAR scriptingComponent: ComponentInstance)
                                : OSAError;
VAR
   descr, descr2: componentDescription;
   comp:          component;
   myErr:         OSErr;
BEGIN
   {fill in the fields of the component description record}
   {first specify component type, subtype, and manufacturer}
   descr.componentType := kOSAComponentType; {must be scripting component}
   descr.componentSubType := OSType(0); {any OSA component matching spec}
   descr.componentManufacturer := OSType(0); {don't care about manufacturer}

   {specify component flags and flags mask}
   descr.componentFlags := kOSASupportsCompiling + kOSASupportsGetSource;
   descr.componentFlagsMask :=
      kOSASupportsCompiling + kOSASupportsGetSource;

   {locate and open the specified component}
   comp := FindNextComponent(Component(0), descr); {0 indicates all }
                                                   { registered components }
                                                   { will be searched}
   {check whether the found component is the generic scripting component; }
   { if so, skip it and find the next matching component}
   myErr := GetComponentInfo(comp, descr2, NIL, NIL, NIL);
   IF descr2.componentSubType = kOSAGenericScriptingComponentSubtype THEN
      comp := FindNextComponent(comp, descr);
   IF comp = 0 THEN
      MyConnectToScripting := kComponentNotFound
   ELSE
   BEGIN
      scriptingComponent := OpenComponent(comp);
      IF scriptingComponent = 0 THEN
         MyConnectToScripting := kComponentNotFound
      ELSE
         MyConnectToScripting := noErr;
   END;
END;
```

Because the generic scripting component supports all the standard scripting component routines, the `MyConnectToScripting` function in Listing 10-1 checks whether the found component is the generic scripting component and, if so, skips it. If for any reason `FindNextComponent` can't locate and open a scripting component that supports the specified routines, `MyConnectToScripting` returns the application-defined constant `kComponentNotFound`.

For more information about locating and opening components with specific characteristics, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

Using Scripting Component Routines

The following sections describe how to use some of the standard scripting component routines to manipulate and execute scripts from within your application. For an overview of these routines, see "Manipulating and Executing Scripts," which begins on page 7-22.

The first section describes how to compile and execute source data for a script. The remaining sections describe how you can use scripting component routines to

n   get a handle to a compiled script and save the data as a resource

n   load and execute a previously saved and compiled script

n   load, modify, recompile, and save a compiled script

n   redirect Apple events to handlers in script contexts

n   supply a resume dispatch function

n   supply an alternative active function

n   supply alternative send and create functions

n   record Apple events in compiled scripts and display equivalent source data to the user

Compiling and Executing Source Data

This section describes how you can use scripting component routines to obtain source data from users, compile the source data, and execute the compiled script. To create and execute a script using the Script Editor application, a user can type the script, then click the Run button to execute it. Your application can provide similar capabilities.

To allow users to write a new script and then execute it, your application must use scripting component routines to compile and execute the source data. To compile source data in a new script with a new script ID, pass the constant `kOSANullScript` (rather than an existing script ID) in the last parameter of the `OSACompile` function. This causes `OSACompile` to return a new script ID in the same parameter.

To execute a compiled script, your application must specify, in addition to the script ID for the compiled script, a script context: either the corresponding scripting component's

default context or a script ID for the global context created by that scripting component. Script contexts maintain state information for the execution of scripts. Your application can use script contexts to control the binding of variables used in scripts that it executes. For example, if your application saves its own global context and reuses it every time a script is executed, the binding of variables used in the script is maintained after the user restarts the computer. If your application does not specify a script context, the AppleScript component uses a single default context whenever it executes the script. A scripting component's default context binds the variables used in the script only until the user quits the application.

To specify a scripting component's default context, pass the constant `kOSANullScript` in the third parameter of the `OSAExecute` function; to specify some other global context, pass its script ID in the third parameter.

The `MyDoNewScript` procedure in Listing 10-2 allows a user to type a script in the appropriate scripting language, then compiles the script, executes the compiled script using a global context provided by the application, and displays the result to the user.

The `MyDoNewScript` procedure begins by calling the `OSAScriptingComponentName` function to obtain the name of the scripting component specified by `gScriptingComponent`. This name is passed to the application-defined function `MyGetUserScriptText`.

Note

If you are using the generic scripting component, you can use the `OSAGetDefaultComponent` function to get the subtype code for the default scripting component (that is, the scripting component used by the generic scripting component for new scripts). You can then get an instance of the default scripting component by passing its subtype code to `OSAGetScriptingComponent`. Finally, you can pass that instance to `OSAScriptingComponentName` to obtain the default scripting component's name. For more information about the default scripting component and routines you can use with the generic scripting component, see "Generic Scripting Component Routines," which begins on page 10-84. u

The `MyGetUserScriptText` function displays the name of the scripting language to use in a script-editing window or message box that allows the user to type and execute a new script. After it obtains the source data for the new script, the `MyDoNewScript` procedure sets the `scriptID` variable to `kOSANullScript`. The procedure then passes the source data and `scriptID` to the `OSACompile` function. When the script ID passed to `OSACompile` is `kOSANullScript`, `OSACompile` returns, in the same parameter, a new script ID for the resulting compiled script. The `MyDoNewScript` procedure then passes the new script ID to the `OSAExecute` function.

In addition to a component instance and the script ID for the compiled script to be executed, `OSAExecute` takes a script ID for a script context and a parameter that contains the mode flags, if any, for script execution. In Listing 10-2, the script ID passed to `OSAExecute` for the script context is `gContext`, a global context provided by the application. The constant `kOSAModeNull` in the next parameter indicates that no mode flags are set for script execution.

Listing 10-2   A routine that compiles and executes source data

```
PROCEDURE MyDoNewScript;
VAR
   componentName, scriptText, resultText: AEDesc;
   scriptID, resultID:                    OSAID;
   myOSAErr, ignoreErr:                   OSAError;
BEGIN
   {get the scripting component's name so you can show }
   { the user which scripting language to use}
   myOSAErr := OSAScriptingComponentName(gScriptingComponent,
                                         componentName);
   IF myOSAErr = noErr THEN
   BEGIN       {get the user's script text, then compile it}
      MyGetUserScriptText(componentName, scriptText);
      {to create a new compiled script using the user's script }
      { text, pass kOSANullScript to OSACompile as the script ID }
      { for the script to be compiled}
      scriptID := kOSANullScript;
      myOSAErr := OSACompile(gScriptingComponent, scriptText,
                             kOSAModeNull, scriptID);
      ignoreErr := AEDisposeDesc(scriptText);
   END;
   IF myOSAErr = noErr THEN
   BEGIN
      {execute the script in a global context}
      myOSAErr := OSAExecute(gScriptingComponent, scriptID,
                             gContext, kOSAModeNull, resultID);
      ignoreErr := OSADispose(gScriptingComponent, scriptID);
      IF myOSAErr = noErr THEN
      BEGIN
         {convert the script value returned by OSAExecute to }
         { text that can be displayed to the user}
         myOSAErr := OSADisplay(gScriptingComponent, resultID,
                                typeChar, kOSAModeNull, resultText);
         ignoreErr := OSADispose(gScriptingComponent, resultID);
         {show result to user}
         MyShowUserResult(resultText);
         ignoreErr := AEDisposeDesc(resultText);
      END;
   END;
   IF myOSAErr = errOSAScriptError THEN
      MyGetScriptErrorInfo;
END;
```

If script execution is successful, the `MyDoNewScript` procedure passes the script ID for the resulting script value to the `OSADisplay` function and calls the `MyShowUserResult` procedure to display the script value to the user. It also disposes of the script data for the compiled script. If `OSAExecute` or `OSACompile` returns the result code `errOSAScriptError`, the `MyDoNewScript` procedure calls the `MyGetScriptErrorInfo` procedure shown in Listing 10-3, which uses the `OSAScriptError` function to obtain more information about the error.

Whenever a scripting component routine returns the result code `errOSAScriptError`, you can use `OSAScriptError` to obtain more information about the error. The second parameter of the `OSAScriptError` function is a constant that specifies the kind of error information to be returned, and the third parameter is the descriptor type for the descriptor record in which the additional error information will be returned.

The `MyGetScriptErrorInfo` procedure in Listing 10-3 calls `OSAScriptError` three times: once to obtain an error number for either a system error or a scripting component error, once to obtain a text description of the error, and once to obtain error-range information. (For more information about specifying descriptor types for `OSAScriptError`, see page 10-37.) Finally, the `MyGetScriptErrorInfo` procedure extracts the starting and ending positions of the error range in the source data and calls the application-defined procedure `MyIndicateError` to display the error information to the user. Note that your application is responsible for disposing of any descriptor records that are created.

You should use the `OSACompile` and `OSAExecute` functions as shown in Listing 10-2 if you expect the user to execute the compiled script several times or manipulate it in some other way. If you want to compile and execute a script just one time and don't need to keep the compiled script in memory after it has been executed, you can use either `OSACompileExecute` or `OSADoScript` if these functions are supported by the scripting component you specify.

The `OSACompileExecute` function takes a component instance, a descriptor record for the source data to be compiled and executed, a context ID, and a `modeFlags` parameter. It executes the resulting compiled script, disposes of the compiled script, and returns the script ID for the resulting script value.

The `OSADoScript` function takes a component instance, a descriptor record for source data, a context ID, a text descriptor type, and a `modeFlags` parameter. It compiles and executes the script, returns a descriptor record for the text that corresponds to the resulting script value, and disposes of both the compiled script and the script value.

Listing 10-3   A procedure that uses `OSAScriptError` to get information about an execution
error

```
PROCEDURE MyGetScriptErrorInfo;
TYPE
    OSErrPtr        = ^OSErr;
    OSErrHandle     = ^OSErrPtr;
VAR
    errorMessage:                 Handle;
    startPos, endPos:             Integer;
    desc, recordDesc:             AEDesc;
    actualType:                   DescType;
    actualSize:                   Size;
    scriptErr, myErr, ignoreErr:  OSErr;
    myOSAErr:                     OSAError;
BEGIN
    myOSAErr := OSAScriptError(gScriptingComponent,
                            kOSAErrorNumber, typeShortInteger,
                            desc);
    scriptErr := OSErrHandle(desc.dataHandle)^^;
    ignoreErr := AEDisposeDesc(desc);
    myOSAErr := OSAScriptError(gScriptingComponent,
                            kOSAErrorMessage, typeChar, desc);
    errorMessage := desc.dataHandle;
    myOSAErr := OSAScriptError(gScriptingComponent,
                            kOSAErrorRange, typeOSAErrorRange,
                            desc);
    myErr := AECoerceDesc (desc, typeAERecord, recordDesc);
    ignoreErr := AEDisposeDesc(desc);
    myErr := AEGetKeyPtr(recordDesc, keySourceStart,
                        typeShortInteger, actualType,
                        Ptr(@startPos), sizeOf(startPos),
                        actualSize);
    myErr := AEGetKeyPtr(recordDesc, keySourceEnd,
                        typeShortInteger, actualType,
                        Ptr(@endPos), sizeOf(endPos),
                        actualSize);
    ignoreErr := AEDisposeDesc(recordDesc);
    MyIndicateError(scriptErr, errorMessage, startPos, endPos);
    {add your own error checking}
END;
```

Saving Script Data

After creating a new script (or after modifying a previously saved script), a user may want to save it.

IMPORTANT

Your application should usually save scripts as script data rather than source data, so that it can reload and execute the data without compiling it. s

Before saving script data, your application can use the `OSAStore` function to obtain a handle to the data. The `OSAStore` function takes four input parameters: a component instance that identifies a connection with a scripting component, a script ID for the script data to be stored, a desired descriptor type for the descriptor record to be returned, and a parameter that contains mode flags for use by individual scripting components. It returns a descriptor record for the script data in the fifth parameter.

The sections that follow describe the storage formats used by `OSAStore` and the resource and file types for script data.

Storage Formats for Script Data

The descriptor record returned by `OSAStore` can be either a generic storage descriptor record or a component-specific storage descriptor record:

n A **generic storage descriptor record** is a special kind of descriptor record of type `typeOSAGenericStorage` that can be used to store script data created by any scripting component.

n A **component-specific storage descriptor record** is a descriptor record whose descriptor type is the scripting component subtype value for the scripting component that created the script data.

Figure 10-1 illustrates the logical arrangement of a generic storage descriptor record. The descriptor type for a generic storage descriptor record is always `typeOSAGenericStorage`, and the data referred to by the descriptor record's handle is always followed by a trailer containing the subtype value for the scripting component that created the script data.

Figure 10-1   A generic storage descriptor record
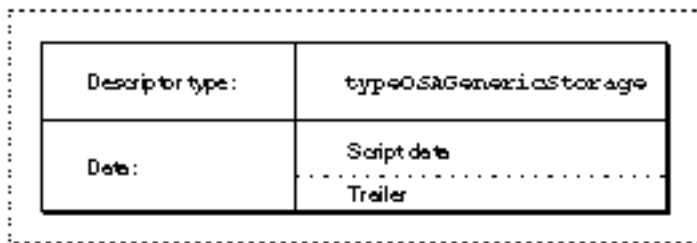
Figure 10-2 illustrates the logical arrangement of a component-specific storage descriptor record. The descriptor type for a component-specific storage descriptor record is the subtype value for the scripting component that created the script data, and the data referred to by the descriptor record's handle consists of the script data only, with no trailer.

Figure 10-2   A component-specific storage descriptor record
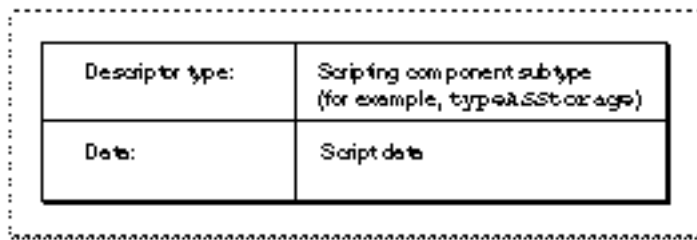
In most cases it is safest to request a handle to script data in the form of a generic storage descriptor record, regardless of the scripting component subtype you pass to the `OSAStore` function.

If the presence of the trailer in a generic storage descriptor record does not interfere with the script data, that data may be used for a wide variety of purposes. For example, if an application uses script IDs to refer to XCMDs, it can call `OSAStore` with a desired type of `typeOSAGenericStorage`. The data for the resulting descriptor record consists of the XCMD data followed by a trailer indicating that the script data was created by a scripting component that executes XCMDs. Because the trailer does not interfere with the use of the data, the data may actually be used as an XCMD. Thus, an application can save XCMDs as script data and load and execute them after it has opened a connection with the generic scripting component.

However, in some cases adding a trailer to script data may interfere with script execution. For example, suppose the data for a generic storage descriptor record consists of sound data. If a scripting component attempts to play the data from beginning to end as sound data, the trailer will interfere with the resulting sound. In this case, an application must open an explicit connection with a scripting component that can play sounds before saving the data, and then call `OSAStore` with a desired type that consists of the subtype for that scripting component.

Resource and File Types for Script Data

When the `OSAStore` function returns a descriptor record of the specified type, your application can save the descriptor record's data as a resource of type `'scpt'` or write it to the data fork of a document.

The generic scripting component subtype, the generic storage descriptor type, and the resource type for stored script data all have the same value, though they serve different purposes.

```
CONST
    kOSAGenericScriptingComponentSubtype = 'scpt';
    kOSAScriptResourceType = kOSAGenericScriptingComponentSubtype;
    typeOSAGenericStorage = kOSAScriptResourceType;
```

If you want to save script data as a compiled script file or as a script application, save it as a resource of type `'scpt'`. The Script Editor application uses resource ID 128, but you can use any valid resource ID. Save the script comment that accompanies the script data as resources of type `'TEXT' and 'styl'` with resource ID 1128. (See Figure 7-1 on page 7-6 for an example of a script comment.) Each script file can contain only one script and one script comment. The file type for a compiled script file should be `'osas'`.

A script application has the file type `'APPL'`. If a script application has the creator signature `'aplt'`, a user can initiate execution of the script it contains by opening it from the Finder. If a script application has the creator signature `'dplt'` and contains a user-defined handler for the Open Documents event, a user can initiate execution of the handler by dragging a document or folder icon over the script application's icon. For more information about the file formats used for script files, see "Script Editors and Script Files" on page 7-6.

Script applications must include a `'SIZE'` resource and two `'CODE'` resources with resource IDs 0 and 1. These resources should be identical to those in the sample script application files provided by Apple Computer, Inc. (except that you can change the size of the memory partition). The `'CODE'` resources contain bootstrap code that instantiates the script application component. The **script application component,** which is registered with the Component Manager at startup, provides the code that loads the script to be run and passes the resulting script ID to the appropriate component.

When the user opens a script application from the Finder, the Finder sends the script application an Open Application event. If the scripting component that created the script supports OSAExecuteEvent, the script application component passes the Open Application event and the script ID for the script to OSAExecuteEvent. If the scripting component doesn't support OSAExecuteEvent, the script application component passes the script ID to OSAExecute.

Loading and Executing Script Data

Figure 7-4 on page 7-13 illustrates how an application might execute a script whenever the user presses the Tab key after entering a customer's name in the "Customer Name" field of an electronic form. To execute a script in response to some user action, your application must be able to load and execute the script data for a compiled script.

This section describes how to load and execute a previously compiled and saved script. The next section, "Modifying and Recompiling a Compiled Script," describes how to allow a user to modify a compiled script.

The `OSALoad` function takes three input parameters: a component instance that identifies a connection with a scripting component; a descriptor record that contains a handle to the script data to be loaded; and a parameter that contains flags for use by individual scripting components. The function returns, in the fourth parameter, a script ID for the script data.

When your application calls `OSALoad` with a component instance that identifies a connection with the generic scripting component, the generic scripting component in turn uses a connection with the scripting component that created the script data (if that component is registered with the Component Manager on the local computer). If the descriptor record passed to `OSALoad` is of type `typeOSAGenericStorage`, the generic scripting component uses the trailer that follows the script data to determine which scripting component to open a connection with. If the descriptor record's type is the subtype value for some other scripting component, the generic scripting component does not look for a trailer and uses the descriptor type to identify the scripting component.

When your application calls `OSALoad` with a component instance that identifies a connection to any scripting component other than the generic scripting component, that component can load script data only if it was saved as the data for a descriptor record whose descriptor type matches the scripting component's subtype. In this case, however, your application easily can take advantage of additional routines and other special capabilities provided by that scripting component.

It is also possible to call `OSALoad` using the generic scripting component, then use generic scripting component routines to identify the specific component associated with the loaded script. This allows you to use component-specific routines with a script originally loaded by the generic scripting component. For information about how to do this, see "Routines Used by Scripting Components," which begins on page 10-92.

The `OSALoad` function returns a script ID for the loaded script data. The generic scripting component always associates the returned script ID with the scripting component that created the script. In this way, it can use a connection with that component again whenever the client application passes the returned script ID to other scripting component routines.

Listing 10-4 shows a procedure that loads and executes a script. The `MyLoadAndExecute` procedure takes a handle to script data that was previously saved using a generic storage descriptor record, obtains a script ID for the equivalent compiled script, executes the compiled script in the default context, and disposes of both the compiled script and the resulting script value ID. If the `OSAExecute` function returns a script execution error, `MyLoadAndExecute` obtains further information about the error and displays it to the user.

Listing 10-4   A routine that loads and executes script data previously saved using a generic
storage descriptor record

```
PROCEDURE MyLoadAndExecute (scriptData: Handle);
VAR
    scriptDesc:                    AEDesc;
    scriptID, resultID:            OSAID;
    scriptText:                    AEDesc;
    myOSAErr, ignoreErr:           OSAError;
BEGIN
    {load the script data}
    scriptDesc.descriptorType := typeOSAGenericStorage;
    scriptDesc.dataHandle := scriptData;
    myOSAErr := OSALoad(gScriptingComponent, scriptDesc,
                         kOSAModeNull, scriptID);
    IF myOSAErr = noErr THEN
    BEGIN
        {execute the resulting compiled script in the default }
        { context}
        myOSAErr := OSAExecute(gScriptingComponent, scriptID,
                               kOSANullScript, kOSAModeNull,
                               resultID);
        ignoreErr := OSADispose(gScriptingComponent, scriptID);
        ignoreErr := OSADispose(gScriptingComponent, resultID);
    END;
    IF myOSAErr = errOSAScriptError THEN
        MyGetScriptErrorInfo;
END;
```

The OSALoad function in Listing 10-4 takes a component instance, a generic storage
descriptor record for the script data to be loaded, and a parameter that contains the
mode flags, if any, for loading the script. In this case the constant kOSAModeNull
indicates that no mode flags are set. The OSALoad function returns a script ID for the
resulting compiled script, which the MyLoadAndExecute procedure then passes to the
OSAExecute function.

In addition to a component instance and the script ID for the compiled script to be
executed, the OSAExecute function takes a script ID for a context and a parameter that
contains the mode flags, if any, for script execution. In Listing 10-4, the script ID passed
to OSAExecute for the script context is kOSANullScript, indicating that the scripting
component can use its default context to bind any variables. The constant
kOSAModeNull in the next parameter indicates that no mode flags are set for script
execution.

After disposing of the compiled script and the resulting script value, `MyLoadAndExecute` checks the result code returned by `OSAExecute`. If it is `errOSAScriptError`, `MyLoadAndExecute` calls the `MyGetScriptErrorInfo` procedure (see Listing 10-3 on page 10-11), which in turn uses the `OSAScriptError` function to obtain more information about the error.

You can use the `OSALoad` and `OSAExecute` functions as shown in Listing 10-4 if you expect the user to execute the compiled script several times or manipulate it in some other way. If you want to load and execute a script just one time and don't need to keep the compiled script in memory after it has been executed, you can use `OSALoadExecute` instead of `OSALoad`, `OSAExecute`, and `OSADispose`. This function takes a component instance, a descriptor record for the script data to be loaded and executed, a context ID, and a `modeFlags` parameter. The `OSALoadExecute` function executes the resulting compiled script, disposes of the compiled script, and returns the script ID for the resulting script data.

Modifying and Recompiling a Compiled Script

In addition to loading and executing a previously compiled and saved script as described in the previous section, your application can use the scripting component routines described in this section to decompile a compiled script, display the equivalent source data to users for editing, and recompile the source data after editing is completed. For example, if a user wants to change the script shown in Figure 7-4 on page 7-13 so that it refers to some other database or looks up other information in addition to the customer's address, the forms application can use scripting component routines to display the compiled script to the user and recompile it after the user has modified it.

You can use the `OSAGetSource` function to obtain the source data for a compiled script. The `OSAGetSource` function takes a component instance, a script ID for the compiled script, and the desired type of the resulting descriptor record. If you specify a component instance that identifies a connection with the generic scripting component, you can use `OSAGetSource` to get the source data for any compiled script created by a scripting component that is registered with the Component Manager on the local computer. If you specify a component instance that identifies an explicit connection with a scripting component, you can use `OSAGetSource` only to get the source data for scripts that were compiled by that scripting component.

The `MyEditGenericScript` procedure in Listing 10-5 shows how you can use the `OSAGetSource` function with a component instance that identifies a connection to the generic scripting component. The `MyEditGenericScript` function gets the source data for a compiled script, allows the user to edit it, and recompiles the script so the original script ID refers to the recompiled script data.

Listing 10-5   A routine that displays a compiled script for editing and recompiles it

```
PROCEDURE MyEditGenericScript (scriptID: OSAID);
VAR
   scriptText:    AEDesc;
   myOSAErr:      OSAError;
   ignoreErr:     OSErr;
BEGIN
   {first get the source data}
   myOSAErr := OSAGetSource(gScriptingComponent, scriptID,
                              typeChar, scriptText);
   {call the application's primitive text editor}
   MyEditText(scriptText.dataHandle);
   {now compile the edited script data in scriptText using }
   { the scripting component that originally created it; }
   { passing the original script ID to OSACompile causes }
   { OSACompile to replace the original script with the new one}
   myOSAErr := OSACompile(gScriptingComponent, scriptText,
                         kOSAModeNull, scriptID);
   ignoreErr := AEDisposeDesc(scriptText);
   IF myOSAErr = errOSAScriptError THEN
      MyGetScriptErrorInfo;
END;
```

After obtaining the source data for the script, the MyEditGenericScript procedure calls the MyEditText function, which displays the application's own primitive text editor and allows the user to edit the source data. After the user has finished editing the script, MyEditGenericScript passes the edited text and the script ID for the original compiled script to the OSACompile function, which updates the script ID so that it refers to the modified and recompiled script. The kOSAModeNull constant passed in the third parameter of OSACompile indicates that no mode flags are specified for compilation.

If the OSACompile function returns errOSAScriptError, the MyEditGenericScript procedure calls the MyGetScriptErrorInfo procedure shown in Listing 10-3 on page 10-11 to obtain information about the error.

After script data has changed as shown in Listing 10-5, your application should save the modified script data. Listing 10-6 shows how this could be done from a function that loads script data, calls the MyEditGenericScript procedure shown in Listing 10-5 to modify and recompile the script, then saves the modified script data.

Listing 10-6   A function that loads and modifies script data, then saves it using a generic
storage descriptor record

```
FUNCTION MyLoadAndModifyScriptData (resourceID: Integer)
                                  : OSAError;
VAR
   scriptDesc:       AEDesc;
   storageDescRec:   AEDesc;
   scriptID:         OSAID;
   myOSAErr:         OSAError;
   ignoreErr:        OSErr;
   myHndl:           Handle;
BEGIN
   scriptDesc.descriptorType := typeOSAGenericStorage;
   scriptDesc.dataHandle := GetResource(kOSAScriptResourceType,
                                     resourceID);
   myOSAErr := OSALoad(gGenericScriptingComponent, scriptDesc,
                     kOSAModeNull, scriptID);
   MyEditGenericScript (scriptID);
   myOSAErr := OSAStore(gScriptingComponent, scriptID,
                     typeOSAGenericStorage, kOSAModeNull,
                     storageDescRec);
   MyWriteResource(storageDescRec.dataHandle, resourceID);
   ignoreErr := AEDisposeDesc(scriptDesc);
   ignoreErr := AEDisposeDesc(storageDescRec);
END;
```

Using a Script Context to Handle an Apple Event

The preceding sections describe how you can load, compile, modify, and execute scripts under circumstances determined by your application. Your application can use these techniques to associate a script with an Apple event object or application object and execute the script when the user manipulates the object in some way.

Another way to execute a script is to use a script context (called a **script object** in AppleScript) to handle an Apple event. To do this, your application passes both the event and the script context to OSAExecuteEvent or OSADoEvent. You can also associate script contexts with Apple event objects—that is, objects in your application that can be identified by object specifier records. If an Apple event acts on an object with which a script context is associated, your application attempts to use the script context to handle the Apple event.

For example, Figure 7-7 on page 7-26 shows how you can use a general Apple event handler to provide initial processing for all Apple events received by your application. Listing 10-7 shows an example of such a handler.

You install a general Apple event handler like the one in Listing 10-7 in your application's special handler dispatch table using the constant `keyPreDispatch`:

```
myErr := AEInstallSpecialHandler(keyPreDispatch,
                                 @MyGeneralAppleEventHandler,
                                 FALSE);
```

When it receives an Apple event, the `MyGeneralAppleEventHandler` function in Listing 10-7 first extracts the event's direct parameter. It then calls another application-defined function, `MyGetAttachedScript`, which checks whether the direct parameter contains an object specifier record, calls `AEResolve` to locate the corresponding Apple event object, and returns a script ID for any script context attached to that object.

If a script context is associated with the object, `MyGeneralAppleEventHandler` passes the script context's script ID and the Apple event to the `OSADoEvent` function. Otherwise, `MyGeneralAppleEventHandler` returns `errAEEventNotHandled`, which causes the Apple Event Manager to look for an appropriate handler in the application's Apple event dispatch table or elsewhere using standard Apple event dispatching.

The `OSADoEvent` function in Listing 10-7 takes a component instance that identifies a connection with the generic scripting component. (If it has not already done so, the generic scripting component in turn opens a connection with the scripting component that created the script context.) In addition to the component instance, the Apple event, and the script ID for the script context, `OSADoEvent` takes a parameter that indicates no mode flags are set and a `VAR` parameter that contains any reply Apple event returned as a result of handling the event.

If the scripting component determines that a script context can't handle the specified event (for example, if an AppleScript script context doesn't include statements that handle the event), `OSADoEvent` returns `errAEEventNotHandled`. If `OSADoEvent` attempts to use the script context to handle the event, the function returns a reply event that contains either the resulting script value or, if an error occurred, information about the error.

The script context shown in Figure 7-7 contains an AppleScript handler for the Move event. Such handlers exist only as AppleScript statements in a script context and do not have corresponding entries in an application's Apple event dispatch table. However, a handler in a script context can modify or override the actions performed by an application's standard Apple event handlers installed in its Apple event dispatch table. The next section, "Supplying a Resume Dispatch Function," describes how this works.

Listing 10-7   A general Apple event handler that uses the `OSADoEvent` function

```
FUNCTION MyGeneralAppleEventHandler (event: AppleEvent;
                                     reply: AppleEvent;
                                     refcon: LongInt): OSErr;
VAR
   dp, resultDesc:    AEDesc;
   scriptID:          OSAID;
   myErr, ignoreErr: OSErr;
   myOSAErr:          OSAError;
BEGIN
   {get the direct parameter}
   myErr := AEGetParamDesc(event, keyDirectObject, typeWildCard,
                           dp);
   {get script ID for script context attached to object }
   { specified in direct parameter}
   IF MyGetAttachedScript(dp, scriptID) THEN
      {execute the handler in the script context handler and, if }
      { necessary, the default Apple event handler}
      myOSAErr := OSADoEvent(gScriptingComponent, event,
                             scriptID, kOSAModeNull, reply)
   ELSE
      myOSAErr := errAEEventNotHandled;
   ignoreErr := AEDisposeDesc(dp);
   MyGeneralAppleEventHandler := OSErr(myOSAErr);
END;
```

For more information about `OSADoEvent`, `OSAExecuteEvent`, and other routines
related to the use of script contexts to handle Apple events, see page 10-71.

Supplying a Resume Dispatch Function

Every scripting component calls a **resume dispatch function** during script execution if
the script contains the equivalent of an AppleScript `continue` statement within an
event handler. (See Figure 7-7 on page 7-26 for an example.) The resume dispatch
function dispatches the event specified by the script directly to the application's standard
handler for that event.

Thus, if the script context passed to OSADoEvent in Listing 10-7 specifies that the event passed in the event parameter should be continued—that is, handled by the application's standard Apple event handler for that event—the scripting component passes the event to the resume dispatch function currently set for that instance of the scripting component. The resume dispatch function attempts to redispatch the event to the handler installed in the application's Apple event dispatch table for that event. If the call to the resume dispatch function is successful, execution of the script proceeds from the point at which the resume dispatch function was called. If the call to the resume dispatch function is not successful, OSADoEvent returns errAEEventNotHandled in the keyErrorNumber parameter of the reply event. (Other routines that execute scripts, such as OSAExecute or OSAExecuteEvent, return errOSAScriptError in this situation, and a subsequent call to OSAScriptError with kOSAErrorNumber in the selector parameter returns errAEEventNotHandled.)

Some scripting components may provide routines that allow your application to set or get the pointer to the resume dispatch function used by a specified instance of a scripting component.

```
TYPE AEHandlerProcPtr = EventHandlerProcPtr;
```

A resume dispatch function takes the same parameters as an Apple event handler.

```
FUNCTION MyResumeDispatch (theAppleEvent: AppleEvent;
                           reply: AppleEvent; refCon: LongInt)
                           :OSErr;
```

To set the resume dispatch function for a scripting component, call OSASetResumeDispatchProc; to get the current dispatch function for a scripting component, call OSAGetResumeDispatchProc. If you do not set a resume dispatch function for a scripting component, it uses standard Apple event dispatching to dispatch the event, starting with the special handler dispatch table.

You can install a resume dispatch function using the OSASetResumeDispatchProc function. However, if you are using a general handler similar to that in Listing 10-7 on page 10-21 and you can rely on standard Apple event dispatching to dispatch the event correctly, you don't need to provide a resume dispatch function. Instead, you can specify kOSAUseStandardDispatch as the resume dispatch function and the constant kOSADontUsePhac as the reference constant when you call OSASetResumeDispatchProc.

```
myErr := OSASetResumeDispatchProc(gScriptingComponent,
              kOSAUseStandardDispatch, kOSADontUsePhac);
```

This causes the Apple Event Manager to redispatch events that would otherwise be passed to a resume dispatch function using standard Apple event dispatching—except that the Apple Event Manager bypasses your application's special handler dispatch table and thus won't call your general Apple event handler recursively.

When a scripting component calls your resume dispatch function, the A5 register is set up for your application, and your application is the current process.

Supplying an Alternative Active Function

Every scripting component calls an **active function** periodically during script compilation and execution. All scripting components support routines that allow your application to set or get the pointer to the active function used by that scripting component.

```
TYPE OSAActiveProcPtr = ProcPtr;
```

A pointer of type `OSAActiveProcPtr` points to a `MyActiveProc` function that takes a reference constant as a parameter.

```
FUNCTION MyActiveProc(refCon: LongInt): OSErr;
```

If you want your application to get time periodically during script compilation and execution for tasks such as spinning the cursor or checking for system-level errors, you should provide an alternative active function that performs those tasks. To set an alternative active function, call `OSASetActiveProc`; to get the current active function, call `OSAGetActiveProc`.

If you do not set an alternative active function for a scripting component, it uses its own default active function. A scripting component's default active function allows a user to cancel script execution by pressing Command-period and calls `WaitNextEvent` to give other processes time.

Your alternative active function can in turn call the scripting component's default active function. To do this, your application can call `OSAGetActiveProc` before calling `OSASetActiveProc` to set the alternative active function, then call the default active function directly when necessary. Some scripting components may also supply building-block routines that your application can use to construct an alternative active function.

Multithreaded applications may need to give time to other threads while one thread is waiting for the scripting component to complete compilation or execution of a script. You can provide an alternative send function and an idle function that allows threads to be switched (see "Alternative Send Functions" on page 10-25). However, the Apple Event Manager calls an idle function only after an Apple event has been sent, whereas a scripting component calls an active function at regular intervals throughout script compilation and execution. Thus, to give time to multiple threads, you may want to provide an alternative active function in addition to an alternative send function and an idle function.

When a scripting component calls your alternative active function, the A5 register is set up for your application, and your application is the current process.

Supplying Alternative Create and Send Functions

Every scripting component calls a **create function** whenever it creates an Apple event during script execution, and a **send function** whenever it sends an Apple event. Scripting components that use Apple events during script compilation, including AppleScript, also call create and send functions during compilation.

Some scripting components may provide routines that allow your application to set or get the pointers to the create and send functions used by that scripting component. If your application does not set alternative send and create functions, the scripting component uses the standard Apple Event Manager functions `AESend` and `AECreateAppleEvent`, which it calls with its own default parameters.

A scripting component that supports the routines you can use to set or get alternative create and send functions has the `kOSASupportsAESending` bit set in its component description record. For more information about using the Component Manager to find a scripting component that supports specific routines, see "Connecting to a Scripting Component," which begins on page 10-3.

When a scripting component calls your alternative send or create function, the A5 register is set up for your application, and your application is the current process.

Alternative Create Functions

A scripting component that allows your application to set or get its create function uses a pointer to identify the current create function.

```
TYPE AECreateAppleEventProcPtr = ProcPtr;
```

A pointer of type `AECreateAppleEventProcPtr` points to a `MyAECreateProc` function that takes the same parameters as the `AECreate` function plus a reference constant.

```
FUNCTION MyAECreateProc (theAEEventClass: AEEventClass;
                         theAEEventID: AEEventID;
                         target: AEAddressDesc;
                         returnID: Integer;
                         transactionID: LongInt;
                         VAR result: AppleEvent;
                         refCon: LongInt): OSErr;
```

Your application can use an alternative create function to gain control over the creation and addressing of Apple events. This can be useful, for example, if your application needs to add its own transaction code to the event.

To set an alternative create function, call `OSASetCreateProc`; to get the current create function, call `OSAGetCreateProc`. If you do not set an alternative create function for a scripting component, it uses the standard Apple Event Manager function `AECreateAppleEvent`, which it calls with its own default parameters.

Your alternative create function can in turn call the scripting component's default create function. To do this, your application can call `OSAGetCreateProc` before calling `OSASetCreateProc` to set the alternative create function, then call the default create function directly when necessary.

Alternative Send Functions

A scripting component that allows your application to set or get its send function uses a pointer to identify the current send function.

```
TYPE AESendProcPtr = ProcPtr;
```

A pointer of type `AESendProcPtr` points to a `MyAESendProc` function that takes the same parameters as the `AECreate` function plus a reference constant.

```
FUNCTION MyAESendProc (theAppleEvent: AppleEvent;
                       VAR reply: AppleEvent;
                       sendMode: AESendMode;
                       sendPriority: AESendPriority;
                       timeOutInTicks: LongInt;
                       idleProc: IdleProcPtr;
                       filterProc: EventFilterProcPtr;
                       refCon: LongInt): OSErr;
```

Your application can use an alternative send function to perform almost any action instead of or in addition to sending Apple events. For example, it can modify Apple events before sending them, save copies of Apple events before sending them, or substitute some other specialized mechanism for sending Apple events.

To set an alternative send function, call `OSASetSendProc`; to get the current send function, call `OSAGetSendProc`. If you do not set an alternative send function for a scripting component, it uses the standard Apple Event Manager function `AESend`, which it calls with its own default parameters.

Your alternative send function can in turn call the scripting component's default send function. To do this, your application can call `OSAGetSendProc` before calling `OSASetSendProc` to set the alternative send function, then call the default send function directly when necessary.

After a scripting component successfully calls a send function, the scripting component proceeds with script execution. If a call to a send function is not successful, the scripting component returns `errOSAScriptError`, and a subsequent call to `OSAScriptError` with `kOSAErrorNumber` in the `selector` parameter returns `errAEEventNotHandled`.

Multithreaded applications need to allow other threads to execute while one thread is waiting for the response to an Apple event. You can accomplish this by supplying an idle function for your alternative send function that allows threads to be switched and by setting the `kAEQueueReply` flag in the `sendMode` parameter of the send function. However, if the call to the send function specifies the `kAENoReply` flag, be careful not to override it, because the user may have explicitly requested that no reply be returned or the `'aete'` resource may indicate that the application cannot reply to that event.

Note

The Apple Event Manager calls an idle function only after an Apple event has been sent, whereas a scripting component calls an active function at regular intervals throughout script compilation and execution. Thus, to give time to multiple threads, you may want to provide an alternative active function in addition to an alternative send function and an idle function. u

Some scripting components (including the current version of AppleScript) can execute only one script at a time per component instance. For this reason, a multithreaded application must provide a separate component instance for each script that it compiles or executes while it is also compiling or executing other scripts.

You should follow the rules for setting `sendMode` flags described in the chapter "Creating and Sending Apple Events" in this book when you set flags for the `sendMode` parameter of an alternative send function. Keep these additional guidelines in mind:

n   If the target application is on the local computer, you can set the `kAECanInteract` and `kAECanSwitchLayer` flags.

n   If the target application is on the local computer and the user has requested no reply, set the `kAENoReply`, `kAECanInteract`, and `kAECanSwitchLayer` flags.

n   If the target application is on a remote computer, set the `kAENeverInteract` flag and do not set the `kAECanSwitchLayer` flag.

Recording Scripts

If you want your application to record Apple events in the form of a compiled script, or if you are writing a script-editing application like Script Editor, you can use the `OSAStartRecording` and `OSAStopRecording` functions to start and stop recording into a specified script ID on a single computer. Both functions take a component instance and a script ID for a compiled script. When your application calls `OSAStartRecording`, the scripting component identified by the component instance

sends a Start Recording event to your application and installs a Receive Recordable Event handler in your application's Apple event dispatch table. When your application calls `OSAStopRecording`, the scripting component removes the handler.

An application acting as a recording process in this manner should not provide a handler for the Start Recording event. Instead, the Apple Event Manager receives the event and responds by sending a Recording On event to all running processes on the local computer. Thereafter, the Apple Event Manager sends copies of subsequent recordable events to the recording process, whose previously installed Receive Recordable Event handler, much like a handler for event class `typeWildCard` and event ID `typeWildCard`, handles those recordable events by recording them in the compiled script specified in the call to `OSAStartRecording`.

Whenever the Receive Recordable Event handler receives a recordable event, the scripting component sends your application a Recorded Text event. The Recorded Text event contains the decompiled source data for the recorded event in the form of styled text.

Recorded Text—append styled text to script editor window

Event class	`kOSASuite`
Event ID	`kOSARecordedText`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeStyledText` or any other text descriptor type
Data:	Decompiled source data for recorded event
Description	Sent by a scripting component to a recording process for each event recorded after a call to `OSAStartRecording`

If your want your application to display the source data for recorded events as they are recorded, you must provide a handler for the Recorded Text event.

For more information about the Receive Recordable Event handler and Apple event recording, see "How Apple Event Recording Works," which begins on page 9-35.

Writing a Scripting Component

It is possible to create scripting components that execute a variety of scripts, including scripts that can be "run" in some sense but do not consist of statements in a scripting language. For example, script data can consist of an XCMD or even sound data that the appropriate scripting component can trigger or play back when it executes the script (see "Storage Formats for Script Data," which begins on page 10-12).

If you are developing a scripting component, you should read the instructions for creating components in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.* Every scripting component should also

n   Provide a component name in the scripting component's component resource that will make sense when displayed to users.

n   Support the standard scripting component routines described in "Required Scripting Component Routines," which begins on page 10-30.

n   Support some, all, or none of the optional scripting component routines, as appropriate for the tasks to be performed by the scripting component. These routines are described in "Optional Scripting Component Routines," which begins on page 10-46.

n   Use the three OSA routines `OSAGetStorageType`, `OSAAddStorageType`, and `OSARemoveStorageType` to inspect, add, or remove the trailers appended to script data in generic storage descriptor records. These routines are described in "Manipulating Trailers for Generic Storage Descriptor Records," which begins on page 10-92.

n   Send the Get AETE event when necessary. This event is described in "Handling the Get AETE Event," which begins on page 8-23.

Scripting Components Reference

This section describes the standard scripting component data structures and routines your application can use to manipulate and execute scripts. This section also describes additional routines provided by the AppleScript scripting component and three routines called by scripting components.

The first section, "Data Structures," describes the principal data types used by scripting component routines. "Required Scripting Component Routines," which begins on page 10-30, describes the standard scripting component routines that all scripting components must support. "Optional Scripting Component Routines," which begins on page 10-46, describes additional standard scripting component routines that scripting components are not required to support.

Your application can use the Component Manager to find a scripting component that supports specific optional routines or to determine whether a particular scripting component supports a specific group of routines. For information about how to do this, see "Connecting to a Scripting Component," which begins on page 10-3.

"AppleScript Component Routines," which begins on page 10-80, describes additional routines supported by the AppleScript component. "Generic Scripting Component Routines" which begins on page 10-84, describes routines you can use to get instances of specific components and perform other useful tasks when you are using multiple scripting components. "Routines Used by Scripting Components," which begins on page 10-92, describes three routines that all scripting components can use to manipulate trailers for generic storage descriptor records.

Data Structures

This section describes the principal data structures and Component Manager type codes used by the standard scripting component routines. Data structures used by individual routines are described with the appropriate routines in the sections that follow.

The Component Manager type code for components that support the standard scripting component routines is `'osa '`, and the subtype code for the generic scripting component is `'scpt'`.

```
CONST
    kOSAComponentType                    = 'osa ';
    kOSAGenericScriptingComponentSubtype  = 'scpt';
```

Because all results returned by the Component Manager are of type `ComponentResult` (a long integer), scripting components also define this type for result codes.

```
TYPE
    OSAError = ComponentResult;
```

Scripting components keep track of script data in memory by means of script IDs of type `OSAID`.

```
TYPE OSAID = LongInt;
```

A scripting component assigns a script ID when it creates the associated script data (that is, a compiled script, a script value, a script context, or other kinds of script data supported by a scripting component) or loads it into memory. The scripting routines that create, load, compile, and execute scripts all return script IDs, and your application must pass valid script IDs to the other routines that manipulate scripts. A script ID remains valid until a client application calls `OSADispose` to reclaim the memory used for the corresponding script data.

If the execution of a script does not result in a value, `OSAExecute` returns the constant `kOSANullScript` as the script ID. If a client application passes `kOSANullScript` to the `OSAGetSource` function instead of a valid script ID, the scripting component should display a null source description (possibly an empty text string). If a client application passes `kOSANullScript` to `OSAStartRecording`, the scripting component creates a new compiled script for editing or recording.

```
CONST kOSANullScript = 0;
```

Required Scripting Component Routines

This section describes the routines that all scripting components must support. Your application can use these routines to save and load script data, execute and dispose of scripts, get script information, and manipulate the active function. "Optional Scripting Component Routines," which begins on page 10-46, describes additional routines your application can use with scripting components that support them.

Saving and Loading Script Data

The `OSAStore` function takes a script ID and returns a copy of the corresponding script data in the form of a storage descriptor record. You can then save the script data as a resource or write it to the data fork of a document. The `OSALoad` function takes script data in a storage descriptor record and returns a script ID.

OSAStore

You can use the `OSAStore` function to get a handle to script data in the form of a storage descriptor record.

```
FUNCTION OSAStore(scriptingComponent: ComponentInstance;
                  scriptID: OSAID;
                  desiredType: DescType;
                  modeFlags: LongInt;
                  VAR resultingScriptData: AEDesc): OSAError;
```

`scriptingComponent`
: A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

`scriptID`
: The script ID for the script data for which to obtain a data handle.

`desiredType`
: The desired type of the descriptor record to be returned. If you want to store the script data in the form used by a generic storage descriptor record, specify `typeOSAGenericStorage`.

`modeFlags`
: Information used by individual scripting components. To avoid setting any mode flags, specify `kOSAModeNull`. To indicate that only the minimum script data required to run the script should be returned, pass `kOSAModePreventGetSource` in this parameter. (In this case the script data returned is not identical to the compiled script data and can't be used to generate source data.) If the `scriptID` parameter identifies a script context, you can pass `kOSAModeDontStoreParent` in this parameter to store the script context without storing its parent context.

`resultingScriptData`
: The resulting descriptor record.

DESCRIPTION

The `OSAStore` function writes script data to a descriptor record so that the data can later be saved in a resource or written to the data fork of a document. You can then reload the data for the descriptor record as a compiled script (although possibly with a different script ID) by passing the descriptor record to `OSALoad`.

If you want the returned script data to be as small as possible and you are sure that you won't need to display the source data to the user, specify the `kOSAModePreventGetSource` flag in the `modeFlags` parameter. If the `scriptID` parameter identifies a script context and you don't want the returned script data to include the associated parent context, specify the `kOSAModeDontStoreParent` flag in the `modeFlags` parameter.

The desired type is either `typeOSAGenericStorage` (for a generic storage descriptor record) or a specific scripting component subtype value (for a component-specific storage descriptor record).

To store either a generic storage descriptor record or a component-specific storage descriptor record with your application's resources, use `'scpt'` as the resource type. The generic scripting component subtype, the generic storage descriptor type, and the resource type for stored script data all have the same value, though they serve different purposes.

```
CONST
    kOSAGenericScriptingComponentSubtype = 'scpt';
    kOSAScriptResourceType = kOSAGenericScriptingComponentSubtype;
    typeOSAGenericStorage = kOSAScriptResourceType;
```

RESULT CODES

`noErr`	0	No error
`errOSASystemError`	–1750	General scripting system error
`errOSAInvalidID`	–1751	Invalid script ID
`errOSABadStorageType`	–1752	Desired type not supported by this scripting component
`badComponentInstance`	$80008001	Invalid component instance

SEE ALSO

For more information about storage formats for script data, see "Saving Script Data" on page 10-12.

For an example of the use of `OSAStore`, see Listing 10-6 on page 10-19.

OSALoad

You can use the OSALoad function to load script data.

```
FUNCTION OSALoad(scriptingComponent: ComponentInstance;
                 scriptData: AEDesc;
                 modeFlags: LongInt;
                 VAR resultingScriptID: OSAID): OSAError;
```

scriptingComponent
: A component instance created by a prior call to the Component Manager function OpenDefaultComponent or OpenComponent (see page 10-4).

scriptData
: The descriptor record containing the script data to be loaded.

modeFlags
: Information used by individual scripting components. To avoid setting any mode flags, specify kOSAModeNull. To indicate that only the minimum script data required to run the script should be loaded, pass kOSAModePreventGetSource in this parameter.

resultingScriptID
: The returned script ID for the compiled script.

DESCRIPTION

The OSALoad function loads script data and returns a script ID. The generic scripting component uses the descriptor record in the scriptData parameter to determine which scripting component should load the script. If the descriptor record is of type typeOSAGenericStorage, the generic scripting component uses the trailer at the end of the script data to identify the scripting component. If the descriptor record's type is the subtype value for another scripting component, the generic scripting component uses the descriptor type to identify the scripting component.

If you want the script ID returned by OSALoad to identify only the minimum script data required to run the script and you are sure that you won't need to display the source data to the user, specify the kOSAModePreventGetSource flag in the modeFlags parameter.

Scripting components other than the generic scripting component can load script data only if it has been saved in a descriptor record whose descriptor type matches the scripting component's subtype.

Script data may change after it has been loaded—for example, if your application allows the user to edit a script's source data. To test whether script data has been modified, pass its script ID to OSAGetScriptInfo. If it has changed, you can call OSAStore again to obtain a handle to the modified script data and save it.

RESULT CODES

noErr	0	No error
errOSACorruptData	–1702	Corrupt data
errOSASystemError	–1750	General scripting system error
errOSABadStorageType	–1752	Script data not for this scripting component
errOSADataFormatObsolete	–1758	Data format is obsolete
errOSADataFormatTooNew	–1759	Data format is too new
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For more information about the way scripting components interpret script data, see "Saving Script Data" on page 10-12.

For examples of the use of OSALoad, see Listing 10-4 on page 10-16 and Listing 10-6 on page 10-19.

Executing and Disposing of Scripts

To execute a script, your application must first obtain a valid script ID for a compiled script or script context. You can use either the OSALoad function described in the preceding section or the optional OSACompile function described on page 10-48 to obtain a script ID.

The OSAExecute function takes a script ID for a compiled script or script context and returns a script ID for a script value. The OSADisplay function converts a script value to text that your application can later display to the user. If the OSAExecute function returns errOSAScriptError, you can use the OSAScriptError function to get more information about the error.

When your application no longer needs the script data associated with a specific script ID, you can use the OSADispose function to release the memory the script data occupies.

OSAExecute

You can use the OSAExecute function to execute a compiled script or a script context.

```
FUNCTION OSAExecute(scriptingComponent: ComponentInstance;
                    compiledScriptID: OSAID;
                    contextID: OSAID;
                    modeFlags: LongInt;
                    VAR resultingScriptValueID: OSAID): OSAError;
```

scriptingComponent
        A component instance created by a prior call to the Component Manager
        function OpenDefaultComponent or OpenComponent (see page 10-4).

compiledScriptID
  The script ID for the compiled script to be executed.

contextID   The script ID for the context to be used during script execution. The
               constant kOSANullScript in this parameter indicates that the scripting
               component should use its default context.

modeFlags   Information used by individual scripting components. To avoid setting
               mode flag values, specify kOSAModeNull. Other possible mode flags are
               listed in the description that follows.

resultingScriptValueID
  The script ID for the script value returned.

DESCRIPTION

The OSAExecute function executes the compiled script identified by the
compiledScriptID parameter, using the script context identified by the contextID
parameter to maintain state information, such as the binding of variables, for the
compiled script. After successfully executing a script, OSAExecute returns the script ID
for a resulting script value, or, if execution does not result in a value, the constant
kOSANullScript.

You can use the OSACoerceToDesc function to coerce the resulting script value to a
descriptor record of a desired descriptor type, or the OSADisplay function to obtain the
equivalent source data for the script value.

You can control the way in which the scripting component executes a script by adding
any of these flags to the modeFlags parameter:

Flag	Description
kOSAModeNeverInteract	Adds kAENeverInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeCanInteract	Adds kAECanInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeAlwaysInteract	Adds kAEAlwaysInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeCantSwitchLayer	Prevents use of kAECanSwitchLayer in sendMode parameter of AESend for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).
kOSAModeDontReconnect	Adds kAEDontReconnect to sendMode parameter of AESend for events sent when script is executed.
kOSAModeDoRecord	Prevents use of kAEDontRecord in sendMode parameter of AESend for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).

If the result code returned by OSAExecute is a general result code, there was some problem in arranging for the script to be run. If the result code is errOSAScriptError, an error occurred during script execution. In this case, you can obtain more detailed error information by calling OSAScriptError.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
errOSAInvalidID	–1751	Invalid script ID
errOSAScriptError	–1753	Error occurred during execution
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For information about the OSAGetSource and OSACoerceToDesc functions, see page 10-51 and page 10-54, respectively.

For examples of the use of the OSAExecute function, see Listing 10-2 on page 10-9 and Listing 10-4 on page 10-16.

For more information about resume dispatch functions, see "Supplying a Resume Dispatch Function," which begins on page 10-21, and the description of a resume dispatch function on page 10-97.

OSADisplay

You can use the OSADisplay function to convert a script value to text. Your application can then use its own routines to display this text to the user.

```
FUNCTION OSADisplay(scriptingComponent: ComponentInstance;
                    scriptValueID: OSAID;
                    desiredType: DescType;
                    modeFlags: LongInt;
                    VAR resultingText: AEDesc): OSAError;
```

scriptingComponent
    A component instance created by a prior call to the Component Manager function OpenDefaultComponent or OpenComponent (see page 10-4).

scriptValueID
    The script ID for the script value to coerce.

desiredType
    The desired text descriptor type, such as typeChar, for the resulting descriptor record.

modeFlags   Information used by individual scripting components. To avoid setting any mode flags, specify `kOSAModeNull`. To make the resulting text readable by humans only, so that it can't be recompiled, specify `kOSAModeDisplayForHumans`.

resultingText
             The resulting descriptor record.

DESCRIPTION

The `OSADisplay` function coerces the script value identified by `scriptValueID` to a descriptor record of the text type specified by the `desiredType` parameter, if possible. Valid types include all the standard text descriptor types defined in the *Apple Event Registry: Standard Suites,* plus any special types supported by the scripting component.

Unlike `OSAGetSource`, `OSADisplay` can coerce only script values and always produces a descriptor record of a text descriptor type. In addition, if you specify the mode flag `kOSAModeDisplayForHumans`, the resulting text cannot be recompiled.

SPECIAL CONSIDERATIONS

If you want to get a script value in a form that you can display for humans to read, use `OSADisplay`. If you want the descriptor type of the descriptor record returned in the `resultingText` parameter to be the same as the descriptor type returned by a scripting component, use `OSACoerceToDesc` and specify `typeWildCard` as the desired type.

RESULT CODES

noErr	0	No error
errOSACantCoerce	–1700	Desired type not supported by scripting component
errOSASystemError	–1750	General scripting system error
errOSAInvalidID	–1751	Invalid script ID
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For descriptions of the `OSAGetSource` and `OSACoerceToDesc` functions, see page 10-51 and page 10-54, respectively.

For an example of the use of `OSADisplay`, see Listing 10-2 on page 10-9.

OSAScriptError

You can use the OSAScriptError function to get information about errors that occur during script execution.

```
FUNCTION OSAScriptError(scriptingComponent: ComponentInstance;
                        selector: OSType;
                        desiredType: DescType;
                        VAR resultingErrorDescription: AEDesc)
                        : OSAError;
```

scriptingComponent
A component instance created by a prior call to the Component Manager function OpenDefaultComponent or OpenComponent (see page 10-4).

selector   A value that determines what OSAScriptError returns. The value can be one of these constants:

```
CONST
    kOSAErrorNumber             = 'errn';
    kOSAErrorMessage            = 'errs';
    kOSAErrorBriefMessage       = 'errb';
    kOSAErrorApp                = 'erap';
    kOSAErrorPartialResult      = 'ptlr';
    kOSAErrorOffendingObject    = 'erob';
    kOSAErrorRange              = 'erng';
```

desiredType
The desired descriptor type of the resulting descriptor record. The description that follows explains how this is determined by the value passed in the selector parameter.

resultingErrorDescription
The resulting descriptor record.

DESCRIPTION

Whenever the OSAExecute function returns the error errOSAScriptError, you can use the OSAScriptError function to get more specific information about the error from the scripting component that encountered it. (This information remains available only until the next call to the same scripting component.) The information returned by

OSAScriptError depends on the value passed in the selector parameter, which also determines the descriptor type you should specify in the desiredType parameter.

Constant	Information returned in resultingErrorDescription parameter
kOSAErrorNumber	Error number for either system error or scripting component error. The value of desiredType must be typeShortInteger.
kOSAErrorMessage	Error message associated with error number, including both the name of the application and a description of the error. This constant is sufficient for simple error reporting. The value of desiredType must be typeChar or another text descriptor type.
kOSAErrorBriefMessage	Brief error message associated with error number, excluding the name of the application, any partial result, and the offending object. The value of desiredType must be typeChar or another text descriptor type.
kOSAErrorApp	Either the name or the process serial number (PSN) of the application that received the error, if it was the result of sending an Apple event. The value of desiredType must be typeProcessSerialNumber (for the PSN) or a text descriptor type such as typeChar (for the name).
kOSAErrorPartialResult	Partial result returned after a call to AESend that failed. This consists of a reply parameter that contains some but not all of the information requested. The value of desiredType must be typeBest (for the best type) or typeWildCard (for the default type).
kOSAErrorOffendingObject	An object specifier record for the object that caused the error. The value of desiredType must be typeObjectSpecifier, typeBest, or typeWildCard. For some scripting components, including AppleScript, these three values are equivalent.
kOSAErrorRange	Range of source data in which error occurred. The value of desiredType must be typeOSAErrorRange.

Every scripting component should support calls to OSAScriptError that pass kOSAErrorNumber, kOSAErrorMessage, or kOSAErrorPartialResult in the selector parameter.

Some scripting components may also support calls that pass other values in the selector parameter, including kOSAErrorRange, which provides start and end positions delimiting the errant expression in the source data. If the value of the selector parameter is kOSAErrorRange, the value of desiredType must be typeOSAErrorRange.

```
CONST typeOSAErrorRange          = 'erng';
```

A descriptor record of type typeOSAErrorRange is an AE record that consists of two descriptor records of typeShortInteger specified by these keywords:

```
CONST
    keyOSASourceStart    = 'srcs'; {start of error range}
    keyOSASourceEnd      = 'srce'; {end of error range}
```

If the value of the selector parameter is kOSAErrorNumber, scripting components may return, in the resultingErrorDescription parameter, one of these general error codes:

errOSACantCoerce	−1700	Same as errAECoercionFail; can't coerce data to requested descriptor type
errOSACantAccess	−1728	Same as errAENoSuchObject; runtime error in resolution of object specifier record
errOSAGeneralError	−2700	General runtime error
errOSADivideByZero	−2701	Attempt to divide by zero
errOSANumericOverflow	−2702	Numeric overflow
errOSACantLaunch	−2703	Can't launch specified file because it isn't an application
errOSAAppNotHighLevelEventAware	−2704	Doesn't respond to Apple events
errOSACorruptTerminology	−2705	The application has a corrupted 'aete' resource
errOSAStackOverflow	−2706	Stack overflow
errOSAInternalTableOverflow	−2707	Internal table overflow
errASDataBlockTooLarge	−2708	Attempt to create a value larger than the allowable size
errOSATypeError	−1703	Same as errAEWrongDataType; wrong descriptor type
errOSAMessageNotUnderstood	−1708	Same as errAEEventNotHandled; event not handled or message not understood
errOSAUndefinedMessage	−1717	Same as errAEHandlerNotFound; handler not found for message
errOSAIllegalIndex	−1728	Same as errAEIllegalIndex; not a valid index
errOSAIllegalRange	−2720	Same as errAEImpossibleRange; range of specified objects not possible
errOSASyntaxError	−2740	General syntax error
errOSASyntaxTypeError	−2741	Syntax error; parser expected one type but found another
errOSATokenTooLong	−2742	Identifier too long

continued

errOSAMissingParameter	−1701	Same as errAEDescNotFound; descriptor record not found
errOSAParameterMismatch	−1721	Same as errAEWrongNumberArgs; wrong number of arguments
errOSADuplicateParameter	−2750	Parameter specified more than once
errOSADuplicateProperty	−2751	Property specified more than once
errOSADuplicateHandler	−2752	Handler defined more than once
errOSAUndefinedVariable	−2753	Undefined variable
errOSAInconsistentDeclarations	−2754	Inconsistent declarations
errOSAControlFlowError	−2755	Control flow error

Although scripting components are not required to support these error codes, their use simplifies error handling for applications that run scripts created by multiple components.

If the value of the selector parameter is kOSAErrorNumber, the AppleScript component may return, in the resultingErrorDescription parameter, one of these error codes:

errAECantConsiderAndIgnore	−2720	Can't both consider and ignore a parameter
errASCantCompareMoreThan32k	−2721	Can't compare text larger than 32K
errASCantCompareMixedScripts	−2722	Can't compare text from different script systems
errASTerminologyNestingTooDeep	−2760	Tell statements nested too deeply
errASInconsistentNames	−2780	Syntax error; names at beginning and end of handler are inconsistent (AppleScript English dialect only)

SPECIAL CONSIDERATIONS

If you call OSAScriptError using an instance of the generic scripting component, the generic scripting component uses the same instance of a scripting component that it used for the previous call.

RESULT CODES

noErr	0	No error
errOSACantCoerce	−1700	Desired type not supported by scripting component
errOSASystemError	−1750	General scripting system error
errOSABadSelector	−1754	Selector value not supported by scripting component
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For an example of the use of OSAScriptError, see Listing 10-3 on page 10-11.

OSADispose

You can use the `OSADispose` function to reclaim the memory occupied by script data.

```
FUNCTION OSADispose(scriptingComponent: ComponentInstance;
                    scriptID: OSAID): OSAError;
```

scriptingComponent
: A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

scriptID
: The script ID for the script data to be disposed of.

DESCRIPTION

The `OSADispose` function releases the memory assigned to the script data identified by the `scriptID` parameter. The script ID passed to the `OSADispose` function is no longer valid if the function returns successfully. A scripting component can then reuse that script ID for other script data.

A call to `OSADispose` returns `noErr` if the script ID is `kOSANullScript`, although it does not dispose of anything.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
errOSAInvalidID	–1751	Invalid script ID
badComponentInstance	$80008001	Invalid component instance

Setting and Getting Script Information

The `OSASetScriptInfo` function sets various kinds of information about script data, and the `OSAGetScriptInfo` function returns information about script data. The kind of information these functions set or get depends on constants you pass to the functions.

OSASetScriptInfo

You can use `OSASetScriptInfo` to set information about script data according to the value you pass in the `selector` parameter.

```
FUNCTION OSASetScriptInfo(scriptingComponent: ComponentInstance;
                          scriptID: OSAID;
                          selector: OSType;
                          value: LongInt): OSAError;
```

scriptingComponent
A component instance created by a prior call to the Component Manager
function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

scriptID   The script ID for the script data whose information is to be set.

selector   A value that determines what kind of information `OSASetScriptInfo`
sets. All scripting components can accept this value:

```
CONST kOSAScriptIsModified            = 'modi';
```

The `kOSAScriptIsModified` constant indicates that the count of
changes since the script data was loaded or created should be set to the
value in the `value` parameter. The AppleScript component provides
limited support for this constant.

value   The value to set.

DESCRIPTION

The `OSASetScriptInfo` function sets script information according to the value you
pass in the `selector` parameter. If you use the `kOSAScriptIsModified` constant,
`OSASetScriptInfo` sets a value that indicates how many times the script data has
been modified since it was created or passed to `OSALoad`. Some scripting components
may provide additional constants.

SPECIAL CONSIDERATIONS

Although you can specify `kOSAScriptIsModified` when you are using the
AppleScript component without generating an error, the current version of AppleScript
doesn't actually set a value for the count of changes since the script data was loaded or
created. For more information, see the description of `OSAGetScriptInfo` that follows.

RESULT CODES

`noErr`	0	No error
`errOSASystemError`	–1750	General scripting system error
`errOSAInvalidID`	–1751	Invalid script ID
`errOSABadSelector`	–1754	Selector value not supported by scripting component
`badComponentInstance`	$80008001	Invalid component instance

OSAGetScriptInfo

You can use OSAGetScriptInfo to obtain information about script data according to the value you pass in the selector parameter.

```
FUNCTION OSAGetScriptInfo(scriptingComponent: ComponentInstance;
                          scriptID: OSAID;
                          selector: OSType;
                          VAR result: LongInt): OSAError;
```

scriptingComponent
          A component instance created by a prior call to the Component Manager
          function OpenDefaultComponent or OpenComponent (see page 10-4).

scriptID   The script ID for the script data about which to obtain information.

selector   A value that determines what kind of information OSAGetScriptInfo
               returns. The value can be one of these constants:

```
CONST   kOSAScriptIsModified            = 'modi';
        kOSAScriptIsTypeCompiledScript  = 'cscr';
        kOSAScriptIsTypeScriptValue     = 'valu';
        kOSAScriptIsTypeScriptContext   = 'cntx';
        kOSAScriptBestType              = 'best';
        kOSACanGetSource                = 'gsrc';
```

               The AppleScript component provides limited support for the constant
               kOSAScriptIsModified (see page 10-44). In addition to the standard
               constants, the AppleScript component also supports this constant:

```
CONST kASHasOpenHandler            = 'hsod';
```

result   The requested information, which you can coerce to the appropriate
               descriptor type for the value specified in the selector parameter.

DESCRIPTION

The OSAGetScriptInfo function returns various results according to the value you
pass in the selector parameter.

Value of selector parameter	Information returned in the result parameter
kOSAScriptIsModified	Long integer that indicates the number of times the script data has been modified since it was passed to OSALoad.
kOSAScriptIsTypeCompiledScript	Boolean value that indicates whether or not the script data is a compiled script.

continued

Value of `selector` parameter	Information returned in the `result` parameter (continued)
`kOSAScriptIsTypeScriptValue`	Boolean value that indicates whether or not the script data is a script value.
`kOSAScriptIsTypeScriptContext`	Boolean value that indicates whether or not the script data is a script context.
`kOSAScriptBestType`	A descriptor type that you can pass to `OSACoerceToDesc`.
`kOSACanGetSource`	Boolean value that indicates whether the script data can be successfully passed to `OSAGetSource`.

The AppleScript component also provides this constant for use in the `selector` parameter.

Value of `selector` parameter	Information returned in the `result` parameter
`kASHasOpenHandler`	Boolean value that indicates whether a script context with the specified script ID contains a handler for the Open Documents event. If the script ID doesn't identify a script context, `OSAGetScriptInfo` returns the result code `errOSAIllegalAccess`.

SPECIAL CONSIDERATIONS

Although you can specify `kOSAScriptIsModified` when you are using the AppleScript component without generating an error, the current version of AppleScript interprets this request conservatively. The AppleScript component stores script data in a network of interlocking structures, and running a script can cause any of these structures to be modified. If you pass a script ID is to `OSAGetScriptInfo` with `kOSAScriptIsModified` as the value of the `selector` parameter, the AppleScript component returns 1 if there is any possibility that the script data or related structures may have been modified, and 0 if there is no possibility that they have been modified.

RESULT CODES

`noErr`	0	No error
`errOSASystemError`	–1750	General scripting system error
`errOSAInvalidID`	–1751	Invalid script ID
`errOSABadSelector`	–1754	Selector value not supported by scripting component
`badComponentInstance`	$80008001	Invalid component instance

Manipulating the Active Function

The `OSASetActiveProc` and `OSAGetActiveProc` functions allow your application to set or to get a pointer to the active function called periodically by the scripting component during script execution. To get time periodically during script execution for its own purposes, your application can substitute its own active function for use by the scripting component. If you do not specify an active function, the scripting component uses its default active function, which allows a user to cancel script execution.

The functions described in this section use the following type for pointers to active functions:

```
TYPE OSAActiveProcPtr = ProcPtr;
```

For more information about active functions, see "Supplying an Alternative Active Function" on page 10-23.

OSASetActiveProc

You can use the `OSASetActiveProc` routine to set the active function that a scripting component calls periodically while executing a script.

```
FUNCTION OSASetActiveProc(scriptingComponent: ComponentInstance;
                          activeProc: OSAActiveProcPtr;
                          refCon: LongInt): OSAError;
```

scriptingComponent
            A component instance created by a prior call to the Component Manager
            function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

activeProc
            A pointer to the active function to set. If the value of this parameter is
            `NIL`, `OSASetActiveProc` sets the scripting component's default active
            function.

refCon   A reference constant to be associated with the active function. This
            parameter can be used for many purposes; for example, it could contain a
            handle to data used by the active function.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

OSAGetActiveProc

You can use the `OSAGetActiveProc` function to get a pointer to the active function that a scripting component is currently using.

```
FUNCTION OSAGetActiveProc(scriptingComponent: ComponentInstance;
                          VAR activeProc: OSAActiveProcPtr;
                          VAR refCon: LongInt): OSAError;
```

scriptingComponent
: A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

activeProc
: The `OSAGetActiveProc` function returns in this parameter a pointer to the active function currently set for the specified scripting component.

refCon
: The `OSAGetActiveProc` function returns in this parameter the reference constant associated with the active function for the specified scripting component.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

Optional Scripting Component Routines

This section describes eight groups of optional routines that scripting components can support. Your application can use the Component Manager to find a scripting component that supports a specific group of routines or to determine whether a particular scripting component supports a specific group of routines.

To specify one or more groups of routines for the Component Manager, use the following constants to set the equivalent bits in the `componentFlags` field of a component description record:

```
CONST
    kOSASupportsCompiling      = $0002;
    kOSASupportsGetSource      = $0004;
    kOSASupportsAECoercion     = $0008;
    kOSASupportsAESending      = $0010;
    kOSASupportsRecording      = $0020;
    kOSASupportsConvenience    = $0040;
    kOSASupportsDialects       = $0080;
    kOSASupportsEventHandling  = $0100;
```

Each of these flags identifies one of the groups of routines that are described in the sections that follow. For information about using these constants to locate scripting components that support specific groups of optional routines, see "Connecting to a Scripting Component," which begins on page 10-3.

Compiling Scripts

Scripting components can provide three optional routines that get the name of a scripting component, compile a script, and update a script ID.

To obtain the name of a scripting component in a form that you can coerce to text, you can use the `OSAScriptingComponentName` function. The `OSACompile` function compiles source data and returns a script ID, and the `OSACopyID` function updates the script data associated with one script ID with the script data associated with another script ID.

A scripting component that supports the routines in this section has the `kOSASupportsCompiling` bit set in the `componentFlags` field of its component description record.

OSAScriptingComponentName

You can use the `OSAScriptingComponentName` function to get the name of a scripting component.

```
FUNCTION OSAScriptingComponentName
        (scriptingComponent: ComponentInstance;
         VAR resultingScriptingComponentName: AEDesc): OSAError;
```

scriptingComponent
            A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

resultingScriptingComponentName
            The name of the scripting component; or, if the component is the generic scripting component, the name of the default scripting component.

DESCRIPTION

The `OSAScriptingComponentName` function returns a descriptor record that you can coerce to a text descriptor type such as `typeChar`. This can be useful if you want to display the name of the scripting language in which the user should write a new script.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For an example of the use of OSAScriptingComponentName, see Listing 10-2 on page 10-9.

OSACompile

You can use the OSACompile function to compile the source data for a script and obtain a script ID for a compiled script or a script context.

```
FUNCTION OSACompile (scriptingComponent: ComponentInstance;
                     sourceData: AEDesc; modeFlags: LongInt;
                     VAR previousAndResultingScriptID: OSAID)
                     : OSAError;
```

scriptingComponent
: A component instance created by a prior call to the Component Manager function OpenDefaultComponent or OpenComponent (see page 10-4).

sourceData
: A descriptor record containing suitable source data for the specified scripting component.

modeFlags
: Information used by individual scripting components. To avoid setting mode flag values, specify kOSAModeNull. Other possible mode flags are listed in the description that follows.

previousAndResultingScriptID
: The script ID for the resulting compiled script. If the value of this parameter on input is kOSANullScript, OSACompile returns a new script ID for the compiled script data. If the value of this parameter on input is an existing script ID, OSACompile updates the script ID so that it refers to the newly compiled script data.

DESCRIPTION

You can pass a descriptor record containing source data suitable for a specific scripting component (usually text) to the OSACompile function to obtain a script ID for the equivalent compiled script or script context. To compile the source data as a script context for use with OSAExecuteEvent or OSADoEvent, you must set the kOSACompileIntoContext flag, and the source data should include appropriate handlers.

After you have successfully compiled the script, you can use the returned script ID to refer to the compiled script when you call OSAExecute and other scripting component routines.

You can control the way a compiled script is executed by adding any of these flags to the modeFlags parameter:

Flag	Description
kOSAModePreventGetSource	Compiled script consists of only the minimum script data required to run the script. It will cause an error if passed to OSAGetSource.
kOSACompileIntoContext	The OSACompile function returns a script context instead of a compiled script.
kOSAModeAugmentContext	Script data associated with script ID passed in previousAndResultingCompiledScriptID is augmented rather than replaced with the new compiled script. Specifying this flag automatically invokes the kOSAModeCompileIntoContext mode flag. If you redefine variables, handlers, and so on that were previously defined in the script context, the new definitions will replace the old ones.
kOSAModeNeverInteract	Adds kAENeverInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeCanInteract	Adds kAECanInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeAlwaysInteract	Adds kAEAlwaysInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeDontReconnect	Adds kAEDontReconnect to sendMode parameter of AESend for events sent when script is executed.
kOSAModeCantSwitchLayer	Prevents use of kAECanSwitchLayer in sendMode parameter of AESend for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).
kOSAModeDoRecord	Prevents use of kAEDontRecord in sendMode parameter of AESend for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).

SPECIAL CONSIDERATIONS

If you use OSACompile with an instance of the generic scripting component and pass kOSANullScript in the previousAndResultingScriptID parameter, the generic scripting component uses the default scripting component to compile the script.

If you're recompiling a script, specify the original script ID in the `previousAndResultingScriptID` parameter. The generic scripting component uses the script ID to determine which scripting component it should use to compile the script.

RESULT CODES

noErr	0	No error
errOSACantCoerce	–1700	Source data incompatible with scripting component
errOSASystemError	–1750	General scripting system error
errOSAInvalidID	–1751	Invalid script ID
errOSAScriptError	–1753	Source data invalid (syntax error)
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For an example of the use of OSACompile to update an existing script ID, see Listing 10-5 on page 10-18. For an example of the use of OSACompile to obtain a new script ID, see Listing 10-2 on page 10-9.

For more information about the default scripting component associated with any instance of the generic scripting component, see "Generic Scripting Component Routines," which begins on page 10-84.

OSACopyID

You can use the OSACopyID function to update script data after editing or recording and to perform undo or revert operations on script data.

```
FUNCTION OSACopyID(scriptingComponent: ComponentInstance;
                    fromID: OSAID;
                    VAR toID: OSAID): OSAError;
```

scriptingComponent
: A component instance created by a prior call to the Component Manager function OpenDefaultComponent or OpenComponent (see page 10-4).

fromID
: The script ID for script data that you want to be associated with the script ID in the toID parameter.

toID
: The script ID for the script data to be replaced. If the value of this parameter is kOSANullScript, the OSACopyID function returns a new script ID.

DESCRIPTION

The OSACopyID function replaces the script data identified by the script ID in the toID parameter with the script data identified by the script ID in the fromID parameter.

RESULT CODES

noErr	**0**	No error
errOSASystemError	–1750	General scripting system error
errOSAInvalidID	–1751	Invalid script ID
badComponentInstance	$80008001	Invalid component instance

Getting Source Data

The OSAGetSource function returns the source data that corresponds to the script data identified by a script ID. The source data it returns can in turn be passed to OSACompile.

A scripting component that supports the OSAGetSource function has the kOSASupportsGetSource bit set in the componentFlags field of its component description record.

OSAGetSource

You can use the OSAGetSource function to decompile the script data identified by a script ID and obtain the equivalent source data.

```
FUNCTION OSAGetSource(scriptingComponent: ComponentInstance;
                      scriptID: OSAID;
                      desiredType: DescType;
                      VAR resultingSourceData: AEDesc): OSAError;
```

scriptingComponent
                A component instance created by a prior call to the Component Manager
                function OpenDefaultComponent or OpenComponent (see page 10-4).

scriptID   The script ID for the script data to decompile. If you pass
                kOSANullScript in this parameter, OSAGetSource returns a null
                source description (such as an empty text string).

desiredType
                The desired descriptor type of the resulting descriptor record, or
                typeBest if any type will do.

resultingSourceData
                The resulting descriptor record.

DESCRIPTION

The OSAGetSource function decompiles the script data identified by the specified script ID and returns a descriptor record containing the equivalent source data. The source data returned need not be exactly the same as the source data originally passed to OSACompile—for example, white space and formatting might be different—but it should be a reasonable equivalent suitable for user viewing and editing.

The difference between OSACoerceToDesc and OSAGetSource is that OSAGetSource creates source data that can be displayed to a user or compiled and executed to generate an appropriate value, whereas OSACoerceToDesc actually returns the value. For example, if you call OSAGetSource and specify a string value, it returns the text surrounded by quotation marks (so that it can be properly compiled). If you call OSACoerceToDesc and specify a string value, it simply returns the text.

The main difference between OSADisplay and OSAGetSource is that OSAGetSource can coerce any form of script data using a variety of descriptor types, whereas OSADisplay can coerce only script values and always produces a descriptor record of a text descriptor type.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
errOSAInvalidID	–1751	Invalid script ID
errOSASourceNotAvailable	–1756	Source data not available
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For an example of the use of OSAGetSource, see Listing 10-5 on page 10-18.

Coercing Script Values

Scripting components can provide support for two optional routines, OSACoerceFromDesc and OSACoerceToDesc, which coerce data in a descriptor record to a script value and coerce a script value to data in a descriptor record, respectively.

A scripting component that supports the routines in this section has the kOSASupportsAECoercion bit set in the componentFlags field of its component description record.

OSACoerceFromDesc

You can use the OSACoerceFromDesc function to obtain the script ID for a script value that corresponds to the data in a descriptor record.

```
FUNCTION OSACoerceFromDesc
                (scriptingComponent: ComponentInstance;
                 scriptData: AEDesc; modeFlags: LongInt;
                 VAR resultingScriptValueID: OSAID): OSAError;
```

scriptingComponent
        A component instance created by a prior call to the Component Manager
        function OpenDefaultComponent or OpenComponent (see page 10-4).

scriptData
        A descriptor record containing the script data to be coerced.

modeFlags   Information used by individual scripting components. To avoid setting
             mode flag values, specify kOSAModeNull. If the scriptData parameter
             contains an Apple event, you can use any of the mode flags listed in the
             description that follows.

resultingScriptValueID
        The resulting script ID for a script value.

DESCRIPTION

The OSACoerceFromDesc function coerces the descriptor record in the scriptData
parameter to the equivalent script value and returns a script ID for that value.

If you pass OSACoerceFromDesc an Apple event in the scriptData
parameter, it returns a script ID for the equivalent compiled script in the
resultingScriptValueID parameter. In this case you can specify any of the
modeFlags values used by OSACompile to control the way the compiled script
is executed:

Flag	Description
kOSAModePreventGetSource	Compiled script consists of only the minimum script data required to run the script. It will cause an error if passed to OSAGetSource.
kOSACompileIntoContext	The OSACoerceFromDesc function returns a script context instead of a compiled script.
kOSAModeNeverInteract	Adds kAENeverInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeCanInteract	Adds kAECanInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeAlwaysInteract	Adds kAEAlwaysInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeDontReconnect	Adds kAEDontReconnect to sendMode parameter of AESend for events sent when script is executed.
kOSAModeCantSwitchLayer	Prevents use of kAECanSwitchLayer in sendMode parameter of AESend for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).
kOSAModeDoRecord	Prevents use of kAEDontRecord in sendMode parameter of AESend for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).

SPECIAL CONSIDERATIONS

If you call `OSACoerceFromDesc` using an instance of the generic scripting component, the generic scripting component uses the default scripting component to perform the coercion.

RESULT CODES

`noErr`	0	No error
`errOSASystemError`	–1750	General scripting system error
`badComponentInstance`	$80008001	Invalid component instance

SEE ALSO

For more information about the default scripting component associated with any instance of the generic scripting component, see "Generic Scripting Component Routines," which begins on page 10-84.

OSACoerceToDesc

You can use the `OSACoerceToDesc` function to coerce a script value to a descriptor record of a desired descriptor type.

```
FUNCTION OSACoerceToDesc(scriptingComponent: ComponentInstance;
                         scriptValueID: OSAID;
                         desiredType: DescType;
                         modeFlags: LongInt;
                         VAR result: AEDesc): OSAError;
```

scriptingComponent
: A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

scriptValueID
: The script ID for the script value to coerce.

desiredType
: The desired descriptor type of the resulting descriptor record.

modeFlags
: Information used by individual scripting components. To avoid setting mode flag values, specify `kOSAModeNull`.

result
: The resulting descriptor record.

DESCRIPTION

The `OSACoerceToDesc` function coerces the script value identified by `scriptValueID` to a descriptor record of the type specified by the `desiredType` parameter, if possible.

Valid types include all the standard descriptor types defined in the *Apple Event Registry: Standard Suites,* plus any special types supported by the scripting component.

SPECIAL CONSIDERATIONS

If you want the descriptor type of the descriptor record returned in the `result` parameter to be the same as the descriptor type returned by a scripting component, use `OSACoerceToDesc` and specify `typeWildCard` as the desired type. If you want to get a script value in a form that you can display for humans to read, use `OSADisplay`.

RESULT CODES

`noErr`	0	No error
`errOSASystemError`	–1750	General scripting system error
`errOSAInvalidID`	–1751	Invalid script ID
`badComponentInstance`	$80008001	Invalid component instance

SEE ALSO

For a description of `OSADisplay`, see page 10-35.

Manipulating the Create and Send Functions

Some scripting components provide routines that allow your application to set or get pointers to the create and send functions used by the scripting component when it sends and creates Apple events during script execution. If you do not set the pointers that specify these functions, the scripting component uses the standard `AECreateAppleEvent` and `AESend` functions with default parameters.

To gain control over the creation and addressing of Apple events, your application can provide its own create function for use by scripting components. To set a new create function, call the `OSASetCreateProc` function; to get the current create function, call `OSAGetCreateProc`.

The send function provided by your application can perform almost any action instead of or in addition to sending Apple events; for example, it can be used to facilitate concurrent script execution. To set a new send function, call the `OSASetSendProc` function; to get the current send function, call `OSAGetSendProc`.

The functions described in this section use the following types for pointers to the create and send functions:

```
TYPE
   AESendProcPtr = ProcPtr;
   AECreateAppleEventProcPtr = ProcPtr;
```

For more information about create and send functions, see "Supplying Alternative Create and Send Functions," which begins on page 10-24.

Scripting components that support manipulation of the create and send functions also support the `OSASetDefaultTarget` function, which allows you to set the default application to which Apple events are sent.

A scripting component that supports the functions described in this section has the `kOSASupportsAESending` bit set in the `componentFlags` field of its component description record.

OSASetCreateProc

You can use the `OSASetCreateProc` function to specify a create function that a scripting component should use instead of the Apple Event Manager's `AECreateAppleEvent` function when creating Apple events.

```
FUNCTION OSASetCreateProc(scriptingComponent: ComponentInstance;
                          createProc: AECreateAppleEventProcPtr;
                          refCon: LongInt): OSAError;
```

scriptingComponent
: A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

createProc
: A pointer to the create function to set.

refCon
: A reference constant.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

OSAGetCreateProc

You can use the `OSAGetCreateProc` function to get a pointer to the create function that a scripting component is currently using to create Apple events.

```
FUNCTION OSAGetCreateProc(scriptingComponent: ComponentInstance;
                          VAR createProc: AECreateAppleEventProcPtr;
                          VAR refCon: LongInt): OSAError;
```

scriptingComponent
: A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

createProc

The OSAGetCreateProc function returns, in this parameter, a pointer to the create function currently set for the specified scripting component.

refCon The OSAGetCreateProc function returns, in this parameter, the reference constant associated with the create function for the specified scripting component.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

OSASetSendProc

You can use the OSASetSendProc function to specify a send function that a scripting component should use instead of the Apple Event Manger's AESend function when sending Apple events.

```
FUNCTION OSASetSendProc(scriptingComponent: ComponentInstance;
                        sendProc: AESendProcPtr;
                        refCon: LongInt): OSAError;
```

scriptingComponent

A component instance created by a prior call to the Component Manager function OpenDefaultComponent or OpenComponent (see page 10-4).

sendProc A pointer to the send function to set.

refCon A reference constant.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

OSAGetSendProc

You can use the OSAGetSendProc function to get a pointer to the send function that a scripting component is currently using.

```
FUNCTION OSAGetSendProc(scriptingComponent: ComponentInstance;
                        VAR sendProc: AESendProcPtr;
                        VAR refCon: LongInt): OSAError;
```

scriptingComponent

A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

sendProc
The `OSAGetSendProc` function returns, in this parameter, a pointer to the send function currently set for the specified scripting component.

refCon
The `OSAGetSendProc` function returns, in this parameter, the reference constant associated with the send function for the specified scripting component.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

OSASetDefaultTarget

You can use the `OSASetDefaultTarget` function to set the default target application for Apple events.

```
FUNCTION OSASetDefaultTarget
                        (scriptingComponent: ComponentInstance;
                         target: AEAddressDesc): OSAError;
```

scriptingComponent

A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

target
The address of the application that is being made the default application. If you pass a null descriptor record in this parameter, the scripting component treats the current process as the default target.

DESCRIPTION

The `OSASetDefaultTarget` function establishes the default target application for Apple event sending and the default application from which the scripting component should obtain terminology information. For example, AppleScript statements that refer to the default application do not need to be enclosed in `tell/end tell` statements.

If your application doesn't call this function, or if you pass a null descriptor record in the `target` parameter, the scripting component treats the current process (that is, the application that calls `OSAExecute` or related routines) as the default target application.

RESULT CODES

noErr	**0**	No error
errOSASystemError	**–1750**	General scripting system error
badComponentInstance	**$80008001**	Invalid component instance

Recording Scripts

The OSAStartRecording function turns on the Apple Event Manager's recording mechanism and specifies a script in which subsequent recordable Apple events are recorded. The scripting component sends the recording process (for example, a script editor) a Recorded Text event that contains the decompiled equivalent for each recordable event it receives. The script editor can then display the decompiled script in a script editor window if a window for that script is currently open. Recording continues until a call to OSAStopRecording turns recording off.

Script editors use these routines to allow users to control recording. Any application can use these routines to provide its own script-recording interface.

For more information about the Apple event recording mechanism, see the chapter "Recording Apple Events" in this book. For more information about the Recorded Text event, see "Recording Scripts" on page 10-26.

A scripting component that supports the functions described in this section has the kOSASupportsRecording bit set in the componentFlags field of its component description record.

OSAStartRecording

You can use the OSAStartRecording routine to turn on Apple event recording and record subsequent Apple events in a compiled script.

```
FUNCTION OSAStartRecording
                    (scriptingComponent: ComponentInstance;
                     VAR compiledScriptToModifyID: OSAID): OSAError;
```

scriptingComponent
           A component instance created by a prior call to the Component Manager
           function OpenDefaultComponent or OpenComponent (see page 10-4).

compiledScriptToModifyID
           The script ID for the compiled script in which to record.

DESCRIPTION

The `OSAStartRecording` routine turns on Apple event recording. Subsequent Apple events are recorded (that is, appended to any existing statements) in the compiled script specified by the `compiledScriptToModifyID` parameter. If the source data for the compiled script is currently displayed in a script editor's window, the script editor's handler for the Recorded Text event should display each new statement in the window as it is recorded. Users should not be able to change a script that is open in a script editor window while it is being recorded into.

To record into a new compiled script, pass the constant `kOSANullScript` in the `compiledScriptToModifyID` parameter. The scripting component should respond by creating a new compiled script and recording into that.

SPECIAL CONSIDERATIONS

The generic scripting component uses its default scripting component to create and record into a new compiled script.

RESULT CODES

`noErr`	0	No error
`errAERecordingIsAlreadyOn`	–1732	Attempt to turn recording on when it is already on for a recording process
`errOSASystemError`	–1750	General scripting system error
`errOSAInvalidID`	–1751	Invalid script ID
`badComponentInstance`	$80008001	Invalid component instance

SEE ALSO

For more information about the default scripting component associated with any instance of the generic scripting component, see "Generic Scripting Component Routines," which begins on page 10-84.

OSAStopRecording

You can use the `OSAStopRecording` function to turn off Apple event recording.

```
FUNCTION OSAStopRecording(scriptingComponent: ComponentInstance;
                          compiledScriptID: OSAID): OSAError;
```

`scriptingComponent`
A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

`compiledScriptID`

A script ID for the compiled script into which Apple events are being recorded.

DESCRIPTION

The `OSAStopRecording` function turns off recording. If the script is not currently open in a script editor window, the `compiledScriptToModifyID` parameter supplied to `OSAStartRecording` is then augmented to contain the newly recorded statements. If the script is currently open in a script editor window, the script data that corresponds to the `compiledScriptToModifyID` parameter supplied to `OSAStartRecording` is updated continuously until the client application calls `OSAStopRecording`.

If the compiled script identified by the script ID in the `compiledScriptID` parameter is not being recorded into or recording is not currently on, `OSAStopRecording` returns `noErr`.

RESULT CODES

`noErr`	0	No error
`errOSASystemError`	−1750	General scripting system error
`errOSAInvalidID`	−1751	Invalid script ID
`badComponentInstance`	$80008001	Invalid component instance

Executing Scripts in One Step

The `OSALoadExecute`, `OSACompileExecute`, and `OSADoScript` functions combine the capabilities of several other scripting component functions so that an application can execute a script in a single step. You can use these functions if you know that the script data to be executed will be executed only once.

A scripting component that supports the functions described in this section has the `kOSASupportsConvenience` bit set in the `componentFlags` field of its component description record.

OSALoadExecute

You can use the `OSALoadExecute` function to load and execute a script in a single step rather than calling `OSALoad` and `OSAExecute`.

```
FUNCTION OSALoadExecute (scriptingComponent: ComponentInstance;
                         scriptData: AEDesc;
                         contextID: OSAID; modeFlags: LongInt;
                         VAR resultingScriptValueID: OSAID)
                         : OSAError;
```

scriptingComponent
A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

scriptData

The descriptor record identifying the script data to be loaded and executed.

contextID   The script ID for the context to be used during script execution. The constant `kOSANullScript` in this parameter indicates that the scripting component should use its default context.

modeFlags   Information used by individual scripting components. To avoid setting mode flag values, specify `kOSAModeNull`. Other possible mode flags are listed in the description that follows.

resultingScriptValueID
The script ID for the script value returned.

DESCRIPTION

The `OSALoadExecute` function loads script data and executes the resulting compiled script, using the script context identified by the `contextID` parameter to maintain state information such as the binding of variables. After successfully executing the script, `OSALoadExecute` disposes of the compiled script and returns either the script ID for the resulting script value or, if execution does not result in a value, the constant `kOSANullScript`.

You can control the way in which the scripting component executes a script by adding any of these flags to the `modeFlags` parameter:

Flag	Description
kOSAModeNeverInteract	Adds `kAENeverInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.
kOSAModeCanInteract	Adds `kAECanInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.
kOSAModeAlwaysInteract	Adds `kAEAlwaysInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.
kOSAModeCantSwitchLayer	Prevents use of `kAECanSwitchLayer` in `sendMode` parameter of `AESend` for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).
kOSAModeDontReconnect	Adds `kAEDontReconnect` to `sendMode` parameter of `AESend` for events sent when script is executed.
kOSAModeDoRecord	Prevents use of `kAEDontRecord` in `sendMode` parameter of `AESend` for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).

If the result code returned by OSALoadExecute is a general result code, there was some problem in arranging for the script to be run. If the result code is errOSAScriptError, an error occurred during script execution. In this case, you can obtain more detailed error information by calling OSAScriptError.

RESULT CODES

noErr	0	No error
errOSACorruptData	–1702	Same as errAECorruptData
errOSASystemError	–1750	General scripting system error
errOSAInvalidID	–1751	Invalid script ID
errOSABadStorageType	–1752	Script data not for this scripting component
errOSAScriptError	–1753	Error occurred during execution
errOSADataFormatObsolete	–1758	Data format is obsolete
errOSADataFormatTooNew	–1759	Data format is too new
badComponentInstance	$80008001	Invalid component instance

OSACompileExecute

You can use the OSACompileExecute routine to compile and execute a script in a single step rather than calling OSACompile and OSAExecute.

```
FUNCTION OSACompileExecute
                (scriptingComponent: ComponentInstance;
                 sourceData: AEDesc;
                 contextID: OSAID; modeFlags: LongInt;
                 VAR resultingScriptValueID: OSAID): OSAError;
```

scriptingComponent
: A component instance created by a prior call to the Component Manager function OpenDefaultComponent or OpenComponent (see page 10-4).

sourceData
: A descriptor record identifying suitable source data for the specified scripting component.

contextID
: The script ID for the context to be used during script execution. The constant kOSANullScript in this parameter indicates that the scripting component should use its default context.

modeFlags
: Information used by individual scripting components. To avoid setting mode flag values, specify kOSAModeNull. Other possible mode flags are listed in the description that follows.

resultingScriptValueID
: The script ID for the script value returned.

DESCRIPTION

The `OSACompileExecute` function compiles source data and executes the resulting compiled script, using the script context identified by the `contextID` parameter to maintain state information such as the binding of variables. After successfully executing the script, `OSACompileExecute` disposes of the compiled script and returns either the script ID for the resulting script value or, if execution does not result in a value, the constant `kOSANullScript`.

You can control the way in which the scripting component executes a script by adding any of these flags to the `modeFlags` parameter:

Flag	Description
`kOSAModeNeverInteract`	Adds `kAENeverInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.
`kOSAModeCanInteract`	Adds `kAECanInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.
`kOSAModeAlwaysInteract`	Adds `kAEAlwaysInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.
`kOSAModeCantSwitchLayer`	Prevents use of `kAECanSwitchLayer` in `sendMode` parameter of `AESend` for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).
`kOSAModeDontReconnect`	Adds `kAEDontReconnect` to `sendMode` parameter of `AESend` for events sent when script is executed.
`kOSAModeDoRecord`	Prevents use of `kAEDontRecord` in `sendMode` parameter of `AESend` for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).

If the result code returned by `OSACompileExecute` is a general result code, there was some problem in arranging for the script to be run. If the result code is `errOSAScriptError`, an error occurred during script execution. In this case, you can obtain more detailed error information by calling `OSAScriptError`.

RESULT CODES

`noErr`	0	No error
`errOSACantCoerce`	–1700	Data could not be coerced to the requested data type
`errOSASystemError`	–1750	General scripting system error
`errOSAInvalidID`	–1751	Invalid script ID
`errOSAScriptError`	–1753	Source data invalid (syntax error) or an execution error occurred
`badComponentInstance`	$80008001	Invalid component instance

OSADoScript

You can use the OSADoScript routine to compile and execute a script and convert the resulting script value to text in a single step rather than calling OSACompile, OSAExecute, and OSADisplay.

```
FUNCTION OSADoScript (scriptingComponent: ComponentInstance;
                      sourceData: AEDesc;
                      contextID: OSAID; desiredType: DescType;
                      modeFlags: LongInt;
                      VAR resultingText: AEDesc): OSAError;
```

scriptingComponent
                A component instance created by a prior call to the Component Manager
                function OpenDefaultComponent or OpenComponent (see page 10-4).

sourceData
                A descriptor record identifying suitable source data for the specified
                scripting component.

contextID   The script ID for the context to be used during script execution. The
                constant kOSANullScript in this parameter indicates that the scripting
                component should use its default context.

desiredType
                The desired text descriptor type, such as typeChar, for the resulting
                descriptor record.

modeFlags   Information used by individual scripting components. To avoid setting
                mode flag values, specify kOSAModeNull. Other possible mode flags are
                listed in the description that follows.

resultingText
                The resulting descriptor record.

DESCRIPTION

Calling the OSADoScript function is equivalent to calling OSACompile followed by
OSAExecute and OSADisplay. After compiling the source data, executing the compiled
script using the script context identified by the contextID parameter, and returning the
text equivalent of the resulting script value in the resultingText parameter,
OSADoScript disposes of both the compiled script and the resulting script value.

You can control the way in which the scripting component executes the script by adding any of these flags to the `modeFlags` parameter:

Flag	Description
kOSAModeNeverInteract	Adds `kAENeverInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.
kOSAModeCanInteract	Adds `kAECanInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.
kOSAModeAlwaysInteract	Adds `kAEAlwaysInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.
kOSAModeCantSwitchLayer	Prevents use of `kAECanSwitchLayer` in `sendMode` parameter of `AESend` for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).
kOSAModeDontReconnect	Adds `kAEDontReconnect` to `sendMode` parameter of `AESend` for events sent when script is executed.
kOSAModeDoRecord	Prevents use of `kAEDontRecord` in `sendMode` parameter of `AESend` for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).
kOSAModeDisplayForHumans	Resulting text is readable by humans only and cannot be recompiled by `OSACompile`.

If the result code returned by `OSADoScript` is a general result code, there was some problem in arranging for the script to be run. If the result code is `errOSAScriptError`, an error occurred during script execution, and the `resultingText` parameter contains the error message associated with the error. In this case, you can obtain more detailed error information by calling `OSAScriptError`.

RESULT CODES

noErr	0	No error
errOSACantCoerce	–1700	Data could not be coerced to the requested data type
errOSASystemError	–1750	General scripting system error
errOSAInvalidID	–1751	Invalid script ID
errOSAScriptError	–1753	Source data invalid (syntax error) or an execution error occurred
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For more information about resume dispatch functions, see "Supplying a Resume Dispatch Function," which begins on page 10-21, and the description of a resume dispatch function on page 10-97.

Manipulating Dialects

Scripting components that provide several dialects may provide five functions that allow you to switch between dialects dynamically and get information about currently available dialects. The codes for specific dialects are provided by the scripting component.

The OSASetCurrentDialect function sets the current dialect, and the OSAGetCurrentDialect function gets the dialect code for the current dialect. The OSAAvailableDialectCodeList function returns a list of codes for a scripting component's dialects. You can pass any of these codes to the OSAGetDialectInfo function to get information about a specific dialect.

Instead of using the OSAAvailableDialectCodeList and OSAGetDialectInfo functions, you can use the OSAAvailableDialects function to get a descriptor list that contains information about all of the currently available dialects for a scripting component. However, it is usually more convenient to get information about just one dialect.

A scripting component that supports the functions described in this section has the kOSASupportsDialects bit set in the componentFlags field of its component description record.

OSASetCurrentDialect

You can use the OSASetCurrentDialect function to set the current dialect for a scripting component.

```
FUNCTION OSASetCurrentDialect
                        (scriptingComponent: ComponentInstance;
                         dialectCode: Integer): OSAError;
```

scriptingComponent
          A component instance created by a prior call to the Component Manager
          function OpenDefaultComponent or OpenComponent (see page 10-4).
dialectCode
          The code for the dialect to be set.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
errOSANoSuchDialect	–1757	Invalid dialect code
badComponentInstance	$80008001	Invalid component instance

OSAGetCurrentDialect

You can use the `OSAGetCurrentDialect` function to get the dialect code for the dialect currently being used by a scripting component.

```
FUNCTION OSAGetCurrentDialect
                    (scriptingComponent: ComponentInstance;
                     VAR resultingDialectCode: Integer): OSAError;
```

scriptingComponent
          A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

resultingDialectCode
          The `OSAGetCurrentDialect` function returns, in this parameter, the code for the current dialect of the specified scripting component.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
errOSANoSuchDialect	–1757	Invalid dialect code
badComponentInstance	$80008001	Invalid component instance

OSAAvailableDialectCodeList

You can use the `OSAAvailableDialectCodeList` function to obtain a descriptor list containing dialect codes for each of a scripting component's currently available dialects.

```
FUNCTION OSAAvailableDialectCodeList
                    (scriptingComponent: ComponentInstance;
                     VAR resultingDialectCodeList: AEDesc): OSAError;
```

scriptingComponent
          A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

resultingDialectCodeList
          The returned descriptor list.

DESCRIPTION

Each item in the descriptor list returned by `OSAAvailableDialectCodeList` is a descriptor record of descriptor type `typeInteger` containing a dialect code for one of the specified scripting component's currently available dialects. Dialect codes are defined by individual scripting components.

You can pass any dialect code you obtain using `OSAAvailableDialectCodeList` to `OSAGetDialectInfo` to get information about the corresponding dialect.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

OSAGetDialectInfo

You can use the `OSAGetDialectInfo` function to get information about a specified dialect provided by a specified scripting component.

```
OSAGetDialectInfo (scriptingComponent: ComponentInstance;
                   dialectCode: Integer; selector: OSType;
                   VAR resultingDialectInfo: AEDesc): OSAError;
```

scriptingComponent
:   A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

dialectCode
:   A code for the dialect about which you want information. You can obtain a list of a scripting component's dialect codes by calling `OSAAvailableDialectCodeList`.

selector
:   A constant that indicates what kind of information you want `OSAGetDialectInfo` to return in the `result` parameter. This constant determines the descriptor type for the descriptor record returned. See the description that follows for a list of the standard constants you can specify in this parameter.

resultingDialectInfo
:   A descriptor record containing the requested information. The descriptor record's descriptor type corresponds to the constant specified in the `selector` parameter.

DESCRIPTION

After you obtain a list of dialect codes by calling `OSAAvailableDialectCodeList`, you can pass any of those codes to `OSAGetDialectInfo` to get information about the corresponding dialect. The descriptor type of the descriptor record returned by `OSAGetDialectInfo` depends on the constant specified in the `selector` parameter. All scripting components support the following constants for this parameter:

```
CONST
    keyOSADialectName       = 'dnam';{used with descriptor record }
                                     { of any text type, such as }
                                     { type typeChar}
    keyOSADialectLangCode   = 'dlcd';{used with descriptor record }
                                     { of type typeShortInteger}
    keyOSADialectScriptCode = 'dscd';{used with descriptor record }
                                     { of type typeShortInteger}
```

Individual scripting components may allow you to specify additional constants.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
errOSABadSelector	–1754	Invalid selector
errOSANoSuchDialect	–1757	Invalid dialect code
badComponentInstance	$80008001	Invalid component instance

OSAAvailableDialects

You can use the OSAAvailableDialects function to obtain a descriptor list containing information about each of the currently available dialects for a scripting component.

```
FUNCTION OSAAvailableDialects
                (scriptingComponent: ComponentInstance;
                 VAR resultingDialectInfoList: AEDesc): OSAError;
```

scriptingComponent
        A component instance created by a prior call to the Component Manager function OpenDefaultComponent or OpenComponent (see page 10-4).

resultingDialectInfoList
        The returned descriptor list.

DESCRIPTION

Each item in the list returned by OSAAvailableDialects is an AE record of descriptor type typeOSADialectInfo.

```
CONST typeOSADialectInfo = 'difo';
```

Each descriptor record in the descriptor list contains, at a minimum, four keyword-specified descriptor records with the following keywords:

```
CONST
    keyOSADialectName      = 'dnam';{used with descriptor record }
                                    { of any text type, such as }
                                    { type typeChar}
    keyOSADialectCode      = 'dcod';{used with descriptor record }
                                    { of type typeShortInteger}
    keyOSADialectLangCode  = 'dlcd';{used with descriptor record }
                                    { of type typeShortInteger}
    keyOSADialectScriptCode = 'dscd';{used with descriptor record }
                                    { of type typeShortInteger}
```

Rather than calling OSAAvailableDialects to obtain complete dialect information for a scripting component, it is usually more convenient to call OSAAvailableDialectCodeList to get a list of codes for a scripting component's dialects, then call OSAGetDialectInfo to get information about the specific dialect you're interested in.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

Using Script Contexts to Handle Apple Events

The optional routines described in this section allow your application to use script contexts to handle Apple events. One way to do this is to install a general Apple event handler in your application's special handler dispatch table. The general Apple event handler provides initial handling for every Apple event received by your application. (For an example of such a handler, see "Using a Script Context to Handle an Apple Event" on page 10-19.)

The general Apple event handler extracts the event's direct parameter, obtains a script ID for the script context associated with the object described in the direct parameter, and passes the Apple event and the script ID to either OSAExecuteEvent or OSADoEvent. The main difference between these two functions is that OSAExecuteEvent returns a script ID for the resulting script value, whereas OSADoEvent returns a reply Apple event that includes either the resulting script value or information about any errors that occurred.

If the scripting component determines that a script context can't handle the specified event (for example, if an AppleScript script context doesn't include statements that handle the event), OSAExecuteEvent and OSADoEvent return errAEEventNotHandled. This causes the Apple Event Manager to look for an appropriate handler in the application's Apple event dispatch table or elsewhere,

using standard Apple event dispatching. If the scripting component determines that a script context passed to OSAExecuteEvent or OSADoEvent can handle the event, the function attempts to use the script context for that purpose.

Script contexts can in turn pass an event to a resume dispatch function with a statement that's equivalent to an AppleScript continue statement. The OSASetResumeDispatchProc and OSAGetResumeDispatchProc functions allow your application to set and get pointers to the resume dispatch function used by a scripting component. These functions use the following type for a pointer to a resume dispatch function:

```
TYPE AEHandlerProcPtr = EventHandlerProcPtr;
```

A resume dispatch function takes the same parameters as an Apple event handler and dispatches an event to an application's standard handler for that event.

If you need to create a new, empty script context, you can use the OSAMakeContext function.

A scripting component that supports the functions described in this section has the kOSASupportsEventHandling bit set in the componentFlags field of its component description record.

OSASetResumeDispatchProc

You can use the OSASetResumeDispatchProc function to set the resume dispatch function called by a scripting component during execution of an AppleScript continue statement or its equivalent.

```
FUNCTION OSASetResumeDispatchProc
                    (scriptingComponent: ComponentInstance;
                     resumeDispatchProc: AEHandlerProcPtr;
                     refCon: LongInt): OSAError;
```

scriptingComponent
          A component instance created by a prior call to the Component Manager
          function OpenDefaultComponent or OpenComponent (see page 10-4).

resumeDispatchProc
          You can specify one of the following in this parameter:

          n   a pointer to a resume dispatch function

          n   the kOSAUseStandardDispatch constant, which causes the Apple
              Event Manager to dispatch the event using standard Apple event
              dispatching

          n   the kOSANoDispatch constant, which tells the Apple Event Manager
              that the processing of the Apple event is complete and that it does not
              need to be dispatched

refCon   A reference constant. Specify kOSADontUsePhac in this parameter and
                kOSAUseStandardDispatch in the resumeDispatchProc parameter
                to request standard Apple event dispatching excluding the special
                handler dispatch table.

DESCRIPTION

The OSASetResumeDispatchProc function sets the resume dispatch function that the
specified instance of a scripting component calls during execution of an AppleScript
continue statement or its equivalent. The resume dispatch function should dispatch
the event to the application's standard handler for that event.

If you are using a general handler similar to that in Listing 10-7 on page 10-21 for
preliminary processing of Apple events, and if you can rely on standard Apple event
dispatching to dispatch the event correctly, you don't need to provide a resume dispatch
function. Instead, you can specify kOSAUseStandardDispatch as the value of the
resumeDispatchProc parameter and the constant kOSADontUsePhac as the value of
the refCon parameter. This causes the Apple Event Manager to use standard Apple
event dispatching except that it bypasses your application's special handler dispatch
table and thus won't call your general Apple event handler recursively.

RESULT CODES

noErr   0   No error
errOSASystemError   –1750   General scripting system error
badComponentInstance $80008001   Invalid component instance

SEE ALSO

For more information about resume dispatch functions, see "Supplying a Resume
Dispatch Function" on page 10-21 and the description of a resume dispatch function on
page 10-97.

OSAGetResumeDispatchProc

You can use the OSAGetResumeDispatchProc function to get the resume dispatch
function currently being used by a scripting component instance during execution of an
AppleScript continue statement or its equivalent.

```
FUNCTION OSAGetResumeDispatchProc
                    (scriptingComponent: ComponentInstance;
                     VAR resumeDispatchProc: AEHandlerProcPtr;
                     VAR refCon: LongInt): OSAError;
```

scriptingComponent

A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

resumeDispatchProc

The `OSAGetResumeDispatchProc` function returns a pointer to the resume dispatch function for the specified scripting component in this parameter. If no resume dispatch function has been registered, `OSAGetResumeDispatchProc` returns `kOSAUseStandardDispatch` (the default).

refCon

The `OSAGetResumeDispatchProc` function returns the reference constant associated with the resume dispatch function in this parameter.

RESULT CODES

`noErr`	0	No error
`errOSASystemError`	–1750	General scripting system error
`badComponentInstance`	$80008001	Invalid component instance

OSAExecuteEvent

You can use the `OSAExecuteEvent` function to handle an Apple event with the aid of a script context and obtain a script ID for the resulting script value.

```
FUNCTION OSAExecuteEvent(scriptingComponent: ComponentInstance;
                         theAppleEvent: AppleEvent;
                         contextID: OSAID;
                         modeFlags: LongInt;
                         VAR resultingScriptValueID: OSAID)
                         : OSAError;
```

scriptingComponent

A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

theAppleEvent

The Apple event to be handled.

contextID   The script ID for the script context to be used to handle the Apple event.

modeFlags   Information used by individual scripting components. To avoid setting any mode flags, specify `kOSAModeNull`. Other possible mode flags are listed in the description that follows.

resultingScriptValueID

A script ID for the resulting script value.

DESCRIPTION

The OSAExecuteEvent function attempts to use the script context specified by the contextID parameter to handle the Apple event specified by the theAppleEvent parameter. If the scripting component determines that the script context can't handle the event (for example, if a script written in AppleScript doesn't include statements that handle the event), OSAExecuteEvent immediately returns errAEEventNotHandled rather than errOSAScriptError.

If the scripting component determines that the script context can handle the event, OSAExecuteEvent executes the script context's handler and returns the resulting script ID. If execution of the script context's handler for the event generates an error, OSAExecuteEvent returns errOSAScriptError, and you can get more detailed error information by calling the OSAScriptError function.

You can control the way in which the scripting component executes a script context by adding any of these flags to the modeFlags parameter:

Flag	Description
kOSAModeNeverInteract	Adds kAENeverInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeCanInteract	Adds kAECanInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeAlwaysInteract	Adds kAEAlwaysInteract to sendMode parameter of AESend for events sent when script is executed.
kOSAModeCantSwitchLayer	Prevents use of kAECanSwitchLayer in sendMode parameter of AESend for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).
kOSAModeDontReconnect	Adds kAEDontReconnect to sendMode parameter of AESend for events sent when script is executed.
kOSAModeDoRecord	Prevents use of kAEDontRecord in sendMode parameter of AESend for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).

If the script context identified by the contextID parameter specifies that the Apple event should be passed to the application's default handler for that event (for example, with an AppleScript continue statement), OSAExecuteEvent passes the event to the resume dispatch function currently being used by the scripting component. The resume dispatch function dispatches the event directly to the application's standard handler for that event (that is, without calling OSAExecuteEvent again). If the contextID parameter is kOSANullScript, the OSAExecuteEvent function passes the event

directly to the resume dispatch function. If a call to the resume dispatch function is successful, execution of the script context proceeds from the point at which the resume dispatch function was called.

IMPORTANT

The `OSAExecuteEvent` function can generate the result code `errAEEventNotHandled` in at least two ways. If the scripting component determines that a script context doesn't declare a handler for a particular event, `OSAExecuteEvent` immediately returns `errAEEventNotHandled`. If a scripting component calls its resume dispatch function during script execution and the application's standard handler for the event fails to handle it, `OSAExecuteEvent` returns `errOSAScriptError` and a call to `OSAScriptError` with `kOSAErrorNumber` in the `selector` parameter returns `errAEEventNotHandled` as the resulting error description. s

RESULT CODES

`noErr`	0	No error
`errAEEventNotHandled`	–1708	Script context doesn't contain handler for event
`errOSASystemError`	–1750	General scripting system error
`errOSAInvalidID`	–1751	Invalid script ID
`errOSAScriptError`	–1753	Error occurred during execution or because of an attempt to pass event to a `NIL` resume dispatch function
`badComponentInstance`	$80008001	Invalid component instance

OSADoEvent

You can use the `OSADoEvent` function to handle an Apple event with the aid of a script context and obtain a reply event.

```
FUNCTION OSADoEvent(scriptingComponent: ComponentInstance;
                    theAppleEvent: AppleEvent;
                    contextID: OSAID;
                    modeFlags: LongInt;
                    VAR reply: AppleEvent): OSAError;
```

`scriptingComponent`
            A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

`theAppleEvent`
            The Apple event to be handled.

contextID   The script ID for the script context to be used to handle the Apple event.

modeFlags   Information used by individual scripting components. To avoid setting any mode flags, specify `kOSAModeNull`. Other possible mode flags are listed in the description that follows.

reply   The reply Apple event.

DESCRIPTION

The `OSADoEvent` function resembles both `OSADoScript` and `OSAExecuteEvent`. However, unlike `OSADoScript`, the script `OSADoEvent` executes must be in the form of a script context, and execution is initiated by an Apple event. Unlike `OSAExecuteEvent`, `OSADoEvent` returns a reply Apple event rather than the script ID of the resulting script value.

The `OSADoEvent` function, like `OSAExecuteEvent`, attempts to use the script context specified by the `contextID` parameter to handle the Apple event specified by the `theAppleEvent` parameter. If the scripting component determines that the script context can't handle the event (for example, if a script written in an AppleScript dialect doesn't include statements that handle the event), `OSADoEvent` immediately returns `errAEEventNotHandled` rather than `errOSAScriptError`.

If the scripting component determines that the script context can handle the event, `OSADoEvent` executes the script context's handler for the event and returns the resulting script ID.

The `OSADoEvent` function returns a reply event that contains either the resulting script value or, if an error occurred during script execution, information about the error. If the error `errOSAScriptError` occurs during script execution, `OSADoEvent` calls `OSAScriptError` and returns the appropriate error information in the reply. The `OSADoEvent` function never returns `errOSAScriptError`.

You can control the way in which the scripting component executes a script context by adding any of these flags to the `modeFlags` parameter:

Flag	Description
`kOSAModeNeverInteract`	Adds `kAENeverInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.
`kOSAModeCanInteract`	Adds `kAECanInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.
`kOSAModeAlwaysInteract`	Adds `kAEAlwaysInteract` to `sendMode` parameter of `AESend` for events sent when script is executed.

continued

Flag	Description (continued)
kOSAModeCantSwitchLayer	Prevents use of kAECanSwitchLayer in sendMode parameter of AESend for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).
kOSAModeDontReconnect	Adds kAEDontReconnect to sendMode parameter of AESend for events sent when script is executed.
kOSAModeDoRecord	Prevents use of kAEDontRecord in sendMode parameter of AESend for events sent when script is executed (the opposite of the Apple Event Manager's interpretation of the same bit).

If the script context specifies that the Apple event should be passed to the application's standard handler for that event (for example, with an AppleScript continue statement), OSADoEvent passes the event to the resume dispatch function currently being used by the scripting component. The resume dispatch function dispatches the event directly to the application's standard handler for that event (that is, without calling OSADoEvent again). If the contextID parameter is kOSANullScript, the OSADoEvent function passes the event directly to the resume dispatch function. If the call to the resume dispatch function is successful, execution of the script context proceeds from the point at which the resume dispatch function was called.

IMPORTANT

Like OSAExecuteEvent, OSADoEvent can generate the result code errAEEventNotHandled in at least two ways. If the scripting component determines that a script context doesn't declare a handler for a particular event, OSADoEvent immediately returns errAEEventNotHandled. If a scripting component calls its resume dispatch function during script execution and the application's standard handler for the event fails to handle it, OSADoEvent returns errAEEventNotHandled in the reply Apple event. s

RESULT CODES

noErr	0	No error
errAEEventNotHandled	–1708	Script context doesn't contain handler for event
errOSASystemError	–1750	General scripting system error
errOSAInvalidID	–1751	Invalid script ID
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For an example of the use of OSADoEvent, see Listing 10-7 on page 10-21.

OSAMakeContext

You can use the `OSAMakeContext` function to get a script ID for a new script context.

```
FUNCTION OSAMakeContext(scriptingComponent: ComponentInstance;
                        contextName: AEDesc;
                        parentContext: OSAID;
                        VAR resultingContextID: OSAID): OSAError;
```

`scriptingComponent`
: A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

`contextName`
: Name of new context. Some scripting components may use context names for semantic purposes. If the value of this parameter is `typeNull`, `OSAMakeContext` creates an unnamed context.

`parentContext`
: Existing context from which new context inherits bindings. If the value of this parameter is `kOSANullScript`, the new context does not inherit bindings from any other context.

`resultingContextID`
: A script ID for the resulting script context.

DESCRIPTION

The `OSAMakeContext` function creates a new script context that you may pass to `OSAExecute` or `OSAExecuteEvent`. The new script context inherits the bindings of the script context specified in the `parentContext` parameter.

SPECIAL CONSIDERATIONS

If you call `OSAMakeContext` using an instance of the generic scripting component, the generic scripting component uses the default scripting component to create the new script context.

RESULT CODES

noErr	0	No error
errOSACantCoerce	–1700	Invalid context name
errOSASystemError	–1750	General scripting system error
errOSAInvalidID	–1751	Invalid script ID
badComponentInstance	$80008001	Invalid component instance D

SEE ALSO

To compile existing source data into a script context, use `OSACompile` as described on page 10-48.

AppleScript Component Routines

The AppleScript component provides routines for initializing the AppleScript component and manipulating the styles used to display AppleScript statements in a script. These routines are used primarily by script editors and other applications that display source data to users.

Initializing AppleScript

Before you call any of the standard scripting component routines, you can call the `ASInit` function to initialize the AppleScript component with desired application-specific stack and heap sizes. If you don't call `ASInit`, the AppleScript component initializes itself using either the values specified in the application's `'scsz'` resource or, for those values not provided by the `'scsz'` resource, default values provided by the AppleScript component.

ASInit

You can use the `ASInit` function to initialize the AppleScript component.

```
FUNCTION ASInit (scriptingComponent: ComponentInstance;
                 modeFlags: LongInt;
                 minStackSize: LongInt;
                 preferredStackSize: LongInt;
                 maxStackSize: LongInt;
                 minHeapSize: LongInt;
                 preferredHeapSize: LongInt;
                 maxHeapSize: LongInt): OSAError;
```

scriptingComponent
:   A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

modeFlags
:   Reserved for future use. Set to `kOSAModeNull`.

minStackSize
:   The minimum size for the portion of the application's heap used by the AppleScript component's application-specific stack.

preferredStackSize
:   The preferred size for the portion of the application's heap used by the AppleScript component's application-specific stack.

maxStackSize
:   The maximum size for the portion of the application's heap used by the AppleScript component's application-specific stack.

minHeapSize

> The minimum size for the portion of the application's heap used by the AppleScript component's application-specific heap.

preferredHeapSize

> The preferred size for the portion of the application's heap used by the AppleScript component's application-specific heap.

maxHeapSize

> The maximum size for the portion of the application's heap used by the AppleScript component's application-specific heap.

DESCRIPTION

Your application should set the modeFlags parameter to kOSAModeNull. You can use the other parameters to specify memory sizes for the portion of your application's heap used by the AppleScript component for its application-specific heap and stack. If your application sets any of these parameters to 0, the AppleScript component uses the corresponding value in your application's 'scsz' resource. If that value is also set to 0, the AppleScript component uses the corresponding default value:

```
CONST
    kASDefaultMinStackSize          = 1 * 1024;
    kASDefaultPreferredStackSize    = 4 * 1024;
    kASDefaultMaxStackSize          = 16 * 1024;
    kASDefaultMinHeapSize           = 4 * 1024;
    kASDefaultPreferredHeapSize     = 64 * 1024;
    kASDefaultMaxHeapSize           = 32 * 1024 * 1024;
```

If your application doesn't call ASInit explicitly, the AppleScript component initializes itself using the values specified in your application's 'scsz' resource when your application first calls any scripting component routine. If any of these values are set to 0, the AppleScript component uses the corresponding default value.

If your application doesn't call ASInit explicitly and doesn't call any scripting component routines, the AppleScript component will not be initialized. For example, if your application opens and closes the AppleScript component or calls Component Manager routines such as OpenDefaultComponent or FindNextComponent but doesn't call any scripting component routines, the AppleScript component is not initialized.

When the AppleScript component is initialized, it uses your application's high memory to create the blocks that it locks for its own use. If you expect to lock any portion of high memory for a shorter time than you expect the AppleScript component to be available, you should call ASInit explicitly.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

Getting and Setting Styles for Source Data

The `ASGetSourceStyles` and `ASSetSourceStyles` functions allow you to get and set the script format styles currently used by the AppleScript component to display scripts. To obtain a list of style names formatted according to the script format styles currently used by the AppleScript component, use the `ASGetSourceStyleNames` function.

ASGetSourceStyles

You can use the `ASGetSourceStyles` function to get the script format styles currently used by the AppleScript component to display scripts.

```
FUNCTION ASGetSourceStyles
                    (scriptingComponent: ComponentInstance;
                     VAR resultingSourceStyles: STHandle): OSAError;
```

scriptingComponent
             A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

resultingSourceStyles
             A handle to a style element array defined by the TextEdit data type `TEStyleTable` that defines the styles used for different kinds of AppleScript terms.

DESCRIPTION

The `ASGetSourceStyles` function returns a style element array that defines the nine styles used for AppleScript terms.

You can use these index constants to identify individual styles returned in the `resultingSourceStyles` parameter:

```
CONST
    kASSourceStyleUncompiledText    = 0;
    kASSourceStyleNormalText        = 1;
    kASSourceStyleLanguageKeyword   = 2;
    kASSourceStyleApplicationKeyword = 3;
    kASSourceStyleComment           = 4;
    kASSourceStyleLiteral           = 5;
```

```
kASSourceStyleUserSymbol        = 6;
kASSourceStyleObjectSpecifier   = 7;
kASNumberOfSourceStyles         = 8;
```

Other AppleScript dialects may define additional styles. When you have finished using the style element array, you must dispose of it.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For information about the `TEStyleTable` array, see *Inside Macintosh: Text.*

ASSetSourceStyles

You can use the `ASSetSourceStyles` function to set the script format styles used by the AppleScript component to display scripts.

```
FUNCTION ASSetSourceStyles (scriptingComponent: ComponentInstance;
                            sourceStyles: STHandle): OSAError;
```

scriptingComponent
            A component instance created by a prior call to the Component Manager
            function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

sourceStyles
            A handle to a style element array defined by the TextEdit data type
            `TEStyleTable` that defines the nine styles used for different kinds of
            AppleScript terms. The style for each kind of term should be identified
            according to the index constants listed for `ASGetSourceStyles` on
            page 10-82.

DESCRIPTION

The `ASSetSourceStyles` function sets the script format styles used to display scripts. If you pass a `NIL` handle in the `sourceStyles` parameter, the AppleScript component uses its default styles.

After you have set the script format styles, you must dispose of the style element array you used to specify them.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For information about the `TEStyleTable` array, see *Inside Macintosh: Text.*

ASGetSourceStyleNames

You can use the `ASGetSourceStyleNames` function to obtain a list of style names that are each formatted according to the script format styles currently used by the AppleScript component.

```
FUNCTION ASGetSourceStyleNames
                (scriptingComponent: ComponentInstance;
                 modeFlags: LongInt;
                 VAR resultingSourceStyleNameList: AEDescList)
                 : OSAError;
```

scriptingComponent
: A component instance created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

modeFlags
: Reserved for future use. Set to `kOSAModeNull`.

resultingSourceStyleNameList
: List of style names (for example, "Uncompiled Text," "Normal Text") that are each formatted according to the current script format styles. The order of the names corresponds to the order of the source style constants listed for `ASGetSourceStyles` on page 10-82.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
badComponentInstance	$80008001	Invalid component instance

Generic Scripting Component Routines

To manipulate and execute scripts written in different scripting languages, your application can either open a connection with each corresponding scripting component individually or open a single connection with the generic scripting component. For information about how to connect with scripting components, see "Connecting to a Scripting Component," which begins on page 10-3.

If you open a connection with the generic scripting component, it in turn attempts to open connections dynamically with the appropriate scripting component for each script that it executes or manipulates. To provide this capability, the generic scripting component must be able to determine which scripting component created any script ID passed as a parameter to a standard scripting component routine. Because different scripting components may end up using the same script ID to refer to different scripts, the generic scripting component uses its own **generic script IDs.** The generic scripting component translates generic scripting IDs into the corresponding component-specific script IDs and vice versa when necessary.

A generic script ID is a script ID of type `GenericID`.

```
TYPE GenericID = OSAID;
```

You don't need to know in detail how the generic scripting component keeps track of script IDs. However, you should be aware that the script IDs to which your application refers when it uses the generic scripting component are not the same as the script IDs used by scripting components that actually manipulate and execute scripts.

If you are writing a script editor or recorder, you must pass the existing script ID to `OSACompile` or `OSAStartRecording` when you are recompiling or recording into an existing script. This ensures that the script is recompiled or recorded using the same scripting component that originally created the script. If instead you pass `kOSANullScript` to these routines, the new script is compiled or recorded using the default scripting component. Each instance of the generic scripting component has its own default scripting component. The section "Getting and Setting the Default Scripting Component," which follows, describes routines provided by the generic scripting component that allow you to get and set the default scripting component.

The generic scripting component supports the standard scripting component routines. However, most scripting components also support their own component-specific routines. You can't use the generic scripting component to call a component-specific routine. Instead, you must use an instance of the specific scripting component that supports the routine.

To facilitate the use of component-specific routines, the generic scripting component allows you to identify the scripting component that created stored script data, get an instance of a specified scripting component, and convert between generic script IDs and component-specific script IDs. The section "Using Component-Specific Routines," which begins on page 10-87, describes the generic scripting component routines that allow you to perform these tasks.

Some generic scripting component routines take or return a component subtype of type `ScriptingComponentSelector`.

```
TYPE ScriptingComponentSelector = OSType;
```

You can use subtype codes of this type to identify specific scripting components.

Getting and Setting the Default Scripting Component

The default scripting component for any instance of the generic
scripting component is initially AppleScript, but you can change
it if necessary. The OSAGetDefaultScriptingComponent and
OSASetDefaultScriptingComponent functions allow you to get
and set the default scripting component.

OSAGetDefaultScriptingComponent

You can use the OSAGetDefaultScriptingComponent function to get the subtype
code for the default scripting component associated with an instance of the generic
scripting component.

```
FUNCTION OSAGetDefaultScriptingComponent
                (genericScriptingComponent: ComponentInstance;
                 VAR scriptingSubType: ScriptingComponentSelector)
                 : OSAError;
```

genericScriptingComponent
                A component instance for the generic scripting component, created by a
                prior call to the Component Manager function OpenDefaultComponent
                or OpenComponent (see page 10-4).

scriptingSubType
                The function returns, in this parameter, the subtype code for the default
                scripting component associated with the instance of the generic scripting
                component specified in the genericScriptingComponent parameter.

DESCRIPTION

The OSAGetDefaultScriptingComponent function returns the subtype code for the
default scripting component. This is the scripting component that will be used by
OSAStartRecording, OSACompile, or OSACompileExecute if no existing script ID
is specified. From the user's point of view, the default scripting component corresponds
to the scripting language selected in the Script Editor application when the user first
creates a new script.

Each instance of the generic scripting component has its own default
scripting component, which is initially AppleScript. You can use
OSASetDefaultScriptingComponent to change the default scripting component.

RESULT CODES

noErr	0	No error
errOSACantOpenComponent	–1762	Can't connect to scripting component
badComponentInstance	$80008001	Invalid component instance

OSASetDefaultScriptingComponent

You can use the `OSASetDefaultScriptingComponent` function to set the default scripting component associated with an instance of the generic scripting component.

```
FUNCTION OSASetDefaultScriptingComponent
                (genericScriptingComponent: ComponentInstance;
                 scriptingSubType: ScriptingComponentSelector)
                 : OSAError;
```

`genericScriptingComponent`
        A component instance for the generic scripting component, created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

`scriptingSubType`
        The subtype code for the scripting component you want to set as the default.

DESCRIPTION

The `OSASetDefaultScriptingComponent` function sets the default scripting component for the specified instance of the generic scripting component to the scripting component identified by the `scriptingSubType` parameter.

Each instance of the generic scripting component has its own default scripting component, which is initially AppleScript. You can use `OSAGetDefaultScriptingComponent` to get the current default scripting component for an instance of the generic scripting component.

RESULT CODES

noErr	0	No error
errOSACantOpenComponent	–1762	Specified component subtype hasn't been registered
badComponentInstance	$80008001	Invalid component instance

Using Component-Specific Routines

You can't use the generic scripting component to call a component-specific routine. Instead, you must use an instance of the specific scripting component that supports the routine.

To facilitate the use of component-specific routines, the generic scripting component allows you to identify the scripting component that created stored script data, get an instance of a specified scripting component, and convert between generic script IDs and component-specific script IDs.

If you want to identify the scripting component that created a storage descriptor record but don't want to load the script, use the OSAGetScriptingComponentFromStored function. When you need to use a specific scripting component, the OSAGetScriptingComponent function allows you to get a component instance for that scripting component.

The OSAGenericToRealID and OSARealToGenericID functions allow you to convert between generic script IDs and component-specific script IDs.

OSAGetScriptingComponentFromStored

You can use the OSAGetScriptingComponentFromStored routine to get the subtype code for a scripting component that created a storage descriptor record.

```
FUNCTION OSAGetScriptingComponentFromStored
            (genericScriptingComponent: ComponentInstance;
             scriptData: AEDesc;
             VAR scriptingSubType: ScriptingComponentSelector)
             : OSAError;
```

genericScriptingComponent
                A component instance for the generic scripting component, created by a
                prior call to the Component Manager function OpenDefaultComponent
                or OpenComponent (see page 10-4).

scriptData
                Either a generic storage descriptor record or a component-specific storage
                descriptor record.

scriptingSubType
                The function returns, in this parameter, a subtype code identifying the
                scripting component that created the descriptor record specified by the
                scriptData parameter.

DESCRIPTION

The OSAGetScriptingComponentFromStored function returns, in the scriptingSubType parameter, the subtype code for the scripting component that created the script data specified by the scriptData parameter.

The generic scripting component automatically identifies the appropriate scripting component for you when you use it to call OSALoad. By calling OSAGetScriptingComponentFromStored, you can determine, without loading a script, which scripting component created the script data.

RESULT CODES

noErr	0	No error
errOSACantOpenComponent	–1762	Can't connect to scripting component
badComponentInstance	$80008001	Invalid component instance

OSAGetScriptingComponent

You can use the OSAGetScriptingComponent function to get the instance of a scripting component for a specified subtype.

```
FUNCTION OSAGetScriptingComponent
                (genericScriptingComponent: ComponentInstance;
                 scriptingSubType: ScriptingComponentSelector;
                 VAR scriptingInstance: ComponentInstance)
                 : OSAError;
```

genericScriptingComponent
              A component instance for the generic scripting component, created by a prior call to the Component Manager function OpenDefaultComponent or OpenComponent (see page 10-4).

scriptingSubType
              A subtype code for a scripting component.

scriptingInstance
              The function returns, in this parameter, a component instance for the scripting component identified by the scriptingSubType parameter.

DESCRIPTION

You can't use the generic scripting component with component-specific routines. Instead, use an instance of the specific scripting component, which you can obtain with OSAGetScriptingComponent.

The OSAGetScriptingComponent function returns, in the scriptingInstance parameter, an instance of the scripting component identified by the scriptingSubType parameter. Each instance of the generic scripting component keeps track of a single instance of each component subtype, so OSAGetScriptingComponent always returns the same instance of a specified scripting component that the generic scripting component uses for standard scripting component routines.

For example, you can use OSAGetDefaultComponent to get the subtype code for the default scripting component (that is, the scripting component used by the generic scripting component for new scripts). You can then get an instance of the default scripting component by passing its subtype code to OSAGetScriptingComponent. Finally, you can pass that instance to OSAScriptingComponentName to obtain the default scripting component's name so you can display it to the user.

Similarly, you can pass `kAppleScriptSubtype` in the `scriptingSubType` parameter to obtain an instance of the AppleScript component. This is necessary, for example, to call AppleScript-specific routines such as `ASGetSourceStyles`.

RESULT CODES

noErr	0	No error
errOSACantOpenComponent	–1762	Can't connect to scripting component
badComponentInstance	$80008001	Invalid component instance

SEE ALSO

For descriptions of the `OSAGetDefaultScriptingComponent` and `OSAScriptingComponentName` functions, see page 10-86 and page 10-47, respectively.

OSAGenericToRealID

You can use the `OSAGenericToRealID` function to convert a generic script ID to the corresponding component-specific script ID.

```
FUNCTION OSAGenericToRealID
                (genericScriptingComponent: ComponentInstance;
                 VAR theScriptID: OSAID;
                 VAR theExactComponent: ComponentInstance)
                 : OSAError;
```

genericScriptingComponent
:   A component instance for the generic scripting component, created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

theScriptID
:   The generic script ID that you want to convert. The `OSAGenericToRealID` function returns, in this parameter, the component-specific script ID that corresponds to the generic script ID that you pass in this parameter.

theExactComponent
:   The `OSAGenericToRealID` function returns, in this parameter, the component instance that created the script ID returned in the `theScriptID` parameter.

DESCRIPTION

You can't use the generic scripting component and a generic script ID with component-specific routines. Instead, you can use the component instance and script ID returned by `OSAGenericToRealID`.

Given a generic script ID (that is, a script ID returned by a call to a standard component routine via the generic scripting component), the `OSAGenericToRealID` function returns the equivalent component-specific script ID and the component instance that created that script ID. The `OSAGenericToRealID` function modifies the script ID in place, changing the generic script ID you pass in the `theScriptID` parameter to the corresponding component-specific script ID.

RESULT CODES

`noErr`	0	No error
`errOSACantOpenComponent`	−1762	Can't connect to scripting component
`badComponentInstance`	$80008001	Invalid component instance

OSARealToGenericID

You can use the `OSARealToGenericID` function to convert a component-specific script ID to the corresponding generic script ID.

```
FUNCTION OSARealToGenericID
                (genericScriptingComponent: ComponentInstance;
                 VAR theScriptID: OSAID;
                 theExactComponent: ComponentInstance)
                 : OSAError;
```

`genericScriptingComponent`
> A component instance for the generic scripting component, created by a prior call to the Component Manager function `OpenDefaultComponent` or `OpenComponent` (see page 10-4).

`theScriptID`
> The component-specific script ID that you want to convert. You must have obtained this script ID from the scripting component instance passed in the `theExactComponent` parameter. The `OSARealToGenericID` function returns, in this parameter, the generic script ID that corresponds to the component-specific script ID that you pass in this parameter.

`theExactComponent`
> A scripting component instance returned by a generic scripting component routine.

DESCRIPTION

The OSARealToGenericID function performs the reverse of the task performed by OSAGenericToRealID. Given a component-specific script ID and an exact scripting component instance (that is, the component instance that created the component-specific script ID), the OSARealToGenericID function returns the corresponding generic script ID. The OSARealToGenericID function modifies the script ID in place, changing the component-specific script ID passed in the theScriptID parameter to the corresponding generic script ID.

You'll need to do this if you have obtained a component-specific script ID using an exact scripting component instance and you want to refer to the same script in calls that use an instance of the generic scripting component. You can't use a component-specific script ID with the generic scripting component.

The script ID you pass in the theScriptID parameter must be a component-specific script ID obtained from a scripting component instance known to the generic scripting component. You can obtain such an instance by calling either OSAGetScriptingComponent or OSAGenericToRealID.

RESULT CODES

noErr	0	No error
errOSAComponentMismatch	–1761	The theScriptID and theExactComponent parameters are for two different scripting components
errOSACantOpenComponent	–1762	Can't connect to scripting component
badComponentInstance	$80008001	Invalid component instance

Routines Used by Scripting Components

Scripting components can call three routines to manipulate the trailers for generic storage descriptor records. "Writing a Scripting Component" on page 10-27 provides general guidelines for writing a scripting component.

Manipulating Trailers for Generic Storage Descriptor Records

All scripting components must use the OSAGetStorageType, OSAAddStorageType, and OSARemoveStorageType functions described in this section to add, remove, and inspect the trailers appended to script data in generic storage descriptor records.

For more information about generic storage descriptor records, see "Saving Script Data," which begins on page 10-12.

OSAGetStorageType

You can use the OSAGetStorageType function to retrieve the scripting component subtype from the script trailer appended to the script data in a generic storage descriptor record.

```
FUNCTION OSAGetStorageType(scriptData: Handle;
                                VAR type: DescType): OSErr;
```

scriptData
                A handle to the script data.

type   The descriptor type specified in the script data trailer.

DESCRIPTION

The OSAGetStorageType function retrieves the scripting component subtype from the trailer. If no trailer can be found, OSAGetStorageType returns the error errOSABadStorageType.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
errOSABadStorageType	–1752	Bad storage type

OSAAddStorageType

You can use the OSAAddStorageType routine to add a trailer to the script data in a generic storage descriptor record.

```
FUNCTION OSAAddStorageType(scriptData: Handle;
                                type: DescType): OSErr;
```

scriptData A handle to the script data.

type   The descriptor type to be specified in the trailer added to the script data.

DESCRIPTION

The OSAAddStorageType routine attaches a trailer to a handle (consequently expanding the data to which the handle refers) or updates an existing trailer.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
errOSABadStorageType	–1752	Bad storage type

OSARemoveStorageType

You can use the OSARemoveStorageType routine to remove a trailer from the script data in a generic storage descriptor record.

```
FUNCTION OSARemoveStorageType (scriptData: Handle): OSErr;
```

scriptData
    A handle to the script data.

DESCRIPTION

The OSARemoveStorageType routine removes an existing trailer (reducing the handle's size). If no trailer can be found, then the handle is not modified, and noErr is returned.

RESULT CODES

noErr	0	No error
errOSASystemError	–1750	General scripting system error
errOSABadStorageType	–1752	Bad storage type

Application-Defined Routines

Your application can provide alternative active, send, and create functions for use by scripting components during script execution. All scripting components support routines that allow you to set and get the current active function called periodically by the scripting component during script execution. Some scripting components also support routines that allow you to set and get the current send and create functions used by the scripting component when it creates and sends Apple events during script execution.

This section provides the syntax declarations for the active, send, create, and resume dispatch functions. When a scripting component calls any of these functions, the A5 register is set up for your application, and your application is the current process.

MyActiveProc

Your application can provide an alternative active function that performs periodic tasks during script compilation such as checking for Command-period, spinning the cursor, and checking for system-level errors.

```
FUNCTION MyActiveProc (refCon: LongInt): OSErr;
```

refCon   A reference constant.

DESCRIPTION

Every scripting component calls an active function periodically during script compilation and execution and provides routines that allow your application to set or get the pointer to the active function.

If you don't set an alternative active function for a scripting component, it uses its own default active function. A scripting component's default active function allows a user to cancel script execution by pressing Command-period and calls `WaitNextEvent` to give other processes time.

SEE ALSO

For descriptions of the scripting component routines you can use to set or get the pointer to a scripting component's active function, see "Manipulating the Active Function" on page 10-45.

For a discussion of the role of an active function, see "Supplying an Alternative Active Function" on page 10-23.

MyAECreateProc

Your application can provide an alternative create function to gain control over the creation and addressing of Apple events. This can be useful, for example, if your application needs to add its own transaction code to the event. An alternative create function takes the same parameters as the `AECreateAppleEvent` function plus a reference constant.

```
FUNCTION MyAECreateProc (theAEEventClass: AEEventClass;
                         theAEEventID: AEEventID;
                         target: AEAddressDesc;
                         returnID: Integer;
                         transactionID: LongInt;
                         VAR result: AppleEvent;
                         refCon: LongInt): OSErr;
```

DESCRIPTION

Every scripting component calls a create function whenever it creates an Apple event during script execution and provides routines that allow you to set or get the pointer to the create function.

If you don't set an alternative create function for a scripting component, it uses the standard Apple Event Manager function `AECreateAppleEvent`, which it calls with its own default parameters.

SEE ALSO

For descriptions of the scripting component routines you can use to set or get the pointer to a scripting component's create function, see "Manipulating the Create and Send Functions," which begins on page 10-55.

For information about create functions, see "Alternative Create Functions" on page 10-24.

For a description of the parameters for the `AECreateAppleEvent` function, see "Creating Apple Events," which begins on page 5-25.

MyAESendProc

Your application can provide an alternative send function that performs almost any action instead of or in addition to sending Apple events. For example, before sending an Apple event, an alternative send function can modify the event or save a copy of the event. An alternative send function takes the same parameters as the `AESend` function plus a reference constant.

```
FUNCTION MyAESendProc (theAppleEvent: AppleEvent;
                       VAR reply: AppleEvent;
                       sendMode: AESendMode;
                       sendPriority: AESendPriority;
                       timeOutInTicks: LongInt;
                       idleProc: IdleProcPtr;
                       filterProc: EventFilterProcPtr;
                       refCon: LongInt): OSErr;
```

DESCRIPTION

Every scripting component calls a send function whenever it sends an Apple event during script execution and provides routines that allow you to set or get the pointer to the send function.

If you don't set an alternative send function for a scripting component, it uses the standard Apple Event Manager function `AESend`, which it calls with its own default parameters.

SEE ALSO

For descriptions of the scripting component routines you can use to set or get the pointer to a scripting component's send function, see "Manipulating the Create and Send Functions," which begins on page 10-55.

For more information about send functions, see "Alternative Send Functions" on page 10-25.

For a description of the parameters for the `AESend` function, see "Sending Apple Events," which begins on page 5-37.

MyResumeDispatch

Your application can provide a resume dispatch function that a scripting component calls during script execution to dispatch Apple events directly to an application's default handler for an Apple event. A resume dispatch function takes the same parameters as an Apple event handler.

```
FUNCTION MyResumeDispatch (theAppleEvent: AppleEvent;
                            reply: AppleEvent; refCon: LongInt)
                            : OSErr;
```

theAppleEvent
            The Apple event to be dispatched.
reply   The default reply Apple event provided by the Apple Event Manager.
refCon   The reference constant stored in the Apple event dispatch table for the
            Apple event.

DESCRIPTION

If a script specifies that the Apple event should be passed to an application's standard handler for that event (for example, with an AppleScript `continue` statement), the scripting component executing the script passes the event to the resume dispatch function currently being used by the scripting component. The resume dispatch function should dispatch the event directly to the application's standard handler for that event. If you use script contexts to handle Apple events, you may need to provide a resume dispatch function.

If you can rely on standard Apple event dispatching to dispatch the event correctly, you don't need to provide a resume dispatch function. Instead, you can use the OSASetResumeDispatchProc routine to specify that the Apple Event Manager should use standard Apple event dispatching instead of a resume dispatch function.

SEE ALSO

For a description of the OSASetResumeDispatchProc function, see page 10-72.

For a discussion of the use of script contexts to handle Apple events, see "Using a Script Context to Handle an Apple Event" on page 10-19.

Summary of Scripting Components

Pascal Summary

Constants

```
CONST
  {Component Manager type and subtype codes}
  kOSAComponentType                  = 'osa ';
  kOSAGenericScriptingComponentSubtype  = 'scpt';

  {null script ID passed to OSAExecute, OSAGetSource, or OSAStartRecording}
  kOSANullScript                     = 0;      {empty script}

  {mode flag that indicates a routine's default mode settings are to be }
  { used}
  kOSAModeNull                       = 0;
  kOSANullMode                       = 0;

  {mode flag used with OSAStore to store a script context without storing }
  { its parent context}
  kOSAModeDontStoreParent            = $00004000;

  {mode flag used with OSAStore, OSALoad, or OSACompile to request }
  { minimum script data}
  kOSAModePreventGetSource           = $00000001;

  {mode flags used with OSACompile, OSAExecute, OSALoadExecute, }
  { OSACompileExecute, OSADoScript, OSAExecuteEvent, and OSADoEvent}

  {these mode flags cause the scripting component to set the corresponding }
  { sendMode flags when it sends the event}
  kOSAModeNeverInteract              = kAENeverInteract;
  kOSAModeCanInteract                = kAECanInteract;
  kOSAModeAlwaysInteract             = kAEAlwaysInteract;
  kOSAModeDontReconnect              = kAEDontReconnect;
```

```
{this mode flag causes the scripting component not to set the }
{ kAECanSwitchLayer sendMode flag when it sends the event}
kOSAModeCantSwitchLayer              = $00000040;



{this mode flag causes the scripting component not to set the }
{ kAEDontRecord sendMode flag when it sends the event}
kOSAModeDoRecord                     = $00001000;


{mode flags used with OSACompile}

{this mode flag causes OSACompile to compile the source data as a script }
{ context}
kOSAModeCompileIntoContext           = $00000002;


{this mode flag causes OSACompile to augment the script data for a }
{ script context rather than replacing it}
kOSAModeAugmentContext               = $00000004;


{mode flags used with OSADisplay or OSADoScript to indicate that output }
{ needs to be readable by humans only and does not have to be recompiled }
{ by OSACompile}
kOSAModeDisplayForHumans             = $00000008;


{suite and event code for the Recorded Text event}
kOSASuite                            = 'ascr';
kOSARecordedText                     = 'recd';


{resource type for stored script data}
kOSAScriptResourceType = kOSAGenericScriptingComponentSubtype;


{descriptor type for generic storage descriptor records}
typeOSAGenericStorage                = kOSAScriptResourceType;


{descriptor types and error range keywords for OSAScriptError}
kOSAErrorNumber                      = 'errn'; {returns error number}
kOSAErrorMessage                     = 'errs'; {returns error message}
kOSAErrorBriefMessage                = 'errb'; {returns brief error }
                                              { message}
kOSAErrorApp                         = 'erap'; {returns PSN or name of }
                                              { errant application}
kOSAErrorPartialResult               = 'ptlr'; {returns partial }
                                              { result, if any}
```

```
kOSAErrorOffendingObject               = 'erob'; {returns info about }
                                                 { offending object, if }
                                                 { any}
kOSAErrorRange                         = 'erng'; {returns error range}

typeOSAErrorRange                      = 'erng'; {descriptor type for }
                                                 { error range}
keyOSASourceStart                      = 'srcs'; {start of error range}
keyOSASourceEnd                        = 'srce'; {end of error range}

{if selector parameter of kOSAScriptError is kOSAErrorNumber, scripting }
{ components may return any of these error codes}

{dynamic errors}
errOSACantCoerce                       = errAECoercionFail;
errOSACantAccess                       = errAENoSuchObject;
errOSAGeneralError                     = -2700;
errOSADivideByZero                     = -2701;
errOSANumericOverflow                  = -2702;
errOSACantLaunch                       = -2703;
errOSAAppNotHighLevelEventAware        = -2704;
errOSACorruptTerminology               = -2705;
errOSAStackOverflow                    = -2706;
errOSAInternalTableOverflow            = -2707;
errOSADataBlockTooLarge                = -2708;

{component-specific dynamic script errors: -2720 through -2739}

{static errors}
errTypeError                           = errAEWrongDataType;
errOSAMessageNotUnderstood             = errAEEventNotHandled;
errOSAUndefinedMessage                 = errAEHandlerNotFound;
errOSAIllegalIndex                     = errAEIllegalIndex;
errOASIllegalRange                     = errAEImpossibleRange;
errOSASyntaxError                      = -2740;
errOSASyntaxTypeError                  = -2741;
errOSATokenTooLong                     = -2742;
errOSAMissingParameter                 = errAEDescNotFound;
errOSAParameterMismatch                = errAEWrongNumberArgs;
errOSADuplicateParameter               = -2750;
errOSADuplicateProperty                = -2751;
errOSADuplicateHandler                 = -2752;
errOSAUndefinedVariable                = -2753;
```

```
errOSAInconsistentDeclarations        = -2754;
errOSAControlFlowError                = -2755;


{component-specific static script errors: -2760 through -2779}


{dialect-specific script errors: -2780 through -2799}


{descriptor type for each item in list returned by OSAAvailableDialects}
typeOSADialectInfo                    = 'difo';


{keywords for descriptor record of descriptor type typeOSADialectInfo; }
{ these can also be used in selector parameter of OSAGetDialectInfo}
keyOSADialectName                     = 'dnam'; {used with descriptor }
                                               { record of any text }
                                               { type, such as typeChar}
keyOSADialectCode                     = 'dcod'; {used with descriptor }
                                               { record of type }
                                               { typeShortInteger}
keyOSADialectLangCode                 = 'dlcd'; {used with descriptor }
                                               { record of type }
                                               { typeShortInteger}
keyOSADialectScriptCode               = 'dscd'; {used with descriptor }
                                               { record of type }
                                               { typeShortInteger}


{constants for use with OSASetResumeDispatchProc}
kOSAUseStandardDispatch               = kAEUseStandardDispatch;
kOSANoDispatch                        = kAENoDispatch;
kOSADontUsePhac                       = $0001;


{selectors for use with OSAGetScriptInfo}
kOSAScriptIsModified                  = 'modi';
kOSAScriptIsTypeCompiledScript        = 'cscr';
kOSAScriptIsTypeScriptValue           = 'valu';
kOSAScriptIsTypeScriptContext         = 'cntx';
kOSAScriptBestType                    = 'best';
kOSACanGetSource                      = 'gsrc';


{OSA component flags}
kOSASupportsCompiling                 = $0002;
kOSASupportsGetSource                 = $0004;
kOSASupportsAECoercion                = $0008;
kOSASupportsAESending                 = $0010;
```

```
kOSASupportsRecording                  = $0020;
kOSASupportsConvenience                = $0040;
kOSASupportsDialects                   = $0080;
kOSASupportsEventHandling              = $0100;


{component selectors}


{basic scripting}
kOSASelectLoad                     = $0001;
kOSASelectStore                    = $0002;
kOSASelectExecute                  = $0003;
kOSASelectDisplay                  = $0004;
kOSASelectScriptError              = $0005;
kOSASelectDispose                  = $0006;
kOSASelectSetScriptInfo            = $0007;
kOSASelectGetScriptInfo            = $0008;
kOSASelectSetActiveProc            = $0009;
kOSASelectGetActiveProc            = $000A;


{compiling}
kOSASelectScriptingComponentName   = $0102;
kOSASelectCompile                  = $0103;
kOSASelectCopyID                   = $0104;


{getting source data}
kOSASelectGetSource                = $0201;


{coercing script values}
kOSASelectCoerceFromDesc           = $0301;
kOSASelectCoerceToDesc             = $0302;


{manipulating send and create functions}
kOSASelectSetSendProc              = $0401;
kOSASelectGetSendProc              = $0402;
kOSASelectSetCreateProc            = $0403;
kOSASelectGetCreateProc            = $0404;
kOSASelectSetDefaultTarget         = $0405;


{recording}
kOSASelectStartRecording           = $0501;
kOSASelectStopRecording            = $0502;
```

```
{convenience}
kOSASelectLoadExecute            = $0601;
kOSASelectCompileExecute         = $0602;
kOSASelectDoScript               = $0603;
{manipulating dialects}
kOSASelectSetCurrentDialect      = $0701;
kOSASelectGetCurrentDialect      = $0702;
kOSASelectAvailableDialects      = $0703;
kOSASelectGetDialectInfo         = $0704;
kOSASelectAvailableDialectCodeList = $0705;


{executing Apple event handlers in script contexts}
kOSASelectSetResumeDispatchProc  = $0801;
kOSASelectGetResumeDispatchProc  = $0802;
kOSASelectExecuteEvent           = $0803;
kOSASelectDoEvent                = $0804;
kOSASelectMakeContext            = $0805;


{scripting-component-specific selectors begin with this value}
kOSASelectComponentSpecificStart = $1001;

{*******AppleScript component constants*******}

  typeAppleScript                = 'ascr';


  {Component Manager subtype for AppleScript component}
  kAppleScriptSubtype            = typeAppleScript;


  {AppleScript constant for storage descriptor records}
  typeASStorage                  = typeAppleScript;


  {AppleScript constant for the selector parameter of OSAGetScriptInfo}
  kASHasOpenHandler              = 'hsod';


  {AppleScript component selectors}
  kASSelectInit                  = $1001;
  kASSelectSetSourceStyles       = $1002;
  kASSelectGetSourceStyles       = $1003;
  kASSelectGetSourceStyleNames   = $1004;


  {default initialization parameters for AppleScript}
  kASDefaultMinStackSize         = 1 * 1024;
  kASDefaultPreferredStackSize   = 4 * 1024;
  kASDefaultMaxStackSize         = 16 * 1024;
```

```
kASDefaultMinHeapSize                  = 4 * 1024;
kASDefaultPreferredHeapSize            = 64 * 1024;
kASDefaultMaxHeapSize                  = 32 * 1024 * 1024;


{AppleScript source style flags}
kASSourceStyleUncompiledText           = 0;
kASSourceStyleNormalText               = 1;
kASSourceStyleLanguageKeyword          = 2;
kASSourceStyleApplicationKeyword       = 3;
kASSourceStyleComment                  = 4;
kASSourceStyleLiteral                  = 5;
kASSourceStyleUserSymbol               = 6;
kASSourceStyleObjectSpecifier          = 7;
kASNumberOfSourceStyles                = 8;


{if selector parameter of kOSAScriptError is kOSAErrorNumber, }
{ AppleScript component may return any of these error codes}


errASCantConsiderAndIgnore             = -2720;
errASCantCompareMoreThan32k            = -2721;
errASCantCompareMixedScripts           = -2722;
errASTerminologyNestingTooDeep         = -2760;
errASInconsistentNames                 = -2780; {English dialect}

{*******generic scripting component constants*******}

{component version this header file describes}
kGenericComponentVersion               = $0100;


{generic scripting component selectors}
kGSSSelectGetDefaultScriptingComponent  = $1001;
kGSSSelectSetDefaultScriptingComponent  = $1002;
kGSSSelectGetScriptingComponent         = $1003;
kGSSSelectGetScriptingComponentFromStored = $1004;
kGSSSelectGenericToRealID               = $1005;
kGSSSelectRealToGenericID               = $1006;
```

Data Types

```
TYPE
    OSAID                      = LongInt;         {script ID}
    OSAError                   = ComponentResult; {type for result codes}
```

```
{pointers for application-defined functions}
OSAActiveProcPtr            = ProcPtr;
AESendProcPtr               = ProcPtr;
AECreateAppleEventProcPtr   = ProcPtr;
AEHandlerProcPtr            = EventHandlerProcPtr;


{generic scripting component data types}
ScriptingComponentSelector = OSType;
GenericID                  = OSAID;
```

Required Scripting Component Routines

Saving and Loading Script Data

```
FUNCTION OSAStore           (scriptingComponent: ComponentInstance;
                             scriptID: OSAID;
                             desiredType: DescType;
                             modeFlags: LongInt;
                             VAR resultingScriptData: AEDesc): OSAError;
FUNCTION OSALoad            (scriptingComponent: ComponentInstance;
                             scriptData: AEDesc;
                             modeFlags: LongInt;
                             VAR resultingScriptID: OSAID): OSAError;
```

Executing and Disposing of Scripts

```
FUNCTION OSAExecute         (scriptingComponent: ComponentInstance;
                             compiledScriptID: OSAID;
                             contextID: OSAID;
                             modeFlags: LongInt;
                             VAR resultingScriptValueID: OSAID): OSAError;
FUNCTION OSADisplay         (scriptingComponent: ComponentInstance;
                             scriptValueID: OSAID;
                             desiredType: DescType;
                             modeFlags: LongInt;
                             VAR resultingText: AEDesc): OSAError;
FUNCTION OSAScriptError     (scriptingComponent: ComponentInstance;
                             selector: OSType;
                             desiredType: DescType;
                             VAR resultingErrorDescription: AEDesc)
                             : OSAError;
FUNCTION OSADispose         (scriptingComponent: ComponentInstance;
                             scriptID: OSAID): OSAError;
```

Setting and Getting Script Information

```
FUNCTION OSASetScriptInfo    (scriptingComponent: ComponentInstance;
                              scriptID: OSAID; selector: OSType;
                              value: LongInt): OSAError;
FUNCTION OSAGetScriptInfo    (scriptingComponent: ComponentInstance;
                              scriptID: OSAID; selector: OSType;
                              VAR result: LongInt): OSAError;
```

Manipulating the Active Function

```
FUNCTION OSASetActiveProc    (scriptingComponent: ComponentInstance;
                              activeProc: OSAActiveProcPtr;
                              refCon: LongInt): OSAError;
FUNCTION OSAGetActiveProc    (scriptingComponent: ComponentInstance;
                              VAR activeProc: OSAActiveProcPtr;
                              VAR refCon: LongInt): OSAError;
```

Optional Scripting Component Routines

Compiling Scripts

```
FUNCTION OSAScriptingComponentName
                             (scriptingComponent: ComponentInstance;
                              VAR resultingScriptingComponentName: AEDesc)
                              : OSAError;
FUNCTION OSACompile          (scriptingComponent: ComponentInstance;
                              sourceData: AEDesc; modeFlags: LongInt;
                              VAR previousAndResultingScriptID: OSAID)
                              : OSAError;
FUNCTION OSACopyID           (scriptingComponent: ComponentInstance;
                              fromID: OSAID; VAR toID: OSAID): OSAError;
```

Getting Source Data

```
FUNCTION OSAGetSource        (scriptingComponent: ComponentInstance;
                              scriptID: OSAID; desiredType: DescType;
                              VAR resultingSourceData: AEDesc): OSAError;
```

Coercing Script Values

```
FUNCTION OSACoerceFromDesc   (scriptingComponent: ComponentInstance;
                              scriptData: AEDesc; modeFlags: LongInt;
                              VAR resultingScriptValueID: OSAID): OSAError;
```

```
FUNCTION OSACoerceToDesc      (scriptingComponent: ComponentInstance;
                               scriptValueID: OSAID;
                               desiredType: DescType; modeFlags: LongInt;
                               VAR result: AEDesc): OSAError;
```

Manipulating the Create and Send Functions

```
FUNCTION OSASetCreateProc     (scriptingComponent: ComponentInstance;
                               createProc: AECreateAppleEventProcPtr;
                               refCon: LongInt): OSAError;
FUNCTION OSAGetCreateProc     (scriptingComponent: ComponentInstance;
                               VAR createProc: AECreateAppleEventProcPtr;
                               VAR refCon: LongInt): OSAError;
FUNCTION OSASetSendProc       (scriptingComponent: ComponentInstance;
                               sendProc: AESendProcPtr;
                               refCon: LongInt): OSAError;
FUNCTION OSAGetSendProc       (scriptingComponent: ComponentInstance;
                               VAR sendProc: AESendProcPtr;
                               VAR refCon: LongInt): OSAError;
FUNCTION OSASetDefaultTarget
                              (scriptingComponent: ComponentInstance;
                               target: AEAddressDesc): OSAError;
```

Recording Scripts

```
FUNCTION OSAStartRecording    (scriptingComponent: ComponentInstance;
                               VAR compiledScriptToModifyID: OSAID): OSAError;
FUNCTION OSAStopRecording     (scriptingComponent: ComponentInstance;
                               compiledScriptID: OSAID): OSAError;
```

Executing Scripts in One Step

```
FUNCTION OSALoadExecute       (scriptingComponent: ComponentInstance;
                               scriptData: AEDesc;
                               contextID: OSAID; modeFlags: LongInt;
                               VAR resultingScriptValueID: OSAID): OSAError;

FUNCTION OSACompileExecute    (scriptingComponent: ComponentInstance;
                               sourceData: AEDesc;
                               contextID: OSAID; modeFlags: LongInt;
                               VAR resultingScriptValueID: OSAID): OSAError;

FUNCTION OSADoScript          (scriptingComponent: ComponentInstance;
                               sourceData: AEDesc;
                               contextID: OSAID;
                               desiredType: DescType; modeFlags: LongInt;
                               VAR resultingText: AEDesc): OSAError;
```

Manipulating Dialects

```
FUNCTION OSASetCurrentDialect
                            (scriptingComponent: ComponentInstance;
                             dialectCode: Integer): OSAError;
FUNCTION OSAGetCurrentDialect
                            (scriptingComponent: ComponentInstance;
                             VAR resultingDialectCode: Integer): OSAError;
FUNCTION OSAAvailableDialectCodeList
                            (scriptingComponent: ComponentInstance;
                             VAR resultingDialectCodeList: AEDesc)
                             : OSAError;
FUNCTION OSAGetDialectInfo  (scriptingComponent: ComponentInstance;
                             dialectCode: Integer; selector: OSType;
                             VAR resultingDialectInfo: AEDesc): OSAError;
FUNCTION OSAAvailableDialects
                            (scriptingComponent: ComponentInstance;
                             VAR resultingDialectCodeList: AEDesc)
                             : OSAError;
```

Using Script Contexts to Handle Apple Events

```
FUNCTION OSASetResumeDispatchProc
                            (scriptingComponent: ComponentInstance;
                             resumeDispatchProc: AEHandlerProcPtr;
                             refCon: LongInt): OSAError;
FUNCTION OSAGetResumeDispatchProc
                            (scriptingComponent: ComponentInstance;
                             VAR resumeDispatchProc: AEHandlerProcPtr;
                             VAR refCon: LongInt): OSAError;
FUNCTION OSAExecuteEvent    (scriptingComponent: ComponentInstance;
                             theAppleEvent: AppleEvent;
                             contextID: OSAID; modeFlags: LongInt;
                             VAR resultingScriptValueID: OSAID): OSAError;
FUNCTION OSADoEvent         (scriptingComponent: ComponentInstance;
                             theAppleEvent: AppleEvent;
                             contextID: OSAID; modeFlags: LongInt;
                             VAR reply: AppleEvent): OSAError;
FUNCTION OSAMakeContext     (scriptingComponent: ComponentInstance;
                             contextName: AEDesc;
                             parentContext: OSAID;
                             VAR resultingContextID: OSAID): OSAError;
```

AppleScript Component Routines

Initializing AppleScript

```
FUNCTION ASInit            (scriptingComponent: ComponentInstance;
                            modeFlags: LongInt;
                            minStackSize: LongInt;
                            preferredStackSize: LongInt;
                            maxStackSize: LongInt;
                            minHeapSize: LongInt;
                            preferredHeapSize: LongInt;
                            maxHeapSize: LongInt): OSAError;
```

Getting and Setting Styles for Source Data

```
FUNCTION ASGetSourceStyles  (scriptingComponent: ComponentInstance;
                            VAR resultingSourceStyles: STHandle): OSAError;

FUNCTION ASSetSourceStyles  (scriptingComponent: ComponentInstance;
                            sourceStyles: STHandle): OSAError;

FUNCTION ASGetSourceStyleNames
                           (scriptingComponent: ComponentInstance;
                            modeFlags: LongInt;
                            VAR resultingSourceStyleNamesList: AEDescList)
                            : OSAError;
```

Generic Scripting Component Routines

Getting and Setting the Default Scripting Component

```
FUNCTION OSAGetDefaultScriptingComponent
                           (genericScriptingComponent: ComponentInstance;
                            VAR scriptingSubType:
                            ScriptingComponentSelector): OSAError;
FUNCTION OSASetDefaultScriptingComponent
                           (genericScriptingComponent: ComponentInstance;
                            scriptingSubType: ScriptingComponentSelector):
                            OSAError;
```

Using Component-Specific Routines

```
FUNCTION OSAGetScriptingComponentFromStored
                           (genericScriptingComponent: ComponentInstance;
                            scriptData: AEDesc;
                            VAR scriptingSubType:
                            ScriptingComponentSelector): OSAError;
```

```
FUNCTION OSAGetScriptingComponent
                          (genericScriptingComponent: ComponentInstance;
                           scriptingSubType: ScriptingComponentSelector;
                           VAR scriptingInstance: ComponentInstance)
                           : OSAError;
FUNCTION OSAGenericToRealID (genericScriptingComponent: ComponentInstance;
                           VAR theScriptID: OSAID;
                           VAR theExactComponent: ComponentInstance)
                           : OSAError;
FUNCTION OSARealToGenericID (genericScriptingComponent: ComponentInstance;
                           VAR theScriptID: OSAID;
                           theExactComponent: ComponentInstance)
                           : OSAError;
```

Routines Used by Scripting Components

Manipulating Trailers for Generic Storage Descriptor Records

```
FUNCTION OSAGetStorageType   (scriptData: Handle; VAR type: DescType): OSErr;

FUNCTION OSAAddStorageType   (scriptData: Handle; type: DescType): OSErr;

FUNCTION OSARemoveStorageType
                          (scriptData: Handle): OSErr;
```

Application-Defined Routines

```
FUNCTION MyActiveProc        (refCon: LongInt): OSErr;
FUNCTION MyAECreateProc      (theAEEventClass: AEEventClass;
                           theAEEventID: AEEventID; target: AEAddressDesc;
                           returnID: Integer; transactionID: LongInt;
                           VAR result: AppleEvent;
                           refCon: LongInt): OSErr;
FUNCTION MyAESendProc        (theAppleEvent: AppleEvent;
                           VAR reply: AppleEvent; sendMode: AESendMode;
                           sendPriority: AESendPriority;
                           timeOutInTicks: LongInt;
                           idleProc: IdleProcPtr;
                           filterProc: EventFilterProcPtr;
                           refCon: LongInt): OSErr;
FUNCTION MyResumeDispatch     (theAppleEvent: AppleEvent; reply: AppleEvent;
                           refCon: LongInt): OSErr;
```

C Summary

Constants

```
/*Component Manager type and subtype codes*/
#define kOSAComponentType               'osa '
#define kOSAGenericScriptingComponentSubtype 'scpt'

/*null script ID passed to OSAExecute, OSAGetSource, or OSAStartRecording*/
#define kOSANullScript                  ((OSAID) 0)

/*mode flag that indicates a routine's default mode settings are to be used*/
#define kOSAModeNull                    0
#define kOSANullMode                    0

/*mode flag used with OSAStore to store a script context without storing */
/* its parent context*/
#define kOSAModeDontStoreParent         0x00004000

/*mode flag used with OSAStore, OSALoad, or OSACompile to request */
/* minimum script data*/
#define kOSAModePreventGetSource        0x00000001

/*mode flags used with OSACompile, OSAExecute, OSALoadExecute, */
/* OSACompileExecute, OSADoScript, OSAExecuteEvent, and OSADoEvent*/

/*these mode flags cause the scripting component to set the corresponding */
/* sendMode flags when it sends the event*/
#define kOSAModeNeverInteract           kAENeverInteract
#define kOSAModeCanInteract             kAECanInteract
#define kOSAModeAlwaysInteract          kAEAlwaysInteract
#define kOSAModeDontReconnect           kAEDontReconnect

/*this mode flag causes the scripting component not to set the */
/* kAECanSwitchLayer sendMode flag when it sends the event*/
#define kOSAModeCantSwitchLayer         0x00000040

/*this mode flag causes the scripting component not to set the */
/* kAEDontRecord sendMode flag when it sends the event*/
#define kOSAModeDoRecord                0x00001000
```

```
/*mode flags used with OSACompile*/

/*this mode flag causes OSACompile to compile the source data as a script */
/* context*/
#define kOSAModeCompileIntoContext      0x00000002

/*this mode flag causes OSACompile to augment the script data for a script */
/* context rather than replacing it*/
#define kOSAModeAugmentContext          0x00000004

/*mode flags used with OSADisplay or OSADoScript to indicate that output */
/* needs to be readable by humans only and does not have to be recompiled */
/* by OSACompile*/
#define kOSAModeDisplayForHumans        0x00000008

/*suite and event code for the Recorded Text event*/
#define kOSASuite                       'ascr'
#define kOSARecordedText                'recd'

/*resource type for stored script data*/
#define kOSAScriptResourceType          kOSAGenericScriptingComponentSubtype

/*descriptor type for generic storage descriptor records*/
#define typeOSAGenericStorage           kOSAScriptResourceType

/*descriptor types and error range keywords for OSAScriptError*/
#define kOSAErrorNumber                 'errn'   /*returns error number*/
#define kOSAErrorMessage                'errs'   /*returns error message*/
#define kOSAErrorBriefMessage           'errb'   /*returns brief error */
                                                 /* message*/
#define kOSAErrorApp                    'erap'   /*returns PSN or name of */
                                                 /* errant application*/

#define kOSAErrorPartialResult          'ptlr'   /*returns partial result, */
                                                 /* if any*/
#define kOSAErrorOffendingObject        'erob'   /*returns info about */
                                                 /* offending object, if any*/
#define kOSAErrorRange                  'erng'   /*returns error range*/
#define typeOSAErrorRange               'erng'   /*descriptor type for */
                                                 /* error range*/
#define keyOSASourceStart               'srcs'   /*start of error range*/
#define keyOSASourceEnd                 'srce'   /*end of error range*/
```

```
/*if selector parameter of kOSAScriptError is kOSAErrorNumber, scripting */
/* components may return any of these error codes*/


/*dynamic errors*/
#define errOSACantCoerce                        errAECoercionFail
#define errOSACantAccess                        errAENoSuchObject
#define errOSAGeneralError                  -2700
#define errOSADivideByZero                  -2701
#define errOSANumericOverflow               -2702
#define errOSACantLaunch                    -2703
#define errOSAAppNotHighLevelEventAware     -2704
#define errOSACorruptTerminology            -2705
#define errOSAStackOverflow                 -2706
#define errOSAInternalTableOverflow         -2707
#define errOSADataBlockTooLarge             -2708


/*component-specific dynamic script errors: -2720 through -2739*/


/*static errors*/
#define errTypeError                            errAEWrongDataType
#define errOSAMessageNotUnderstood              errAEEventNotHandled
#define errOSAUndefinedMessage                  errAEHandlerNotFound
#define errOSAIllegalIndex                      errAEIllegalIndex
#define errOASIllegalRange                      errAEImpossibleRange
#define errOSASyntaxError                   -2740
#define errOSASyntaxTypeError               -2741
#define errOSATokenTooLong                  -2742
#define errOSAMissingParameter                  errAEDescNotFound
#define errOSAParameterMismatch                 errAEWrongNumberArgs
#define errOSADuplicateParameter            -2750
#define errOSADuplicateProperty             -2751
#define errOSADuplicateHandler              -2752
#define errOSAUndefinedVariable             -2753
#define errOSAInconsistentDeclarations      -2754
#define errOSAControlFlowError              -2755


/*component-specific static script errors: -2760 through -2779*/


/*dialect-specific script errors: -2780 through -2799*/


/*descriptor type for each item in list returned by OSAAvailableDialects*/
#define typeOSADialectInfo              'difo'
```

```
/*keywords for descriptor record of descriptor type typeOSADialectInfo; */
/* these can also be used in selector parameter of OSAGetDialectInfo*/
#define keyOSADialectName             'dnam'   /*used with descriptor */
                                               /* record of any text */
                                               /* type, such as typeChar*/
#define keyOSADialectCode             'dcod'   /*used with descriptor */
                                               /* record of type */
                                               /* typeShortInteger*/
#define keyOSADialectLangCode         'dlcd'   /*used with descriptor */
                                               /* record of type */
                                               /* typeShortInteger*/
#define keyOSADialectScriptCode       'dscd'   /*used with descriptor */
                                               /* record of type */
                                               /* typeShortInteger*/


/*constants for use with OSASetResumeDispatchProc*/
#define kOSAUseStandardDispatch       kAEUseStandardDispatch
#define kOSANoDispatch                kAENoDispatch
#define kOSADontUsePhac               $0001


/*selectors for use with OSAGetScriptInfo*/
#define kOSAScriptIsModified          'modi'
#define kOSAScriptIsTypeCompiledScript 'cscr'
#define kOSAScriptIsTypeScriptValue   'valu'
#define kOSAScriptIsTypeScriptContext 'cntx'
#define kOSAScriptBestType            'best'
#define kOSACanGetSource              'gsrc'


/*OSA component flags*/
#define kOSASupportsCompiling         0x0002
#define kOSASupportsGetSource         0x0004
#define kOSASupportsAECoercion        0x0008
#define kOSASupportsAESending         0x0010
#define kOSASupportsRecording         0x0020
#define kOSASupportsConvenience       0x0040
#define kOSASupportsDialects          0x0080
#define kOSASupportsEventHandling     0x0100



/*component selectors*/

/*basic scripting*/
#define kOSASelectLoad                0x0001
```

```
#define kOSASelectStore                 0x0002
#define kOSASelectExecute               0x0003
#define kOSASelectDisplay               0x0004
#define kOSASelectScriptError           0x0005
#define kOSASelectDispose               0x0006
#define kOSASelectSetScriptInfo         0x0007
#define kOSASelectGetScriptInfo         0x0008
#define kOSASelectSetActiveProc         0x0009
#define kOSASelectGetActiveProc         0x000A


/*compiling*/
#define kOSASelectScriptingComponentName 0x0102
#define kOSASelectCompile               0x0103
#define kOSASelectCopyID                0x0104


/*getting source data*/
#define kOSASelectGetSource             0x0201


/*coercing script values*/
#define kOSASelectCoerceFromDesc        0x0301
#define kOSASelectCoerceToDesc          0x0302


/*manipulating send and create functions*/
#define kOSASelectSetSendProc           0x0401
#define kOSASelectGetSendProc           0x0402
#define kOSASelectSetCreateProc         0x0403
#define kOSASelectGetCreateProc         0x0404
#define kOSASelectSetDefaultTarget      0x0405


/*recording*/
#define kOSASelectStartRecording        0x0501
#define kOSASelectStopRecording         0x0502


/*convenience*/
#define kOSASelectLoadExecute           0x0601
#define kOSASelectCompileExecute        0x0602
#define kOSASelectDoScript              0x0603


/*manipulating dialects*/
#define kOSASelectSetCurrentDialect     0x0701
#define kOSASelectGetCurrentDialect     0x0702
#define kOSASelectAvailableDialects     0x0703
#define kOSASelectGetDialectInfo        0x0704
```

```
#define kOSASelectAvailableDialectCodeList 0x0705

/*executing Apple event handlers in script contexts*/
#define kOSASelectSetResumeDispatchProc     0x0801
#define kOSASelectGetResumeDispatchProc     0x0802
#define kOSASelectExecuteEvent              0x0803
#define kOSASelectDoEvent                   0x0804
#define kOSASelectMakeContext               0x0805

/*scripting-component-specific selectors are added beginning with this */
/* value*/
#define kOSASelectComponentSpecificStart  0x1001

/********AppleScript component constants********/

#define typeAppleScript                     'ascr'

/*Component Manager subtype for AppleScript component*/
#define kAppleScriptSubtype                 typeAppleScript

/*AppleScript constant for storage descriptor records*/
#define typeASStorage                       typeAppleScript

/*AppleScript constant for the selector parameter of OSAGetScriptInfo*/
#define kASHasOpenHandler                   'hsod'

/*AppleScript component selectors*/
#define kASSelectInit                  0x1001
#define kASSelectSetSourceStyles       0x1002
#define kASSelectGetSourceStyles       0x1003
#define kASSelectGetSourceStyleNames   0x1004

/*default initialization parameters for AppleScript*/
#define kASDefaultMinStackSize         1 * 1024
#define kASDefaultPreferredStackSize   4 * 1024
#define kASDefaultMaxStackSize         16 * 1024
#define kASDefaultMinHeapSize          4 * 1024
#define kASDefaultPreferredHeapSize    64 * 1024
#define kASDefaultMaxHeapSize          32 * 1024 * 1024

/*AppleScript source style flags*/
#define kASSourceStyleUncompiledText   0
#define kASSourceStyleNormalText       1
#define kASSourceStyleLanguageKeyword  2
```

```
#define kASSourceStyleApplicationKeyword     3
#define kASSourceStyleComment                4
#define kASSourceStyleLiteral                5
#define kASSourceStyleUserSymbol             6
#define kASSourceStyleObjectSpecifier        7
#define kASNumberOfSourceStyles              8


/*if selector parameter of kOSAScriptError is kOSAErrorNumber, AppleScript */
/* component may return any of these error codes*/


#define errASCantConsiderAndIgnore           -2720
#define errASCantCompareMoreThan32k          -2721
#define errASCantCompareMixedScripts         -2722
#define errASTerminologyNestingTooDeep       -2760
#define errASInconsistentNames               -2780 /*English dialect*/


/*******generic scripting component constants*******/


/*component version this header file describes*/
kGenericComponentVersion                     0x0100


/*generic scripting component selectors*/
#define kGSSSelectGetDefaultScriptingComponent  0x1001
#define kGSSSelectSetDefaultScriptingComponent  0x1002
#define kGSSSelectGetScriptingComponent         0x1003
#define kGSSSelectGetScriptingSystemFromStored  0x1004
#define kGSSSelectGenericToRealID               0x1005
#define kGSSSelectRealToGenericID               0x1006
```

Data Types

```
typedef unsigned long           OSAID;          /*script ID*/
typedef ComponentResult         OSAError;       /*type for result codes*/


/*pointers for application-defined functions*/
typedef pascal OSErr (*OSAActiveProcPtr) (long refCon);
typedef pascal OSErr (*AESendProcPtr)
                        (const AppleEvent* theAppleEvent,
                         AppleEvent* reply, AESendMode sendMode,
                         AESendPriority sendPriority,
                         long timeOutInTicks, IdleProcPtr idleProc,
                         EventFilterProcPtr filterProc, long refCon);
```

```
typedef pascal OSErr (*AECreateAppleEventProcPtr)
                            (AEEventClass theAEEventClass,
                             AEEventID theAEEventID,
                             const AEAddressDesc* target, short returnID,
                             long transactionID, AppleEvent* result,
                             long refCon);
typedef pascal OSErr (*AEHandlerProcPtr)
                            (const AppleEvent* the AppleEvent,
                             AppleEvent* reply, long refCon);


/*generic scripting component data types*/
typedef OSType ScriptingComponentSelector;
typedef OSAID GenericID;
```

Required Scripting Component Routines

Saving and Loading Script Data

```
pascal OSAError OSAStore      (ComponentInstance scriptingComponent,
                               OSAID scriptID, DescType desiredType,
                               long modeFlags, AEDesc* resultingScriptData);
pascal OSAError OSALoad       (ComponentInstance scriptingComponent,
                               const AEDesc* scriptData, long modeFlags,
                               OSAID* resultingScriptID);
```

Executing and Disposing of Scripts

```
pascal OSAError OSAExecute    (ComponentInstance scriptingComponent,
                               OSAID compiledScriptID, OSAID contextID,
                               long modeFlags, OSAID* resultingScriptValueID);
pascal OSAError OSADisplay    (ComponentInstance scriptingComponent,
                               OSAID scriptValueID, DescType desiredType,
                               long modeFlags, AEDesc* resultingText);
pascal OSAError OSAScriptError
                            (ComponentInstance scriptingComponent,
                             OSType selector, DescType desiredType,
                             AEDesc* resultingErrorDescription);
pascal OSAError OSADispose    (ComponentInstance scriptingComponent,
                               OSAID scriptID);
```

Setting and Getting Script Information

```
pascal OSAError OSASetScriptInfo
                            (ComponentInstance scriptingComponent,
                             OSAID scriptID, OSType selector, long value);
```

```
pascal OSAError OSAGetScriptInfo
                            (ComponentInstance scriptingComponent,
                             OSAID scriptID, OSType selector, long* result);
```

Manipulating the Active Function

```
pascal OSAError OSASetActiveProc
                            (ComponentInstance scriptingComponent,
                             OSAActiveProcPtr activeProc, long refCon);
pascal OSAError OSAGetActiveProc
                            (ComponentInstance scriptingComponent,
                             OSAActiveProcPtr* activeProc, long* refCon);
```

Optional Scripting Component Routines

Compiling Scripts

```
pascal OSAError OSAScriptingComponentName
                            (ComponentInstance scriptingComponent,
                             AEDesc* resultingScriptingComponentName);
pascal OSAError OSACompile  (ComponentInstance scriptingComponent,
                             const AEDesc* sourceData,
                             long modeFlags,
                             OSAID* previousAndResultingScriptID);
pascal OSAError OSACopyID   (ComponentInstance scriptingComponent,
                             OSAID fromID, OSAID* toID);
```

Getting Source Data

```
pascal OSAError OSAGetSource
                            (ComponentInstance scriptingComponent,
                             OSAID scriptID, DescType desiredType,
                             AEDesc* resultingSourceData);
```

Coercing Script Values

```
pascal OSAError OSACoerceFromDesc
                            (ComponentInstance scriptingComponent,
                             const AEDesc* scriptData, long modeFlags,
                             OSAID* resultingScriptValueID);
pascal OSAError OSACoerceToDesc
                            (ComponentInstance scriptingComponent,
                             OSAID scriptValueID, DescType desiredType,
                             long modeFlags, AEDesc* result);
```

Manipulating the Create and Send Functions

```
pascal OSAError OSASetCreateProc
                        (ComponentInstance scriptingComponent,
                         AECreateAppleEventProcPtr createProc,
                         long refCon);
pascal OSAError OSAGetCreateProc
                        (ComponentInstance scriptingComponent,
                         AECreateAppleEventProcPtr* createProc,
                         long* refCon);
pascal OSAError OSASetSendProc
                        (ComponentInstance scriptingComponent,
                         AESendProcPtr sendProc, long refCon);
pascal OSAError OSAGetSendProc
                        (ComponentInstance scriptingComponent,
                         AESendProcPtr* sendProc, long* refCon);
pascal OSAError OSASetDefaultTarget
                        (ComponentInstance scriptingComponent,
                         const AEAddressDesc* target);
```

Recording Scripts

```
pascal OSAError OSAStartRecording
                        (ComponentInstance scriptingComponent,
                         OSAID* compiledScriptToModifyID);
pascal OSAError OSAStopRecording
                        (ComponentInstance scriptingComponent,
                         OSAID compiledScriptID);
```

Executing Scripts in One Step

```
pascal OSAError OSALoadExecute
                        (ComponentInstance scriptingComponent,
                         const AEDesc* scriptData, OSAID contextID,
                         long modeFlags, OSAID* resultingScriptValueID);
pascal OSAError OSACompileExecute
                        (ComponentInstance scriptingComponent,
                         const AEDesc* sourceData, OSAID contextID,
                         long modeFlags, OSAID* resultingScriptValueID);
pascal OSAError OSADoScript (ComponentInstance scriptingComponent,
                         const AEDesc* sourceData, OSAID contextID,
                         DescType desiredType, long modeFlags,
                         AEDesc* resultingText);
```

Manipulating Dialects

```
pascal OSAError OSASetCurrentDialect
                        (ComponentInstance scriptingComponent,
                         short dialectCode);
pascal OSAError OSAGetCurrentDialect
                        (ComponentInstance scriptingComponent,
                         short* resultingDialectCode);
pascal OSAError OSAAvailableDialectCodeList
                        (ComponentInstance scriptingComponent,
                         AEDesc* resultingDialectCodeList);
pascal OSAError OSAGetDialectInfo
                        (ComponentInstance scriptingComponent,
                         short dialectCode, OSType selector,
                         AEDesc* resultingDialectInfo);
pascal OSAError OSAAvailableDialects
                        (ComponentInstance scriptingComponent,
                         AEDesc* resultingDialectInfoList);
```

Using Script Contexts to Handle Apple Events

```
pascal OSAError OSASetResumeDispatchProc
                        (ComponentInstance scriptingComponent,
                         AEHandlerProcPtr resumeDispatchProc,
                         long refCon);
pascal OSAError OSAGetResumeDispatchProc
                        (ComponentInstance scriptingComponent,
                         AEHandlerProcPtr* resumeDispatchProc,
                         long* refCon);
pascal OSAError OSAExecuteEvent
                        (ComponentInstance scriptingComponent,
                         const AppleEvent* theAppleEvent,
                         OSAID contextID, long modeFlags,
                         OSAID* resultingScriptValueID);
pascal OSAError OSADoEvent   (ComponentInstance scriptingComponent,
                         const AppleEvent* theAppleEvent,
                         OSAID contextID, long modeFlags,
                         AppleEvent* reply);
pascal OSAError OSAMakeContext
                        (ComponentInstance scriptingComponent,
                         const AEDesc* contextName,
                         OSAID parentContext,
                         OSAID* resultingContextID);
```

AppleScript Component Routines

Initializing AppleScript

```
pascal OSAError ASInit      (ComponentInstance scriptingComponent,
                             long modeFlags, long minStackSize,
                             long preferredStackSize, long maxStackSize,
                             long minHeapSize, long preferredHeapSize,
                             long maxHeapSize);
```

Getting and Setting Styles for Source Data

```
pascal OSAError ASGetSourceStyles
                            (ComponentInstance scriptingComponent,
                             STHandle* resultingSourceStyles);
pascal OSAError ASSetSourceStyles
                            (ComponentInstance scriptingComponent,
                             STHandle sourceStyles);
pascal OSAError ASGetSourceStyleNames
                            (ComponentInstance scriptingComponent,
                             long modeFlags,
                             AEDescList* resultingSourceStyleNamesList);
```

Generic Scripting Component Routines

Getting and Setting the Default Scripting Component

```
pascal OSAError OSAGetDefaultScriptingComponent
                            (ComponentInstance genericScriptingComponent,
                             ScriptingComponentSelector* scriptingSubType);
pascal OSAError OSASetDefaultScriptingComponent
                            (ComponentInstance genericScriptingComponent,
                             ScriptingComponentSelector scriptingSubType);
```

Using Component-Specific Routines

```
pascal OSAError OSAGetScriptingComponentFromStored
                            (ComponentInstance genericScriptingComponent,
                             const AEDesc *scriptData,
                             ScriptingComponentSelector scriptingSubType);
pascal OSAError OSAGetScriptingComponent
                            (ComponentInstance genericScriptingComponent,
                             ScriptingComponentSelector scriptingSubType,
                             ComponentInstance* scriptingInstance);
```

```
pascal OSAError OSAGenericToRealID
                            (ComponentInstance genericScriptingComponent,
                             OSAID *theScriptID,
                             ComponentInstance *theExactComponent);
pascal OSAError OSARealToGenericID
                            (ComponentInstance genericScriptingComponent,
                             OSAID *theScriptID,
                             ComponentInstance theExactComponent);
```

Routines Used by Scripting Components

Manipulating Trailers for Generic Storage Descriptor Records

```
pascal OSErr OSAGetStorageType
                            (Handle scriptData, DescType* type);
pascal OSErr OSAAddStorageType
                            (Handle scriptData, DescType type);
pascal OSErr OSARemoveStorageType
                            (Handle scriptData);
```

Application-Defined Routines

```
pascal OSErr MyActiveProc    (long refCon);
pascal OSErr MyAECreateProc
                            (AEEventClass theAEEventClass,
                             AEEventID theAEEventID, AEAddressDesc target,
                             short returnID, long transactionID,
                             AppleEvent* result, long refCon);
pascal OSErr MyAESendProc    (AppleEvent theAppleEvent, AppleEvent* reply,
                             AESendMode sendMode,
                             AESendPriority sendPriority,
                             long timeOutInTicks, IdleProcPtr idleProc,
                             EventFilterProcPtr filterProc, long refCon);
pascal OSErr MyResumeDispatch
                            (const AppleEvent* theAppleEvent,
                             AppleEvent* reply, long refCon);
```

Result Codes

noErr	0	No error
errOSACantCoerce	−1700	Same as errAECoercionFail; data could not be coerced to the requested data type
errOSACorruptData	−1702	Same as errAECorruptData
errAEEventNotHandled	−1708	Event wasn't handled by an Apple event handler
errAERecordingIsAlreadyOn	−1732	Attempt to turn recording on when it is already on for a recording process
errOSASystemError	−1750	General scripting system error
errOSAInvalidID	−1751	Invalid script ID
errOSABadStorageType	−1752	Illegal storage type
errOSAScriptError	−1753	Error occurred during compilation or execution
errOSABadSelector	−1754	Selector not supported by scripting component
errOSASourceNotAvailable	−1756	Source data not available
errOSANoSuchDialect	−1757	Invalid dialect code
errOSADataFormatObsolete	−1758	Data format is obsolete
errOSADataFormatTooNew	−1759	Data format is too new
errOSAComponentMismatch	−1761	Generic scripting component error; parameters are for two different scripting components instead of the same one
errOSACantOpenComponent	−1762	Generic scripting component error; can't connect to scripting component
badComponentInstance	$80008001	Invalid component instance

Program-to-Program Communications Toolbox

Contents

This chapter describes how you can use the Program-to-Program Communications (PPC) Toolbox to send and receive low-level message blocks between applications.

The PPC Toolbox can be used by different applications located on the same computer or across a network of Macintosh computers. The PPC Toolbox is available only in System 7 or later. To test for the existence of the PPC Toolbox, use the `Gestalt` function, described in *Inside Macintosh: Operating System Utilities.*

Read this chapter if you want your application to transmit and receive data from other applications that support the PPC Toolbox. Applications that utilize the PPC Toolbox must be open and connected to each other to exchange data. The PPC Toolbox allows you to send large amounts of data to other applications; it is typically useful for code that is not event-based. The PPC Toolbox is called by the Macintosh Operating System and can also be called by applications, device drivers, desk accessories, or other programs.

The PPC Toolbox provides a method of communication that is particularly useful for applications that are specifically designed to work together and are dependent on each other for information. For example, suppose one user organizes large amounts of data using a database application and another user filters and plots the same data using a plotting application. If both applications use the PPC Toolbox, these two applications can directly transmit data to each other when both applications are open and connected to each other.

You can also use the PPC Toolbox if your application communicates with other applications using high-level events or Apple events, and your application allows the user to choose another application to communicate with. You can use a PPC Toolbox routine that provides a standard user interface to display a dialog box that lists other applications that are available to exchange information. See "Browsing for Ports Using the Program Linking Dialog Box" beginning on page 11-22 for detailed information. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on high-level events, and see earlier chapters in this book for information on Apple events.

The PPC Toolbox uses the AppleTalk Data Stream Protocol (ADSP) and the Name-Binding Protocol (NBP). For detailed information on ADSP and NBP, see *Inside Macintosh: Networking.*

Note
The sample applications "store data," "display data," "send and receive," "make memo," and "spell quick" used in this chapter are not actual products of Apple Computer, Inc. They are used for illustrative purposes only. u

About the PPC Toolbox

The PPC Toolbox provides you with the ability to

n   exchange data with other open applications on the same computer or across a
network of Macintosh computers

n   browse through a listing of applications that are available to exchange data

n   verify user identities for communication across a network

To utilize the PPC Toolbox to exchange data between open applications, each application
involved must support the PPC Toolbox.

This chapter first defines the main elements of the PPC Toolbox and then discusses
how to

n   set up your application for communication

n   use security features prior to establishing communication

n   locate other applications that can exchange data

n   initiate communication between applications

n   accept or reject incoming communications requests

n   transmit and receive data between applications

n   terminate communication between applications

Ports, Sessions, and Message Blocks

To initiate communication between applications, you must first open a port. A **port** is a
portal through which your application can exchange information with another
application. A port is designated by a port name and a location name.

A **port name** is a unique identifier for a particular application on a computer. The port
name contains a name string, a type string, and a script code for localization. The
location name identifies the location of the computer on the network. The location name
contains an object string, a type string, and a zone. An application can specify an alias
location name by modifying its type string.

Your application can open as many ports as it requires as long as each port name is
unique within a particular computer. See "Specifying Port Names and Location Names"
beginning on page 11-17 for detailed information on port names and location names.

Through its port, an open application can communicate with another open application during a **session.** One port can support any number of communication sessions. During a session, an application sends and receives data in the form of a **message block.** The PPC Toolbox treats each block of data as a byte stream and delivers it in the same sequence in which it was sent.

The words *port name, location name, session,* and *message block* are programmatic terms. You should not use them in the user interface of your application or in your user documentation. Instead, refer to a file that contains executable code as an *application program.* An application program that opens and uses PPC ports supports *program linking.* When you *link* two application programs together, you are forming a *program link.* A link allows two application programs to communicate with each other— you *unlink* two application programs when you break the link between them. You can compare the link between two application programs to the communication established using telephones. For example, a program link is similar to a telephone connection that enables various forms of communication such as human-to-human, modem-to-modem, and facsimile machine–to–facsimile machine.

Figure 11-1 shows a database application on one computer that has initiated a session with a spreadsheet application located on another computer on the network.

Figure 11-1   A PPC Toolbox session between two applications
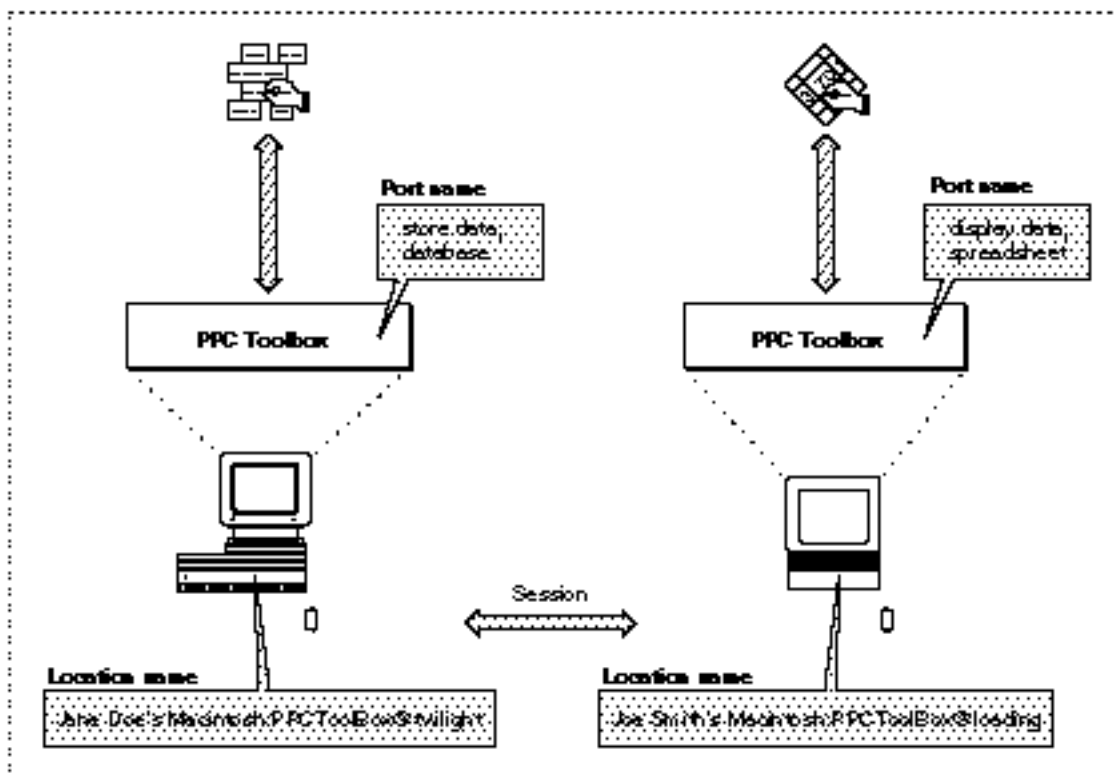
The database application's port name consists of "store data" (the name string) and "database" (the type string). Its location name consists of "Jane Doe's Macintosh" (the object string), "PPCToolBox" (the type string), and "twilight" (the AppleTalk zone).

The spreadsheet application's port name consists of "display data" (the name string) and "spreadsheet" (the type string). Its location name consists of "Joe Smith's Macintosh" (the object string), "PPCToolBox" (the type string), and "loading" (the AppleTalk zone).

Setting Up Authenticated Sessions

Network communication must be active to initiate sessions with other computers across a network. The user must activate AppleTalk in the Chooser and enable program linking using the Sharing Setup control panel located in the Control Panels folder inside the System Folder. Figure 11-2 displays the icon for the Sharing Setup control panel.

Figure 11-2   The icon for the Sharing Setup control panel

Figure 11-3 shows the Sharing Setup control panel.

Figure 11-3   The Sharing Setup control panel
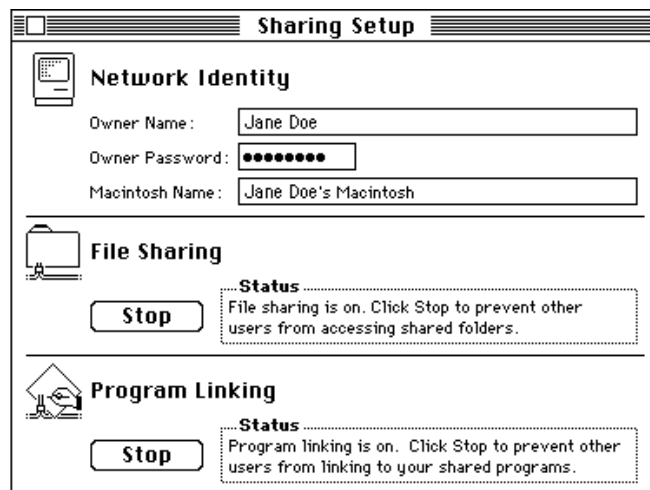
To permit other computers to initiate sessions with the owner's computer, the owner of the computer must click the Start button underneath Program Linking (Start toggles with Stop). The Sharing Setup control panel then indicates "Program linking is on. Click Stop to prevent other users from linking to your shared programs." To prevent other computers from initiating sessions, an owner simply clicks Stop underneath Program Linking. The Sharing Setup control panel then indicates "Program linking is off. Click Start to allow other users to link to your shared programs." Clicking the Start or Stop button also enables or disables the transmission of incoming Apple events across the network.

If a user clicks the Stop button while there are active incoming sessions (sessions initiated by other users), an alert box (shown in Figure 11-4) appears on the user's screen.

Figure 11-4   The session termination alert box
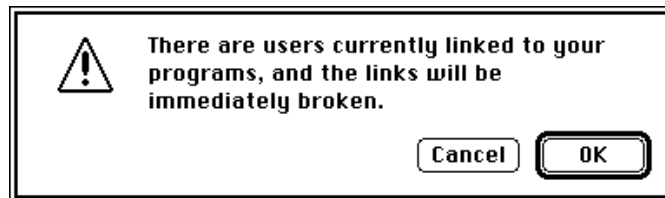
If a user clicks OK, all active sessions initiated by other users are immediately terminated. Note that it is still possible for the owner of the computer to initiate sessions, even though other users may not initiate sessions with the owner's computer.

The PPC Toolbox establishes the identity of users through the process of **authentication.** The authentication mechanism of the PPC Toolbox identifies each user through an assigned name and password. Each session initiated with a port that is located on a remote computer requires authentication (unless guest access is enabled) before a session is permitted. Sessions between applications located on the same computer never require authentication.

A computer's owner can establish access for other users and guests by opening the Users & Groups control panel located in the Control Panels folder. The Users & Groups control panel allows an owner to specify the names and passwords of other users whose computers can initiate sessions with his or her ports across the network. When the computer's owner opens the Users & Groups control panel, the Guest icon appears. If the owner's name is specified in the Sharing Setup control panel, an icon with the owner's name also appears.

To specify a particular new user, the owner chooses New User from the File menu. The owner should type in the user's name. When the owner opens a user icon in the Users & Groups control panel, the Finder displays the users and groups dialog box on the owner's screen. Figure 11-5 shows the users and groups dialog box for a particular user.

Figure 11-5   The users and groups dialog box
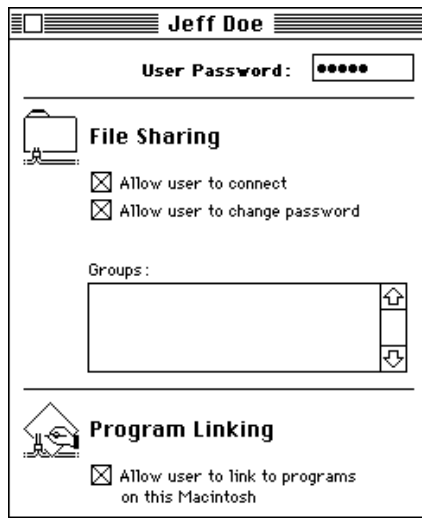
To permit authenticated session requests, the owner can specify a password for each user. The owner allows other users to utilize the PPC Toolbox by clicking the checkbox under Program Linking. If the owner clicks the checkbox again, all active sessions initiated by this particular user are immediately terminated. The user termination alert box (shown in Figure 11-6) is displayed as a warning.

Figure 11-6   The user termination alert box
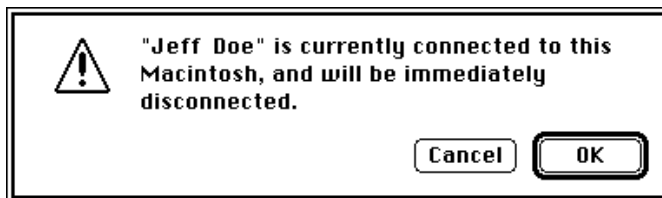
When the owner opens a Guest icon in the Users & Groups control panel, the Finder displays the guest dialog box on the owner's screen. Authentication is not required if the owner permits guest access. Figure 11-7 shows the guest dialog box.

Figure 11-7   The guest dialog box
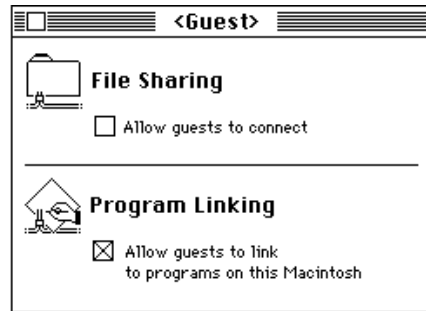
By clicking the checkbox under Program Linking, the owner permits guests to communicate using the PPC Toolbox or Apple events.

Consider this example of the authentication process: one user decides to make a dictionary service available to other users. A second user wishes to employ this service in a word-processing program. Assuming both programs support the PPC Toolbox, the word-processing program attempts to gain access to the dictionary service that is open on the first user's computer by initiating a session. When the word-processing application requests a session, the PPC Toolbox attempts to authenticate the second user by requesting a user name and a password (unless guest access is enabled). If the authentication process verifies the user's identity and the dictionary application accepts the request for a session, a session is established and the second user can access the dictionary's data.

Figure 11-8 illustrates the authentication process that occurs when a user attempts to initiate a session.

Figure 11-8   The PPC Toolbox authentication process
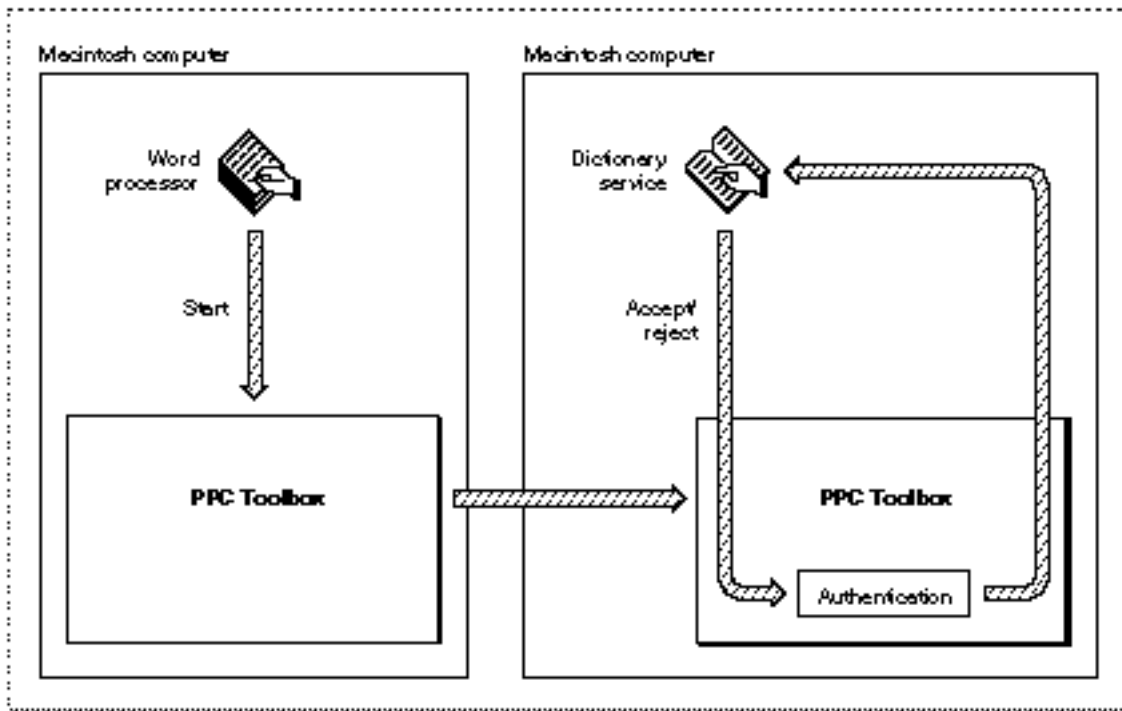
Using the PPC Toolbox

This section describes how to

n   use PPC Toolbox calling conventions

n   open a port

n   list all available port locations on the network

n   indicate that a port is available to accept session requests

n   initiate a session

n   accept and reject session requests

n   read and write data during a session

n   end a session after data is transmitted and received

n   close a port when it is no longer needed to transmit or receive data

n   invalidate users

To begin, you must determine whether the PPC Toolbox is available on the user's computer system by using the `Gestalt` function with the selector `gestaltPPCToolboxAttr`. A `noErr` result code indicates that the PPC Toolbox is present.

The `Gestalt` function returns a combination of the following constants in the `response` parameter: `gestaltPPCToolboxPresent`, `gestaltPPCSupportsRealTime`, `gestaltPPCSupportsOutGoing`, and `gestaltPPCSupportsIncoming`.

The PPC Toolbox currently supports only sessions in real time. The `Gestalt` function returns `gestaltPPCSupportsRealTime` by default. If this bit is not set, you need to initialize the PPC Toolbox.

The `Gestalt` function returns `gestaltPPCSupportsOutGoing` to indicate support of outgoing sessions across a network of Macintosh computers. If this bit is not set, the user hasn't enabled AppleTalk in the Chooser.

The `Gestalt` function returns `gestaltPPCSupportsIncoming` if the user has enabled program linking in the Sharing Setup control panel. If this bit is not set, the user either hasn't enabled AppleTalk in the Chooser or hasn't enabled program linking in the Sharing Setup control panel.

Use the `PPCInit` function to initialize the PPC Toolbox.

```
err := PPCInit;
```

Listing 11-1 illustrates how you use the `PPCInit` function to initialize the PPC Toolbox.

Listing 11-1   Initializing the PPC Toolbox using the `PPCInit` function

```
FUNCTION MyPPCInit: OSErr;
VAR
   PPCAttributes: LongInt;
   err:           OSErr;
BEGIN
   err := Gestalt(gestaltPPCToolboxAttr, PPCAttributes);
   IF err = noErr THEN            {PPC Toolbox is present}
   BEGIN
      IF BAND(PPCAttributes, gestaltPPCSupportsRealTime) = 0 THEN
      BEGIN
         MyPPCInit := PPCInit;   {initialize the PPC Toolbox}
         {test the attributes for the PPC Toolbox}
         err := Gestalt(gestaltPPCToolboxAttr, PPCAttributes);
      END;
      IF BAND(PPCAttributes, gestaltPPCSupportsOutGoing) <> 0 THEN
         {ports can be opened to the outside world}
      ELSE     {it's likely that AppleTalk is disabled, so you }
         ;       { may want to tell the user to activate AppleTalk }
                { from the Chooser}
      IF BAND(PPCAttributes, gestaltPPCSupportsIncoming) <> 0 THEN
         {ports can be opened with location names that the }
         { outside world can see}
      ELSE     {it's likely that program linking is disabled, so }
         ;       { you may want to tell the user to start program }
                { linking from the Sharing Setup control panel}
   END
   ELSE
      MyPPCInit := err;
END;
```

Figure 11-9 illustrates a spreadsheet application (on the left) that has initiated a session with a database application (on the right) to exchange data using the PPC Toolbox. This figure includes an example of the sequence of PPC Toolbox routines executed by these applications. Detailed descriptions of the functions appear in the sections that follow.

Figure 11-9 Database and spreadsheet applications using the PPC Toolbox
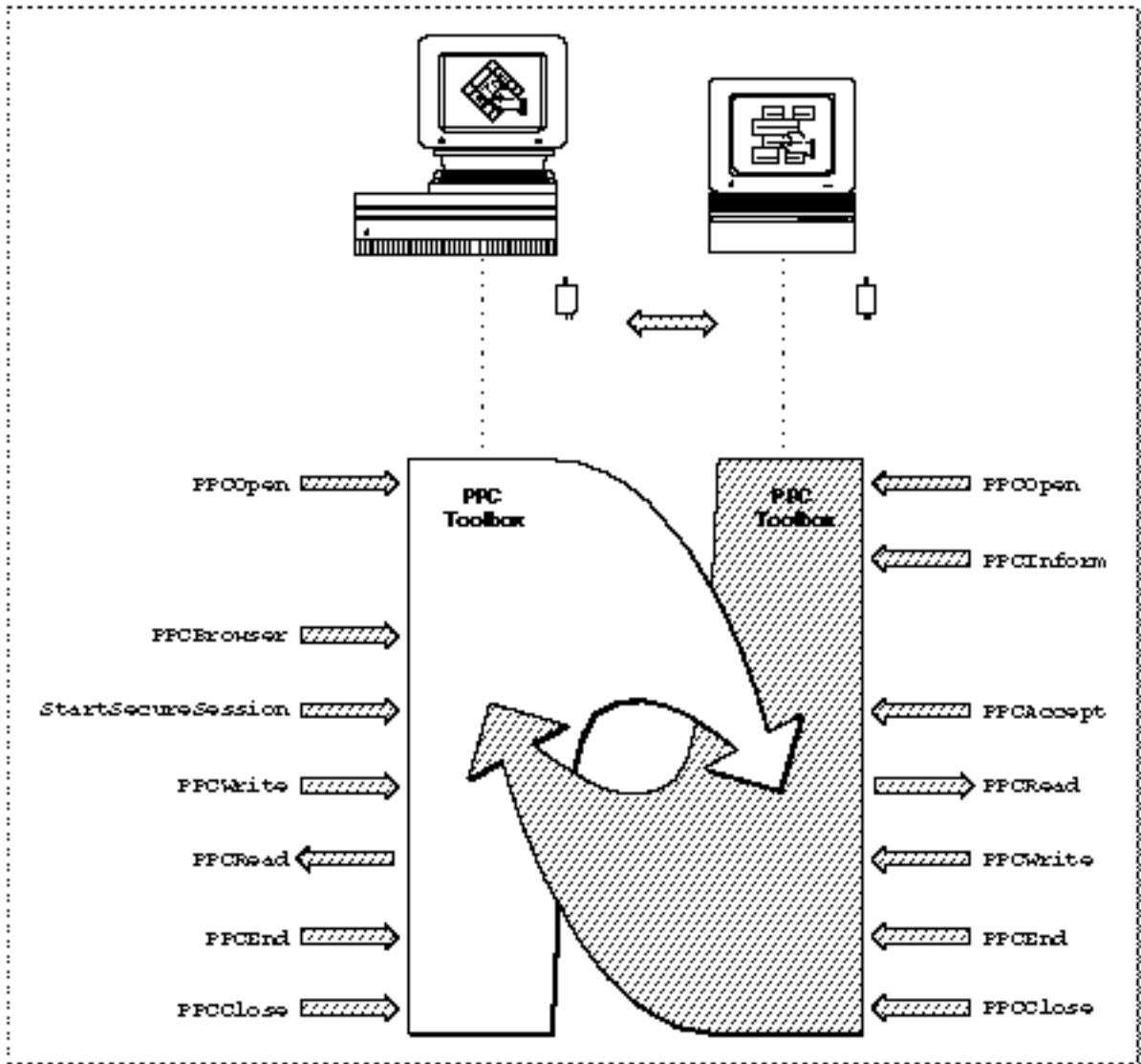
To establish a session, each application must first open a port using the `PPCOpen` function. The spreadsheet application prepares to receive session requests by calling the `PPCInform` function.

Before initiating a session or opening a port, the database application can let the user browse through the list of available ports (using the `PPCBrowser` function). If the user decides to communicate with the spreadsheet application, the database application initiates a session with the spreadsheet application's port using the `StartSecureSession` function. After the PPC Toolbox authenticates the user name and password of the initiating port, the spreadsheet application accepts the session request (using the `PPCAccept` function).

Once the session is established, the applications exchange information in the form of message blocks (using the `PPCRead` and `PPCWrite` functions). During a session, an application can both read from and write message blocks to another application. After the information exchange is done, each application ends the session (`PPCEnd`) and then closes its port (`PPCClose`) when it quits.

The `PPCOpen` function returns a port reference number. The port reference number is a reference number for the port through which you are requesting a session. The database application uses the port reference number in subsequent calls to the `StartSecureSession` and `PPCClose` functions. The `StartSecureSession` function returns a session reference number. The session reference number is used to identify the session during the exchange of data. It is used in subsequent calls to the `PPCWrite`, `PPCRead`, and `PPCEnd` functions.

The `PPCOpen` function returns a port reference number that the spreadsheet uses in subsequent calls to the `PPCInform` and `PPCClose` functions. The `PPCInform` function returns a session reference number that is used in subsequent calls to the `PPCAccept`, `PPCRead`, `PPCWrite`, and `PPCEnd` functions.

PPC Toolbox Calling Conventions

Most PPC Toolbox functions can execute synchronously (meaning that the application cannot continue until the function completes execution) or asynchronously (meaning that the application is free to perform other tasks while the function is executing). The PPC Toolbox functions that can only be executed synchronously include `PPCInit`, `PPCBrowser`, `StartSecureSession`, `DeleteUserIdentity`, and `GetDefaultUser`. All other PPC Toolbox functions can execute asynchronously or synchronously. Here's an example:

```
FUNCTION PPCFunction (pb: PPCParamBlockPtr;
                      async: Boolean): OSErr;
```

The pb parameter should point to a PPC parameter block. Set the async parameter to TRUE if you want the function to execute asynchronously; set it to FALSE if you want the function to execute synchronously.

Note

The PPCInform, PPCRead, and PPCWrite functions should always be executed asynchronously, because they require interaction from the other application in the session before they complete execution. u

The PPCParamBlockRec data type defines the PPC parameter block.

```
TYPE PPCParamBlockRec =
   RECORD
      CASE Integer OF
      0: (openParam:       PPCOpenPBRec);       {PPCOpen params}
      1: (informParam:     PPCInformPBRec);     {PPCInform params}
      2: (startParam:      PPCStartPBRec);      {PPCStart params}
      3: (acceptParam:     PPCAcceptPBRec);     {PPCAccept params}
      4: (rejectParam:     PPCRejectPBRec);     {PPCReject params}
      5: (writeParam:      PPCWritePBRec);      {PPCWrite params}
      6: (readParam:       PPCReadPBRec);       {PPCRead params}
      7: (endParam:        PPCEndPBRec);        {PPCEnd params}
      8: (closeParam:      PPCClosePBRec);      {PPCClose params}
      9: (listPortsParam:  IPCListPortsPBRec);  {IPCListPorts }
                                                { params}

   END;
```

For an illustration of the fields of each individual parameter block (such as PPCInformPBRec or IPCListPortsPBRec), see Figure 11-18 on page 11-47.

Your application transfers ownership of the PPC parameter block (and any buffers or records pointed to by the PPC parameter block) to the PPC Toolbox until a PPC function completes execution. Once the function completes, ownership of the parameter block (and any buffers or records it points to) is transferred back to your application. If a PPC Toolbox function is executed asynchronously, your program cannot alter memory that might be used by the PPC Toolbox until that function completes.

A PPC Toolbox function that is executed asynchronously must specify NIL or the address of a completion routine in the ioCompletion field of the PPC parameter block. You should use the ioResult field to determine the actual result code when an asynchronously executed PPC Toolbox function completes.

If you specify NIL in the ioCompletion field, you should poll the ioResult field of the PPC parameter block after the function is called to determine whether the PPC function has completed the requested operation. You should poll the ioResult field within the event loop of your application. If the ioResult field contains a value other than 1, the function has completed execution. Note that you must not poll the ioResult field at interrupt time to determine whether the function has completed execution.

If you specify a completion routine in the ioCompletion field, it is called at interrupt time when the PPC Toolbox function completes execution.

s   **W A R N I N G**

Completion routines execute at the interrupt level and must preserve all registers other than A0, A1, and D0–D2. (Note that MPW C and MPW Pascal do this automatically.) Your completion routine must not make any calls to the Memory Manager directly or indirectly, and it can't depend on the validity of handles to unlocked blocks. The PPC Toolbox preserves the application global register A5. s

You can write completion routines in C, Pascal, or assembly language. A completion routine declared in Pascal has this format:

```
PROCEDURE MyCompletionRoutine (pb: PPCParamBlockPtr);
```

The pb parameter points to the PPC parameter block passed to the PPC Toolbox function.

You may call another PPC Toolbox function from within a completion routine, but the function called must be executed asynchronously. It is recommended that you allocate parameter blocks of data type PPCParamBlockRec so that you may reuse the pb parameter to call another PPC Toolbox function from within a completion routine. For example, you should call either the PPCAccept function or the PPCReject function asynchronously from within a PPCInform completion routine to accept or reject the session request.

If your application is executing PPC Toolbox functions asynchronously, you may want to define your own record type to hold all data associated with a session. You can attach the data to the end of the parameter block. Here's an example:

```
TYPE
   SessRecHndl = ^SessRecPtr;
   SessRecPtr = ^SessRec;
   SessRec =
   RECORD
      pb:                     PPCParamBlockRec; {must be first }
                                                { item in record}
      thePPCPortRec:      PPCPortRec;
      theLocationNameRec: LocationNameRec;
      theUserName:        Str32;
   END;
```

The additional data elements in your record can be accessed during execution of a completion routine by coercing the pb parameter to a pointer to your record type.

Specifying Port Names and Location Names

Before initiating a session, you must open a port to communicate with other programs. A port name and location name identify each port. An application can open as many ports as it requires as long as each port name is unique within a particular computer. You specify both the port name and the location name in the PPC parameter block.

Figure 11-10 illustrates a single Macintosh computer with two applications, and their corresponding port names and location names.

To open a port, you need to specify a port name. A port name consists of a name string, a type string, and a script code for localization. For example, you can designate "make memo" as the application's name string, "word processor" as its type string, and "smRoman" as its script code.

A port name is defined by a PPC port record. The PPC port record contains a script code, name string, port kind selector, and type string. The script code is an integer script identifier used for localization. The name string consists of a 32-byte character string that designates the application name. You should keep both the script code and the name string in a resource. The port kind selector is an integer that selects the kind of type string. You should make it consistent internationally. The type string can be either a 32-byte character string or a 4-character creator and a 4-character file type. See the chapter "Finder Interface" of *Inside Macintosh: Macintosh Toolbox Essentials* for information on creators and file types. See *Inside Macintosh: Text* for information on script codes and localization.

Figure 11-10 Two Macintosh applications and their corresponding ports
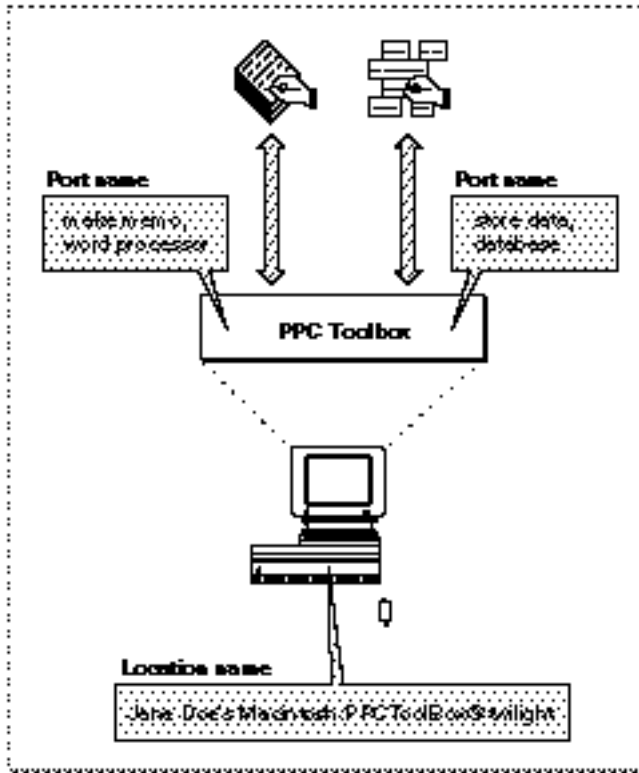
The `PPCPortRec` data type defines the PPC port record.

```
TYPE PPCPortRec =
   RECORD
      nameScript:        ScriptCode;        {script identifier}
      name:              Str32;             {port name in program }
                                            { linking dialog box}
      portKindSelector:  PPCPortKinds;      {general category of }
                                            { application}
      CASE PPCPortKinds OF
                         ppcByString:       (portTypeStr: Str32);
                         ppcByCreatorAndType:
                                            (portCreator: OSType;
                                            portType: OSType);
   END;
```

The location name identifies the location of the computer on the network. The PPC Toolbox provides the location name when the user starts up the computer. The location name is specified in the standard Name-Binding Protocol (NBP) form, *<object string>*:PPCToolBox @*<AppleTalk zone>*. The object string is the name provided in the Sharing Setup control panel in the Control Panels folder. By default, the type string is "PPCToolBox". The AppleTalk zone is the zone to which the particular Macintosh computer belongs. For example, "Jane Doe's Macintosh:PPCToolBox@twilight" specifies the object string, type string, and AppleTalk zone for a particular computer.

The `LocationNameRec` data type defines the location name record. The `locationKindSelector` field can be set to `ppcNoLocation`, `ppcNBPLocation`, or `ppcNBPTypeLocation`.

```
TYPE LocationNameRec =
   RECORD
      locationKindSelector: PPCLocationKind;    {which variant}
      CASE PPCLocationKind OF
         {ppcNoLocation: storage not used by this value}
         ppcNBPLocation:
                        (nbpEntity: EntityName); {NBP name entity}
         ppcNBPTypeLocation:
                        (nbpType: Str32); {just the NBP type }
                                          { string for the }
                                          { PPCOpen function}
   END;
```

The `ppcNoLocation` constant is used when the location received from or passed to a PPC Toolbox function is the location of the local machine.

The `ppcNBPLocation` constant is used when a full NBP entity name is received from or passed to a PPC Toolbox function.

Note

You should assign an NBP value directly—do not pack it using `nbpSetEntity`. u

The `ppcNBPTypeLocation` constant is used only by the `PPCOpen` function when an alias location name is needed.

The NBP type to be used for the alias location name is passed in the location name record's `nbpType` field. Alias location names allow you to filter the NBP objects (Macintosh computers) displayed by the program linking dialog box (shown in Figure 11-12 on page 11-22) using the `PPCBrowser` function. See "Browsing for Ports Using the Program Linking Dialog Box" beginning on page 11-22 for information on the `PPCBrowser` function.

An alias location name could be used to advertise a service (such as a dictionary service) that is available to any application located on the network. For example, "Joe Smith's Macintosh: dictionary@ozone" specifies the object string, type string, and AppleTalk zone for a particular dictionary service.

To search for all dictionary services available within a zone, you use the `PPCBrowser` function and a filter. Figure 11-11 illustrates a Macintosh dictionary service application, its corresponding port name, and its alias location name.

Figure 11-11 The PPC Toolbox and a dictionary service application
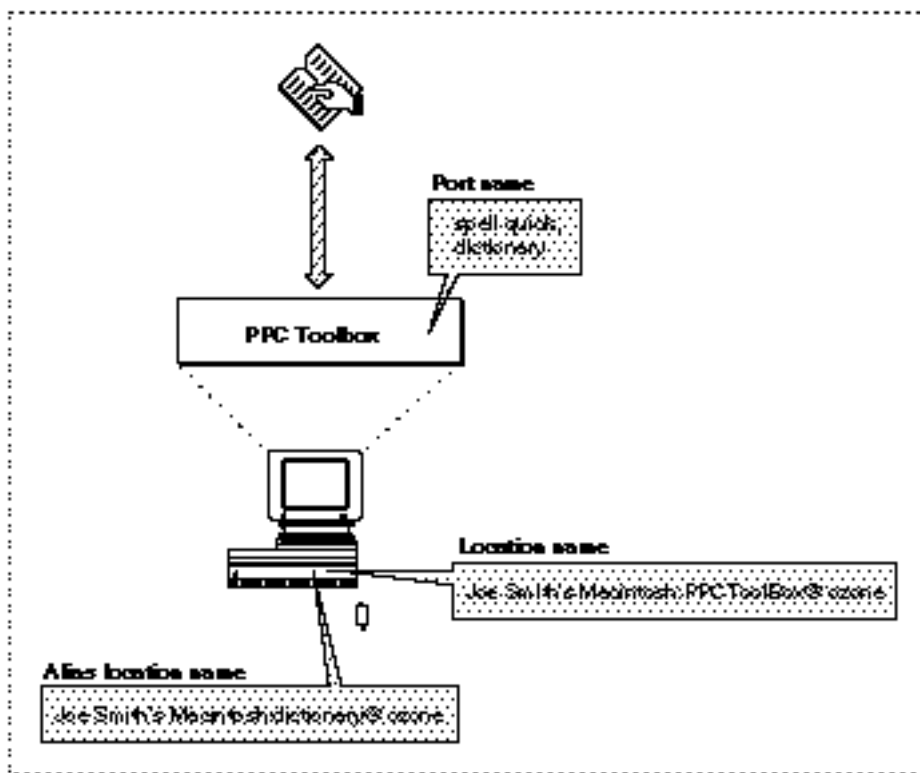
Opening a Port

To open a port and associate a name with it, use the `PPCOpen` function. Listing 11-2 illustrates how you use the `PPCOpen` function to open a port. In this listing, the name is "Inside Macintosh" and the port type string is "Example". The location name is *<object string>*:**PPC Example@***<AppleTalk zone>*.

Listing 11-2   Opening a PPC port

```
FUNCTION MyPPCOpen(VAR thePortRefNum: PPCPortRefNum;
                   VAR nbpRegisteredFlag: Boolean): OSErr;
VAR
   thePPCOpenPBRec:        PPCOpenPBRec;
   thePPCPortRec:         PPCPortRec;
   theLocationNameRec:    LocationNameRec;
BEGIN
   WITH thePPCPortRec DO
   BEGIN
      {nameScript and name should be resources to allow }
      { easy localization}
      nameScript := smRoman;  {Roman script}
      name := 'Inside Macintosh';
      {the port type should always be hard-coded to allow the }
      { application to find ports of a particular type even }
      { after the name is localized}
      portKindSelector := ppcByString;
      portTypeStr := 'Example';
   END;
   WITH theLocationNameRec DO
   BEGIN
      locationKindSelector := ppcNBPTypeLocation;
      nbpType := 'PPC Example';
   END;
   WITH thePPCOpenPBRec DO
   BEGIN
      serviceType := ppcServiceRealTime;
      resFlag := 0;                      {must be 0 for 7.0}
      portName := @thePPCPortRec;
      locationName := @theLocationNameRec;
      networkVisible := TRUE;            {make this a visible }
                                         { entity on the network}
   END;
   MyPPCOpen := PPCOpen(@thePPCOpenPBRec, FALSE);{synchronous}
   thePortRefNum := thePPCOpenPBRec.portRefNum;
   nbpRegisteredFlag := thePPCOpenPBRec.nbpRegistered;
END;
```

The PPCOpen function opens a port with the port name and location name specified in the name and location fields of the parameter block. When the PPCOpen function completes execution, the portRefNum field returns the port reference number. You can use the port reference number in the PPCInform, PPCStart, StartSecureSession, and PPCClose functions to refer to the port you have opened.

Browsing for Ports Using the Program Linking Dialog Box

Before initiating a session, you can use either the PPCBrowser function or the IPCListPorts function to locate a port to communicate with.

Use the PPCBrowser function to display the program linking dialog box (shown in Figure 11-12) on the user's screen.

Note
Because this function displays a dialog box on the user's screen, you must not call the PPCBrowser function from an application that is running in the background. u

Figure 11-12   The program linking dialog box
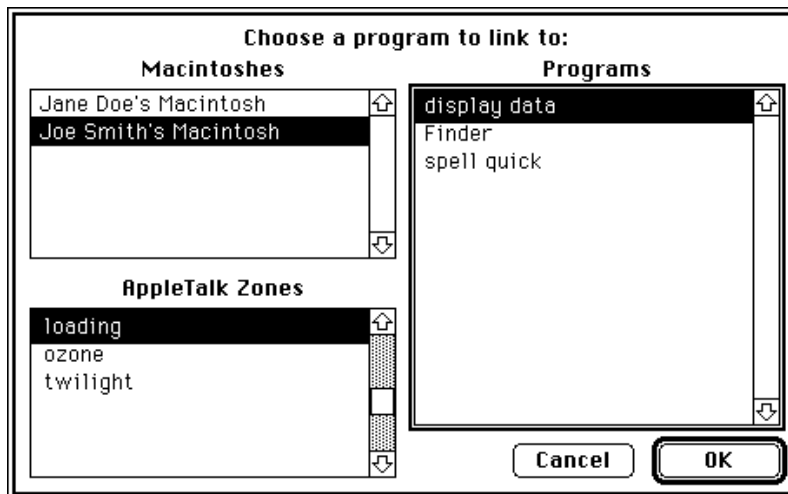
In the program linking dialog box, the user selects the computer, zone, and application. The zone list is not displayed if there is no network connection. Figure 11-13 shows the dialog box without the zone list.

Figure 11-13 The program linking dialog box without a zone list
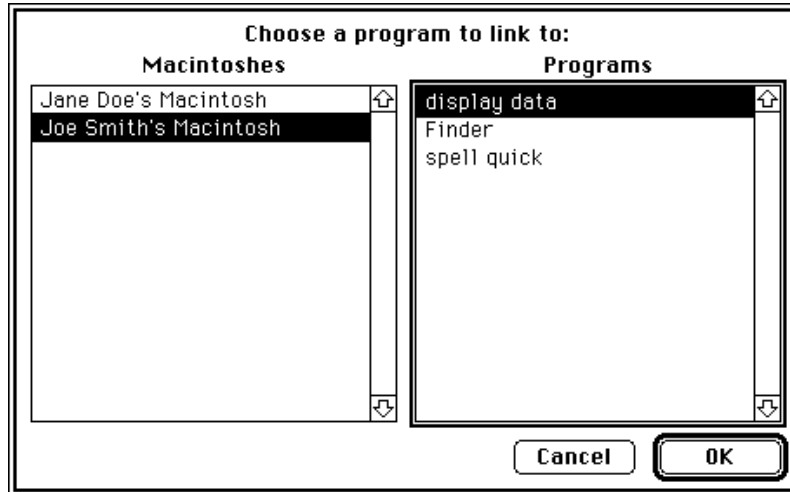
As shortcuts for the user, the program linking dialog box supports standard keyboard equivalents. Pressing Command-period or the Esc (Escape) key selects Cancel—pressing Enter or Return selects the OK button.

Each list is sorted in alphabetical order. As in the Chooser, the current list is indicated by a thick outline around its border. The program linking dialog box supports keyboard navigation and use of the arrow keys to select items from the current list. Pressing Tab or clicking the rectangle of another list switches the current list. Pressing Shift-Tab reverses the order in which the lists are selected. In addition, double-clicking an application name in the Programs list of the program linking dialog box is equivalent to clicking the OK button.

The `PPCBrowser` function allows users to browse for PPC ports.

```
err := PPCBrowser (prompt, applListLabel, defaultSpecified,
                   theLocation, thePortInfo, portFilter,
                   theLocNBPType);
```

If the `defaultSpecified` parameter is `TRUE`, the `PPCBrowser` function tries to select the PPC port specified by the parameters `theLocation` and `thePortInfo` when the program linking dialog box first appears. If the default cannot be found, the `PPCBrowser` function selects the first PPC port in the list.

An application can open multiple ports as long as each port name is unique within a particular computer. Unique ports can have duplicate name fields but different types. For example, you can designate "make memo" as the application's name string and "word processor" as its type string. You can also designate a separate port as "make memo" (the application's name string) and "text only" (its type string).

In such a case, the `PPCBrowser` function does a secondary sort based on the port type. Ports with a type selector of `ppcByCreatorAndType` are displayed before `ppcByString` ports, and types are sorted alphabetically within each type selector.

The `PPCBrowser` function uses the `IPCListPorts` function to obtain the list of existing ports on a particular computer within a particular zone. The `portFilter` parameter of the `PPCBrowser` function allows you to filter the list of PPC ports before it displays them in the program linking dialog box. If you set the `portFilter` parameter to `NIL`, the `PPCBrowser` function displays the names of all the existing PPC ports returned by the `IPCListPorts` function. If you do not set the `portFilter` parameter to `NIL`, you must set it to a pointer to a port filter function that you create.

Listing 11-3 illustrates how you use a sample port filter function. In this listing, the `MyBrowserPortFilter` function returns `TRUE` for ports with the port type string "Example".

Listing 11-3   Using a port filter function

```
FUNCTION MyBrowserPortFilter(theLocationNameRec: LocationNameRec;
                             thePortInfoRec: PortInfoRec)
                             : Boolean;
BEGIN
   IF thePortInfoRec.name.portKindSelector = ppcByString THEN
      IF thePortInfoRec.name.portTypeStr = 'Example' THEN
         MyBrowserPortFilter := TRUE
      ELSE
         MyBrowserPortFilter := FALSE
   ELSE
      MyBrowserPortFilter := FALSE;
END;
```

The PPCBrowser function calls your filter function once for each port on the selected computer. Your function should return TRUE for each port you want to display in the program linking dialog box, and FALSE for each port that you do not want to display. Do not modify the data in the filter function parameters theLocationNameRec and thePortInfoRec.

The PPCBrowser function returns the selected port name in the parameter thePortInfo. The IPCListPorts function returns the port names in the area of memory pointed to by the bufferPtr field of the IPCListPorts parameter block. Both functions specify each port name in a port information record.

```
TYPE PortInfoRec =
   RECORD
      filler1:      SignedByte;        {space holder}
      authRequired: Boolean;           {authentication required}
      name:         PPCPortRec;        {port name}
   END;
```

If the authRequired field returns TRUE, the port requires authentication before a session can begin. You should use the StartSecureSession function to initiate a session with this port. If this field returns FALSE, you can use either the PPCStart function or the StartSecureSession function to initiate a session. See "Initiating a PPC Session" beginning on page 11-29 for detailed information. The name field of the port information record specifies an available port name.

Listing 11-4 illustrates how you use the PPCBrowser function to display the program linking dialog box in order to obtain the location and name of a port chosen by the user. In this listing, the PPCBrowser function builds lists of zones (shown in the AppleTalk Zones list of the program linking dialog box), objects (shown in the Macintoshes list), and ports (shown in the Programs list). In this example, the PPCBrowser function next tries to default to object "Moof™" in the "Twilight" zone. If it matches the object and zone, it also tries to default to the port "Inside Macintosh" with the port type "Example".

Note that the data in the records `LocationNameRec` and `PortInfoRec` is used to match the names in the program linking dialog box. The data has nothing to do with the NBP type used by `NBPLookup` or the filtered PPC ports that show up in the program linking dialog box. The `NBPLookup` function uses the NBP type supplied in `theLocNBPType`. The PPC port names are filtered using the `MyBrowserPortFilter` function shown in Listing 11-3 on page 11-24.

Listing 11-4   Browsing through dictionary service ports

```
FUNCTION MyPPCBrowser(VAR theLocationNameRec: LocationNameRec;
                      VAR thePortInfoRec: PortInfoRec): OSErr;
VAR
   prompt:            Str255;
   applListLabel:     Str255;
   defaultSpecified:  Boolean;
   theLocNBPType:     Str32;
BEGIN
   prompt := 'Choose an example to link to:';
   applListLabel := 'Examples';
   defaultSpecified := TRUE;
   WITH theLocationNameRec DO
   BEGIN
      locationKindSelector := ppcNBPLocation;
      WITH nbpEntity DO
      BEGIN
         objStr := 'Moof™';
         {typeStr is ignored}
         zoneStr := 'Twilight';
      END;
   END;
   WITH thePortInfoRec.name DO
   BEGIN
      {nameScript and name should be resources to allow easy }
      { localization}
      nameScript := smRoman;  {Roman script}
      name := 'Inside Macintosh';
      {the port type should always be hard-coded to allow the }
      { application to find ports of a particular type even }
      { after the name is localized}
      portKindSelector := ppcByString;
      portTypeStr := 'Example';
   END;
```

```
{when building the list of objects (Macintoshes), }
{ show only those with the NBP type "PPC Example"}
theLocNBPType := 'PPC Example';  {match this NBP type}
MyPPCBrowser := PPCBrowser(prompt, applListLabel,
                              defaultSpecified,
                              theLocationNameRec,
                              thePortInfoRec,
                              @MyBrowserPortFilter,
                              theLocNBPType);
END;
```

Obtaining a List of Available Ports

To generate a list of ports without displaying dialog boxes, you can use the
IPCListPorts function. The IPCListPorts function allows you to obtain a list of
ports on a particular computer within a particular zone. To obtain a list of ports, several
steps are required. First, use the GetZoneList function to obtain a list of zones. Next,
you must use the PLookupName function to obtain a list of computers with ports. After
establishing the zone and the computer, you can use the IPCListPorts function to
obtain the list of available ports. See *Inside Macintosh: Networking* for information on the
GetZoneList and PLookupName functions.

Listing 11-5 illustrates how you use the IPCListPorts function to obtain a list of ports
on a particular computer. This function returns a list of port information records in the
buffer pointed to by the parameter thePortInfoBufferPtr. The actual number of
port information records is returned in the parameter theActualCount.

Listing 11-5   Using the `IPCListPorts` function to obtain a list of ports

```
FUNCTION MyIPCListPorts
        (theStartIndex: Integer;
         theRequestCount: Integer; VAR theActualCount: Integer;
         theObjStr: Str32; theZoneStr: Str32;
         thePortInfoBufferPtr: PortInfoArrayPtr): OSErr;
VAR
   theIPCListPortsPBRec:       IPCListPortsPBRec;
   thePPCPortRec:              PPCPortRec;
   theLocationNameRec:         LocationNameRec;
BEGIN
   {list all PPC ports at the specified location}
   WITH thePPCPortRec DO
   BEGIN
      nameScript := smRoman;
      name := '=';            {match all names}
      portKindSelector := ppcByString;
      portTypeStr := '=';   {match all types}
   END;
   WITH theLocationNameRec DO
   BEGIN
      locationKindSelector := ppcNBPLocation;
      WITH nbpEntity DO
      BEGIN
         {set NBP object from the list returned by NBPLookup}
         objStr := theObjStr;
         {set NBP type, in this example to "PPC Example"; if you }
         { don't supply your own NBP type, use "PPCToolBox"}
         typeStr := 'PPC Example';
         {set NBP zone from the list returned by GetZoneList}
         zoneStr := theZoneStr;
      END;
   END;
   WITH theIPCListPortsPBRec DO
   BEGIN
      startIndex := theStartIndex;
      requestCount := theRequestCount;
      portName := @thePPCPortRec;
      locationName := @theLocationNameRec;
      bufferPtr := thePortInfoBufferPtr;
   END;
```

```
   MyIPCListPorts := IPCListPorts(@theIPCListPortsPBRec, FALSE);
   theActualCount := theIPCListPortsPBRec.actualCount;
END;
```

The `IPCListPorts` function returns information about ports that are on the computer specified in the `locationName` field of the list ports parameter block. If you set the `locationName` field to `NIL` or if you set the `locationKindSelector` field in the location name record to `ppcNoLocation`, the `IPCListPorts` function returns only the port names for the local computer.

The `bufferPtr` field points to an area of memory that contains the requested port names. You are responsible for allocating enough memory to hold the requested port names. The buffer length must be equal to

```
sizeof(PortInfoRec) * requestCount
```

Preparing for a Session

To communicate, you can open a port for your application and make it available to receive session requests, to initiate sessions, or both. Applications that are able to receive session requests can choose to accept or reject incoming session requests.

Before an application can accept and establish a session with another application, the PPC Toolbox authenticates the initiating user (unless guest access is enabled or the applications are located on the same computer). Once a session begins, the two applications can exchange data with each other.

Initiating a PPC Session

Once you have established the name and the location of the port that you want to communicate with, you can initiate a session. You can use either the `StartSecureSession` function or the `PPCStart` function to initiate a session. The `StartSecureSession` function displays several dialog boxes to identify each user who requests a session. You may prefer to use the `PPCStart` function for low-level code such as that used for drivers, which typically do not provide a user interface. You may also prefer to use `PPCStart` when the application you are initiating a session with does not require authentication. The `IPCListPorts` and `PPCBrowser` functions return information about whether a particular port requires authentication.

Note
Do not call the `StartSecureSession` function from an application that is running in the background, because the function displays several dialog boxes on the user's screen. u

The `StartSecureSession` function provides authentication services to identify each user who requests a session. This function combines the processes of prompting for user name and password and initiating a session into one synchronous procedure call. If authentication fails, the PPC Toolbox rejects the incoming session request.

```
err := StartSecureSession (pb, userName, useDefault, allowGuest,
                           guestSelected, prompt);
```

Set the `useDefault` parameter to `TRUE` if you want the `StartSecureSession` function to use the default user identity (described later in this section). If the default user identity cannot be authenticated, the `StartSecureSession` function displays a dialog box to allow a user to log on. Figure 11-14 shows the user identity dialog box.

Figure 11-14   The user identity dialog box
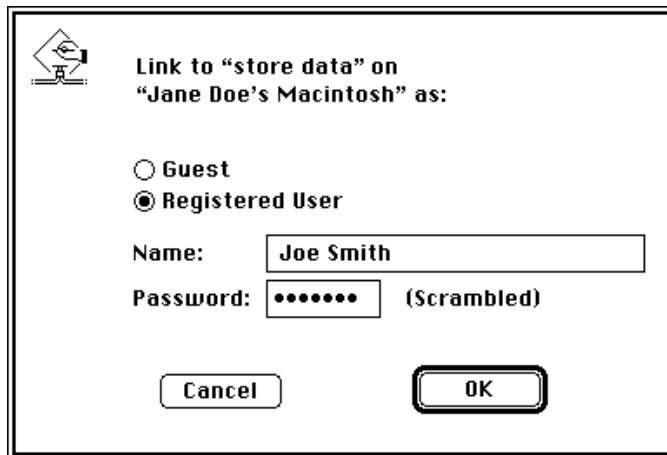
The `prompt` parameter of the `StartSecureSession` function allows you to specify a line of text that the dialog box can display. The `allowGuest` parameter specifies whether to enable the Guest radio button. If a port requires authentication, you should set this parameter to `FALSE`.

The `userName` parameter specifies the name of the user who is attempting to initiate a session. If the user name is not specified, the user identity dialog box appears on the user's screen with the owner name provided from the Sharing Setup control panel.

If the user enters an invalid password, the StartSecureSession function displays the dialog box shown in Figure 11-15.

Figure 11-15   The incorrect password dialog box

After the user clicks OK, the user identity dialog box reappears in the foreground so that the user can enter the password again.

If the user's name is invalid, the StartSecureSession function displays the dialog box shown in Figure 11-16.

Figure 11-16   The invalid user name dialog box

After the user clicks OK, the user identity dialog box reappears so that the user can enter a new user name.

The StartSecureSession function remains in this loop until a secure session is initiated or the user clicks Cancel in the user identity dialog box. If a secure session is initiated, StartSecureSession returns the user reference number in the corresponding field in the PPCStart parameter block. The user reference number represents the user name and password. A user reference number of 0 indicates that a session has been initiated with guest access. See "Setting Up Authenticated Sessions" beginning on page 11-6 for more information.

Before your application quits, you need to invalidate all user reference numbers obtained with the StartSecureSession function except for the default user reference number and the guest reference number (0). See "Invalidating Users" on page 11-44 for detailed information.

Listing 11-6 illustrates how to use the StartSecureSession function to establish an authenticated session. This listing shows only one session, although your application may conduct multiple sessions at one time.

Listing 11-6   Using the StartSecureSession function to establish a session

```
FUNCTION MyStartSecureSession(thePortInfoPtr: PortInfoPtr;
                              theLocationNamePtr: LocationNamePtr;
                              thePortRefNum: PPCPortRefNum;
                              VAR theSessRefNum: PPCSessRefNum;
                              VAR theUserRefNum: LongInt;
                              VAR theRejectInfo: LongInt;
                              VAR userName: Str32;
                              VAR guestSelected: Boolean): OSErr;
VAR
   thePPCStartPBRec: PPCStartPBRec;
   useDefault:       Boolean;
   allowGuest:       Boolean;
   err:              OSErr;
BEGIN
   WITH thePPCStartPBRec DO
   BEGIN
      ioCompletion := NIL;
      portRefNum := thePortRefNum;      {from the PPCOpen function}
      serviceType := ppcServiceRealTime;
      resFlag := 0;
      portName := @thePortInfoPtr^.name;  {from the PPCBrowser}
      locationName := theLocationNamePtr; {from the PPCBrowser}
      userData := 0;      {application-specific data that the }
                          { PPCInform function sees}
   END;
   {try to connect with default user identity}
   useDefault := TRUE;
   {highlight the Guest button appropriately}
   allowGuest := NOT thePortInfoPtr^.authRequired;
   err := StartSecureSession(@thePPCStartPBRec, userName,
                             useDefault, allowGuest,
                             guestSelected, stringPtr(NIL)^);
```

```
   IF err = noErr THEN
   BEGIN
      theSessRefNum := thePPCStartPBRec.sessRefNum;
      theUserRefNum := thePPCStartPBRec.userRefNum;
   END
   ELSE
      IF err = userRejectErr THEN
          {return rejectInfo from the PPCReject function}
          theRejectInfo := thePPCStartPBRec.rejectInfo;
   MyStartSecureSession := err;
END;
```

For low-level code such as that used for drivers (which typically do not provide a user interface), you can use the PPCStart function instead of the StartSecureSession function to initiate a session. You can also use the IPCListPorts function (instead of displaying the program linking dialog box) to obtain a list of ports.

If the authRequired field of the port information record contains FALSE, the port allows guest access. If the authRequired field of the port information record contains TRUE, use the PPCStart function and the user reference number obtained previously from the StartSecureSession function to reestablish an authenticated session.

You can also attempt to log on as the default user using the GetDefaultUser function to obtain the default user reference number and the default user name. The default user name is established after the owner starts up the computer.

```
err := GetDefaultUser (userRef, userName);
```

The userRef parameter is a reference number that represents the user name and password of the default user. The userName parameter contains the owner name that is specified in the Sharing Setup control panel.

The GetDefaultUser function returns an error when the default user identity does not exist (no name is specified in the Sharing Setup control panel) or the user is not currently logged on.

Listing 11-7 illustrates how you use the PPCStart function to initiate a session. The PPCStart function uses the port information record and the location name record to attempt to open a session with the selected PPC port.

Listing 11-7   Initiating a session using the `PPCStart` function

```pascal
FUNCTION MyPPCStart(thePortInfoPtr: PortInfoPtr;
                    theLocationNamePtr: LocationNamePtr;
                    thePortRefNum: PPCPortRefNum;
                    VAR theSessRefNum: PPCSessRefNum;
                    VAR theUserRefNum: LongInt;
                    VAR theRejectInfo: LongInt): OSErr;
VAR
   thePPCStartPBRec: PPCStartPBRec;
   userName:         Str32;
   err:              OSErr;
BEGIN
   WITH thePPCStartPBRec DO
   BEGIN
      ioCompletion := NIL;
      portRefNum := thePortRefNum;     {from the PPCOpen function}
      serviceType := ppcServiceRealTime;
      resFlag := 0;
      portName := @thePortInfoPtr^.name; {destination port}
      locationName := theLocationNamePtr; {destination location}
      userData := 0;     {application-specific data for PPCInform}
   END;
   err := GetDefaultUser(thePPCStartPBRec.userRefNum, userName);
   IF err <> noErr THEN
      thePPCStartPBRec.userRefNum := 0;
   IF thePortInfoPtr^.authRequired AND
      (thePPCStartPBRec.userRefNum = 0) THEN
      {port selected doesn't allow guests & you don't have a }
      { default user ref number so you can't log on to this port}
      err := authFailErr
   ELSE  {attempt to log on}
      err := PPCStart(@thePPCStartPBRec, FALSE);
   IF err = noErr THEN
   BEGIN
      theSessRefNum := thePPCStartPBRec.sessRefNum;
      theUserRefNum := thePPCStartPBRec.userRefNum;
   END
   ELSE
      IF err = userRejectErr THEN
         {return rejectInfo from the PPCReject function}
         theRejectInfo := thePPCStartPBRec.rejectInfo;
   MyPPCStart := err;
END;
```

The port to which you wish to connect must have an outstanding `PPCInform` function to successfully start a session. You cannot initiate a session with a port that is not able to receive session requests.

If the port is open, has an outstanding `PPCInform` function posted, and accepts your session request, the `PPCStart` function returns a `noErr` result code and a valid session reference number. This session reference number is used to identify the session during the exchange of data.

Receiving Session Requests

Your application can open as many ports as it requires as long as each port name is unique within a particular computer. A single port can support a number of communication sessions. To allow a port to receive session requests, use the `PPCInform` function. (Note that you must open a port to obtain a port reference number before calling the `PPCInform` function.) A port may have any number of outstanding `PPCInform` requests.

Listing 11-8 illustrates how you use the `PPCInform` function to allow a port to receive session requests. In this listing, the parameter `thePPCParamBlockPtr` points to a PPC parameter block record allocated by the application. The `portRefNum`, `autoAccept`, `portName`, `locationName`, `userName`, and `ioCompletion` parameters of the PPC parameter block record must be supplied. If you want to automatically accept all incoming session requests, you can set the `autoAccept` field in the `PPCInform` parameter block.

Listing 11-8   Using the PPCInform function to enable a port to receive sessions

```
FUNCTION MyPPCInform(thePPCParamBlockPtr: PPCParamBlockPtr;
                     thePPCPortPtr: PPCPortPtr;
                     theLocationNamePtr: LocationNamePtr;
                     theUserNamePtr: stringPtr;
                     thePortRefNum: PPCPortRefNum): OSErr;
BEGIN
   WITH thePPCParamBlockPtr^.informParam DO
   BEGIN
      ioCompletion := @MyInformCompProc;
      portRefNum := thePortRefNum;   {from the PPCOpen function}
      autoAccept := FALSE;           {the completion routine }
                                     { handles accepting or }
                                     { rejecting requests}
      portName := thePPCPortPtr;
      locationName := theLocationNamePtr;
      userName := theUserNamePtr;
   END;
   MyPPCInform := PPCInform(PPCInformPBPtr(thePPCParamBlockPtr),
                           TRUE);   {asynchronous}
END;
```

A PPC parameter block record is used instead of a PPCInform parameter block record
so that the same parameter block can be reused to make other PPC Toolbox calls from
the PPCInform completion routine. The parameter block and the records it points to
cannot be deallocated until all calls that use the parameter block and records have
completed.

You should make the call to PPCInform asynchronously. For each function that you use
asynchronously, you should provide a completion routine. The completion routine gets
called at interrupt time when the PPCInform function completes.

Listing 11-9 illustrates a completion routine for a `PPCInform` function. You can use the data passed into your `PPCInform` completion routine (user name, user data, port name, and location name) to determine whether to accept or reject the session request.

Listing 11-9 Completion routine for a `PPCInform` function

```
PROCEDURE MyInformCompProc(pb: PPCParamBlockPtr);
BEGIN
   IF pb^.informParam.ioResult = noErr THEN
   BEGIN
      {decide if this session should be accepted or rejected by }
      { looking at data supplied by the session requester}
      IF pb^.informParam.userData <> -1 THEN
         DoPPCAccept(pb)
      ELSE
         DoPPCReject(pb);
   END
   ELSE
      {use a global to tell the application that }
      { PPCParamBlockRec and the records it points to }
      { can be deallocated}
      gPBInUse := FALSE;
END;
```

When the `PPCInform` function completes, the `MyInformCompProc` procedure determines whether to accept or reject the incoming session request. It does this by calling `PPCAccept` or `PPCReject`, as described in the next section.

Accepting or Rejecting Session Requests

Use the `PPCAccept` function or the `PPCReject` function to accept or reject an incoming session request.

s **WARNING**

If the `PPCInform` function (with the `autoAccept` parameter set to `FALSE`) returns a `noErr` result code, you must call either the `PPCAccept` function or the `PPCReject` function. The computer trying to initiate a session (using the `StartSecureSession` function or the `PPCStart` function) waits (hangs) until the session attempt is either accepted or rejected, or until an error occurs. s

Listing 11-10 illustrates how you use the PPCAccept function to accept a session request. This listing reuses the parameter block used in the PPCInform function, so the sessRefNum field already contains the session reference number needed by the PPCAccept function.

Listing 11-10   Accepting a session request using the PPCAccept function

```
PROCEDURE DoPPCAccept(pb: PPCParamBlockPtr);
VAR
   err:  OSErr;
BEGIN {accept the session}
   pb^.acceptParam.ioCompletion := @MyAcceptCompProc;
   {the sessRefNum field is set by the PPCInform function}
   err := PPCAccept(@pb^.acceptParam, TRUE); {asynchronous}
END;
```

For each function that you use asynchronously, you should provide a completion routine. Listing 11-11 illustrates a completion routine for a PPCAccept function. This procedure gets called at interrupt time when the PPCAccept function completes. If there are no errors, it sets the global variable gSessionOpen to TRUE. The global variable gPBInUse is set to FALSE to inform the application that the parameter block and the records it points to are no longer in use.

You can use the session reference number in subsequent PPCWrite, PPCRead, and PPCEnd functions once a session is accepted.

Listing 11-11   Completion routine for a PPCAccept function

```
PROCEDURE MyAcceptCompProc(pb: PPCParamBlockPtr);
BEGIN
   IF pb^.acceptParam.ioResult = noErr THEN
      {accept completed so the session is completely open}
      gSessionOpen := TRUE;
   {use a global to tell the application that PPCParamBlockRec }
   { and the records it points to can be deallocated}
   gPBInUse := FALSE;
END;
```

Use the PPCReject function to reject an incoming session request. Listing 11-12 illustrates how you use the PPCReject function to reject a session request.

This listing reuses the parameter block used in the `PPCInform` function, so the `sessRefNum` field already contains the session reference number needed by the `PPCReject` function.

Listing 11-12   Rejecting a session request using the `PPCReject` function

```
PROCEDURE DoPPCReject(pb: PPCParamBlockPtr);
VAR
   err:  OSErr;
BEGIN {reject the session}
   WITH pb^.rejectParam DO
   BEGIN
      ioCompletion := @MyRejectCompProc;
      {the sessRefNum field is set by the PPCInform function}
      rejectInfo := -1;
   END;
   err := PPCReject(@pb^.rejectParam, TRUE); {asynchronous}
END;
```

Listing 11-13 illustrates a completion routine for a `PPCReject` function. This procedure is called at interrupt time when the `PPCReject` function completes. In this example, the global variable `gPBInUse` is set to `FALSE` to inform the application that the parameter block and the records it points to are no longer in use.

Listing 11-13   Completion routine for a `PPCReject` function

```
PROCEDURE MyRejectCompProc(pb: PPCParamBlockPtr);
BEGIN
   {use a global to tell the application that PPCParamBlockRec }
   { and the records it points to can be deallocated}
   gPBInUse := FALSE;
END;
```

Exchanging Data During a PPC Session

After a session begins, each application can send data to and receive data from the other using a sequence of message blocks. The PPC Toolbox treats each message block as a byte stream and does not interpret the contents of the message block. The size of a message block can be between 0 and ($2^{32}-1$) bytes. The PPC Toolbox treats the buffer size as an unsigned long integer.

The PPC Toolbox delivers the message blocks in the same sequence as they are sent and without duplicates. In Figure 11-17, an application transmits message blocks during a session.

Figure 11-17   Transmitting message blocks
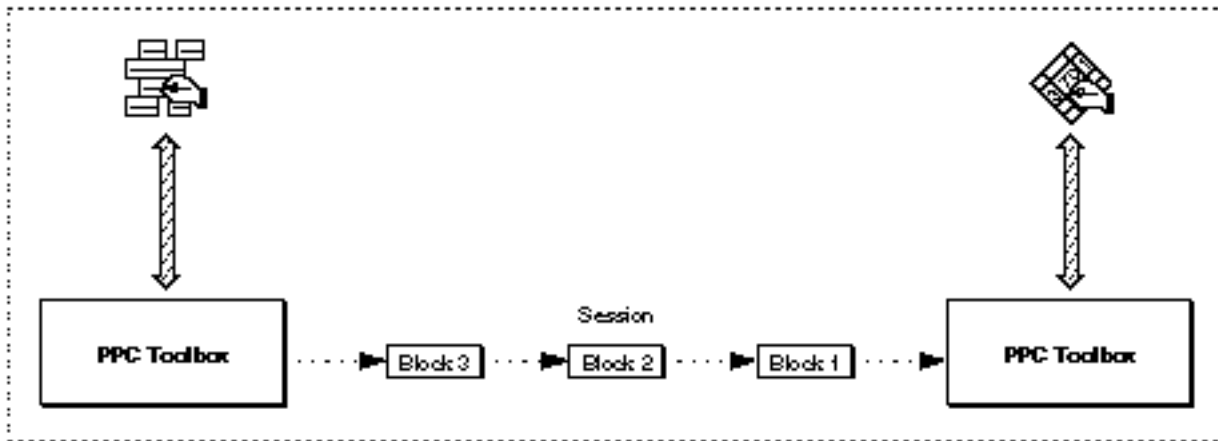
For each message block, you specify a block creator, block type, and user data. The first `PPCWrite` function that you use to create a new message block sets the attributes for the block. The `PPCRead` function returns the block creator, block type, and user data attributes for the current message block when the call completes.

Although the PPC Toolbox does not interpret these attributes, they can give the receiving application information about how to process the contents of the message block. For example, a database application may specify, in the block creator field, a counter to indicate the block number (block number 20 of 30 total blocks). This application could also specify a code, such as `'DREC'`, in the block type field to indicate that the information it contains is a database record. In addition, this application could specify, in the user data field, the length of the message block.

Reading Data From an Application

An application can both read from and write data to another application during a session. Use the `PPCRead` function during a session to read incoming blocks of data from another application.

Once a session is initiated, you should have a `PPCRead` function pending. You can issue a `PPCRead` function from inside a completion routine. This provides you with immediate notification if an error condition arises or the session closes.

The `blockCreator`, `blockType`, and `userData` fields are returned for the block you are reading. (These fields are set by the `PPCWrite` function.) To determine whether there is additional data to be read, check the `more` field. The value `FALSE` indicates the end of a message block.

Listing 11-14 illustrates how you use the `PPCRead` function to read data during a session.

Listing 11-14   Using the `PPCRead` function to read data during a session

```
FUNCTION MyPPCRead(thePPCReadPBPtr: PPCReadPBPtr;
                    theSessRefNum: PPCSessRefNum;
                    theBufferLength: Size;
                    theBufferPtr: Ptr): OSErr;
BEGIN
   WITH thePPCReadPBPtr^ DO
   BEGIN
      ioCompletion := NIL;
      sessRefNum := theSessRefNum;   {from PPCStart or PPCInform}
      bufferLength := theBufferLength;
      bufferPtr := theBufferPtr;
   END;
   MyPPCRead := PPCRead(thePPCReadPBPtr, TRUE); {asynchronous}
END;
```

You should make any calls to `PPCRead` asynchronously. You can provide a completion routine that will be called when the `PPCRead` function has completed, or you can poll the `ioResult` field of the PPC parameter block to determine whether the `PPCRead` function has completed. A `PPCRead` completion routine can issue another asynchronous PPC Toolbox call or set global variables. If another PPC Toolbox call is made from a completion routine, then the `PPCRead` function must use a record of data type `PPCParamBlockRec` instead of type `PPCReadPBRec`.

Listing 11-15 illustrates a function that can be used to poll the `ioResult` field of a record of data type `PPCReadPBRec`. The function returns `TRUE` when the `PPCRead` function associated with `PPCReadPBRec` has completed.

Listing 11-15   Polling the `ioResult` field to determine if a `PPCRead` function has completed

```
FUNCTION MyReadComplete(thePPCReadPBPtr: PPCReadPBPtr;
                         VAR err: OSErr): Boolean;
BEGIN
   err := thePPCReadPBPtr^.ioResult;
   MyReadComplete := err <> 1;
END;
```

Sending Data to an Application

Use the `PPCWrite` function to send a message block during a session specified by the session reference number.

You should call the `PPCWrite` function asynchronously. You can provide a completion routine that will be called when the `PPCWrite` function has completed, or you can poll the `ioResult` field of the PPC parameter block to determine whether the `PPCWrite` function has completed. A `PPCWrite` completion routine can issue another PPC Toolbox call or set global variables. If another PPC Toolbox call is made from a completion routine, then the `PPCWrite` function must use a record of data type `PPCParamBlockRec` instead of type `PPCWritePBRec`. Note that message blocks are sent in the order in which they are written.

Listing 11-16 illustrates how you use the `PPCWrite` function to write data during a session.

Listing 11-16   Using the `PPCWrite` function to write data during a session

```
FUNCTION MyPPCWrite(thePPCWritePBPtr: PPCWritePBPtr;
                    theSessRefNum: PPCSessRefNum;
                    theBufferLength: Size;
                    theBufferPtr: Ptr): OSErr;
BEGIN
   WITH thePPCWritePBPtr^ DO
   BEGIN
      ioCompletion := NIL;
      sessRefNum := theSessRefNum;  {from PPCStart or PPCInform}
      bufferLength := theBufferLength;
      bufferPtr := theBufferPtr;
      more := FALSE;           {no more data to read}
      userData := 0;           {application-specific data}
      blockCreator := '????'; {application-specific data}
      blockType := '????';     {application-specific data}
   END;
   MyPPCWrite := PPCWrite(thePPCWritePBPtr, TRUE); {asynchronous}
END;
```

The first `PPCWrite` function that you use to create a new message block sets the block creator, block type, and user data attributes for the block. These attributes are returned to the application when it reads from the message block. Set the `more` field to `FALSE` to indicate the end of the message block or set this field to `TRUE` if you want to append additional data to a message block.

Listing 11-17 illustrates a function that can be used to poll the `ioResult` field of a record of data type `PPCWritePBRec`. The function returns `TRUE` when the `PPCWrite` function associated with `PPCWritePBRec` has completed.

Listing 11-17   Polling the `ioResult` field to determine if a `PPCWrite` function has completed

```
FUNCTION MyWriteComplete(thePPCWritePBPtr: PPCWritePBPtr;
                         VAR err: OSErr): Boolean;
BEGIN
   err := thePPCWritePBPtr^.ioResult;
   MyWriteComplete := err <> 1;
END;
```

Ending a Session and Closing a Port

After data is written and read in, use the `PPCEnd` function to end the session (identified by the session reference number). You may receive an error if you use the `PPCEnd` function to end a session that has already been terminated.

Listing 11-18 illustrates how you use the `PPCEnd` function to end a session.

Listing 11-18   Ending a PPC session using the `PPCEnd` function

```
FUNCTION MyPPCEnd(theSessRefNum: PPCSessRefNum): OSErr;
VAR
   thePPCEndPBRec: PPCEndPBRec;
BEGIN
   thePPCEndPBRec.sessRefNum := theSessRefNum;
   MyPPCEnd := PPCEnd(@thePPCEndPBRec, FALSE);      {synchronous}
END;
```

The `PPCEnd` function causes all calls to the `PPCRead` and `PPCWrite` functions to complete (with a `sessClosedErr` result code) and invalidates the session reference number. The `PPCEnd` function also releases any PPC Toolbox resources so that they can be reused.

Use the `PPCClose` function to close the port specified by the port reference number. When you close a port, all sessions associated with a port are ended. Any active asynchronous calls associated with a session then call their completion routines (if they have one).

Listing 11-19 illustrates how you use the PPCClose function to close a port.

Listing 11-19   Closing a PPC port using the PPCClose function

```
FUNCTION MyPPCClose(thePortRefNum: PPCPortRefNum): OSErr;
VAR
    theClosePBRec: PPCClosePBRec;
BEGIN
    theClosePBRec.portRefNum := thePortRefNum;        {from PPCOpen}
    MyPPCClose := PPCClose(@theClosePBRec, FALSE);  {synchronous}
END;
```

In this example, the call to PPCClose is made synchronously.

Invalidating Users

It is your responsibility to invalidate all user reference numbers obtained with the StartSecureSession function before your application quits. However, while your application remains open, you may want to keep track of a user reference number to start a session with a port, end it, and then later start another session with the same port.

Use the DeleteUserIdentity function to invalidate the user reference number for a particular user.

```
err := DeleteUserIdentity (userRef);
```

The DeleteUserIdentity function removes a user by invalidating the specified user reference number. Note that you cannot invalidate the guest reference number (0) and, in most cases, you should not dispose of the default user reference number.

Listing 11-20 illustrates how you use the `DeleteUserIdentity` function to invalidate a user reference number obtained from a `StartSecureSession` function. The sample code does not invalidate the user reference number if it is either the default user reference number or the guest reference number (0).

Listing 11-20 Using the `DeleteUserIdentity` function to invalidate a user identity

```
FUNCTION MyDeleteNewUserRefNum(newUserRef: LongInt): OSErr;
VAR
   err:          OSErr;
   defUserRef:   LongInt;
   defUserName:  Str32;
BEGIN
   IF newUserRef <> 0 THEN
   BEGIN {user reference number passed was not the guest}
      err := GetDefaultUser(defUserRef, defUserName);
      IF err = noErr THEN
      BEGIN    {there is a default user}
         IF newUserRef <> defUserRef THEN
            {new user ref number isn't the default user ref num, }
            { so ok to delete}
            err := DeleteUserIdentity(newUserRef);
      END
      ELSE     {there is no default, so delete new user ref num}
         err := DeleteUserIdentity(newUserRef);
      MyDeleteNewUserRefNum := err;
   END
   ELSE {user reference number passed was the guest}
      MyDeleteNewUserRefNum := noErr;
END;
```

PPC Toolbox Reference

This section describes the data structures and routines that are specific to the PPC Toolbox. The section "PPC Toolbox Routines" beginning on page 11-50 describes PPC Toolbox routines. "Application-Defined Routines" beginning on page 11-77 describes completion routines and port filter functions.

Data Structures

This section describes the PPC parameter block, PPC port record, location name record, and port information record.

The PPC Toolbox Parameter Block

PPC Toolbox functions require a pointer to a PPC parameter block. You must fill out any fields of the parameter block that the specific PPC Toolbox function requires.

```
TYPE PPCParamBlockRec =
   RECORD
      CASE Integer OF
      0: (openParam:      PPCOpenPBRec);      {PPCOpen params}
      1: (informParam:    PPCInformPBRec);    {PPCInform params}
      2: (startParam:     PPCStartPBRec);     {PPCStart params}
      3: (acceptParam:    PPCAcceptPBRec);    {PPCAccept params}
      4: (rejectParam:    PPCRejectPBRec);    {PPCReject params}
      5: (writeParam:     PPCWritePBRec);     {PPCWrite params}
      6: (readParam:      PPCReadPBRec);      {PPCRead params}
      7: (endParam:       PPCEndPBRec);       {PPCEnd params}
      8: (closeParam:     PPCClosePBRec);     {PPCClose params}
      9: (listPortsParam: IPCListPortsPBRec); {IPCListPorts }
                                              { params}
      END;
```

Figure 11-18 on the next page shows the PPC Toolbox parameter blocks. Note that the reserved fields are not included in the illustration. The `qLink`, `csCode`, `intUse`, `intUsePtr`, and reserved fields are used internally by the PPC Toolbox. Your application should not rely on the PPC Toolbox to preserve these fields across calls.

Your application transfers ownership of the PPC Toolbox parameter block (and any buffers or records pointed to by the PPC Toolbox parameter block) to the PPC Toolbox until a PPC function is complete. Once the function completes, ownership of the parameter block (and any buffers or records it points to) is transferred back to your application. If a PPC Toolbox function is executed asynchronously, your program cannot alter memory that might be used by the PPC Toolbox until that function completes.

Program-to-Program Communications Toolbox

Figure 11-18   The PPC Toolbox parameter blocks
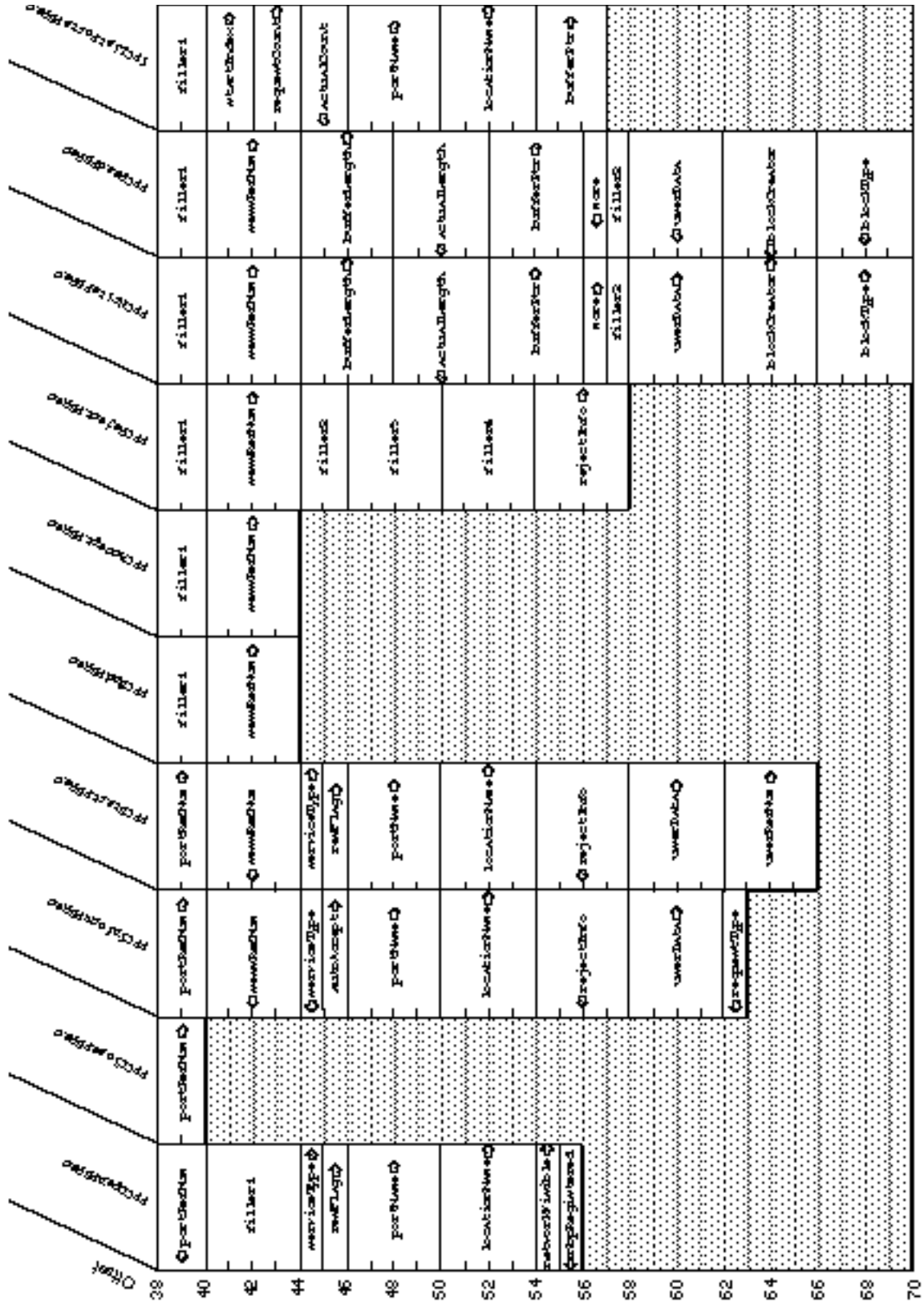
11-47

A PPC Toolbox function that is executed asynchronously must specify NIL or the address of a completion routine in the ioCompletion field of the PPC parameter block. The ioResult field should be used to determine the actual result code when an asynchronously executed PPC Toolbox function completes. If you specify a completion routine in the ioCompletion field, it is called at interrupt time when the PPC Toolbox function completes execution. See page 11-78 for the routine declaration for a completion routine.

The PPC Port Record

A PPC port name is defined by a PPC port record. The PPCPortRec data type defines the PPC port record.

```
TYPE PPCPortRec =
   RECORD
      nameScript:      ScriptCode;      {script identifier}
      name:            Str32;           {port name in program }
                                        { linking dialog box}
      portKindSelector: PPCPortKinds;   {general category of }
                                        { application}
      CASE PPCPortKinds OF
                      ppcByString:      (portTypeStr: Str32);
                      ppcByCreatorAndType:
                                        (portCreator: OSType;
                                        portType: OSType);
   END;
```

Field descriptions

nameScript	An integer script code.
name	A string that designates the application name.
portKindSelector	An integer that selects the kind of type string (either ppcByString or ppcByCreatorAndType).
portTypeStr	If the portKindSelector field specifies ppcByString, the portTypeStr field contains a 32-byte character string.
portCreator	If the portKindSelector field specifies ppcByCreatorAndType, the portCreator field contains a 4-character creator code.
portType	If the portKindSelector field specifies ppcByCreatorAndType, the portType field contains a 4-character type code.

To open a port, you need to specify a port name. As previously described, a port name consists of a script code, a name string, and a type string. For example, you can designate "smRoman" as the script code, "make memo" as the application's name string, and "word processor" as its type string.

The Location Name Record

A location name identifies the location of a computer on the network. A location name is specified in the standard Name-Binding Protocol (NBP) form, *<object string>*:PPCToolBox @*<AppleTalk zone>*. The object string is the name provided in the Sharing Setup control panel in the Control Panels folder. By default, the type string is "PPCToolBox". The AppleTalk zone is the zone to which the particular Macintosh computer belongs. For example, "Jane Doe's Macintosh:PPCToolBox@twilight" specifies the object string, type string, and AppleTalk zone for a particular computer.

The `LocationNameRec` data type defines the location name record. The `locationKindSelector` field can be set to `ppcNoLocation`, `ppcNBPLocation`, or `ppcNBPTypeLocation`.

```
TYPE LocationNameRec =
   RECORD
      locationKindSelector: PPCLocationKind;    {which variant}
      CASE PPCLocationKind OF
         {ppcNoLocation: storage not used by this value}
         ppcNBPLocation:
                        (nbpEntity: EntityName); {NBP name entity}
         ppcNBPTypeLocation:
                        (nbpType: Str32); {just the NBP type }
                                          { string for the }
                                          { PPCOpen function}
   END;
```

Field descriptions

`locationKindSelector`
                      An integer that determines how the location is specified. You can use either of the constants `ppcNBPLocation` or `ppcNBPTypeLocation`. (The PPC Toolbox uses the constant `ppcNoLocation` when the location received from or passed to a PPC Toolbox function is the location of the local machine.)

`nbpEntity`   If the `locationKindSelector` field specifies `ppcNBPLocation`, the `nbpEntity` field specifies a full NBP entity name.

`nbpType`   If the `locationKindSelector` field specifies `ppcNBPTypeLocation`, the `nbpType` field specifies an alias location name. This location kind is used only by the `PPCOpen` function when an alias location name is needed.

Note

You should assign an NBP value directly—do not pack it using `nbpSetEntity`. u

The Port Information Record

A port information record identifies whether a particular port requires authentication and specifies the port's port name. Both the `PPCBrowser` and `IPCListPorts` functions return information about ports using port information records. In addition, if you provide a port filter function, the PPC Toolbox provides information to your function about the current port in a port information record. The `PortInfoRec` data type defines a port information record.

```
TYPE PortInfoRec =
   RECORD
      filler1:      SignedByte;        {space holder}
      authRequired: Boolean;           {authentication required}
      name:         PPCPortRec;        {port name}
   END;
```

Field descriptions

filler1   Reserved.

authRequired   Specifies whether the port requires authentication. This field is
                TRUE if the port requires authentication before a session can begin.
                Otherwise, this field is FALSE.

name   Specifies an available port name.

For information on the `PPCBrowser` and `IPCListPorts` functions, see page 11-52 and page 11-54, respectively. For information on port filter functions, see page 11-78.

PPC Toolbox Routines

This section describes the routines for

n   initializing the PPC Toolbox

n   displaying the program linking dialog box

n   listing available ports

n   opening and closing a port

n   starting and ending a session

n   accepting and rejecting a session

n   reading and writing data

n   obtaining the default user reference number and name

n   invalidating a user reference number

Result codes appear after each function where applicable.

Initializing the PPC Toolbox

You use the PPCInit function to initialize the PPC Toolbox.

PPCInit

Use the PPCInit function to initialize the PPC Toolbox.

```
FUNCTION PPCInit: OSErr;
```

DESCRIPTION

After initialization, most PPC Toolbox routines can execute either synchronously or asynchronously.

Note that a noGlobalsErr result code indicates that the PPC Toolbox is not loaded properly.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the PPCInit function are

Trap macro	Selector
_PPCBrowser	$0000

The registers on entry and exit for this routine are

Registers on entry

D0   Selector code

Registers on exit

D0   Result code

RESULT CODES

noErr	0	No error
noGlobalsErr	–904	System unable to allocate memory, critical error

Using the Program Linking Dialog Box

You can use either the PPCBrowser function or the IPCListPorts function to locate a port to communicate with. Use the PPCBrowser function to display the program linking dialog box. For the description of IPCListPorts, see page 11-54.

PPCBrowser

Use the `PPCBrowser` function to display the program linking dialog box, which allows a user to select a port to communicate with.

```
FUNCTION PPCBrowser (prompt: Str255; applListLabel: Str255;
                     defaultSpecified: Boolean;
                     VAR theLocation: LocationNameRec;
                     VAR thePortInfo: PortInfoRec;
                     portFilter: PPCFilterProcPtr;
                     theLocNBPType: Str32): OSErr;
```

prompt   A line of text that the `PPCBrowser` function displays as a prompt in the program linking dialog box. If you specify `NIL` or an empty string is passed, the default prompt "Choose a program to link to:" is used.

applListLabel
                The title of the list of PPC ports. If you specify `NIL` or an empty string is passed, the default title "Programs" is used.

defaultSpecified
                A value that determines which port is initially selected in the program linking dialog box. If you specify `TRUE`, you must provide information in the parameters `theLocation` and `thePortInfo`. In this case, the `PPCBrowser` function tries to select the PPC port specified by the parameters `theLocation` and `thePortInfo` when the program linking dialog box first appears. If you specify `FALSE`, the `PPCBrowser` function selects the first port in the list and you can leave the location name record and the port information record (in the parameters `theLocation` and `thePortInfo`) uninitialized.

theLocation
                The port location. For information on this data structure, see "The Location Name Record" on page 11-49.

thePortInfo
                The port name. For information on this data structure, see "The Port Information Record" on page 11-50.

portFilter
                Determines how the list of PPC ports is filtered. If this parameter is `NIL`, the names of all existing PPC ports are displayed. If this parameter isn't `NIL`, it must be a pointer to a port filter function.

theLocNBPType
                The NBP type passed to `NBPLookup` to generate the list of computers. If you specify `NIL` or an empty string is passed, the default, "PPCToolBox", is used.

DESCRIPTION

The `PPCBrowser` function builds the list of ports and then displays the program linking dialog box.

If you set the `defaultSpecified` parameter to `TRUE`, the `PPCBrowser` function tries to select the PPC port specified by the parameters `theLocation` and `thePortInfo` when the program linking dialog box first appears. The `locationKindSelector` field in the location name record must be set to the `ppcNoLocation` constant (which specifies the local computer) or the `ppcNBPLocation` constant (which specifies the NBP object and NBP zone). The `ppcNBPTypeLocation` constant is not supported for matching. When matching the location, only the object string and the zone string of the entity name are used—the type string is ignored. When matching the port, the entire PPC port record (script, name, and port type) is used in the port information record. The `authRequired` field of the port information record is ignored.

The parameter `theLocNBPType` of the `PPCBrowser` function specifies the NBP type passed to `NBPLookup` to generate the list of computers. If you specify `NIL` or an empty string is passed, the default, "PPCToolBox", is used. Note that the current computer is always included in the list of computers (even if a location with the specified type does not exist for it). If the parameter `theLocNBPType` contains either of the NBP wildcard characters (= or), the `PPCBrowser` function returns a `paramErr` result code.

If the `PPCBrowser` function returns `noErr`, the parameters `theLocation` and `thePortInfo` specify the port chosen by the user. If the `PPCBrowser` function returns a `userCanceledErr` result code, the user clicked the Cancel button, and no port was selected. If the function returns a `memFullErr` result code, there was not enough memory to load the `PPCBrowser` package, and the dialog box did not appear.

Note

You must not call the `PPCBrowser` function from an application that is running in the background, since this function displays a dialog box on the user's screen. u

RESULT CODES

noErr	0	No error
paramErr	−50	Illegal parameter
memFullErr	−108	Not enough memory to load `PPCBrowser` package
userCanceledErr	−128	User decided not to conduct a session

SEE ALSO

For an example of the use of the `PPCBrowser` function, see Listing 11-4 on page 11-26. For an example of the program linking dialog box, see Figure 11-12 on page 11-22. For information on port filter functions, see page 11-78.

Obtaining a List of Ports

Use the `IPCListPorts` function to generate a list of existing ports without displaying a dialog box. The `IPCListPortsPBRec` data type defines the parameter block used by the `IPCListPorts` function.

IPCListPorts

Use the `IPCListPorts` function to generate a list of existing ports without displaying a dialog box.

```
FUNCTION IPCListPorts (pb: IPCListPortsPBPtr;
                       async: Boolean): OSErr;
```

pb   A pointer to an `IPCListPorts` parameter block.

async   A value that specifies whether the function is to be executed
            asynchronously (`TRUE`) or synchronously (`FALSE`).

Parameter block

ioCompletion	PPCCompProcPtr	**Address of a completion routine**
ioResult	OSErr	**Result code**
startIndex	Integer	**Index to the port entry list**
requestCount	Integer	**Number of port names requested**
actualCount	Integer	**Number of port names returned**
portName	PPCPortPtr	**Pointer to a** `PPCPortRec`
locationName	LocationNamePtr	**Pointer to a** `LocationNameRec`
bufferPtr	PortInfoArrayPtr	**Pointer to an array of** `PortInfoRec`

DESCRIPTION

If your application calls the `IPCListPorts` function asynchronously, you must specify in the `ioCompletion` field either the address of a completion routine or `NIL`. If you set `ioCompletion` to `NIL`, you should poll the `ioResult` field of the PPC parameter block (from your application's main event loop) to determine whether the PPC Toolbox has completed the requested operation. A value in the `ioResult` field other than 1 indicates that the call is complete. Note that it is unsafe to poll the `ioResult` field at interrupt time since the PPC Toolbox may be in the process of completing a call. See "PPC Toolbox Calling Conventions" beginning on page 11-14 for detailed information.

If you call the `IPCListPorts` function asynchronously, you must not change any of the fields in the parameter block until the call completes. The port name, location name, and buffer pointed to by `IPCListPortsPBRec` are owned by the PPC Toolbox until the call completes. These objects must not be deallocated or moved in memory while the call is in progress.

The startIndex field specifies the index to the list of ports on the remote machine from which the PPC Toolbox begins to get the list. In most cases, you'll want to start at the beginning, so set the startIndex field to 0. The requestCount field specifies the maximum number of port information records that can fit into your buffer.

The actualCount field returns the actual number of entries returned. Your program can use the IPCListPorts function repeatedly to obtain the entire list of ports. Ports that are not visible to the network are not included in the ports listing on a remote machine. (If you specify FALSE for the networkVisible field in the PPCOpen function, the port is not included in the listing of available ports across a network.)

The portName field must contain a pointer to a PPC port record that specifies which PPC ports to list. You can specify particular values in the PPC port record or you can use an equal sign (=) in the name or the portTypeStr fields as a wildcard to match all port names or port types.

The locationName field should contain a pointer to a location name record that designates the computer that contains the PPC ports you want returned. If the locationKindSelector field in the location name record is ppcNoLocation or if the locationName pointer is NIL, then the location is the local machine. If the locationKindSelector field in the location name record is ppcNBPLocation, then the location is a remote machine designated by the location name record's nbpEntity field.

The IPCListPorts function returns an array (list) of port information records in the area of memory pointed to by bufferPtr. Make sure that the buffer pointed to by the bufferPtr field is at least sizeof(PortInfoRec) * requestCount.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the IPCListPorts function are

Trap macro	Selector
_PPC	$000A

The registers on entry and exit for this routine are

Registers on entry

A0	Pointer to a parameter block
D0	Selector code

Registers on exit

D0	Result code

RESULT CODES

noErr	0	No error
notInitErr	−900	PPC Toolbox has not been initialized yet
nameTypeErr	−902	Invalid or inappropriate `locationKindSelector` in location name
noGlobalsErr	−904	System unable to allocate memory, critical error
localOnlyErr	−905	Network activity is currently disabled
sessTableErr	−907	PPC Toolbox is unable to create a session
noResponseErr	−915	Unable to contact application
badPortNameErr	−919	PPC port record is invalid
networkErr	−925	An error has occurred in the network
badLocNameErr	−931	Location name is invalid

SEE ALSO

Opening and Closing a Port

You open a port using the `PPCOpen` function and close a port using the `PPCClose` function.

PPCOpen

You open a port using the `PPCOpen` function.

```
FUNCTION PPCOpen (pb: PPCOpenPBPtr; async: Boolean): OSErr;
```

pb   A pointer to a `PPCOpen` parameter block.

async   A value that specifies whether the function is to be executed asynchronously (`TRUE`) or synchronously (`FALSE`).

Parameter block

ioCompletion	PPCCompProcPtr	Address of a completion routine
ioResult	OSErr	Result code
portRefNum	PPCPortRefNum	Port reference number of port opened
serviceType	PPCServiceType	Service type requested—must be `ppcServiceRealTime`
resFlag	SignedByte	Reserved field—must be 0
portName	PPCPortPtr	Pointer to a `PPCPortRec`
locationName	LocationNamePtr	Pointer to a `LocationNameRec`
networkVisible	Boolean	Make this port network visible
nbpRegistered	Boolean	Port location was registered on the network

DESCRIPTION

If your application calls the `PPCOpen` function asynchronously, you must specify in the `ioCompletion` field either the address of a completion routine or `NIL`. If you set `ioCompletion` to `NIL`, you should poll the `ioResult` field of the PPC parameter block (from your application's main event loop) to determine whether the PPC Toolbox has completed the requested operation. A value in the `ioResult` field other than 1 indicates that the call is complete. Note that it is unsafe to poll the `ioResult` field at interrupt time since the PPC Toolbox may be in the process of completing a call. See "PPC Toolbox Calling Conventions" beginning on page 11-14 for detailed information.

If you call the `PPCOpen` function asynchronously, you must not change any of the fields in the parameter block until the call completes. The port name and location name pointed to by the `PPCOpen` parameter block record are owned by the PPC Toolbox until the call completes. These objects must not be deallocated or moved in memory while the call is in progress.

The `portRefNum` field returns the PPC port identifier. Use this port reference number to initiate a session for this particular port. Set the `serviceType` field to indicate that this port accepts sessions in real time. For System 7, this field must always be set to the `ppcServiceRealTime` constant. You must set the `resFlag` field to 0.

The `portName` field must contain a pointer to a PPC port record that specifies the name of the PPC port to be opened.

The `locationName` field should contain a pointer to a location name record that designates the location of the PPC port to be opened. If the `locationName` pointer is `NIL`, then the default name PPC Toolbox is used. If a location name record is used, then the `locationKindSelector` field in the location name record must be `ppcNBPTypeLocation`, and an alias location name specified by the location name record's `nbpType` field is used.

The `networkVisible` field indicates whether the port should be made visible (for browsing as well as incoming network requests). If you specify `FALSE`, this port is not visible in the listing of available ports across a network (although it is still included within the local machine's listing of available ports).

The `nbpRegistered` field returns `TRUE` if the location name specified was registered on the network.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
_PPC	$0001

The registers on entry and exit for this routine are

Registers on entry

A0   Pointer to a parameter block

D0   Selector (1)

Registers on exit

D0   Result code

RESULT CODES

noErr	0	No error
notInitErr	–900	PPC Toolbox has not been initialized yet
nameTypeErr	–902	Invalid or inappropriate locationKindSelector in location name
noPortErr	–903	Unable to open port or bad port reference number
noGlobalsErr	–904	System unable to allocate memory, critical error
badReqErr	–909	Bad parameter or invalid state for this operation
portNameExistsErr	–910	Another port is already open with this name
badPortNameErr	–919	PPC port record is invalid
badServiceMethodErr	–930	Service method is other than ppcServiceRealTime
badLocNameErr	–931	Location name is invalid
nbpDuplicateName	–1027	Location name represents a duplicate on this computer

SEE ALSO

For an example of the use of the PPCOpen function, see Listing 11-2 on page 11-21.

PPCClose

You use the PPCClose function to close the port specified by the port reference number.

```
FUNCTION PPCClose (pb: PPCClosePBPtr; async: Boolean): OSErr;
```

pb   A pointer to a PPCClose parameter block.

async   A value that specifies whether the function is to be executed
            asynchronously (TRUE) or synchronously (FALSE).

Parameter block

ioCompletion	PPCCompProcPtr	Address of a completion routine
ioResult	OSErr	Result code
portRefNum	PPCPortRefNum	Port reference number of port to close

DESCRIPTION

If your application calls this function asynchronously, you must specify in the `ioCompletion` field either the address of a completion routine or `NIL`. If you set `ioCompletion` to `NIL`, you should poll the `ioResult` field of the PPC parameter block (from your application's main event loop) to determine whether the PPC Toolbox has completed the requested operation. A value in the `ioResult` field other than 1 indicates that the call is complete. Note that it is unsafe to poll the `ioResult` field at interrupt time since the PPC Toolbox may be in the process of completing a call. See "PPC Toolbox Calling Conventions" beginning on page 11-14 for detailed information.

The `portRefNum` field specifies the PPC port identifier of the port to close. The port reference number must be a valid port reference number returned from a previous call to the `PPCOpen` function.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
`_PPC`	$0009

The registers on entry and exit for this routine are

Registers on entry

A0	Pointer to a parameter block
D0	Selector code

Registers on exit

D0	Result code

RESULT CODES

`noErr`	0	No error
`notInitErr`	−900	PPC Toolbox has not been initialized yet
`noPortErr`	−903	Bad port reference number
`noGlobalsErr`	−904	System unable to allocate memory, critical error

SEE ALSO

For an example of the use of the `PPCClose` function, see Listing 11-19 on page 11-44.

Starting and Ending a Session

You use the `PPCStart` or `StartSecureSession` function to initiate a session with another port, and you use the `PPCEnd` function to end a session.

PPCStart

The `PPCStart` function initiates a session with the destination port specified in the name and location fields.

```
FUNCTION PPCStart (pb: PPCStartPBPtr; async: Boolean): OSErr;
```

pb	A pointer to a `PPCStart` parameter block.
async	A value that specifies whether the function is to be executed asynchronously (`TRUE`) or synchronously (`FALSE`).

Parameter block

ioCompletion	PPCCompProcPtr	Address of a completion routine
ioResult	OSErr	Result code
portRefNum	PPCPortRefNum	Port reference number of this session
sessRefNum	PPCSessRefNum	Session reference number of this session
serviceType	PPCServiceType	Service type requested—must be `ppcServiceRealTime`
resFlag	SignedByte	Reserved field—must be 0
portName	PPCPortPtr	Pointer to a `PPCPortRec`
locationName	LocationNamePtr	Pointer to a `LocationNameRec`
rejectInfo	LongInt	Value from `PPCReject` if session was rejected
userData	LongInt	Application-specific data
userRefNum	LongInt	User reference number

DESCRIPTION

If your application calls the `PPCStart` function asynchronously, you must specify in the `ioCompletion` field either the address of a completion routine or `NIL`. If you set `ioCompletion` to `NIL`, you should poll the `ioResult` field of the PPC parameter block (from your application's main event loop) to determine whether the PPC Toolbox has completed the requested operation. A value in the `ioResult` field other than 1 indicates that the call is complete. Note that it is unsafe to poll the `ioResult` field at interrupt time, since the PPC Toolbox may be in the process of completing a call. See "PPC Toolbox Calling Conventions" beginning on page 11-14 for detailed information.

If you call the `PPCStart` function asynchronously, you must not change any of the fields in the parameter block until the call completes. The port name and location name pointed to by the `PPCStart` parameter block record are owned by the PPC Toolbox until the call completes. These objects must not be deallocated or moved in memory while the call is in progress.

You specify the PPC port identifier in the portRefNum field. The port reference number is a reference number for the port through which you are requesting a session. The value you specify must correspond to the port reference number returned from the PPCOpen function.

The sessRefNum field returns a session identifier. This number, which is provided by the PPC Toolbox, is used while data is being exchanged to identify a particular session. You must set the serviceType field to indicate that the session is to be connected in real time. For System 7, this field must always be set to the ppcServiceRealTime constant. You must set the resFlag field to 0.

The portName field must contain a pointer to a PPC port record. The locationName field should contain a pointer to a location name record or NIL. The PPC port record and the location name record specify the name and location of the PPC port to initiate a session with, and they are usually obtained from the PPCBrowser function. If the locationKindSelector field in the location name record is ppcNoLocation or if the locationName pointer is NIL, then the location is the local machine. If the locationKindSelector field in the location name record is ppcNBPLocation, then the location is a remote machine designated by the location name record's nbpEntity field.

If the ioResult field of the PPC parameter block returns a userRejectErr result code, the rejectInfo field contains the same value as the rejectInfo field in the PPCReject parameter block. The rejectInfo field is defined by your application.

The initiating port can specify any information in the userData field. The PPCInform function reports this data to the responding port upon its completion.

The userRefNum field specifies an authenticated user. The authentication mechanism of the PPC Toolbox identifies each user through an assigned name and a password. A user reference number of 0 indicates that you want to specify a guest.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the PPCStart function are

Trap macro	Selector
_PPC	$0002

The registers on entry and exit for this routine are

Registers on entry

A0	Pointer to a parameter block
D0	Selector code

Registers on exit

D0	Result code

RESULT CODES

noErr	0	No error
notInitErr	–900	PPC Toolbox has not been initialized yet
nameTypeErr	–902	locationKindSelector is not ppcNBPLocation or ppcNoLocation
noPortErr	–903	Bad port reference number
noGlobalsErr	–904	System unable to allocate memory, critical error
localOnlyErr	–905	Network activity is currently disabled
destPortErr	–906	Port does not exist at destination
sessTableErr	–907	PPC Toolbox is unable to create a session
noUserNameErr	–911	User name unknown on destination machine
userRejectErr	–912	Destination rejected the session request
noResponseErr	–915	Unable to contact application
portClosedErr	–916	The port was closed
badPortNameErr	–919	PPC port record is invalid
networkErr	–925	An error has occurred in the network
noInformErr	–926	PPCStart failed because target application did not have an inform pending
authFailErr	–927	User's password is wrong
noUserRecErr	–928	Invalid user reference number
badServiceMethodErr	–930	Service method is other than ppcServiceRealTime
guestNotAllowedErr	–932	Destination port requires authentication

SEE ALSO

For an example of the use of the PPCStart function, see Listing 11-7 on page 11-34.

StartSecureSession

The StartSecureSession function prompts for user name and password and calls PPCStart—all in one synchronous procedure call. Use the StartSecureSession function whenever a port destination requires authentication.

```
FUNCTION StartSecureSession (pb: PPCStartPBPtr;
                             VAR userName: Str32;
                             useDefault: Boolean;
                             allowGuest: Boolean;
                             VAR guestSelected: Boolean;
                             prompt: Str255): OSErr;
```

pb   A pointer to a PPCStart parameter block.

userName   A pointer to a 32-byte character string to be displayed as the user's name.

useDefault

            A Boolean value that indicates whether you want the StartSecureSession function to use the default user identity (and possibly prevent the user identity dialog box from appearing). If so, specify TRUE; otherwise, specify FALSE.

allowGuest

>A Boolean value that determines whether the Guest radio button in the user identity dialog box is active (TRUE) or inactive (FALSE).

guestSelected

>Returns TRUE if the user has logged on as a guest.

prompt   A line of text that the dialog box displays in place of the default prompt. Specify NIL or an empty string to use the default prompt.

DESCRIPTION

Your program fills out a parameter block just as though it were calling the PPCStart function. You specify all input fields in the parameter block except for the userRefNum field. The userRefNum field is returned when the StartSecureSession function successfully completes.

The userName parameter is a pointer to a 32-byte character string to be displayed as the user's name. If the Pascal string length is 0, the default user name is used. The default user name is the name specified in the Sharing Setup control panel. The default user name is returned in the userName buffer.

Set the useDefault parameter to TRUE if you want the StartSecureSession function to use the default user identity (and possibly prevent the user identity dialog box from appearing). The allowGuest parameter specifies whether the Guest radio button in the user identity dialog box is active. You usually set it to the inverse of the authRequired field in the port information record. For example, if authRequired is TRUE, then allowGuest should be set to FALSE.

The guestSelected parameter returns TRUE if the user has logged on as a guest. The prompt parameter of the StartSecureSession function allows you to specify a line of text that the dialog box can display. Specify NIL or an empty string for the prompt parameter to enable the PPC Toolbox to use the default prompt. The PPC Toolbox uses the default string "Link to <*port name*> on <*object string*> as:". The port name is obtained from the name string of the port name, and the object string is obtained from the object string of the location name.

Note

Do not call the StartSecureSession function from an application that is running in the background, because the function displays several dialog boxes on the user's screen. u

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the StartSecureSession function are

Trap macro	Selector
_PPC	$000E

The registers on entry and exit for this routine are

Registers on entry

A0   Pointer to a `StartSecureParams` record

D0   Selector code

Registers on exit

D0   Result code

RESULT CODES

noErr	0	No error
userCanceledErr	−128	User decided not to conduct a session
notInitErr	−900	PPC Toolbox has not been initialized yet
nameTypeErr	−902	`locationKindSelector` is not `ppcNBPLocation` or `ppcNoLocation`
noPortErr	−903	Bad port reference number
noGlobalsErr	−904	System unable to allocate memory, critical error
localOnlyErr	−905	Network activity is currently disabled
destPortErr	−906	Port does not exist at destination
sessTableErr	−907	PPC Toolbox is unable to create a session
noResponseErr	−915	Unable to contact application
portClosedErr	−916	The port was closed
badPortNameErr	−919	PPC port record is invalid
noUserRefErr	−924	Unable to create a new user reference number
networkErr	−925	An error has occurred in the network
noInformErr	−926	`PPCStart` failed because target application did not have an inform pending
badServiceMethodErr	−930	Service method is other than `ppcServiceRealTime`
guestNotAllowedErr	−932	Destination port requires authentication

SEE ALSO

For an example of the use of the `StartSecureSession` function, see "Initiating a PPC Session" beginning on page 11-29.

PPCEnd

Use the `PPCEnd` function to end a session. This function completes all outstanding asynchronous calls associated with the session reference number.

```
FUNCTION PPCEnd (pb: PPCEndPBPtr; async: Boolean): OSErr;
```

pb   A pointer to a `PPCEnd` parameter block.

async   A value that specifies whether the function is to be executed asynchronously (`TRUE`) or synchronously (`FALSE`).

Parameter block

ioCompletion	PPCCompProcPtr	Address of a completion routine
ioResult	OSErr	Result code
sessRefNum	PPCSessRefNum	Session reference number of session to end

DESCRIPTION

If your application calls the PPCEnd function asynchronously, you must specify in the ioCompletion field either the address of a completion routine or NIL. If you set ioCompletion to NIL, you should poll the ioResult field of the PPC parameter block (from your application's main event loop) to determine whether the PPC Toolbox has completed the requested operation. A value in the ioResult field other than 1 indicates that the call is complete. Note that it is unsafe to poll the ioResult field at interrupt time since the PPC Toolbox may be in the process of completing a call. See "PPC Toolbox Calling Conventions" beginning on page 11-14 for detailed information.

You provide a session identifier in the sessRefNum field to identify the session that you are terminating. The PPCStart, StartSecureSession, or PPCInform function returns the session reference number.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the PPCEnd function are

Trap macro	Selector
_PPC	$0008

The registers on entry and exit for this routine are

Registers on entry

A0	Pointer to a parameter block
D0	Selector code

Registers on exit

D0	Result code

RESULT CODES

noErr	0	No error
notInitErr	−900	PPC Toolbox has not been initialized yet
noGlobalsErr	−904	System unable to allocate memory, critical error
noSessionErr	−908	Invalid session reference number

SEE ALSO

For an example of the use of the PPCEnd function, see Listing 11-18 on page 11-43.

Receiving, Accepting, and Rejecting a Session

You use the `PPCInform` function to receive session requests. After the `PPCInform` function completes (with the `autoAccept` field set to `FALSE`), you must accept or reject the session request using the `PPCAccept` or `PPCReject` function.

PPCInform

As long as a port has been opened, you can call the `PPCInform` function at any time. You can have any number of outstanding `PPCInform` functions.

```
FUNCTION PPCInform (pb: PPCInformPBPtr; async: Boolean): OSErr;
```

pb   A pointer to a `PPCInform` parameter block.

async   A value that specifies whether the function is to be executed
                asynchronously (`TRUE`) or synchronously (`FALSE`). You should execute
                the `PPCInform` function asynchronously.

Parameter block

ioCompletion	PPCCompProcPtr	Address of a completion routine
ioResult	OSErr	Result code
portRefNum	PPCPortRefNum	Port reference number of this session
sessRefNum	PPCSessRefNum	Session reference number of this session
serviceType	PPCServiceType	Service type of this session
autoAccept	Boolean	If `TRUE`, session is accepted automatically
portName	PPCPortPtr	Pointer to `PPCPortRec`, may be `NIL`
locationName	LocationNamePtr	Pointer to `LocationNameRec`, may be `NIL`
userName	StringPtr	Pointer to `Str32`, may be `NIL`
userData	LongInt	Application-specific data
requestType	PPCSessionOrigin	Network or local request

DESCRIPTION

If your application calls the `PPCInform` function asynchronously, you must specify in the `ioCompletion` field either the address of a completion routine or `NIL`. If you set `ioCompletion` to `NIL`, you should poll the `ioResult` field of the PPC parameter block (from your application's main event loop) to determine whether the PPC Toolbox has completed the requested operation. A value in the `ioResult` field other than 1 indicates that the call is complete. Note that it is unsafe to poll the `ioResult` field at interrupt time since the PPC Toolbox may be in the process of completing a call. See "PPC Toolbox Calling Conventions" beginning on page 11-14 for detailed information.

If you call the `PPCInform` function asynchronously, you must not change any of the fields in the parameter block until the call completes. The port name, location name, user name, and buffer pointed to by the record of type `PPCInformPBRec` are owned by the PPC Toolbox until the call completes. These objects must not be deallocated or moved in memory while the call is in progress.

You provide the PPC port identifier in the `portRefNum` field. A `PPCOpen` function returns the port identifier. The `sessRefNum` field returns a session identifier.

The `serviceType` field indicates the service type. For system software version 7.0, this field always returns the `ppcServiceRealTime` constant.

If you set the `autoAccept` field to `TRUE`, session requests are automatically accepted as they are received. When the `PPCInform` function completes execution with a `noErr` result code and you set the `autoAccept` field to `FALSE`, you need to accept or reject the session.

s **WARNING**

If the `PPCInform` function (with the `autoAccept` parameter set to `FALSE`) returns a `noErr` result code, you must call either the `PPCAccept` function or the `PPCReject` function. The computer trying to initiate a session using the `StartSecureSession` function or the `PPCStart` function waits (hangs) until the session attempt is either accepted or rejected, or until an error occurs. s

The `portName` field must contain `NIL` or a pointer to a PPC port record. If the `portName` field contains `NIL`, then the name of the PPC port that initiated the session is not returned. If the `portName` field points to a PPC port record, then the PPC port record is filled with the name of the PPC port that initiated the session when the `PPCInform` function completes.

The `locationName` field must contain `NIL` or a pointer to a location name record. If the `locationName` field contains `NIL`, then the location of the PPC port that initiated the session is not returned. If the `locationName` field points to a location name record, then the location name record is filled with the location of the PPC port that initiated the session when the `PPCInform` function completes. If the `locationKindSelector` field of the location name record returned is `ppcNoLocation`, then the location is the local machine. If the `locationKindSelector` field of the location name record returned is `ppcNBPLocation`, then the location is a remote machine designated by the location name record's `nbpEntity` field.

The `userName` field must contain `NIL` or a pointer to a 32-byte character string. If the `userName` field contains `NIL`, then the user name string is not returned. If the `userName` field points to a 32-byte character string, then the 32-byte character string is filled with the name of the user making the session request (if authenticated) when the `PPCInform` function completes.

When the `PPCInform` function completes, the `userData` field contains the user data provided by the application making the session request. This field is transparent to the PPC Toolbox. The application can send any data in this field.

When the `PPCInform` function completes, the `requestType` field contains either `ppcRemoteOrigin` or `ppcLocalOrigin`, depending on whether the session request is initiated by a computer across the network or by a port on the same computer.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
_PPC	$0003

The registers on entry and exit for this routine are

Registers on entry

A0   Pointer to a parameter block

D0   Selector code

Registers on exit

D0   Result code

RESULT CODES

noErr	0	No error
notInitErr	–900	PPC Toolbox has not been initialized yet
noPortErr	–903	Unable to open port or bad port reference number
noGlobalsErr	–904	System unable to allocate memory, critical error
portClosedErr	–916	The port was closed

SEE ALSO

For an example of the use of the `PPCInform` function, see Listing 11-8 on page 11-36.

PPCAccept

Use the `PPCAccept` function to indicate that an application is willing to accept an incoming session request after a `PPCInform` function completes.

```
FUNCTION PPCAccept (pb: PPCAcceptPBPtr; async: Boolean): OSErr;
```

pb   A pointer to a `PPCAccept` parameter block.

async   A value that specifies whether the function is to be executed asynchronously (TRUE) or synchronously (FALSE).

Parameter block

ioCompletion	PPCCompProcPtr	Address of a completion routine
ioResult	OSErr	Result code
sessRefNum	PPCSessRefNum	Session reference number of session to accept

DESCRIPTION

If your application calls the PPCAccept function asynchronously, you must specify in the ioCompletion field either the address of a completion routine or NIL. If you set ioCompletion to NIL, you should poll the ioResult field of the PPC parameter block (from your application's main event loop) to determine whether the PPC Toolbox has completed the requested operation. A value in the ioResult field other than 1 indicates that the call is complete. Note that it is unsafe to poll the ioResult field at interrupt time since the PPC Toolbox may be in the process of completing a call. See "PPC Toolbox Calling Conventions" beginning on page 11-14 for detailed information.

The sessRefNum field specifies a session identifier. Use the session reference number returned from the completed PPCInform parameter block to accept the session request.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the PPCAccept function are

**Trap
macro   Selector**

_PPC $0004

The registers on entry and exit for this routine are

Registers on entry

A0   Pointer to a parameter block

D0   Selector code

Registers on exit

D0   Result code

RESULT CODES

noErr	0	No error
notInitErr	−900	PPC Toolbox has not been initialized yet
noGlobalsErr	−904	System unable to allocate memory, critical error
noSessionErr	−908	Invalid session reference number
badReqErr	−909	Bad parameter or invalid state for this operation

SEE ALSO

For an example of the use of the PPCAccept function, see "Accepting or Rejecting Session Requests" beginning on page 11-37.

PPCReject

Use the PPCReject function to reject a session request after a PPCInform function completes.

```
FUNCTION PPCReject (pb: PPCRejectPBPtr; async: Boolean): OSErr;
```

pb   A pointer to a PPCReject parameter block.

async   A value that specifies whether the function is to be executed asynchronously (TRUE) or synchronously (FALSE).

Parameter block

ioCompletion	PPCCompProcPtr	Address of a completion routine
ioResult	OSErr	Result code
sessRefNum	PPCSessRefNum	Session reference number of session to reject
rejectInfo	LongInt	Value to return if session is rejected

DESCRIPTION

If your application calls the PPCReject function asynchronously, you must specify in the ioCompletion field either the address of a completion routine or NIL. If you set ioCompletion to NIL, you should poll the ioResult field of the PPC parameter block (from your application's main event loop) to determine whether the PPC Toolbox has completed the requested operation. A value in the ioResult field other than 1 indicates that the call is complete. Note that it is unsafe to poll the ioResult field at interrupt time since the PPC Toolbox may be in the process of completing a call. See "PPC Toolbox Calling Conventions" beginning on page 11-14 for detailed information.

The sessRefNum field specifies a session to be rejected. This must be a valid session reference number returned from a previous PPCInform function. The rejectInfo field is an optional field. The application receiving a session request may specify any data in this field. The initiating application receives this information in the PPCStart parameter block.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the PPCReject function are

Trap macro   **Selector**

_PPC $0005

The registers on entry and exit for this routine are

Registers on entry

A0   Pointer to a parameter block

D0   Selector code

Registers on exit

D0   Result code

RESULT CODES

noErr	0	No error
notInitErr	–900	PPC Toolbox has not been initialized yet
noGlobalsErr	–904	System unable to allocate memory, critical error
noSessionErr	–908	Invalid session reference number
badReqErr	–909	Bad parameter or invalid state for this operation

SEE ALSO

For an example of the use of the PPCReject function, see page 11-39.

Reading and Writing Data

The PPCRead function reads incoming data from an application, and the PPCWrite function writes data to an application during a session.

PPCRead

Use the PPCRead function to read message blocks during a session.

```
FUNCTION PPCRead (pb: PPCReadPBPtr; async: Boolean): OSErr;
```

pb   A pointer to a PPCRead parameter block.

async   A value that specifies whether the function is to be executed
            asynchronously (TRUE) or synchronously (FALSE). You should execute
            the PPCRead function asynchronously.

Parameter block

ioCompletion	PPCCompProcPtr	Address of a completion routine
ioResult	OSErr	Result code
sessRefNum	PPCSessRefNum	Session reference number
bufferLength	Size	Length of data buffer
actualLength	Size	Actual length of data read
bufferPtr	Ptr	Pointer to data buffer
more	Boolean	TRUE if more data in this block to be read
userData	LongInt	Application-specific data
blockCreator	OSType	Creator of block read
blockType	OSType	Type of block read

DESCRIPTION

If your application calls the `PPCRead` function asynchronously, you must specify in the `ioCompletion` field either the address of a completion routine or `NIL`. If you set `ioCompletion` to `NIL`, you should poll the `ioResult` field of the PPC parameter block (from your application's main event loop) to determine whether the PPC Toolbox has completed the requested operation. A value in the `ioResult` field other than 1 indicates that the call is complete. Note that it is unsafe to poll the `ioResult` field at interrupt time since the PPC Toolbox may be in the process of completing a call. See "PPC Toolbox Calling Conventions" beginning on page 11-14 for detailed information.

If you call the `PPCRead` function asynchronously, you must not change any of the fields in the parameter block until the call completes. The buffer pointed to by the record of data type `PPCReadPBRec` is owned by the PPC Toolbox until the call completes. These objects must not be deallocated or moved in memory while the call is in progress.

The `sessRefNum` field specifies a session to read data from. This must be a valid session reference number returned from a previous `PPCStart`, `StartSecureSession`, or `PPCInform` function. The `bufferLength` and `bufferPtr` fields specify the length and location of a buffer the message block will be read into. Your application must allocate the storage for the buffer. The `actualLength` field returns the actual size of the data read into your data buffer.

The `more` field returns `TRUE` if the provided buffer cannot hold the remainder of the message block. Your application may read a message block in several pieces. It is not necessary to have a buffer large enough to read in the entire message block, so a message block can span multiple calls to the `PPCRead` function.

Upon completion of the `PPCRead` function, the `userData`, `blockCreator`, and `blockType` fields contain information regarding the contents of the message block. You specify these fields using the `PPCWrite` function.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
`_PPC`	$0007

The registers on entry and exit for this routine are

Registers on entry

A0	Pointer to a parameter block
D0	Selector code

Registers on exit

D0	Result code

RESULT CODES

noErr	0	No error
notInitErr	–900	PPC Toolbox has not been initialized yet
noGlobalsErr	–904	System unable to allocate memory, critical error
noSessionErr	–908	Invalid session reference number
badReqErr	–909	Bad parameter or invalid state for this operation
sessClosedErr	–917	The session has closed

SEE ALSO

For an example of the use of the PPCRead function in conjunction with the PPCWrite function, see "Exchanging Data During a PPC Session" beginning on page 11-39.

PPCWrite

Use the PPCWrite function to write message blocks during a session.

```
FUNCTION PPCWrite (pb: PPCWritePBPtr; async: Boolean): OSErr;
```

pb   A pointer to a PPCWrite parameter block.

async   A value that specifies whether the function is to be executed asynchronously (TRUE) or synchronously (FALSE). You should execute the PPCWrite function asynchronously.

Parameter block

ioCompletion	PPCCompProcPtr	Address of a completion routine
ioResult	OSErr	Result code
sessRefNum	PPCSessRefNum	Session reference number
bufferLength	Size	Length of data buffer
actualLength	Size	Actual length of data written
bufferPtr	Ptr	Pointer to data buffer
more	Boolean	TRUE if more data in this block to be written
userData	LongInt	Application-specific data
blockCreator	OSType	Creator of block written
blockType	OSType	Type of block written

DESCRIPTION

If your application calls the PPCWrite function asynchronously, you must specify in the ioCompletion field either the address of a completion routine or NIL. If you set ioCompletion to NIL, you should poll the ioResult field of the PPC parameter block (from your application's main event loop) to determine whether the PPC Toolbox has

completed the requested operation. A value in the `ioResult` field other than 1 indicates that the call is complete. Note that it is unsafe to poll the `ioResult` field at interrupt time since the PPC Toolbox may be in the process of completing a call. See "PPC Toolbox Calling Conventions" beginning on page 11-14.

If you call the `PPCWrite` function asynchronously, you must not change any of the fields in the parameter block until the call completes. The buffer pointed to by the record of data type `PPCWritePBRec` is owned by the PPC Toolbox until the call completes. These objects must not be deallocated or moved in memory while the call is in progress.

The `sessRefNum` field specifies a session identifier. This must be a valid session reference number returned from a previous `PPCStart`, `StartSecureSession`, or `PPCInform` function.

The `bufferLength` and `bufferPtr` fields specify the length and location of a buffer the message block is sent to. If the `PPCWrite` function returns a `noErr` result code, the `actualLength` field returns the actual size of the message block that was written.

Set the `more` field to `TRUE` to indicate that you will be using the `PPCWrite` function again to append data to this message block. Set the `more` field to `FALSE` to indicate that this is the end of the data in this message block.

The initiating port can specify any information in the `userData` field. The `PPCRead` function reports this data to the responding port upon its completion.

Set the `userData`, `blockCreator`, and `blockType` fields for each message block that you create. These fields can give the receiving application information about how to process the contents of the message block. They are ignored when you append information to a message block.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
_PPC	$0006

The registers on entry and exit for this routine are

Registers on entry

A0   Pointer to a parameter block

D0   Selector code

Registers on exit

D0   Result code

RESULT CODES

noErr	0	No error
notInitErr	–900	PPC Toolbox has not been initialized yet
noGlobalsErr	–904	System unable to allocate memory, critical error
noSessionErr	–908	Invalid session reference number
badReqErr	–909	Bad parameter or invalid state for this operation
sessClosedErr	–917	The session has closed

SEE ALSO

For an example of the use of the PPCWrite function in conjunction with the PPCRead function, see "Exchanging Data During a PPC Session" beginning on page 11-39.

Locating a Default User and Invalidating a User

You use the GetDefaultUser function to obtain a user reference number and the name of the default user. To invalidate a particular user name and corresponding password, use the DeleteUserIdentity function.

GetDefaultUser

The GetDefaultUser function returns the user reference number and the name of the default user.

```
FUNCTION GetDefaultUser (VAR userRef: LongInt;
                         VAR userName: Str32): OSErr;
```

userRef   If the GetDefaultUser function completes with no errors, then the userRef parameter returns the user reference number that represents the user name and password of the default user.

userName   The name of the default user.

DESCRIPTION

The default user is specified in the Sharing Setup control panel. This function is useful if your application uses the PPCStart function to initiate a session with an application that does not support guest access.

If the GetDefaultUser function completes with no errors, then the userRef parameter returns the user reference number that represents the user name and password of the default user. The userName parameter must contain NIL or a 32-byte character string. If the userName parameter contains NIL, then the user name string is

not returned. If the `userName` parameter is a 32-byte character string, the 32-byte character string contains the user name that is specified in the Sharing Setup control panel when the `GetDefaultUser` function completes (with no errors).

s **WARNING**
If you are using Pascal, you cannot pass `NIL` for the `userName` parameter. For example, you cannot pass `StringPtr(NIL)^` because Pascal performs range checking of string bounds. s

ASSEMBLY-LANGUAGE INFORMATION

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

**Trap
macro   Selector**

`_PPC` $000D

The registers on entry and exit for this routine are

Registers on entry

A0   Pointer to a `GetDefaultUserParams` record

D0   Selector code

Registers on exit

D0   Result code

RESULT CODES

`noErr`	0	No error
`noDefaultUserErr`	–922	User has not specified owner name in Sharing Setup control panel
`notLoggedInErr`	–923	Default user reference number does not yet exist

SEE ALSO

For an example of the use of the `GetDefaultUser` function, see Listing 11-20 on page 11-45.

DeleteUserIdentity

To invalidate a particular user name and corresponding password, use the `DeleteUserIdentity` function.

```
FUNCTION DeleteUserIdentity (userRef: LongInt): OSErr;
```

`userRef`   The reference number representing the user and password to be deleted.

DESCRIPTION

The `DeleteUserIdentity` function deletes the user name and password corresponding to the user reference number.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
_PPC	$000C

The registers on entry and exit for this routine are

Registers on entry

A0	Pointer to a `DeleteUserParams` record
D0	Selector code

Registers on exit

D0	Result code

RESULT CODES

noErr	0	No error
noUserRecErr	−928	Invalid user reference number

SEE ALSO

For an example of the use of the `DeleteUserIdentity` function, see "Invalidating Users" on page 11-44.

Application-Defined Routines

This section describes the routine syntax for completion routines and port filter functions.

Completion Routines for PPC Toolbox Routines

Your application can provide a pointer to a completion routine in the `ioCompletion` field of a PPC parameter block. You can provide completion routines only for PPC Toolbox routines that you execute asynchronously.

MyCompletionRoutine

You can provide a completion routine for a PPC Toolbox routine that you execute asynchronously.

```
PROCEDURE MyCompletionRoutine (pb: PPCParamBlockPtr);
```

pb   A pointer to the PPC parameter block passed to the PPC Toolbox function.

DESCRIPTION

If you specify a completion routine in the `ioCompletion` field of a PPC parameter block, it is called at interrupt time when the PPC Toolbox routine completes execution. The PPC Toolbox passes to your completion routine a pointer to the same PPC parameter block that your application passed to the PPC Toolbox routine.

s **WARNING**
Completion routines execute at the interrupt level and must preserve all registers other than A0, A1, and D0–D2. (Note that MPW C and MPW Pascal do this automatically.) Your completion routine must not make any calls to the Memory Manager, directly or indirectly, and it can't depend on the validity of handles to unlocked blocks. The PPC Toolbox preserves the application global register A5. s

SEE ALSO

For examples of completion routines, see Listing 11-9 on page 11-37, Listing 11-11 on page 11-38, and Listing 11-13 on page 11-39.

Port Filter Functions

This section describes the port filter function that can be used by the `PPCBrowser` function.

MyPortFilter

You can provide a pointer to a port filter function in the `portFilter` parameter of the `PPCBrowser` function.You can use a port filter function to refine the list of PPC ports that the `PPCBrowser` function displays in the program linking dialog box.

```
FUNCTION MyPortFilter (locationName: LocationNamePtr;
                       thePortInfo: PortInfoPtr): Boolean;
```

locationName

A pointer to a location name record. This record specifies the location of the PPC port currently under consideration for display in the program linking dialog box.

thePortInfo

A pointer to a port information record. This record specifies the port information for the PPC port currently under consideration for display in the program linking dialog box.

DESCRIPTION

The PPCBrowser function calls your port filter function once for each port before it adds that port to the dialog list. Your port filter function should return TRUE for each port that should be displayed in the program linking dialog box, and FALSE for each port that shouldn't be displayed.

SEE ALSO

For an example of a port filter function, see Listing 11-3 on page 11-24. For a description of the location name record, see page 11-49. For a description of the port information record, see page 11-50.

Summary of the PPC Toolbox

Pascal Summary

Constants

```
CONST
   {gestalt selectors}
   gestaltPPCToolboxAttr      = 'ppc ';    {PPC Toolbox attributes}
   gestaltPPCToolboxPresent   = $0000;     {PPC Toolbox is present}
   gestaltPPCSupportsRealTime = $1000;     {real time only in system }
                                           { software version 7.0}
   gestaltPPCSupportsOutGoing = $0002;     {support of outgoing }
                                           { sessions across a network}
   gestaltPPCSupportsIncoming = $0001;     {user enabled program }
                                           { linking in Sharing Setup }
                                           { control panel}
   {service type)
   ppcServiceRealTime         = 1;         {real time only in System 7}
   {look-up type}
   ppcNoLocation              = 0;         {there is no PPCLocName}
   ppcNBPLocation             = 1;         {use AppleTalk NBP}
   ppcNBPTypeLocation         = 2;         {use just the NBP type, fill }
                                           { in the rest with default}
   {port type}
   ppcByCreatorAndType        = 1;         {port type is specified as }
                                           { standard creator and type}
   ppcByString                = 2;         {port type is in Pascal }
                                           { string format}
   {session request type returned in the PPCInform function}
   ppcLocalOrigin             = 1;         {session initiated on }
                                           { local computer}
   ppcRemoteOrigin            = 2;         {session initiated on }
                                           { remote computer}
```

Data Types

```
TYPE
   PPCServiceType          = SignedByte;      {service type}
   PPCLocationKind         = Integer;         {look-up type}
   PPCPortKinds            = Integer;         {port type}
   PPCSessionOrigin        = SignedByte;      {local or remote}
   PPCPortRefNum           = Integer;         {port reference number}
   PPCSessRefNum           = LongInt;         {session reference number}

   LocationNamePtr = ^LocationNameRec;
   LocationNameRec =
   RECORD
      locationKindSelector: PPCLocationKind; {which variant}
      CASE PPCLocationKind OF                 {ppcNoLocation: storage not }
                                              { used by this value}
         ppcNBPLocation:                      {NBP name entity}
                        (nbpEntity: EntityName);
         ppcNBPTypeLocation:(nbpType: Str32);{just the NBP type string }
                                              { for the PPCOpen function}
   END;
   PortInfoPtr = ^PortInfoRec;
   PortInfoRec =
   RECORD
      filler1:           SignedByte;          {space holder}
      authRequired:      Boolean;             {authentication required}
      name:              PPCPortRec;          {port name}
   END;

   PPCPortPtr = ^PPCPortRec;
   PPCPortRec =
   RECORD
      nameScript:        ScriptCode;          {script identifier}
      name:              Str32;               {port name shown in program }
                                              { linking dialog box}
         portKindSelector: PPCPortKinds;      {general category of }
                                              { application}
         CASE PPCPortKinds OF
            ppcByString:   (portTypeStr: Str32);{32 characters}
            ppcByCreatorAndType:              {4-character creator and type}
                        (portCreator: OSType; portType: OSType);
   END;
```

```
PPCParamBlockPtr = ^PPCParamBlockRec;
PPCParamBlockRec =
RECORD
    CASE Integer OF
    0: (openParam:       PPCOpenPBRec);        {PPCOPen params}
    1: (informParam:     PPCInformPBRec);      {PPCInform params}
    2: (startParam:      PPCStartPBRec);       {PPCStart params}
    3: (acceptParam:     PPCAcceptPBRec);      {PPCAccept params}
    4: (rejectParam:     PPCRejectPBRec);      {PPCReject params}
    5: (writeParam:      PPCWritePBRec);       {PPCWrite params}
    6: (readParam:       PPCReadPBRec);        {PPCRead params}
    7: (endParam:        PPCEndPBRec);         {PPCEnd params}
    8: (closeParam:      PPCClosePBRec);       {PPCClose params}
    9: (listPortsParam:  IPCListPortsPBRec);   {IPCListPorts params}
END;


PortInfoArrayPtr = ^PortInfoArray;
PortInfoArray    = ARRAY[0..0] OF PortInfoRec;


PPCOpenPBPtr = ^PPCOpenPBRec;
PPCOpenPBRec =
RECORD
    qLink:         Ptr;                  {private}
    csCode:        Integer;              {private}
    intUse:        Integer;              {private}
    intUsePtr:     Ptr;                  {private}
    ioCompletion:  PPCCompProcPtr;       {address of a }
                                         { completion routine}
    ioResult:      OSErr;                {completion of operation}
    reserved:      ARRAY[1..5] OF LongInt;
                                         {private}
    portRefNum:    PPCPortRefNum;        {PPC port identifier}
    filler1:       LongInt;              {space holder}
    serviceType:   PPCServiceType;       {real time only}
    resFlag:       SignedByte;           {reserved field}
    portName:      PPCPortPtr;           {name of port to be opened}
    locationName:  LocationNamePtr;      {location of port to be }
                                         { opened}
    networkVisible: Boolean;             {port is visible for }
                                         { browsing}
    nbpRegistered:  Boolean;             {location name registered }
                                         { on network}
END;
```

```
PPCInformPBPtr = ^PPCInformPBRec;
PPCInformPBRec =
RECORD
   qLink:          Ptr;                {private}
   csCode:         Integer;            {private}
   intUse:         Integer;            {private}
   intUsePtr:      Ptr;                {private}
   ioCompletion:   PPCCompProcPtr;     {address of a completion }
                                       { routine}
   ioResult:       OSErr;              {completion of operation}
   reserved:       ARRAY[1..5] OF LongInt;
                                       {private}
   portRefNum:     PPCPortRefNum;      {port identifier}
   sessRefNum:     PPCSessRefNum;      {session identifier}
   serviceType:    PPCServiceType;     {real time only}
   autoAccept:     Boolean;            {automatic session }
                                       { acceptance}
   portName:       PPCPortPtr;         {name of port that }
                                       { initiated a session}
   locationName:   LocationNamePtr;    {location of port that }
                                       { initiated a session}
   userName:       StringPtr;          {name of user that }
                                       { initiated a session}
   userData:       LongInt;            {application-defined}
   requestType:    PPCSessionOrigin;   {local or remote}
END;

PPCStartPBPtr = ^PPCStartPBRec;
PPCStartPBRec =
RECORD
   qLink:          Ptr;                {private}
   csCode:         Integer;            {private}
   intUse:         Integer;            {private}
   intUsePtr:      Ptr;                {private}
   ioCompletion:   PPCCompProcPtr;     {address of a completion }
                                       { routine}
   ioResult:       OSErr;              {completion of operation}
   reserved:       ARRAY[1..5] OF LongInt;
                                       {private}
   portRefNum:     PPCPortRefNum;      {identifier for requested }
                                       { port}
   sessRefNum:     PPCSessRefNum;      {session identifier}
   serviceType:    PPCServiceType;     {real time only}
```

```
   resFlag:            SignedByte;         {reserved field}
   portName:           PPCPortPtr;         {name of port to be opened}
   locationName:       LocationNamePtr;    {location of port to be }
                                           { opened}
   rejectInfo:         LongInt;            {rejection of session}
   userData:           LongInt;            {application-specific}
   userRefNum:         LongInt;            {specifies an authenticated }
                                           { user}
END;


PPCAcceptPBPtr = ^PPCAcceptPBRec;
PPCAcceptPBRec =
RECORD
   qLink:              Ptr;                {private}
   csCode:             Integer;            {private}
   intUse:             Integer;            {private}
   intUsePtr:          Ptr;                {private}
   ioCompletion:       PPCCompProcPtr;     {address of a completion }
                                           { routine}
   ioResult:           OSErr;              {completion of operation}
   reserved:           ARRAY[1..5] OF LongInt;
                                           {private}
   filler1:            Integer;            {space holder}
   sessRefNum:         PPCSessRefNum;      {session identifier}
END;


PPCRejectPBPtr = ^PPCRejectPBRec;
PPCRejectPBRec =
RECORD
   qLink:              Ptr;                {private}
   csCode:             Integer;            {private}
   intUse:             Integer;            {private}
   intUsePtr:          Ptr;                {private}
   ioCompletion:       PPCCompProcPtr;     {address of a completion }
                                           { routine}
   ioResult:           OSErr;              {completion of operation}
   reserved:           ARRAY[1..5] OF LongInt;
                                           {private}
   filler1:            Integer;            {space holder}
   sessRefNum:         PPCSessRefNum;      {session identifier}
   filler2:            Integer;            {space holder}
   filler3:            LongInt;            {space holder}
   filler4:            LongInt;            {space holder}
```

```
   rejectInfo:      LongInt;              {rejection of session}
END;


PPCWritePBPtr = ^PPCWritePBRec;
PPCWritePBRec =
RECORD
   qLink:           Ptr;                  {private}
   csCode:          Integer;              {private}
   intUse:          Integer;              {private}
   intUsePtr:       Ptr;                  {private}
   ioCompletion:    PPCCompProcPtr;       {address of a completion }
                                          { routine}
   ioResult:        OSErr;                {completion of operation}
   reserved:        ARRAY[1..5] OF LongInt;
                                          {private}
   filler1:         Integer;              {space holder}
   sessRefNum:      PPCSessRefNum;        {session identifier}
   bufferLength:    Size;                 {length of buffer to be }
                                          { written}
   actualLength:    Size;                 {actual size of data written}
   bufferPtr:       Ptr;                  {location of buffer to be }
                                          { written}
   more:            Boolean;              {additional data to be }
                                          { written}
   filler2:         SignedByte;           {space holder}
   userData:        LongInt;              {application-specific}
   blockCreator:    OSType;               {creator of block to be }
                                          { written}
   blockType:       OSType;               {type of block to be written}
END;


PPCReadPBPtr = ^PPCReadPBRec;
PPCReadPBRec =
RECORD
   qLink:           Ptr;                  {private}
   csCode:          Integer;              {private}
   intUse:          Integer;              {private}
   intUsePtr:       Ptr;                  {private}
   ioCompletion:    PPCCompProcPtr;       {address of a completion }
                                          { routine}
   ioResult:        OSErr;                {completion of operation}
   reserved:        ARRAY[1..5] OF LongInt;
                                          {private}
```

```
   filler1:          Integer;            {space holder}
   sessRefNum:       PPCSessRefNum;      {session identifier}
   bufferLength:     Size;               {length of buffer to be read}
   actualLength:     Size;               {actual size of the data }
                                         { read}
   bufferPtr:        Ptr;                {location of buffer to be }
                                         { read}
   more:             Boolean;            {additional data to be read}
   filler2:          SignedByte;         {space holder}
   userData:         LongInt;            {application-specific}
   blockCreator:     OSType;             {creator of block to be read}
   blockType:        OSType;             {type of block to be read}
END;


PPCEndPBPtr = ^PPCEndPBRec;
PPCEndPBRec =
RECORD
   qLink:            Ptr;                {private}
   csCode:           Integer;            {private}
   intUse:           Integer;            {private}
   intUsePtr:        Ptr;                {private}
   ioCompletion:     PPCCompProcPtr;     {address of a completion }
                                         { routine}
   ioResult:         OSErr;              {completion of operation}
   reserved:         ARRAY[1..5] OF LongInt;
                                         {private}
   filler1:          Integer;            {space holder}
   sessRefNum:       PPCSessRefNum;      {identifier of session to }
                                         { be terminated}
END;


PPCClosePBPtr = ^PPCClosePBRec;
PPCClosePBRec =
RECORD
   qLink:            Ptr;                {private}
   csCode:           Integer;            {private}
   intUse:           Integer;            {private}
   intUsePtr:        Ptr;                {private}
   ioCompletion:     PPCCompProcPtr;     {address of a completion }
                                         { routine}
   ioResult:         OSErr;              {completion of operation}
   reserved:         ARRAY[1..5] OF LongInt;
                                         {private}
```

```
    portRefNum:        PPCPortRefNum;      {identifier of port to }
                                           { be closed}
END;


IPCListPortsPBPtr = ^IPCListPortsPBRec;
IPCListPortsPBRec =
RECORD
    qLink:             Ptr;                {private}
    csCode:            Integer;            {private}
    intUse:            Integer;            {private}
    intUsePtr:         Ptr;                {private}
    ioCompletion:      PPCCompProcPtr;     {address of a completion }
                                           { routine}
    ioResult:          OSErr;              {completion of operation}
    reserved:          ARRAY[1..5] OF LongInt;
                                           {private}
    filler1:           Integer;            {space holder}
    startIndex:        Integer;            {index to the port entry }
                                           { list}
    requestCount:      Integer;            {number of entries to }
                                           { be returned}
    actualCount:       Integer;            {actual number of port names}
    portName:          PPCPortPtr;         {list of port names}
    locationName:      LocationNamePtr;    {location of port names}
    bufferPtr:         PortInfoArrayPtr;   {pointer to a buffer}
END;
```

PPC Toolbox Routines

Initializing the PPC Toolbox

```
FUNCTION PPCInit: OSErr;
```

Using the Program Linking Dialog Box

```
FUNCTION PPCBrowser         (prompt: Str255; applListLabel: Str255;
                             defaultSpecified: Boolean;
                             VAR theLocation: LocationNameRec;
                             VAR thePortInfo: PortInfoRec;
                             portFilter: PPCFilterProcPtr;
                             theLocNBPType: Str32): OSErr;
```

Obtaining a List of Ports

```
FUNCTION IPCListPorts       (pb: IPCListPortsPBPtr; async: Boolean): OSErr;
```

Opening and Closing a Port

```
FUNCTION PPCOpen             (pb: PPCOpenPBPtr; async: Boolean): OSErr;
FUNCTION PPCClose            (pb: PPCClosePBPtr; async: Boolean): OSErr;
```

Starting and Ending a Session

```
FUNCTION PPCStart            (pb: PPCStartPBPtr; async: Boolean): OSErr;
FUNCTION StartSecureSession  (pb: PPCStartPBPtr; VAR userName: Str32;
                              useDefault: Boolean; allowGuest: Boolean;
                              VAR guestSelected: Boolean; prompt: Str255)
                              : OSErr;
FUNCTION PPCEnd              (pb: PPCEndPBPtr; async: Boolean): OSErr;
```

Receiving, Accepting, and Rejecting a Session

```
FUNCTION PPCInform           (pb: PPCInformPBPtr; async: Boolean): OSErr;
FUNCTION PPCAccept           (pb: PPCAcceptPBPtr; async: Boolean): OSErr;
FUNCTION PPCReject           (pb: PPCRejectPBPtr; async: Boolean): OSErr;
```

Reading and Writing Data

```
FUNCTION PPCRead             (pb: PPCReadPBPtr; async: Boolean): OSErr;
FUNCTION PPCWrite            (pb: PPCWritePBPtr; async: Boolean): OSErr;
```

Locating a Default User and Invalidating a User

```
FUNCTION GetDefaultUser      (VAR userRef: LongInt; VAR userName: Str32)
                              : OSErr;
FUNCTION DeleteUserIdentity  (userRef: LongInt): OSErr;
```

Application-Defined Routines

```
PROCEDURE MyCompletionRoutine
                             (pb: PPCParamBlockPtr);
FUNCTION MyPortFilter        (locationName: LocationNameRec;
                              thePortInfo: PortInfoRec): Boolean;
```

C Summary

Constants

```
CONST
enum {
   /*gestalt selectors*/
   #define gestaltPPCToolboxAttr 'ppc '   /*PPC Toolbox attributes*/
   gestaltPPCToolboxPresent   = $0000,    /*PPC Toolbox is present*/
   gestaltPPCSupportsRealTime = $1000,    /*real time only in system */
                                          /* software version 7.0*/
   gestaltPPCSupportsOutGoing = $0002,    /*support of outgoing */
                                          /* sessions across a network*/
   gestaltPPCSupportsIncoming = $0001     /*user enabled program */
                                          /* linking in Sharing Setup */
                                          /* control panel*/
};
enum {
   /*service type*/
   ppcServiceRealTime         = 1         /*real time only in System 7*/
};
enum {
   /*look-up type*/
   ppcNoLocation              = 0,        /*there is no PPCLocName*/
   ppcNBPLocation             = 1,        /*use AppleTalk NBP*/
   ppcNBPTypeLocation         = 2         /*use just the NBP type, fill */
                                          /* in the rest with default*/
};
enum {
   /*port type*/
   ppcByCreatorAndType        = 1,        /*port type is specified as */
                                          /* standard Mac creator and type*/
   ppcByString                = 2         /*port type is in Pascal */
                                          /* string format*/
};
enum {
   /*session request type returned in the PPCInform function*/
   ppcLocalOrigin             = 1,        /*session initiated on */
                                          /* local computer*/
   ppcRemoteOrigin            = 2         /*session initiated on */
                                          /* remote computer*/
};
```

Data Types

```
typedef unsigned char PPCServiceType;          /*service type*/
typedef short PPCLocationKind;                 /*look-up type*/
typedef short PPCPortKinds;                     /*port type*/
typedef unsigned char PPCSessionOrigin;        /*local or remote*/
typedef short PPCPortRefNum;                    /*port reference number*/
typedef long PPCSessRefNum;                     /*session reference number*/

struct PPCPortRec {
    ScriptCode nameScript;                      /*script identifier*/
    Str32 name;                                 /*port name shown in program */
                                                /* linking dialog box*/
    PPCPortKinds portKindSelector;              /*general category of */
                                                /* application*/
    union
        Str32 portTypeStr;                      /*32 characters*/
        struct
           OSType creator;                      /*4-character creator and */
           OSType type;                         /* type*/
           } port;
        } u;
};
typedef struct PPCPortRec PPCPortRec;
typedef PPCPortRec *PPCPortPtr;

struct LocationNameRec {
    PPCLocationKind locationKindSelector;      /*which variant*/
    union {
        EntityName  nbpEntity;                  /*NBP name entity*/
        Str32       nbpType;                    /*just the NBP type string */
                                                /* for the PPCOpen function*/
        } u;
};

typedef struct LocationNameRec LocationNameRec;
typedef LocationNameRec *LocationNamePtr;

struct PortInfoRec {
    unsigned char  filler1;                     /*space holder*/
    Boolean        authRequired;                /*authentication required*/
    PPCPortRec     name;                        /*port name*/
};
```

```
typedef struct PortInfoRec PortInfoRec;
typedef PortInfoRec *PortInfoPtr;

typedef PortInfoRec *PortInfoArrayPtr;
typedef pascal Boolean (*PPCFilterProcPtr) (LocationNamePtr, PortInfoPtr);
/*procedures you need to write*/
/*ex: void MyCompletionRoutine(PPCParamBlkPtr pb)*/
/*ex: pascal Boolean MyPortFilter(LocationNamePtr, PortInfoPtr)*/
typedef ProcPtr PPCCompProcPtr;

#define PPCHeader \
    Ptr             qLink;              /*private*/
    unsigned short  csCode;             /*private*/
    unsigned short  intUse;             /*private*/
    Ptr             intUsePtr;          /*private*/
    PPCCompProcPtr  ioCompletion;       /*address of a */
                                        /* completion routine*/
    OSErr           ioResult;           /*completion of operation*/
    unsigned long   Reserved[5];        /*private*/

struct PPCOpenPBRec {
    PPCHeader
    PPCPortRefNum  portRefNum;          /*PPC port identifier*/
    long           filler1;             /*space holder*/
    PPCServiceType serviceType;         /*real time only*/
    unsigned char  resFlag;             /*reserved field*/
    PPCPortPtr     portName;            /*name of port to be opened*/
    LocationNamePtr locationName;       /*location of port to be */
                                        /* opened*/
    Boolean        networkVisible;      /*port is visible for */
                                        /* browsing*/
    Boolean        nbpRegistered;       /*location name registered */
                                        /* on network*/
};

typedef struct PPCOpenPBRec PPCOpenPBRec;
typedef PPCOpenPBRec *PPCOpenPBPtr;

struct PPCInformPBRec {
    PPCHeader
    PPCPortRefNum     portRefNum;       /*port identifier*/
    PPCSessRefNum     sessRefNum;       /*session identifier*/
    PPCServiceType    serviceType;      /*real time only*/
```

```
    Boolean             autoAccept;                 /*automatic session acceptance*/
    PPCPortPtr          portName;                   /*name of port that */
                                                    /* initiated a session*/
    LocationNamePtr     locationName;               /*location of port that */
                                                    /* initiated a session*/
    StringPtr           userName;                   /*name of user that */
                                                    /* initiated a session*/
    unsigned long       userData;                   /*application-defined*/
    PPCSessionOrigin    requestType;                /*local or remote*/
};


typdef struct PPCInformPBRec PPCInformPBPtr;

struct PPCStartPBRec {
    PPCHeader
    PPCPortRefNum       portRefNum;             /*identifier for requested */
                                                /* port*/
    PPCSessRefNum       sessRefNum;             /*session identifier*/
    PPCServiceType      serviceType;            /*real time only*/
    unsigned char       resFlag;                /*reserved field*/
    PPCPortPtr          portName;               /*name of port to be opened*/
    LocationNamePtr     locationName;           /*location of port to be opened*/
    unsigned long       rejectInfo;             /*rejection of session*/
    unsigned long       userData;               /*application-specific*/
    unsigned long       userRefNum;             /*specifies an authenticated user*/
};


typedef struct PPCStartPBRec PPCStartPBRec;
typedef PPCStartPBRec *PPCStartPBPtr;

struct PPCAcceptPBRec {
     PPCHeader
     short          filler1;              /*space holder*/
     PPCSessRefNum  sessRefNum;           /*session identifier*/
};


typedef struct PPCAcceptPBRec PPCAcceptPBRec;
typedef PPCAcceptPBRec *PPCAcceptPBPtr;

struct PPCRejectPBRec {
     PPCHeader
     short          filler1;              /*space holder*/
     PPCSessRefNum  sessRefNum;           /*session identifier*/
```

```
      short          filler2;               /*space holder*/
      long           filler3;               /*space holder*/
      long           filler4;               /*space holder*/
      unsigned long  rejectInfo;            /*rejection of session*/
};

typedef struct PPCRejectPBRec PPCRejectPBRec;
typedef PPCRejectPBRec *PPCRejectPBPtr;

struct PPCWritePBRec {
   PPCHeader
   short          filler1;               /*space holder*/
   PPCSessRefNum  sessRefNum;            /*session identifier*/
   Size           bufferLength;          /*length of buffer to be written*/
   Size           actualLength;          /*actual size of data written*/
   Ptr            bufferPtr;             /*location of buffer to be */
                                         /* written*/
   Boolean        more;                  /*additional data to be written*/
   unsigned char  filler2;               /*space holder*/
   unsigned long  userData;              /*application-specific*/
   OSType         blockCreator;          /*creator of block to be written*/
   OSType         blockType;             /*type of block to be written*/
};

typedef struct PPCWritePBRec PPCWritePBRec;
typedef PPCWritePBRec *PPCWritePBPtr;

struct PPCReadPBRec {
   PPCHeader
   short          filler1;               /*space holder*/
   PPCSessRefNum  sessRefNum;            /*session identifier*/
   Size           bufferLength;          /*length of buffer to be read*/
   Size           actualLength;          /*actual size of the data read*/
   Ptr            bufferPtr;             /*location of buffer to be read*/
   Boolean        more;                  /*additional data to be read*/
   unsigned char  filler2;               /*space holder*/
   unsigned long  userData;              /*application-specific*/
   OSType         blockCreator;          /*creator of block to be read*/
   OSType         blockType;             /*type of block to be read*/
};

typedef struct PPCReadPBRec PPCReadPBRec;
typdef PPCReadPBRec *PPCReadPBPtr;
```

```
struct PPCEndPBRec {
   PPCHeader
   short           filler1;                /*space holder*/
   PPCSessRefNum   sessRefNum;             /*identifier of session to */
                                           /* be terminated*/
};

typedef struct PPCEndPBRec PPCEndPBRec;
typedef PPCEndPBRec *PPCEndPBPtr;

struct PPCClosePBRec {
   PPCHeader
   PPCPortRefNum   portRefNum;             /*identifier of port to */
                                           /* be closed*/
};

typedef struct PPCClosePBRec PPCClosePBRec;
typedef PPCClosePBRec *PPCClosePBPtr;

struct IPCListPortsPBRec {
   PPCHeader
   short           filler1;                /*space holder*/
   unsigned short  startIndex;             /*index to the port entry list*/
   unsigned short  requestCount;           /*number of entries to */
                                           /* be returned*/
   unsigned short  actualCount;            /*actual number of port names*/
   PPCPortPtr      portName;               /*list of port names*/
   LocationNamePtr locationName;           /*location of port names*/
   PortInfoArrayPtr bufferPtr;             /*pointer to a buffer*/
};
typedef struct IPCListPortsPBRec IPCListPortsPBRec;
typedef IPCListPortsPBRec *IPCListPortsPBPtr;

union PPCParamBlockRec {
   PPCOpenPBRec      openParam;            /*PPCOpen params*/
   PPCInformPBRec    informParam;          /*PPCInform params*/
   PPCStartPBRec     startParam;           /*PPCStart params*/
   PPCAcceptPBRec    acceptParam;          /*PPCAccept params*/
   PPCRejectPBRec    rejectParam;          /*PPCReject params*/
   PPCWritePBRec     writeParam;           /*PPCWrite params*/
   PPCReadPBRec      readParam;            /*PPCRead params*/
   PPCEndPBRec       endParam;             /*PPCEnd params*/
   PPCClosePBRec     closeParam;           /*PPCClose params*/
```

```
   IPCListPortsPBRec listPortsParam;        /*IPCListPorts params*/
};


typdef union PPCParamBlockRec PPCParamBlockRec;
typdef PPCParamBlockRec *PPCParamBlockPtr;
```

PPC Toolbox Routines

Initializing the PPC Toolbox

```
pascal OSErr PPCInit          (void);
```

Using the Program Linking Dialog Box

```
pascal OSErr PPCBrowser       (ConstStr255Param prompt,
                               ConstStr255Param applListLabel,
                               Boolean defaultSpecified,
                               LocationNameRec *theLocation,
                               PortInfoRec *thePortInfo,
                               PPCFilterProcPtr portFilter,
                               ConstStr32Param theLocNBPType);
```

Obtaining a List of Ports

```
pascal OSErr IPCListPorts     (IPCListPortsPBPtr pb, Boolean async);
```

Opening and Closing a Port

```
pascal OSErr PPCOpen          (PPCOpenPBPtr pb, Boolean async);
pascal OSErr PPCClose         (PPCClosePBPtr pb, Boolean async);
```

Starting and Ending a Session

```
pascal OSErr PPCStart         (PPCStartPBPtr pb, Boolean async);
pascal OSErr StartSecureSession
                              (PPCStartPBPtr pb, Str32 userName,
                               Boolean useDefault, Boolean allowGuest,
                               Boolean *guestSelected,
                               ConstStr255Param prompt);
pascal OSErr PPCEnd           (PPCEndPBPtr pb, Boolean async);
```

Receiving, Accepting, and Rejecting a Session

```
pascal OSErr PPCInform        (PPCInformPBPtr pb, Boolean async);
pascal OSErr PPCAccept        (PPCAcceptPBPtr pb, Boolean async);
pascal OSErr PPCReject        (PPCRejectPBPtr pb, Boolean async);
```

Reading and Writing Data

```
pascal OSErr PPCRead          (PPCReadPBPtr pb, Boolean async);
pascal OSErr PPCWrite         (PPCWritePBPtr pb, Boolean async);
```

Locating a Default User and Invalidating a User

```
pascal OSErr GetDefaultUser (unsigned long *userRef, Str32 userName);
pascal OSErr DeleteUserIdentity
                             (unsigned long userRef);
```

Application-Defined Routines

```
void MyCompletionRoutine     (PPCParamBlockPtr pb);
pascal Boolean MyPortFilter (LocationNameRec locationName,
                              PortInfoRec thePortInfo);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

_Pack9

Selector	Routine
$0D00	PPCBrowser

_PPC

Selector	Routine
$0000	PPCInit
$0001	PPCOpen
$0002	PPCStart
$0003	PPCInform
$0004	PPCAccept
$0005	PPCReject
$0006	PPCWrite
$0007	PPCRead
$0008	PPCEnd
$0009	PPCClose

Selector	Routine
$000A	IPCListPorts
$000C	DeleteUserIdentity
$000D	GetDefaultUser
$000E	StartSecureSession

Result Codes

noErr	0	No error
paramErr	–50	Illegal parameter
memFullErr	–108	Not enough memory to load PPCBrowser package
userCanceledErr	–128	User decided not to conduct a session
notInitErr	–900	PPC Toolbox has not been initialized yet
nameTypeErr	–902	Invalid or inappropriate locationKindSelector in location name
noPortErr	–903	Unable to open port or bad port reference number
noGlobalsErr	–904	System unable to allocate memory, critical error
localOnlyErr	–905	Network activity is currently disabled
destPortErr	–906	Port does not exist at destination
sessTableErr	–907	PPC Toolbox is unable to create a session
noSessionErr	–908	Invalid session reference number
badReqErr	–909	Bad parameter or invalid state for this operation
portNameExistsErr	–910	Another port is already open with this name
noUserNameErr	–911	User name unknown on destination machine
userRejectErr	–912	Destination rejected the session request
noResponseErr	–915	Unable to contact application
portClosedErr	–916	The port was closed
sessClosedErr	–917	The session has closed
badPortNameErr	–919	PPC port record is invalid
noDefaultUserErr	–922	User has not specified owner name in Sharing Setup control panel
notLoggedInErr	–923	Default user reference number does not yet exist
noUserRefErr	–924	Unable to create a new user reference number
networkErr	–925	An error has occurred in the network
noInformErr	–926	PPCStart failed because target application did not have an inform pending
authFailErr	–927	User's password is wrong
noUserRecErr	–928	Invalid user reference number
badServiceMethodErr	–930	Service method is other than ppcServiceRealTime
badLocNameErr	–931	Location name is invalid
guestNotAllowedErr	–932	Destination port requires authentication
nbpDuplicate	–1027	Location name represents a duplicate on this computer

Data Access Manager

Contents

This chapter describes how your application can use the Data Access Manager to gain access to data in another application. It also tells you how to provide templates to be used for data transactions.

The Data Access Manager is available in System 7 and later versions. Use the Gestalt Manager to determine whether the Data Access Manager is present. To determine whether the Data Access Manager is available, use the `Gestalt` function with the `gestaltDBAccessMgrAttr` environmental selector. If the Data Access Manager is not available, the `Gestalt` function returns an error. For more information on the Gestalt Manager, see *Inside Macintosh: Operating System Utilities*.

The Data Access Manager allows your application to communicate with a database or other data source even if you do not know anything about databases in general or the specific data source with which the users of your software will be communicating. All your application needs are a few high-level Data Access Manager functions and access to a file called a **query document.** The query document, provided by another application, contains commands and data in the format appropriate for the database or other data source. The string of commands and data sent to the data source are referred to as a **query.** Note that a query does not necessarily extract data from a data source; it might only send data or commands to a database or other application.

The Data Access Manager makes it easy for your application to communicate with data sources. You need only add a menu item that opens a query document, using a few standard Data Access Manager functions to implement the menu selection. Users of your application can then access a database or other data source whenever they have the appropriate query documents. A user of a word-processing program might use this feature, for example, to obtain access to archived material, dictionaries in a variety of languages, or a database of famous quotations. A user of a spreadsheet program might use a query document to obtain tax records, actuarial tables, or other data. A user of an art or computer-aided design program might download archived illustrations or designs. And for the user of a database application for the Macintosh computer, the Data Access Manager can provide the resources and power of a mainframe database.

The Data Access Manager also provides a low-level interface for use by applications that are capable of creating their own queries and that therefore do not have to use query documents.

If your application uses only the high-level interface and relies on query documents created by other programs, then all the routines you need to know are described in this chapter. However, if you want to create a query document or an application that uses the low-level interface, then you must also be familiar with the command language used by the data server.

You need the information in this chapter if you want your application to access data in other applications or if you want to write a query document.

Note

The Data Access Manager makes it easy for your application to communicate with a database running on a remote computer, and this chapter generally assumes that you are using it for that purpose. However, there is no reason why the database could not be local—that is, running on the same computer as your application. To implement such a system, you would have to have a database that runs on a Macintosh computer and that has a command-language interface, plus a database extension that can use that command language. In most cases, it would be much simpler to run the database as a separate application and use the Clipboard to transfer data into and out of the database. u

Note also that the program containing the data need not be a database. With the appropriate database extension, your application could read data from a spreadsheet, for example, or any other program that stores data.

Apple Computer, Inc. provides a database extension that uses Data Access Language (DAL). A **database extension** provides an interface between the Data Access Manager and the database or other program that contains the data. If you want to write an application that uses the low-level interface to communicate with a Data Access Language server, or if you want to create a query document that uses Data Access Language, you must be familiar with that language. *Data Access Language Programmer's Reference,* available from APDA, fully describes this language.

About the Data Access Manager

The Data Access Manager constitutes a standard interface that allows Macintosh applications to communicate with any number of databases or other data sources through a variety of data servers. As used in this chapter, a **data server** is the application that acts as an interface between the database extension on the Macintosh computer and the data source, which can be on the Macintosh computer or on a remote host computer. A data server can be a database server program, such as a Data Access Language server, which can provide an interface to a variety of different databases, or it can be the data source itself, such as a Macintosh application.

The Data Access Manager has two application interfaces: the high-level interface and the low-level interface. If the proper database extension and query documents are available in the user's system, you can use the high-level interface to communicate with a data source without having any knowledge of the command language that the data server uses. Even if you use the low-level interface, your application can isolate the user from any specific knowledge of the data source or the data server's command language.

This section presents an overview and description of the Data Access Manager, including diagrams and conceptual descriptions of the components and processes involved in using the high-level and low-level interfaces. Next, "Using the Data Access Manager" beginning on page 12-12 includes descriptions, flowcharts, and program fragments that provide a step-by-step guide to the use of the high-level and low-level interfaces. "Creating a Query Document" beginning on page 12-46 describes the contents and function of a query document. You do not have to read this section unless you are writing an application that creates query documents, although if you are using the high-level interface you might be interested to know just how a query document works.

Figure 12-1 illustrates connections between Macintosh applications and a database on a remote computer. The arrows in Figure 12-1 show the flow of information, not the paths of commands or control signals. See Figure 12-2 on page 12-8 and Figure 12-3 on page 12-10 for the sequences involved in sending and retrieving data.

Figure 12-1   A connection with a database
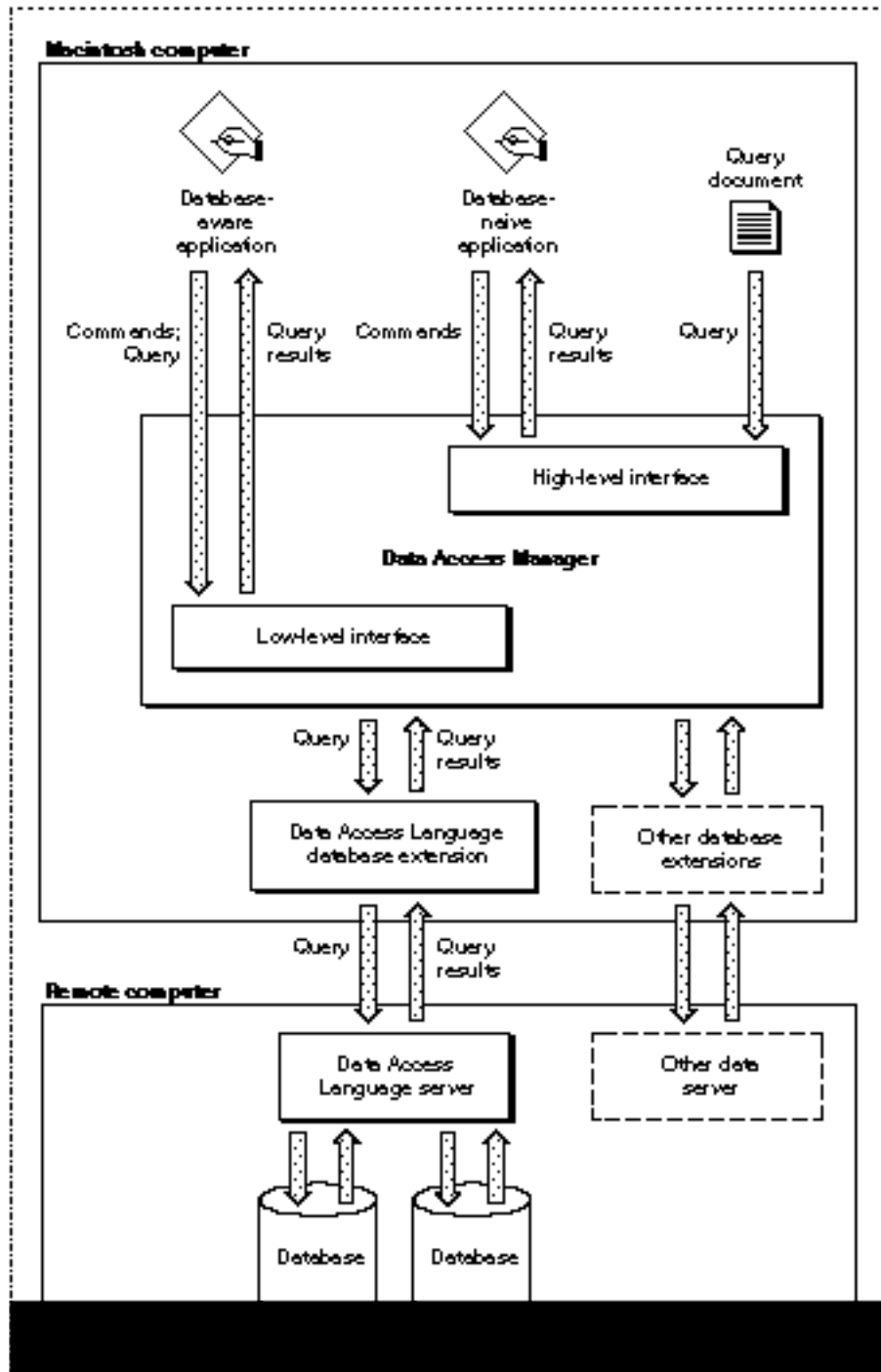
The High-Level Interface

As Figure 12-1 on page 12-6 shows, a database-naive application—that is, one that cannot prepare a query for a specific data server—uses the Data Access Manager's high-level routines to communicate with a data server. Because the application cannot prepare a query, it must use a query document to provide one. A query document can contain code, called a **query definition function,** that prompts the user for information and modifies the query before the Data Access Manager sends it to the data server. The exact format of a query definition function is described in "Writing a Query Definition Function" on page 12-51.

Note

The term *query* refers to any string of commands (and associated data) that can be executed by a data server. A query can send data to a data source, retrieve data from a data source, or reorganize the data in a data source. The Data Access Manager does not interpret or execute the query; it only implements the interface (sometimes called the *application program interface,* or API) that allows you to send the query to the data server. u

When you want to use the high-level routines to execute a query on a data server, you first select a query document or allow the user to select one. You use high-level routines to

n   get the query from the query document

n   execute the query definition function to modify the query

n   send the query to the data server

n   retrieve the results from any query that asks for information from the data source

n   convert to text the results returned by a query

For example, suppose a company that makes rubber ducks has a database on a minicomputer that contains a mailing list of all its customers. The database has a Data Access Language interface, and the company's marketing manager has a Macintosh computer with an application that uses high-level Data Access Manager routines to communicate with the remote database server. As Figure 12-2 illustrates, the marketing manager must also have a query document, created by another application, that she can use to get an address from the mailing list on the remote minicomputer. The query document can be as complex or as simple as its creator cares to make it; in this example, the query document is designed specifically to obtain addresses from the rubber duck mailing list. The marketing manager might have several other query documents available as well: one to extract a mailing list for a specific zip code, one to list all of the customers who have made a purchase within the last year, and so on.

Notice that once the query document has sent the query to the data server, the Data Access Manager handles the data retrieval. Although query documents and high-level Data Access Manager routines make it very easy for you to *request* data from a data source, there is no way for a query document to verify that data *sent* to a data source has been successfully received. For that reason, it is recommended that you use the low-level interface to send data to a data source or update data in a data source.

Figure 12-2   Using high-level Data Access Manager routines
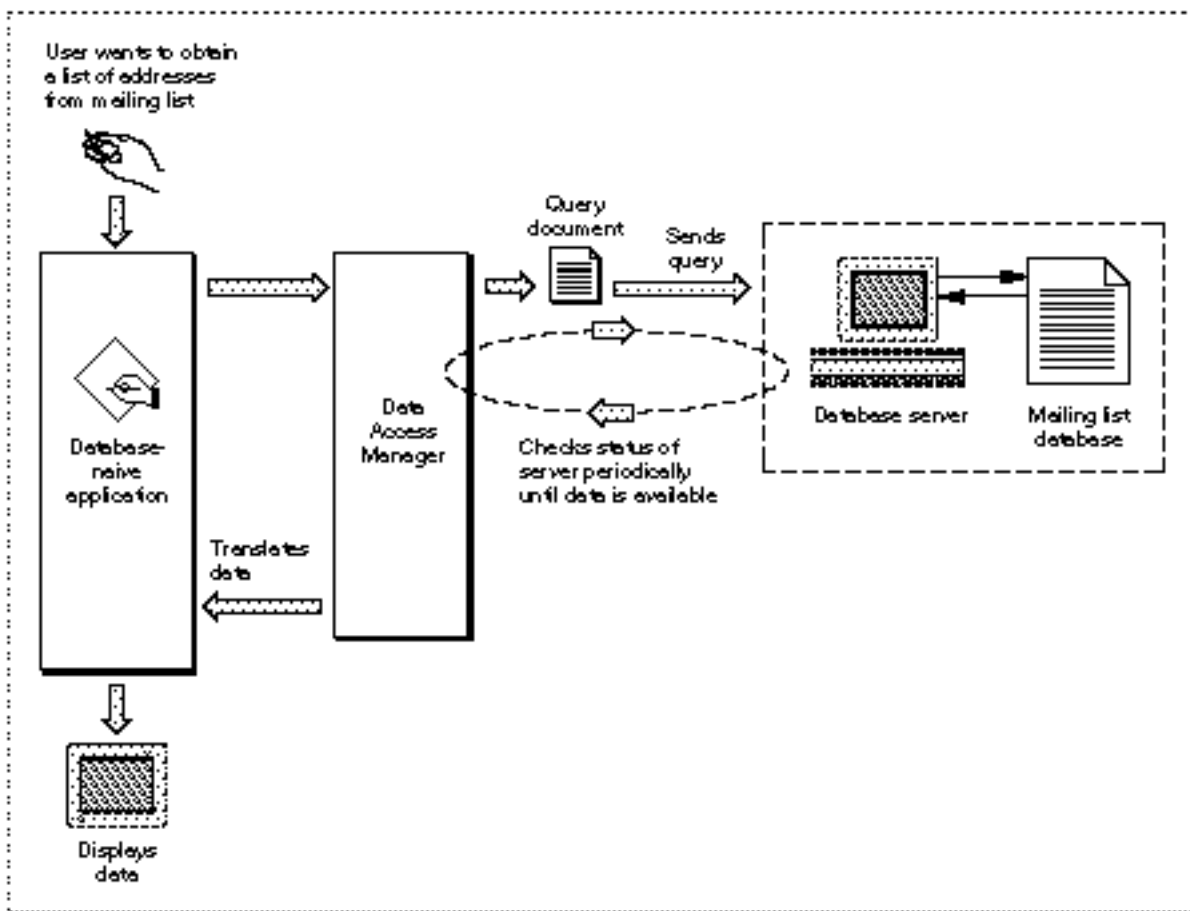
Sending a Query Through the High-Level Interface

To obtain a list of addresses from the mailing list, the marketing manager chooses the Open Query menu command from the File menu in her application. From the list of query documents displayed, she chooses one named Rubber Duck Address List.

The application calls the Data Access Manager function DBGetNewQuery, specifying the resource ID of the query ('qrsc') resource in the Rubber Duck Address List query document. The DBGetNewQuery function creates a query record and a partial query from the information in the query resource. The partial query specifies the type of data (character strings) and the columns from which the data items should come (the name and address columns). The partial query lacks some specific data (the rows that should be searched) that is needed to complete the search criteria.

Next, the application calls the DBStartQuery function, which in turn calls the query definition function in the query document. The query definition function displays a dialog box that asks for the purchase dates to search. When the marketing manager types in the requested information and clicks OK, the query definition function adds the data to the partial query in memory. The query is now ready to be executed.

Next, the DBStartQuery function sends the query to the Data Access Language database extension, and the database extension sends the query over a communications network to the remote Data Access Language server. Finally, the DBStartQuery function commands the Data Access Language server to execute the query.

Retrieving Data Through the High-Level Interface

When the application is ready to retrieve the data that it requested from the database, the application calls the DBGetQueryResults function. This function determines when the data is available, retrieves it from the data server, and places the data in a record in memory. The application can then call the DBResultsToText function, which uses routines called **result handlers** to convert each data item to a character string. The DBResultsToText function passes to the application a handle to the converted data. The application then displays the list of customers for the marketing manager.

Data items and result handlers are described in "Processing Query Results" beginning on page 12-36.

The Low-Level Interface

A database-aware application communicates through the low-level interface of the Data Access Manager. You can use the low-level interface to

n initiate communication with the data server, sending the user name, password, and other information to the data server

n send a query to the data server

n execute the query that you have sent to the data server

n halt execution of the query

n return status and errors from the data server

n send data to the data source

n retrieve data from the data source

For example, suppose once again that a company that makes rubber ducks has a mailing list of all of its customers in a database on a minicomputer, and the database has a Data Access Language interface. This time, suppose the Macintosh application the marketing manager is using calls low-level Data Access Manager routines to communicate with the remote database server. Figure 12-3 illustrates the use of the low-level interface. Notice that if you use the high-level interface (Figure 12-2), the query document and the Data Access Manager prepare the query, send the query, retrieve the query results, and translate the data for you. If you use the low-level interface, however, you must perform these functions yourself.

Figure 12-3   Using low-level Data Access Manager routines
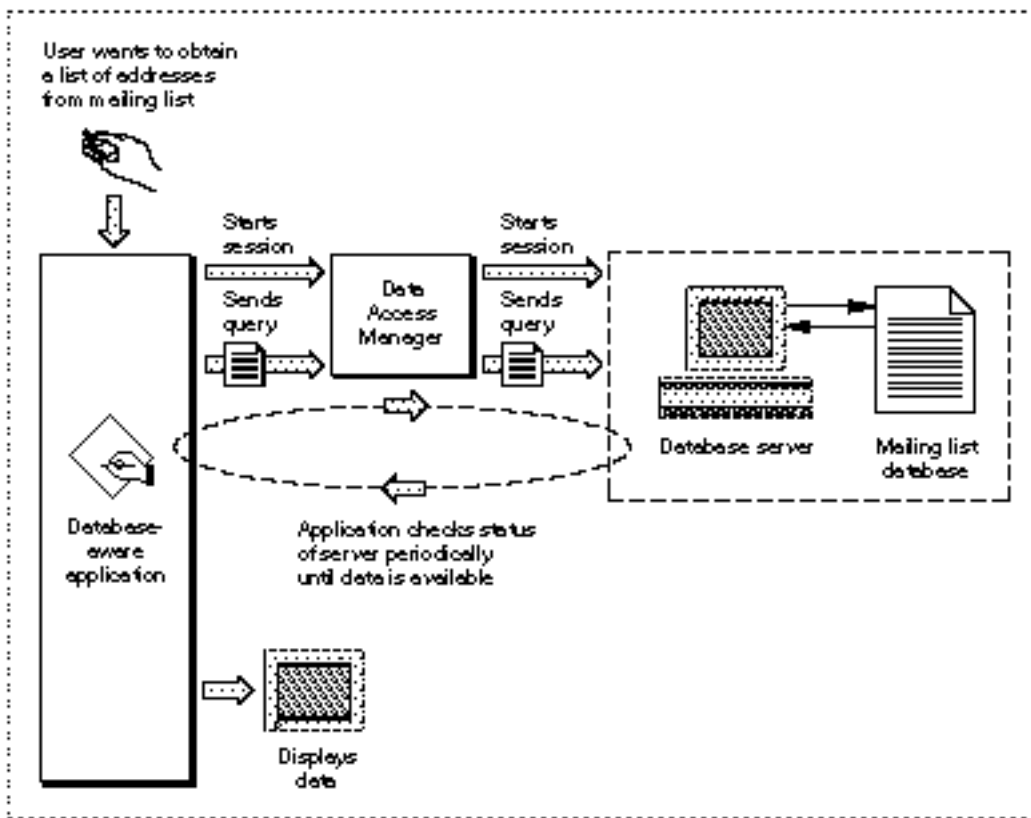
Sending a Query Through the Low-Level Interface

To update the mailing list with a new address for customer Marvin M., the marketing manager enters the new address into her application. The application prepares a Data Access Language statement (a query) that specifies the type of data (a character string), the column into which the data item should go (the address column), the row to be modified (the Marvin M. row), plus the actual data the application wishes to send (Marvin M.'s address). The application then passes this query to the Data Access

Manager using the low-level interface. (The application can send the query in several pieces or all at once.) The Data Access Manager sends the query to the Data Access Language database extension in the Macintosh computer, and the database extension sends the query to the remote Data Access Language server.

Retrieving Data Through the Low-Level Interface

Once the query begins executing, the application can periodically check with the data server to determine whether the data is ready (Figure 12-3). When the data is available, the application must retrieve it one data item at a time. An application that uses the low-level interface must determine the data type of each data item, convert the data into a format that is meaningful to the user, and store the data in memory allocated by the application. Data types are described in "Getting Query Results" beginning on page 12-36.

Note that neither the Data Access Manager nor the DAL database extension reads, modifies, or acts on the query that an application sends to the data server. The data server does execute the query, causing the data source to accept new data or prepare data for the application. To use the low-level interface to communicate with a data server, your application must be capable of preparing a query that can be executed by the data server.

Comparison of the High-Level and Low-Level Interfaces

An application that uses the low-level interface to send a query to the data server must prepare the query, initiate communication with the data server, send the query to the data server, and execute the query. If it requested data to be returned, the application must determine when the data is ready, and retrieve the data one item at a time. Each step in this process requires calling one or more low-level routines.

The high-level interface between the Data Access Manager and the application, in contrast, consists of only a few routines, each of which might call several low-level routines to accomplish its tasks. For example, a single high-level function can call the query definition function, initiate communication with the data server, send the query to the data server, and execute the query.

Because the high-level interface is very easy to use and requires no specific knowledge of the data source or database server, you can add high-level data access to your application very easily. Then, whenever someone provides a query document for use with a specific data server, the user can take advantage of the data access capability included in your application. However, because there is no way for a query document to verify that data *sent* to a data source has been successfully received, it is recommended that you use the low-level interface to send data to a data source or update data in a data source.

Although in concept the low-level routines and high-level routines serve separate purposes, there is nothing to prevent you from using calls to both in a single application. For example, you might use low-level routines to send a query to a data server and high-level routines to read the results and convert them to text.

Using the Data Access Manager

There are at least three different ways in which you can use the Data Access Manager to communicate with a data source. You can

n use low-level interface routines to send queries and retrieve data from the data source. In this case, your application must be capable of preparing a query in a language appropriate for the data server.

n use high-level interface routines to send queries and retrieve data from the data source. In this case, you must have one or more query documents provided by another application.

n create your own query documents and use high-level interface routines to send queries and retrieve data from the data source. In this case, your application must be capable of preparing a query, but it can use the same query repeatedly once it has been prepared.

This section describes how to use the high-level and low-level interfaces to the Data Access Manager to send queries to a data server. This section also describes how to call Data Access Manager functions asynchronously, how to determine the status of the high-level functions at various points in their execution (and cancel execution if you so desire), how to obtain information about Data Access Manager sessions that are in progress, and how to retrieve query results and convert them to text.

Executing Routines Asynchronously

All of the Data Access Manager low-level routines and some of the high-level routines can execute asynchronously—that is, the routine returns control to your application before the routine has completed execution. Your application must call the Event Manager's `WaitNextEvent` function periodically to allow an asynchronous routine to complete execution.

Note
The database extension is responsible for implementing asynchronous execution of Data Access Manager routines. For example, if you call the `DBSend` function to send a query to a data server, and the database extension calls a device driver, the database extension can return control to your application as soon as the device driver has placed its routine in the driver input/output (I/O) queue. If you attempt to execute a routine asynchronously and the database extension that the user has selected does not support asynchronous execution, the routine returns a result code of `rcDBAsyncNotSupp` and terminates execution. u

All Data Access Manager routines that can execute asynchronously take as a parameter a pointer to a parameter block known as the *asynchronous parameter block.* If the value of this pointer is `NIL`, the function is executed synchronously—that is, the routine does not return control to your application until execution is complete.

General Guidelines for the User Interface

When you use the Data Access Manager to provide data access, you should keep two important principles in mind: keep the user in control, and provide feedback to the user.

Keep the User in Control

When designing a data access feature or application, keep in mind that the user should have as much access to the Macintosh computer's abilities as possible. Design your application so that most of the data access process happens in the background. Call the Data Access Manager asynchronously whenever the database extension you are using supports asynchronous calls. Because data retrieval queries can take minutes or even hours to complete, they should always run in the background.

After issuing a query, return control of the computer to users so that they may work on other tasks or switch to other applications while the query runs. Whenever a background task requires the user's attention, follow the suggestions in *Macintosh Human Interface Guidelines* regarding user notification. A background task should never take control from the user by posting an alert box in front of the active application's windows. Any message that you post should identify the query that requires attention. For example, an alert box might display the message "The query Get Employee Information was canceled because the connection was unexpectedly broken."

If your application allows more than one simultaneous connection to data sources or allows more than one query document to run, provide a modeless window that lists the open connections and queries, displays the status of each, and allows the user to cancel them if necessary.

Allow the user to limit the amount of disk space that must remain free after any transaction. For example, a user may wish to specify that 1 MB of space always be free. Cancel any transaction that would exceed the user's limit and notify the user.

Provide Feedback to the User

Keep the user informed about status, progress, and error conditions, and allow the user to cancel an interaction whenever possible. Inform the user before the application becomes modal and the computer becomes unavailable. Use the spinning beach ball cursor or the animated wristwatch cursor to indicate a process that takes several seconds to complete. Use a dialog box to indicate any process that lasts longer than a few seconds. For example, connecting to a remote database could take a couple of minutes. In this case include a Cancel button in the dialog box so that the user can cancel the operation. When possible, display a progress indicator to show how long a process lasts. Warn the user before doing anything potentially dangerous or irreversible, such as deleting all of a user's data files to replace them with data retrieved from a data source.

When a data retrieval query terminates prematurely, make the retrieved data available to the user but warn the user that it is incomplete. The user can then evaluate the partial data before deciding whether to run the query again.

Using the High-Level Interface

Use the high-level interface to the Data Access Manager if you want to use a query document to do the work of communicating with a data source. You can use the high-level interface to open a query document, execute the query definition function in the query document, establish communication (initiate a session) with a data server, send the query to the data server, execute the query, retrieve any data requested by the query, and convert the retrieved data to text. Although two or three high-level routines accomplish most of these tasks, you may need to call a few low-level routines as well to control a session with a data server.

Applications that implement this type of data access must provide user control and feedback as described in "General Guidelines for the User Interface" on page 12-13. In addition, you should include an Open Query command in the File menu. The Open Query command is equivalent to the Open (file) command in meaning. When the user chooses this command, display an open file dialog box filtered to show only query documents (file type `'qery'`). The user can then select the desired query document. The query document contains the query to be sent to the data source. Depending on the type of query, the data source could receive information, send back information, report the status of the data source, or perform some other task.

Figure 12-4 is a flowchart of a typical session using the high-level interface. As Figure 12-4 illustrates, you must follow this procedure to use the high-level interface:

1. Call the `InitDBPack` function to initialize the Data Access Manager.

2. Select the query document that you want to use and determine the resource ID of the `'qrsc'` resource in that query document. You can use any method you like to select the query document. One possibility is to use the `StandardGetFile` procedure to let the user select the query document. A query document should contain only one `'qrsc'` resource; you can then use the Resource Manager to determine the resource ID of the `'qrsc'` resource in the document that the user selected. For further information, see the description of the `StandardGetFile` procedure in the chapter "Standard File Package" in *Inside Macintosh: Files* and the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox.*

3. Call the `DBGetNewQuery` function. The `DBGetNewQuery` function creates in memory a data structure called a *query record* from the `'qrsc'` resource that you specify.

4. Call the `DBStartQuery` function specifying the handle to the query record that you created with the `DBGetNewQuery` function (step 3).

   You should also provide the `DBStartQuery` function with a handle to your status routine. A **status routine** is a routine that you provide to update windows, check the results of the low-level calls made by the `DBStartQuery` and `DBGetQueryResults` functions, and cancel execution of these functions when you consider it appropriate to do so.

Figure 12-4   A flowchart of a session using the high-level interface
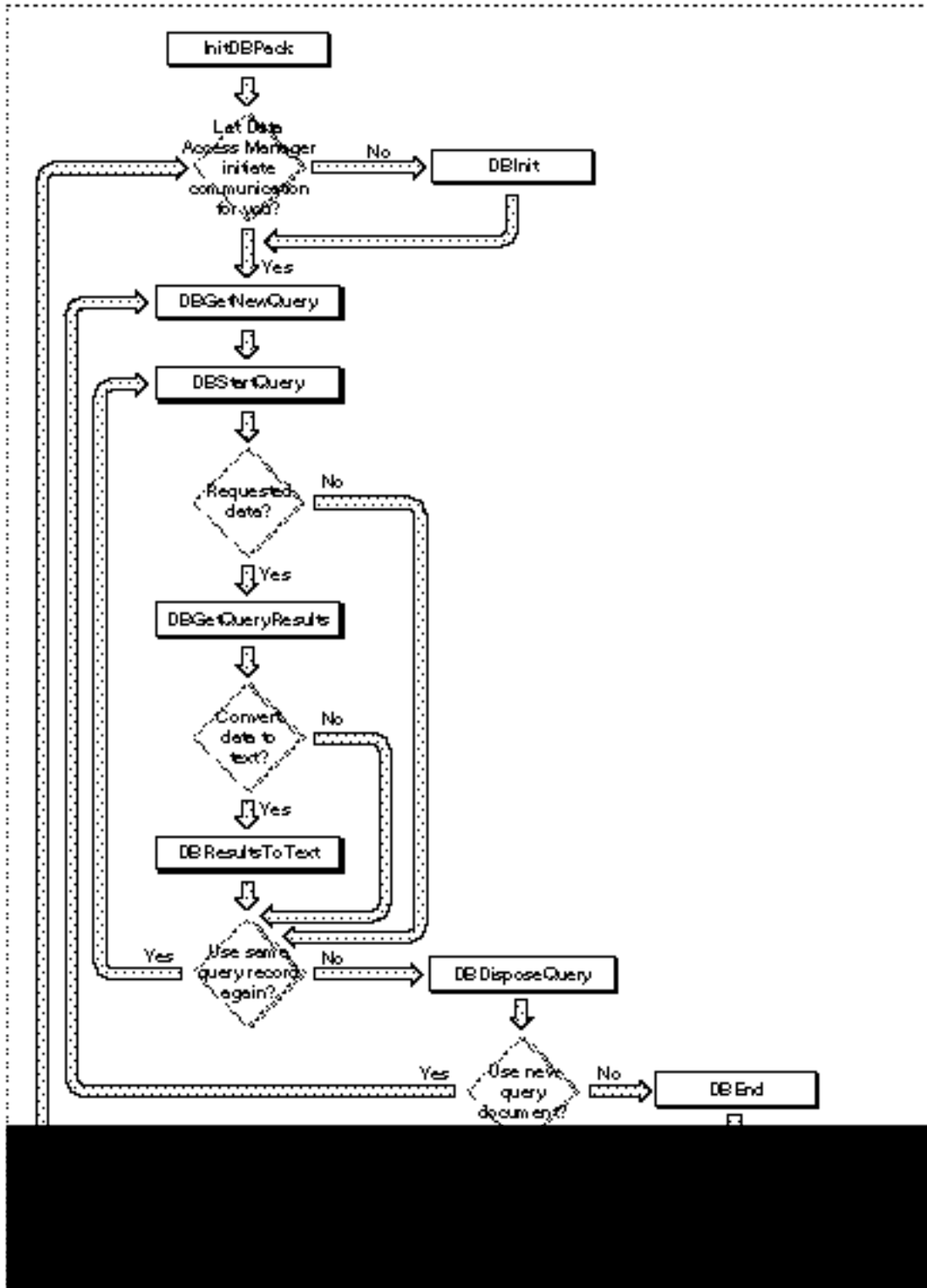
The `DBStartQuery` function calls the query definition function (if any) referred to by the query record. The query definition function can prompt the user for information and modify the query record.

After the query definition function has completed execution, the `DBStartQuery` function calls your status routine so that you can update your windows if necessary. The `DBStartQuery` function then checks whether communication has been established with the data server. If not, it calls your status routine so that you can display a status dialog box and then calls the `DBInit` function to establish communication (initiate a session) with the data server. The `DBStartQuery` function obtains the values it needs for the `DBInit` function parameters from the query record. When the `DBInit` function completes execution, the `DBStartQuery` function calls your status routine again.

The `DBInit` function returns an identification number, called a **session ID.** This session ID is unique; no other current session, for any database extension, has the same session ID. You must specify the session ID any time you want to send data to or retrieve data from this session. If you prefer, you can use the `DBInit` function to establish communication before you call the `DBStartQuery` function. In that case, you must specify the session ID as an input parameter to the `DBStartQuery` function. See "Using the Low-Level Interface" beginning on page 12-28 for more information on using the `DBInit` function.

Once communication has been established, the `DBStartQuery` function calls the `DBSend` function to send the data server the query specified by the query record. When the `DBSend` function has completed execution, the `DBStartQuery` function calls your status routine. Finally, the `DBStartQuery` function uses the `DBExec` function to execute the query. The `DBStartQuery` function calls your status routine after the `DBExec` function has completed execution (that is, the query has started executing and the `DBExec` function has returned control to the `DBStartQuery` function) and again just before the `DBStartQuery` function completes execution.

5. If you requested data and want to know when the data is available, but do not want to retrieve the data immediately, you can call the `DBState` function. This function tells you when the data server has finished executing the query, but it does not retrieve the data. If you requested data and want to retrieve it as soon as it is available, you do not have to call the `DBState` function; go to step 6 instead.

If you did not request data, you can use the `DBState` function to determine the status of the query. When the data server has finished executing the query, skip to step 8.

6. Call the `DBGetQueryResults` function. If the query has not finished executing, this function returns the `rcDBExec` result code. If the query has finished executing, the `DBGetQueryResults` function calls the `DBGetItem` function repeatedly until the data server has returned all of the data available.

The `DBGetQueryResults` function puts the returned data into a record that contains handles to arrays that contain the data, the type of data in each column, and the length of each data item. The Data Access Manager allocates the memory for this data in the application heap.

The `DBGetQueryResults` function calls your status routine after it retrieves each data item. You can use this opportunity to display the data item for the user and to give the user the opportunity to cancel execution of the function. The `DBGetQueryResults` function also calls your status routine just before completing execution, so that you can dispose of any memory allocated by the status routine, remove any dialog box that you displayed, and update your windows if necessary.

To convert the returned data to text, go to the next step. If you do not want to convert the returned data to text, skip to step 9.

7. Call the `DBResultsToText` function. This function calls a result handler function for each data type. The result handler converts the data to text, places it in a buffer, and returns a handle to the buffer. Some result handlers are provided with the Data Access Manager; you can provide as many with your application as you wish. Result handlers are discussed in "Converting Query Results to Text" beginning on page 12-42.

8. If you are finished using the query record, call the `DBDisposeQuery` function to dispose of the query record and free all the memory associated with the query record. If you want to reuse the same query, return to step 5. You should close the query document when you are finished using it.

If you want to use a new query document, return to step 3.

9. When you are finished using the data source, you must use the `DBEnd` function to terminate the session. You must call the `DBEnd` function after the `DBInit` function has returned a nonzero session ID, even if it also returned an error.

Listing 12-1 illustrates the use of the high-level interface. This code initiates a session with a remote database, lets the user select a query document to execute, opens the selected file, finds a `'qsrc'` resource, and creates a query record. Next, it executes the query, checks the status of the remote database server, retrieves the data when it's available, and converts this data to text. When the query has finished executing, the code disposes of the query record, ends the session, and closes the user-selected query document. In general, there's no reason why there can't be multiple sessions open at once. You can identify each session by its session ID. Listing 12-1 shows just one session.

Listing 12-1 assumes that you are using a database extension that supports asynchronous execution of Data Access Manager routines. This listing shows just one possible approach to sending a query and retrieving data asynchronously.

Listing 12-1   Using the high-level interface

```pascal
PROCEDURE MyHiLevel(VAR rr: ResultsRecord; myTextHdl: Handle;
                    VAR thisSession: LongInt; VAR sessErr: OSErr);
TYPE
   {define a record to include space for the current value in }
   { A5 so a completion routine can find it}
   CRRec = RECORD
      QPB: DBAsyncParamBlockRec;       {the parameter block}
      appsA5: LongInt;                 {append A5 to the }
                                       { parameter block}
   END;
   CRRecPtr = ^CRRec;

VAR
   StartPB, GetQRPB:                      CRRec;
   SFR:                                   StandardFileReply;
   packErr, startQErr, getQErr, disposeQErr:   OSErr;
   getnewQErr, gStartQErr, gGetQRErr:     OSErr;
   endErr, fsopenErr, fscloseErr, resultsErr:   OSErr;
   gStart, gQueryResults:                 Boolean;
   qrscHandle:                            Handle;
   rsrcID:                                Integer;
   rsrcType:                              ResType;
   rsrcName:                              Str255;
   myQHandle:                             QueryHandle;
   savedResFile:                          Integer;
   typeList:                              SFTypeList;
   fsRefNum:                              Integer;

FUNCTION GetQPB: CRRecPtr;
   INLINE $2E88;                       {MOVE.L  A0,(SP)}

BEGIN
   gStart := FALSE;
   gQueryResults := FALSE;
   sessErr := noErr;               {assume everything went fine}
   packErr := InitDBPack;          {initialize the Data Access Mgr}
   {display a dialog box to let the user pick a query document}
   typeList[0] := 'qery';
   StandardGetFile(NIL, 1, typeList, SFR);
   IF SFR.sfGood = TRUE THEN
      fsopenErr := FSpOpenRF(SFR.sfFile, fsCurPerm, fsRefNum);
   IF (fsopenErr <> noErr) OR (SFR.sfGood = FALSE) THEN
```

```
BEGIN
   sessErr := fsopenErrOrUserCanceled;
   EXIT(MyHiLevel);
END;
savedResFile := CurResFile;    {save current resource file}
UseResFile(fsRefNum);          {get query info from here}
{a query document should have only one 'qrsc' resource}
qrscHandle := Get1IndResource('qrsc', 1);
IF ResError <> noErr THEN
BEGIN
   sessErr := ResError;
   EXIT(MyHiLevel);
END;
{get the resource ID of the 'qrsc' resource}
GetResInfo(qrscHandle, rsrcID, rsrcType, rsrcName);
{create a query record using the resource ID}
getnewQErr := DBGetNewQuery(rsrcID, myQHandle);
IF getnewQErr <> noErr THEN
BEGIN
   sessErr := getnewQErr;
   endErr := DBEnd(thisSession, NIL);
   EXIT(MyHiLevel);
END;
StartPB.QPB.completionProc := @MyStartCompRoutine;
StartPB.appsA5 := SetCurrentA5;      {save this for the }
                                     { completion routine}
{MyStartStatus is a status routine that handles messages sent }
{ by the DBStartQuery function when it calls a low-level }
{ function}
startQErr := DBStartQuery(thisSession, myQHandle,
                          @MyStartStatus, @StartPB);
IF startQErr <> noErr THEN
BEGIN
   sessErr := startQErr;
   IF thisSession <> 0 THEN
      endErr := DBEnd(thisSession, NIL);
   EXIT(MyHiLevel);
END;
WHILE NOT gStart DO  {while waiting for gStart to go TRUE, }
BEGIN                { MyGoDoSomething calls WaitNextEvent }
   MyGoDoSomething;  { to give other routines a chance to run}
END;     {while}
{the DBStartQuery call has completed}
```

```
IF gStartQErr <> noErr THEN
BEGIN
   sessErr := gStartQErr;
   IF thisSession <> 0 THEN
      endErr := DBEnd(thisSession, NIL);
   EXIT(MyHiLevel);
END;
GetQRPB.QPB.completionProc := @MyGetQRCompRoutine;
GetQRPB.appsA5 := SetCurrentA5;          {save this for the }
                                         { completion routine}
{MyGetQRStatus is a status routine that handles messages sent }
{ by the DBGetQueryResults function when it calls a low-level }
{ function.}
getQErr := DBGetQueryResults(thisSession, rr, kDBWaitForever,
                                @MyGetQRStatus, @GetQRPB);
IF getQErr <> noErr THEN
BEGIN
   sessErr := getQErr;
   endErr := DBEnd(thisSession, NIL);
   EXIT(MyHiLevel);
END;
WHILE NOT gQueryResults DO
BEGIN
   MyGoDoSomething;
END; {while}
{The DBGetQueryResults call has completed. Assuming the call }
{ completed successfully, you may want to convert the }
{ retrieved data to text, return memory you have borrowed, }
{ and end the session.}
IF gGetQRErr <> noErr THEN
BEGIN
   sessErr := gGetQRErr;
   endErr := DBEnd(thisSession, NIL);
   EXIT(MyHiLevel);
END;
{the data has been retrieved; convert it to text}
resultsErr := DBResultsToText(rr, myTextHdl);
{The current query is finished.  You can elect to execute }
{ the next 'qrsc' resource of the file, or select another }
{ query document.  This example just returns to the caller.}
disposeQErr := DBDisposeQuery(myQHandle);
UseResFile(savedResFile);{restore current resource file}
fscloseErr := FSClose(fsRefNum); {close the query document}
```

```
IF fscloseErr <> noErr THEN
    DoError(fscloseErr);
endErr := DBEnd(thisSession, NIL);
IF endErr <> noErr THEN
    DoError(endErr);
END;
```

Listing 12-2 shows the completion routines `MyStartCompRoutine` and
`MyGetQRCompRoutine` used in Listing 12-1.

Listing 12-2   Two completion routines

```
PROCEDURE MyStartCompRoutine(aCRRecPtr: CRRecPtr);
VAR
    curA5:    LongInt;
BEGIN
    aCRRecPtr := GetQPB;                        {get the param block}
    curA5 := SetA5(aCRRecPtr^.appsA5);  {set A5 to the app's A5}
    gStart := TRUE;                             {query has been started}
    gStartQErr := aCRRecPtr^.QPB.result;{send back result code}
    {do whatever else you want to do}
    curA5 := SetA5(curA5);{restore original A5}
END;  {MyStartCompRoutine}

PROCEDURE MyGetQRCompRoutine(aCRRecPtr: CRRecPtr);
VAR
    curA5:    LongInt;
BEGIN
    aCRRecPtr := GetQPB;                        {get the param block}
    curA5 := SetA5(aCRRecPtr^.appsA5);  {set A5 to the app's A5}
    gQueryResults := TRUE;{query results are complete}
    gGetQRErr := aCRRecPtr^.QPB.result; {send back the result code}
    {do whatever else you want to do}
    curA5 := SetA5(curA5);                      {restore original A5}
END;  {MyGetQRCompRoutine}
```

The next section provides information about status routines.

Writing a Status Routine for High-Level Functions

Both of the two main high-level functions, DBStartQuery and DBGetQueryResults, call low-level functions repeatedly. After each time they call a low-level function, these high-level functions call a routine that you provide, called a *status routine.* Your status routine can check the result code returned by the low-level function and can cancel execution of the high-level function before it calls the next low-level function. Your status routine can also update your application's windows after the DBStartQuery function has displayed a dialog box.

You provide a pointer to your status routine in the statusProc parameter to the DBStartQuery and DBGetQueryResults functions.

Here is a function declaration for a status routine:

```
FUNCTION MyStatusFunc (message: Integer; result: OSErr;
                       dataLen: Integer; dataPlaces: Integer;
                       dataFlags: Integer; dataType: DBType;
                       dataPtr: Ptr): Boolean;
```

Your status routine should return a value of TRUE if you want to continue execution of the DBStartQuery or DBGetQueryResults function, or a value of FALSE if you want to cancel execution of the function. In the latter case, the high-level function returns the userCanceledErr result code.

Note

If you call the DBStartQuery or DBGetQueryResults function asynchronously, you cannot depend on the A5 register containing a pointer to your application's global variables when the Data Access Manager calls your status routine. u

The message parameter tells your status routine the current status of the high-level function that called it. The possible values for the message parameter depend on which function called your routine.

The value of the result parameter depends on the value of the message parameter, as summarized in the following list:

Message	Result
kDBUpdateWind	**0**
kDBAboutToInit	**0**
kDBInitComplete	**Result of** DBInit
kDBSendComplete	**Result of** DBSend
kDBExecComplete	**Result of** DBExec
kDBStartQueryComplete	**Result of** DBStartQuery
kDBGetItemComplete	**Result of** DBGetItem
kDBGetQueryResultsComplete	**Result of** DBGetQueryResults

The dataLen, dataPlaces, dataFlags, dataType, and dataPtr parameters are returned only by the DBGetQueryResults function, and only when the message parameter equals kDBGetItemComplete. When the DBGetQueryResults function calls your status routine with this message, the dataLen, dataPlaces, and dataType parameters contain the length, decimal places, and type of the data item retrieved, respectively, and the dataPtr parameter contains a pointer to the data item.

The least significant bit of the dataFlags parameter is set to 1 if the data item is in the last column of the row. The third bit of the dataFlags parameter is set to 1 if the data item is NULL. You can use this information, for example, to check whether the data meets some criteria of interest to the user, or to display each data item as the DBGetItem function receives it. You can use the constants kDBLastColFlag and kDBNullFlag to test for these flag bits.

The DBGetQueryResults function returns a results record, which contains a handle to the retrieved data. The address in the dataPtr parameter points inside the array specified by this handle. Because the dataPtr parameter is not a pointer to a block of memory allocated by the Memory Manager, but just a pointer to a location inside such a block, you cannot use this pointer in any Memory Manager routines (such as the GetPtrSize function). Note also that you cannot rely on this pointer remaining valid after you return control to the DBGetQueryResults function.

The DBStartQuery function can send to your status routine the following constants in the message parameter:

```
CONST {DBStartQuery status messages}
      kDBUpdateWind          = 0;      {update windows}
      kDBAboutToInit         = 1;      {about to call DBInit}
      kDBInitComplete        = 2;      {DBInit has completed}
      kDBSendComplete        = 3;      {DBSend has completed}
      kDBExecComplete        = 4;      {DBExec has completed}
      kDBStartQueryComplete  = 5;      {DBStartQuery is about to }
                                       { complete}
```

DBStartQuery message constant	Meaning
kDBUpdateWind	The DBStartQuery function has just called a query definition function. Your status routine should process any update events that your application has received for its windows.
kDBAboutToInit	The DBStartQuery function is about to call the DBInit function to initiate a session with a data server. Because initiating the session might involve establishing communication over a network, and because in some circumstances the execution of a query can tie up the user's computer for some length of time, you might want to display a dialog box giving the user the option of canceling execution at this time.

continued

`DBStartQuery` message constant	Meaning (continued)
`kDBInitComplete`	The `DBInit` function has completed execution. When the `DBStartQuery` function calls your status routine with this message, the `result` parameter contains the result code returned by the `DBInit` function. If the `DBInit` function returns the `noErr` result code, the `DBStartQuery` function calls the `DBSend` function next. If the `DBInit` function returns any other result code, you can display a dialog box informing the user of the problem before returning control to the `DBStartQuery` function. The `DBStartQuery` function then returns an error code and stops execution.
`kDBSendComplete`	The `DBSend` function has completed execution. When the `DBStartQuery` function calls your status routine with this message, the `result` parameter contains the result code returned by the `DBSend` function. If the `DBSend` function returns the `noErr` result code, the `DBStartQuery` function calls the `DBExec` function next. If the `DBSend` function returns any other result code, you can display a dialog box informing the user of the problem before returning control to the `DBStartQuery` function. The `DBStartQuery` function then returns an error code and stops execution.
`kDBExecComplete`	The `DBExec` function has completed execution. When the `DBStartQuery` function calls your status routine with this message, the `result` parameter contains the result code returned by the `DBExec` function. If the `DBExec` function returns the `noErr` result code, the `DBStartQuery` function returns control to your application next. If the `DBExec` function returns any other result code, you can display a dialog box informing the user of the problem before returning control to the `DBStartQuery` function. The `DBStartQuery` function then returns an error code and stops execution.
`kDBStartQueryComplete`	The `DBStartQuery` function has completed execution and is about to return control to your application. The function result is in the `result` parameter passed to your status routine. Your status routine can use this opportunity to perform any final tasks, such as disposing of memory that it allocated or removing from the screen any dialog box that it displayed.

The `DBGetQueryResults` function can send to your status routine the following constants in the `message` parameter:

```
CONST {DBGetQueryResults status messages}
     kDBGetItemComplete        = 6;  {DBGetItem has completed}
     kDBGetQueryResultsComplete = 7; {DBGetQueryResults has }
                                     { completed}
```

DBGetQueryResults message constant	Meaning
kDBGetItemComplete	The DBGetItem function has completed execution. When the DBGetQueryResults function calls your status routine with this message, the result parameter contains the result code returned by the DBGetItem function. The DBGetQueryResults function also returns values for the dataLen, dataPlaces, dataType, dataFlags, anddataPtr parameters, as discussed earlier in this section.
	For each data item that it retrieves, the DBGetQueryResults function calls the DBGetItem function twice: once to obtain information about the next data item and once to retrieve the data item. The DBGetQueryResults function calls your status routine only after calling the DBGetItem function to retrieve a data item.
	If your status routine returns a function result of FALSE in response to the kDBGetItemComplete message, the results record returned by the DBGetQueryResults function to your application contains data through the last full row retrieved.
	Data types and results records are described in "Getting Query Results" beginning on page 12-36.
kDBGetQueryResultsComplete	The DBGetQueryResults function has completed execution and is about to return control to your application. The function result is in the result parameter passed to your status routine. Your status routine can use this opportunity to perform any final tasks, such as disposing of memory that it allocated or removing from the screen any dialog box that it displayed.

Listing 12-3 shows a status routine for the DBStartQuery function. This routine updates the application's windows in response to the kDBUpdateWind message, displays a dialog box giving the user the option of canceling before the data access is initiated, and checks the results of calls to the DBInit, DBSend, and DBExec functions. If one of these functions returns an error, the status routine displays a dialog box describing the error.

Listing 12-3   A sample status routine

```
FUNCTION MyStartStatus(message: Integer; result: OSErr;
                       dataLen: Integer; dataPlaces: Integer;
                       dataFlags: Integer; dataType: DBType;
                       dataPtr: Ptr): Boolean;
VAR
   myString:    Str255;
   continue:    Boolean;
BEGIN
   continue := TRUE; {assume user wants to continue with query}
   CASE message OF
   kDBUpdateWind:     {a qdef function has just been called; }
      BEGIN           { handle activate and update events}
         MyDoActivate;  {find and handle activate events}
         MyDoUpdate;    {find and handle update events}
      END;  {kDBUpdateWind}
   kDBAboutToInit:    {about to initiate a session}
      BEGIN {MyDisplayDialog displays a dialog box. The value }
            { returned in the continue variable indicates }
            { whether DBStartQuery should continue.}
         myString := 'The Data Access Manager is about to open a
                      session.  This could take a while.  Do you
                      want to continue?';
         MyDisplayDialog(@myString, continue);
      END;  {kDBAboutToInit}
   kDBInitComplete:  {the DBInit function has completed execution}
      BEGIN
         IF result <> noErr THEN {if there's an error, }
         BEGIN                   { let the user know what it is}
            CASE result OF
            rcDBError:
            BEGIN
               myString := 'The Data Access Manager was unable to
                            open the session.  Please check your
                            connections and try again later.';
               MyDisplayString(@myString);
            END;  {rcDBError}
            rcDBBadDDev:
            BEGIN
               myString := 'The Data Access Manager cannot find
                            the database extension file it needs to
                            open a session.  Check with your system
                            administrator for a copy of the file.';
```

```
            MyDisplayString(@myString);
         END;  {rcDBBadDDev}
      OTHERWISE
      BEGIN
         myString := 'The Data Access Manager was unable to
                      open the session.  The error code
                      returned was';
         MyDisplayError(@myString, result);
      END; {of otherwise}
   END;  {of CASE result}
   END; {of result <> noErr}
END; {kDBInitComplete}
kDBSendComplete:  {the DBSend function has completed execution}
   BEGIN
      {if there's an error, let the user know what it is}
      IF result <> noErr THEN
      BEGIN
         IF result = rcDBError THEN
         BEGIN
            myString := 'An error occurred while the Data
                         Access Manager was trying to send the
                         query. Please try again later.';
            MyDisplayString(@myString);
         END
         ELSE
         BEGIN
            myString := 'An error occurred while the Data
                         Access Manager was trying to send the
                         query. The error code returned was';
            MyDisplayError(@myString, result);
         END;
      END;  {of result <> noErr}
   END;  {kDBSendComplete}
kDBExecComplete:  {the DBExec function has completed execution}
   BEGIN
      IF result <> noErr THEN {if there's an error, }
      BEGIN                    { let the user know what it is}
         IF result = rcDBError THEN
         BEGIN
            myString := 'The Data Access Manager was unable to
                         execute the query. There may be a problem
                         with the query document or the database.
                         Check with your system administrator.';
```

```
            MyDisplayString(@myString);
        END
        ELSE
        BEGIN
            myString := 'An error occurred while the Data
                     Access Manager was trying to execute the
                     query.  The error code returned was';
            MyDisplayError(@myString, result);
        END;
    END;  {of result <> noErr}
  END;  {kDBExecComplete}
kDBStartQueryComplete:{the DBStartQuery function is about }
    BEGIN              { to return control to your application}
        {clean up memory and any dialog boxes left on the screen}
        MyCleanUpWindows;
    END;  {kDBStartQueryComplete}
  END; {CASE message}
  MyStartStatus := continue;
END;
```

Using the Low-Level Interface

You can use the low-level interface to establish communication (initiate a session) with a data server, send a query to the data server, execute the query, and retrieve any data requested by the query. You call one or more low-level routines to accomplish each of these tasks.

Applications that implement this type of data access must provide user control and feedback, as described in "General Guidelines for the User Interface" on page 12-13. When the data source is ready to return data, you can retrieve it all and then display it to the user, or you can display the data as it arrives. If the data arrives slowly, it's best to display it one record at a time as it arrives. This way the user can preview the data, decide if it's the desired information, and cancel the query if not.

Figure 12-5 is a flowchart of a typical session using the low-level interface. As Figure 12-5 illustrates, you must follow this procedure to use the low-level interface:

1. Call the `InitDBPack` function to initialize the Data Access Manager.

2. Call the `DBInit` function to establish communication with the data server. The `DBInit` function returns an identification number, called a *session ID.* This session ID is unique; no other current session, for any database extension, has the same session ID.

Figure 12-5   A flowchart of a session using the low-level interface
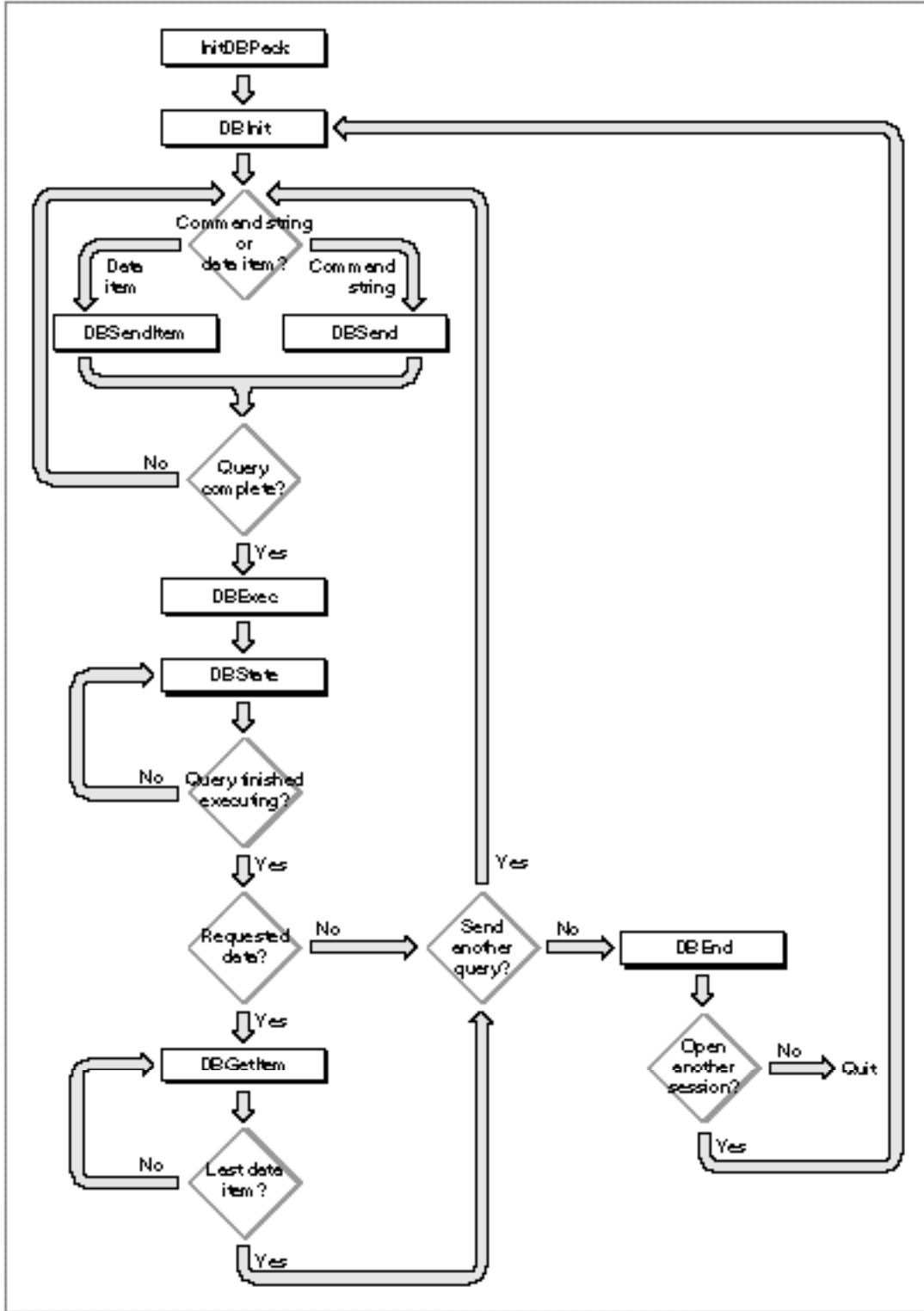
You must specify the session ID any time you want to send data to or retrieve data from this session.

The `DBInit` function requires as input parameters the name of the database extension and character strings for the host system, user name, password, and connection string. All of these parameters depend on the user and the user's computer system, including the specific database extension, host computer, data server, and database management software in use. You will not know the user name and password when you are writing an application, and you might not know the values of any of these parameters. Therefore, you must display a dialog box that prompts the user for the necessary information.

Depending on the database extension you are using, the `DBInit` function might return a session ID of zero if it fails to initiate a session, or it might return a nonzero session ID and a result code other than `noErr`. In the latter case, you can pass the session ID to the `DBGetErr` function to determine the cause of the error. If the `DBInit` function returns a nonzero session ID and a result code other than `noErr`, you must call the `DBEnd` function before making another attempt to open the session.

3. Prepare a query, and send it to the data server by calling the `DBSend` and `DBSendItem` functions one or more times.

An application that uses the low-level interface must be capable of creating a query for the data server in the language and format required by that data server.

The `DBSend` function sends a query or a portion of a query to the data server. The data server appends this portion of the query to any portion you sent previously. Because the Data Access Manager and data server do not modify the string you send in any way, they do not insert any delimiter between fragments of queries that you send to the data server. If you want a blank or a semicolon to be included between query fragments, or if you want to use return characters to divide the query into lines of text, you must include them in the character string that you send with the `DBSend` function. The data string that you send with the `DBSend` function can be any length up to 64 KB.

The `DBSendItem` function sends a single data item to the data server. Use the `DBSendItem` function to send data items to the data source in the same format as they are retrieved from the data source by the `DBGetItem` function. You must specify the data type as an input parameter and, for any data type that does not have an implied length, you must specify the length as well. The database extension or the data server (depending on how the system is implemented) converts the data item to a character string and appends it to the query, just as a query program fragment is appended to the query by the `DBSend` function.

You can call the `DBSend` and `DBSendItem` functions as many times as you wish to send your query to the data server.

Listing 12-4 sends the Data Access Language query fragment "`print 451+222;`" to the Data Access Language server.

Listing 12-4   Sending a query fragment

```
FUNCTION MySendFragment(sessID: LongInt): OSErr;
VAR
   value1:                        LongInt;
   value2:                        LongInt;
   text1, text2, text3:           Str15;
   text1Ptr, text2Ptr, text3Ptr: Ptr;
   rc:       OSErr;
BEGIN
   text1 := 'print ';
   value1 := 451;
   text2 := '+';
   value2 := 222;
   text3 := ';';
   MySetTextPtrs(text1, text1Ptr, text2, text2Ptr,
                 text3, text3Ptr);
   rc := DBSend (sessID, text1Ptr, LENGTH(text1), NIL);
   IF rc = noErr THEN
      rc := DBSendItem (sessID, typeInteger, 0, 0, 0,
                        Ptr(ORD(@value1)), NIL);
   IF rc = noErr THEN
      rc := DBSend (sessID, text2Ptr, LENGTH(text2), NIL);
   IF rc = noErr THEN
      rc := DBSendItem (sessID, typeInteger, 0, 0, 0,
                        Ptr(ORD(@value2)), NIL);
   IF rc = noErr THEN
      rc := DBSend (sessID, text3Ptr, LENGTH(text3), NIL);
   MySendFragment := rc;
END;
```

4. Use the `DBExec` function to initiate execution of the query.

   Depending on the way the system you are using is implemented, the `DBExec` function might return control to your application as soon as the query has begun execution.

5. Use the `DBState` function to determine the status of the data source.

   The `DBState` function tells you when the data server has finished executing the query you just sent. If you have requested data, the data server stores the data you requested but does not send it to your application until you request it explicitly. The `DBState` function tells you when the data is available; if data is available, go to step 6. If you wish to send another query, return to step 3. If you are finished using the data source, skip to step 7.

6. Call the `DBGetItem` function repeatedly to retrieve the data.

   The `DBGetItem` function retrieves the next data item from the data server. You can also use this function to obtain information about the next data item without retrieving the data. When you use the `DBGetItem` function to retrieve a data item, you must specify the location and size of the buffer into which the function is to place that item. If you know beforehand what kind of data to expect, you can allocate a buffer of the exact size you need. If you do not know what type of data to expect, you can first call the `DBGetItem` function with a `NIL` pointer to the data buffer. The `DBGetItem` function then returns information about the next data item without actually retrieving it. You can then allocate the appropriate buffer and call `DBGetItem` again.

   Alternatively, to avoid calling `DBGetItem` twice for each data item, you can allocate a buffer that you expect to be of sufficient size for any data item and call the `DBGetItem` function. If the buffer is not large enough for the data item, the `DBGetItem` function returns the `rcDBError` result code, but still returns information about the data item. You can then allocate the necessary buffer, call the `DBUnGetItem` function to go back one data item, and call the `DBGetItem` function again to retrieve the data item a second time.

   The `DBGetItem` function includes a `timeout` parameter that you can use to specify the maximum amount of time that the database extension should wait to receive results from the data server before canceling the command. If the database extension you are using does not support asynchronous execution of routines, you can use the `timeout` parameter to return control to your application while a query is executing. To use the `timeout` parameter in this way, call the `DBGetItem` function periodically, specifying a brief period of time for the `timeout` parameter. Your application can then retrieve the next data item as soon as execution of the query is complete without having to call the `DBState` function to determine when data is available. The `DBGetItem` function ignores the `timeout` parameter if you make an asynchronous call to this function.

7. When you are finished using the data source, you must use the `DBEnd` function to terminate the session. You must call the `DBEnd` function after the `DBInit` function has returned a nonzero session ID, even if it also returned an error.

The procedure in Listing 12-5 uses the low-level interface to send a Data Access Language routine to the Data Access Language server on a remote computer and then retrieves the results. The procedure initiates a session with a remote database and calls the `MySendFragment` routine (Listing 12-4) to send a query. Next, it executes the query, checks the status of the remote database server, and retrieves the data when it's available. This example retrieves only one data item. To retrieve more than one data item, put the data-retrieval code in a loop.

Listing 12-5 assumes that the database extension does not support asynchronous execution of Data Access Manager routines. For an example of asynchronous execution of routines, see Listing 12-1 beginning on page 12-18.

Listing 12-5   Using the low-level interface

```pascal
PROCEDURE MyLoLevel(VAR thisSession: LongInt; VAR sessErr: OSErr);
VAR
   theDDevName:                        Str63;
   theHost, theUser:                   Str255;
   thePasswd, theConnStr:              Str255;
   packErr, initErr, sendErr, execErr: OSErr;
   stateErr, getErr, endErr:           OSErr;
   myTimeout:                          LongInt;
   myType:                             DBType;
   len, places, flags:                 Integer;
   myBuffer:                           Ptr;
   myDataInfo:                         Boolean;
   myDataReturned:                     Boolean;
BEGIN
   sessErr := noErr;       {assume everything went fine}
   packErr := InitDBPack;  {init the Data Access Mgr}
   {Set up values for theDDevName, theHost, theUser, thePasswd, }
   { and theConnStr. You can display a dialog box prompting }
   { the user to supply some of these parameters.}
   theDDevName := 'DAL';
   theHost := 'The Host System Name';
   theUser := 'Joe User';
   thePasswd := 'secret';
   theConnStr := 'extra stuff as needed';
   initErr := DBInit(thisSession, theDDevName, theHost, theUser,
                     thePasswd, theConnStr, NIL);
   IF initErr <> noErr THEN
   BEGIN
      sessErr := initErr;
      IF thisSession <> 0 THEN endErr := DBEnd(thisSession, NIL);
      EXIT(MyLoLevel);
   END;
   {send a query or query fragment to the remote data server}
   sendErr := MySendFragment(thisSession);
   {If there's an error, then probably something went wrong with }
   { DBSend or DBSendItem.  Don't forget to end the session.}
   IF sendErr <> noErr THEN
   BEGIN
      sessErr := sendErr;
      endErr := DBEnd(thisSession, NIL);
      EXIT(MyLoLevel);
   END;
```

```
{The query has been sent.  This example assumes that }
{ the query will return data.}
execErr := DBExec(thisSession, NIL);
IF execErr = noErr THEN
BEGIN
   stateErr := rcDBExec;
   WHILE (stateErr = rcDBExec) DO
   BEGIN                {while waiting for stateErr <> rcDBExec, }
      MyGoDoSomething; { let other apps run}
      stateErr := DBState(thisSession, NIL);
   END;
   {DBState returned a result code other than rcDBExec. }
   { If it's rcDBValue, there are results to retrieve. }
   { Otherwise, it's probably an error.}
   IF stateErr = rcDBValue THEN
   BEGIN
      {call DBGetItem once to get info on the data item and }
      { call DBGetItem a second time to get the data item}
      myTimeout := 2*60;   {2*60 ticks = 2 seconds}
      myType := DBType(typeAnyType);
      myDataInfo := FALSE;
      WHILE NOT myDataInfo DO
      BEGIN
         getErr := DBGetItem(thisSession, myTimeout, myType,
                              len, places, flags, NIL, NIL);
         {If you timed out, then give up control.  When }
         { control returns, continue getting the info.}
         IF getErr = rcDBBreak THEN MyGoDoSomething
         ELSE IF (getErr = noErr) OR (getErr = rcDBValue) THEN
            myDataInfo := TRUE
         ELSE
         BEGIN
            sessErr := getErr;
            endErr := DBEnd(thisSession, NIL);
            EXIT(MyLoLevel);
         END;
      END; {while}
      {At this point, you may want to examine the info }
      { about the data item before calling DBGetItem a }
      { second time to actually retrieve it.}
      {MyGimmeSpace returns a pointer to where you want }
      { the data item to go.}
      myBuffer := MyGimmeSpace(len);
```

```
            myDataReturned := FALSE;
            WHILE NOT myDataReturned DO
            BEGIN
                getErr := DBGetItem(thisSession, myTimeout, myType,
                                    len, places, flags, myBuffer,
                                    NIL);
                {If you timed out, then give up control.  When }
                { control returns, continue getting the data.}
                IF getErr = rcDBBreak THEN MyGoDoSomething
                ELSE IF (getErr = noErr) OR
                        (getErr = rcDBValue) THEN myDataReturned := TRUE
                ELSE
                BEGIN
                    sessErr := getErr;
                    endErr := DBEnd(thisSession, NIL);
                    EXIT(MyLoLevel);
                END;
            END; {while}
        END
        ELSE sessErr := stateErr;
    END
    ELSE sessErr := execErr;
    endErr := DBEnd(thisSession, NIL);
END;
```

Note that, even if you are using the low-level interface to send queries to the data server, you might want to use the high-level functions to retrieve data and convert it to text.

Getting Information About Sessions in Progress

If your application is only one of several on a single Macintosh computer connected to data servers, you can use the DBGetConnInfo and DBGetSessionNum functions to obtain information about the sessions in progress. If you know the session ID (which is returned by the DBInit function when you open a session), you can use the DBGetConnInfo function to determine the database extension being used, the name of the host system on which the session is running, the user name and connection string used to initiate the session, the time at which the session started, and the status of the session. The status of the session specifies whether the data server is executing a query or waiting for another query fragment, whether there is output data available, and whether execution of a query ended in an error.

If you do not know the session ID, or if you want to get information about all open sessions, you can specify a database extension and a session number when you call the DBGetConnInfo function. Although there can be only one active session with a given session ID, session numbers are unique only for a specific database extension. Because the database extension assigns session numbers sequentially, starting with 1, you can call the DBGetConnInfo function repeatedly for a given database extension, incrementing the session number each time, to obtain information about all sessions open for that database extension. Your application need not have initiated the session to obtain information about it in this fashion.

The DBGetSessionNum function returns the session number when you specify the session ID. You can use this function to determine the session numbers for the sessions opened by your own application. You might want this information, for example, so you can distinguish your own sessions from those opened by other applications when you use the DBGetConnInfo function to get information about all open sessions.

Processing Query Results

You can use the low-level function DBGetItem to retrieve a single data item returned by a query, or you can use the high-level function DBGetQueryResults to retrieve all of the query results at once. If you use the DBGetQueryResults function, you can then use the DBResultsToText function to convert the results to ASCII text. The DBResultsToText function calls routines called *result handlers,* which are installed in memory by applications or by system extensions (files containing 'INIT' resources). This section discusses the use of the DBGetItem and DBGetQueryResults functions and describes how to write and install a result handler.

Getting Query Results

The DBGetItem function retrieves a single data item that was returned by a data source in response to a query. When you call the DBGetItem function, you specify the data type to be retrieved. If you do not know what data type to expect, you can specify the typeAnyType constant for the dataType parameter, and the data server returns the next data item regardless of data type. It also returns information about the data item, including data type and length.

If you do not know the length of the next data item, you can specify NIL for the buffer parameter in the DBGetItem function, and the data server returns the data type, length, and number of decimal places without retrieving the data item. The next time you call the DBGetItem function with a nonzero value for the buffer parameter, the function retrieves the data item.

If you want to skip a data item, specify the typeDiscard constant for the dataType parameter. Then the next time you call the DBGetItem function, it retrieves the following data item.

You should use the DBGetItem function if you want complete control over the retrieval of each item of data. If you want the Data Access Manager to retrieve the data for you, use the DBGetQueryResults function instead.

Table 12-1 shows the data types recognized by the Data Access Manager. You use a constant to specify each data type, as follows:

```
CONST {data types}
    typeAnyType    =  0;        {can be any data type}
    typeNone       = 'none';    {no more data expected}
    typeBoolean    = 'bool';    {Boolean}
    typeSMInt      = 'shor';    {short integer}
    typeInteger    = 'long';    {integer}
    typeSMFloat    = 'sing';    {short floating point}
    typeFloat      = 'doub';    {floating point}
    typeDate       = 'date';    {date}
    typeTime       = 'time';    {time}
    typeTimeStamp  = 'tims';    {date and time}
    typeChar       = 'TEXT';    {character}
    typeDecimal    = 'deci';    {decimal number}
    typeMoney      = 'mone';    {money value}
    typeVChar      = 'vcha';    {variable character}
    typeVBin       = 'vbin';    {variable binary}
    typeLChar      = 'lcha';    {long character}
    typeLBin       = 'lbin';    {long binary}
    typeDiscard    = 'disc';    {discard next data item}
    typeUnknown    = 'unkn';    {result handler for unknown }
                                { data type}
    typeColBreak   = 'colb';    {result handler for column break}
    typeRowBreak   = 'rowb';    {result handler for end of line}
```

The writer of a database extension can define other data types to support specific data sources or data servers.

Each data type has a standard definition, shown in Table 12-1. For example, if the DBGetItem function returns the typeInteger constant for the dataType parameter, you know that the data item represents an integer value and that a 4-byte buffer is necessary to hold it. Similarly, if you are using the DBSendItem function to send to the data server a data item that you identify as typeFloat, the data server expects to receive an 8-byte floating-point value.

Notice that some of these data types are defined to have a specific length (referred to as an *implied length*), and some do not. The len parameter of the DBSendItem and DBGetItem functions indicates the length of an individual data item. The DBGetQueryResults function returns a handle to an array of lengths, decimal places, and flags in the colInfo field of the results record. The typeAnyType, typeColBreak, and typeRowBreak constants do not refer to specific data types, and therefore the length specification is not applicable for these constants.

Table 12-1   Data types defined by the Data Access Manager

Constant	Length	Definition
typeAnyType	NA	Any data type (used as an input parameter to the DBGetItem function only; never returned by the function).
typeNone	0	Empty.
typeBoolean	1 byte	TRUE (1) or FALSE (0).
typeSMInt	2 bytes	Signed integer value.
typeInteger	4 bytes	Signed long integer value.
typeSMFloat	4 bytes	Signed floating-point value.
typeFloat	8 bytes	Signed floating-point value.
typeDate	4 bytes	Date; a long integer value consisting of a year (most significant 16 bits), month (8 bits), and day (least significant 8 bits).
typeTime	4 bytes	Time; a long integer value consisting of an hour (0–23; most significant 8 bits), minute (8 bits), second (8 bits), and hundredths of a second (least significant 8 bits).
typeTimeStamp	8 bytes	Date and time. A long integer date value followed by a long integer time value.
typeChar	Any	Fixed-length character string, not NULL terminated. The length of the string is defined by the specific data source.

Table 12-1 Data types defined by the Data Access Manager (continued)

Constant	Length	Definition
typeDecimal	Any	Packed decimal string. A contiguous string of 4-bit nibbles, each of which contains a decimal number, except for the low nibble of the highest-addressed byte (that is, the last nibble in the string), which contains a sign. The value of the sign nibble can be 10, 12, 14, or 15 for a positive number or 11 or 13 for a negative number; 12 is recommended for a positive number and 13 is recommended for a negative number. The most significant digit is the high-order nibble of the lowest-addressed byte (that is, the first nibble to appear in the string).

The total number of nibbles (including the sign nibble) must be even; therefore, the high nibble of the highest-addressed byte of a number with an even number of digits must be 0.

For example, the number +123 is represented as $123C.

The number –1234 is represented as $01234D.

The length of a packed decimal string is defined as the number of bytes, including any extra leading 0 and the sign nibble. A packed decimal string can have from 0 to 31 digits, not including the sign nibble.

In addition to the length of a packed decimal string, each data item has an associated value that indicates the number of digits that follow the decimal place. The `places` parameter in the `DBGetItem` and `DBSendItem` functions indicates the number of decimal places in an individual data item. The `DBGetQueryResults` function returns the number of decimal places.

Constant	Length	Definition
typeMoney	Any	Same as `typeDecimal`, but always has two decimal places.
typeVChar	Any	Variable-length character string, `NULL` terminated.

continued

Table 12-1   Data types defined by the Data Access Manager (continued)

Constant	Length	Definition
typeVBin	Any	Not defined. Reserved for future use.
typeLChar	Any	Not defined. Reserved for future use.
typeLBin	Any	Not defined. Reserved for future use.
typeDiscard	NA	Do not retrieve the next data item (used as an input parameter to the DBGetItem function only; never returned by the function).
typeUnknown	NA	A dummy data type for the result handler that processes any data type for which no other result handler is available (used as an input parameter to the DBInstallResultHandler, DBRemoveResultHandler, and DBGetResultHandler functions only; never returned by the DBGetItem function).
typeColBreak	NA	A dummy data type for the result handler that the DBGetQueryResults function calls after each item that is not the last item in a row (used as an input parameter to the DBInstallResultHandler, DBRemoveResultHandler, and DBGetResultHandler functions only; never returned by the DBGetItem function).
typeRowBreak	NA	A dummy data type for the result handler that the DBGetQueryResults function calls at the end of each row (used as an input parameter to the DBInstallResultHandler, DBRemoveResultHandler, and DBGetResultHandler functions only; never returned by the DBGetItem function).

The DBGetQueryResults function retrieves all of the data that was returned by a data source in response to a query, unless insufficient memory is available to hold the data, in which case it retrieves as many complete rows of data as possible. The DBGetQueryResults function stores the data in a structure called a *results record.* You must allocate the results record data structure and pass this record to the DBGetQueryResults function. The Data Access Manager allocates the handles inside the results record. When your application is finished using the results record, you must deallocate both the results record and the handles inside the results record.

The results record is defined by the `ResultsRecord` data type.

```
TYPE ResultsRecord =
    RECORD
        numRows:    Integer;            {number of rows retrieved}
        numCols:    Integer;            {number of columns per row}
        colTypes:   ColTypesHandle;     {type of data in each column}
        colData:    Handle;             {array of data items}
        colInfo:    ColInfoHandle;      {info about each data item}
    END;
```

The `numRows` field in the results record indicates the total number of rows retrieved. If the `DBGetQueryResults` function returns a result code other than `rcDBValue`, then not all of the data actually returned by the data source was retrieved. This could happen, for instance, if the user's computer does not have sufficient memory space to hold all the data. In this case, your application can make more space available (by writing the data in the data record to disk, for example) and then call the `DBGetQueryResults` function again to complete retrieval of the data.

Note

The `DBGetQueryResults` function retrieves whole rows only; if it runs out of space in the middle of a row, it stores the partial row in a private buffer so that the data in the results record ends with the last complete row. Because the last partial row is no longer available from the data server, you cannot start to retrieve data with the `DBGetQueryResults` function and then switch to the `DBGetItem` function to complete the data retrieval. u

The `numCols` field indicates the number of columns in each row of data.

The `colTypes` field is a handle to an array of data types, specifying the type of data in each column. The number of elements in the array is equal to the value in the `numCols` field. Table 12-1 beginning on page 12-38 shows the standard data types.

The `colData` field is a handle to the data retrieved by the `DBGetQueryResults` function.

The `colInfo` field is a handle to an array of records of type `DBColInfoRecord`, each of which specifies the length, places, and flags for a data item. There are as many records in the array as there are data items retrieved by the `DBGetQueryResults` function. Here is the `DBColInfoRecord` type definition:

```
TYPE DBColInfoRecord =
   RECORD
      len:        Integer;        {length of data item}
      places:     Integer;        {places for decimal and }
                                  { money data items}
      flags:      Integer;        {flags for data item}
   END;
```

The `len` field indicates the length of the data item. The `DBGetQueryResults` function returns a value in this field only for those data types that do not have implied lengths; see Table 12-1 beginning on page 12-38.

The `places` field indicates the number of decimal places in data items of types `typeMoney` and `typeDecimal`. For all other data types, the `places` field returns 0.

The least significant bit of the `flags` field is set to 1 if the data item is in the last column of the row. The third bit of the `flags` field is 1 if the data item is NULL. You can use the constants `kDBLastColFlag` and `kDBNullFlag` to test for these flag bits.

Converting Query Results to Text

The `DBResultsToText` function provided by the high-level interface converts the data retrieved by the `DBGetQueryResults` function into strings of ASCII text. This function makes it easier for you to display retrieved data for the user.

For the `DBResultsToText` function to convert data of a specific type to text, either the application or the system software must have a routine called a *result handler.* With System 7, Apple Computer, Inc., provides system result handlers for the data types listed here. (These data types are described in Table 12-1 beginning on page 12-38.)

Data type	Constant	Data type	Constant
Boolean	`typeBoolean`	Time	`typeTime`
Short integer	`typeSMInt`	Date and time	`typeTimeStamp`
Integer	`typeInteger`	Character	`typeChar`
Short floating point	`typeSMFloat`	Decimal number	`typeDecimal`
Floating point	`typeFloat`	Money value	`typeMoney`
Date	`typeDate`	Variable character	`typeVChar`

Note

Apple's system result handler for the variable character (`typeVChar`) data type strips trailing spaces from the character string. u

In addition to the result handlers for these standard data types, Apple provides the following three system result handlers, which correspond to no specific data type:

Data type	Constant
Unknown	`typeUnknown`
Column break	`typeColBreak`
End of line	`typeRowBreak`

The `typeUnknown` result handler processes any data type for which no other result handler is available. The `DBResultsToText` function calls the `typeColBreak` result handler after each item that is not the last item in a row. This result handler does not correspond to any data type, but adds a delimiter character to separate columns of text. The default `typeColBreak` result handler inserts a tab character. Similarly, the `DBResultsToText` function calls the `typeRowBreak` result handler at the end of each row of data to add a character that separates the rows of text. The default `typeRowBreak` result handler inserts a return character. Your application can install your own `typeColBreak` and `typeRowBreak` result handlers to insert whatever characters you wish—or to insert no character at all, if you prefer.

You can install result handlers for any data types you know about. When you call the `DBInstallResultHandler` function, you can specify whether the result handler you are installing is a system result handler. A **system result handler** is available to all applications that use the system. All other result handlers (called **application result handlers**) are associated with a particular application. The `DBResultsToText` function always uses a result handler for the current application in preference to a system result handler for the same data type. When you install a system result handler for the same data type as an already installed system result handler, the new result handler replaces the old one. Similarly, when you install an application result handler for the same data type as a result handler already installed for the same application, the new result handler replaces the old one for that application.

Result handlers are stored in memory. The Data Access Manager installs its system result handlers the first time the Macintosh Operating System loads the Data Access Manager into memory. You must reinstall your own application result handlers each time your application starts up. You can also install your own system result handlers each time your application starts up, or you can provide a system extension (that is, a file with an `'INIT'` resource) that installs system result handlers each time the user starts up the system.

Here is a function declaration for a result handler function:

```
FUNCTION MyResultHandler (dataType: DBType;
                          theLen, thePlaces, theFlags: Integer;
                          theData: Ptr; theText: Handle): OSErr;
```

The `dataType` parameter specifies the data type of the data item that the `DBResultsToText` function is passing to the result handler. Table 12-1 beginning on page 12-38 describes the standard data types.

The parameters `theLen` and `thePlaces` specify the length and number of decimal places of the data item that the `DBResultsToText` function wants the result handler to convert to text.

The parameter `theFlags` is the value returned for the `flags` parameter by the `DBGetItem` function. If the least significant bit of this parameter is set to 1, the data item is in the last column of the row. If the third bit of this parameter is set to 1, the data item is `NULL`. You can use the constants `kDBLastColFlag` and `kDBNullFlag` to test for these flag bits.

The parameter `theData` is a pointer to the data that the result handler is to convert to text.

The parameter `theText` is a handle to the buffer that is to hold the text version of the data. The result handler should use the Memory Manager's `SetHandleSize` function to increase the size of the buffer as necessary to hold the new text, and append the new text to the end of the text already in the buffer. The `SetHandleSize` function is described in the chapter "Memory Manager" in *Inside Macintosh: Memory.*

If the result handler successfully converts the data to text, it should return a result code of 0 (`noErr`).

You can use the `DBInstallResultHandler` function to install a result handler and the `DBRemoveResultHandler` function to remove an application result handler. You can install and replace system result handlers, but you cannot remove them.

The following line of code installs an application result handler. The first parameter (`typeInteger`) specifies the data type that this result handler processes. The second parameter (`MyTypeIntegerHandler`) is a pointer to the result handler routine. The last parameter (`FALSE`) is a Boolean value specifying that this routine is not a system result handler.

```
err := DBInstallResultHandler
                    (typeInteger,@MyTypeIntegerHandler,FALSE);
```

Listing 12-6 shows a result handler that converts the integer data type to text.

Listing 12-6   A result handler

```
FUNCTION MyTypeIntegerHandler(datatype: DBType; theLen: Integer;
                              theData: Ptr;
                              theText: Handle): OSErr;
VAR
   theInt:        LongInt;
   theTextLen:    LongInt;
   temp:          Str255;
   atemp1:        Ptr;
   atemp2:        LongInt;
   atemp3:        LongInt;
BEGIN
   BlockMove(theData, @theInt, sizeof(theInt));
   NumToString(theInt, temp);               {convert to text}
   theTextLen := GetHandleSize(theText);    {get current size }
                                            { of theText}
                                            {size text handle}
   SetHandleSize(theText, theTextLen + LongInt(LENGTH(temp)));
   IF (MemError <> noErr) THEN
      MyTypeIntegerHandler := MemError
   ELSE
   BEGIN
      atemp1 := Ptr(ORD(@temp));
      atemp2 := LongInt(theText^) + theTextLen;
      atemp3 := LongInt(LENGTH(temp));
      {use BlockMove to append text}
      BlockMove(P2CStr(atemp1), Ptr(atemp2), atemp3);
      MyTypeIntegerHandler := MemError;
   END;
END;
```

Creating a Query Document

A query document is a file of type `'qery'` that contains a `'qrsc'` resource and one or more `'wstr'` resources, and may contain a `'qdef'` resource plus other resources. Query documents make it possible for you to write applications that can communicate with data servers without requiring familiarity with the command language used by the data server. Because a query document is most useful if it can be used by many different applications, no query document should depend on the presence of a particular application in order to function.

An application can call the `DBGetNewQuery` function to convert a `'qrsc'` resource into a **query record** in memory. A query record specifies connection information and also contains a handle to an array of queries; each query can be either a complete query or a template for a query. If the `'wstr'` resource is a template, it contains the commands and data necessary to create a query, without any information that the user must add just before the query is sent. The `'qdef'` resource contains a query definition function, which can modify the query record and, if necessary, fill in the query template to create a complete query. The `DBStartQuery` function sends the query pointed to by a query record to a data server. The following sections describe the contents of a query document, describe query records, and define the `'qrsc'`, `'wstr'`, and `'qdef'` resources.

User Interface Guidelines for Query Documents

All query documents should behave in fundamentally the same way. They should be self-explanatory and should never execute a query without an explicit command from the user. When your application opens a query document, the query document should display a dialog box with enough information about the query so that the user can decide if it's the right query. The dialog box should describe the purpose of the query, what kind of data it transfers and in which direction, the type of data source it accesses, and any warnings or instructions. The dialog box can describe how the user interprets the data, such as the name of each field in a record. Figure 12-6 shows an example of a query document dialog box.

Figure 12-6   A query document dialog box

This dialog box should allow the user to cancel the request for data. In addition, it may be useful to allow the user to set parameters with text boxes, checkboxes, or radio buttons. For example, a query to a database of financial information could provide a list of these options: a trial balance, profit-and-loss statements, or net worth reports. Save the last set of user-specified parameters with the query document. This way the user can review the parameters used to generate the data or use the same parameters the next time.

Once a query starts running, it must be able to complete its task without user intervention. If a query must run modally (that is, it must run to completion before returning control to the user), display a dialog box that shows the query's progress and be sure to return control to the user as soon as possible. The philosophy of this process is similar to that of receiving electronic mail—that is, inform the user when the information arrives, but let the user decide when to read it.

Whenever possible, query documents should check that data is compatible before transmitting it to a data source. Establish a connection with a data source only after you have checked the data.

Contents of a Query Document

The query document must contain

n one 'qrsc' resource, as defined in the next section, "Query Records and Query Resources"

n one 'STR#' resource that contains the name of the database extension to be used, plus any host, user name, password, and connection string needed for the DBInit function

n one or more 'wstr' resources containing queries—that is, strings of commands and data that the DBSend function sends to the data server and that the DBExec function executes

A 'wstr' resource consists of a 2-byte length field followed by a character string. (The *w* in 'wstr' refers to the length word as opposed to the length byte used in an 'STR ' resource.) Each 'wstr' resource contains one query (or one query template, to be modified by the query definition function before it is sent to the data server). The 'qrsc' resource includes an array that lists the resource ID numbers of all of the 'wstr' resources in the query document and an index into the array that specifies which one of the 'wstr' resources should be sent to the data server.

In addition, the query document may contain

n a 'qdef' resource that contains a query definition function

n any resources needed by the query definition function, such as 'DLOG' and 'DITL' resources (which support dialog boxes)

n resources to support an icon (to replace the default icon that the Finder uses for files of type 'qery'); see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for more information on icon resources and for guidelines on designing icons

Figure 12-7 illustrates the relationship between the resources in a query document and the query record.

Figure 12-7   The relationship between resources in a query document and the query record
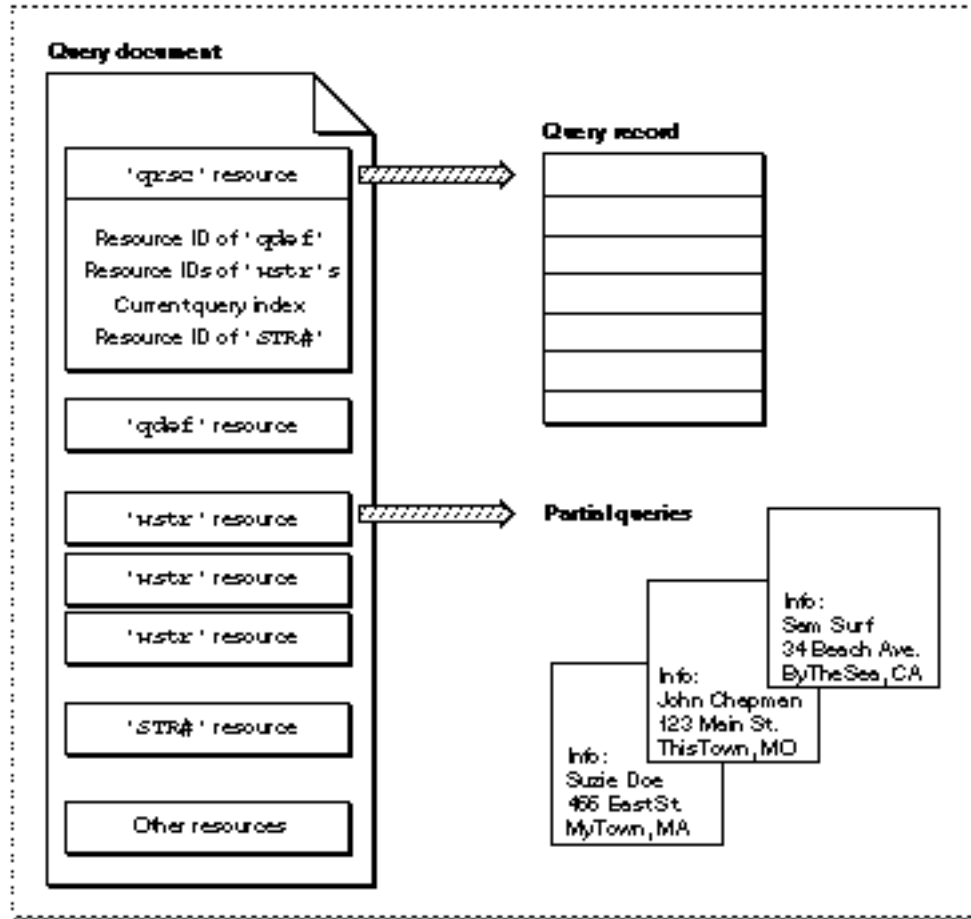
The query document in Figure 12-7 contains a `'qrsc'` resource that specifies the
resource ID of a `'qdef'` resource, the resource IDs of three `'wstr'` resources, and
the resource ID of an `'STR#'` resource. It also specifies which of the three `'wstr'`
resources represents the current query.

The DBGetNewQuery function creates the query record and partial queries from this information. Your application can use the DBStartQuery function to send a query to a data server. The DBStartQuery function calls the query definition function referred to by the query record (if any). The query definition function can prompt the user for information and modify the query as needed. Figure 12-8 illustrates a query record that contains a handle to an array of queries, a handle to a query definition function, and an index that identifies the current query. The query definition function displays a dialog box and modifies the current query according to the user's input. Once the query definition function modifies the current query and returns, the DBStartQuery function sends the query to the data server.

Figure 12-8   The relationship between a query definition function and queries
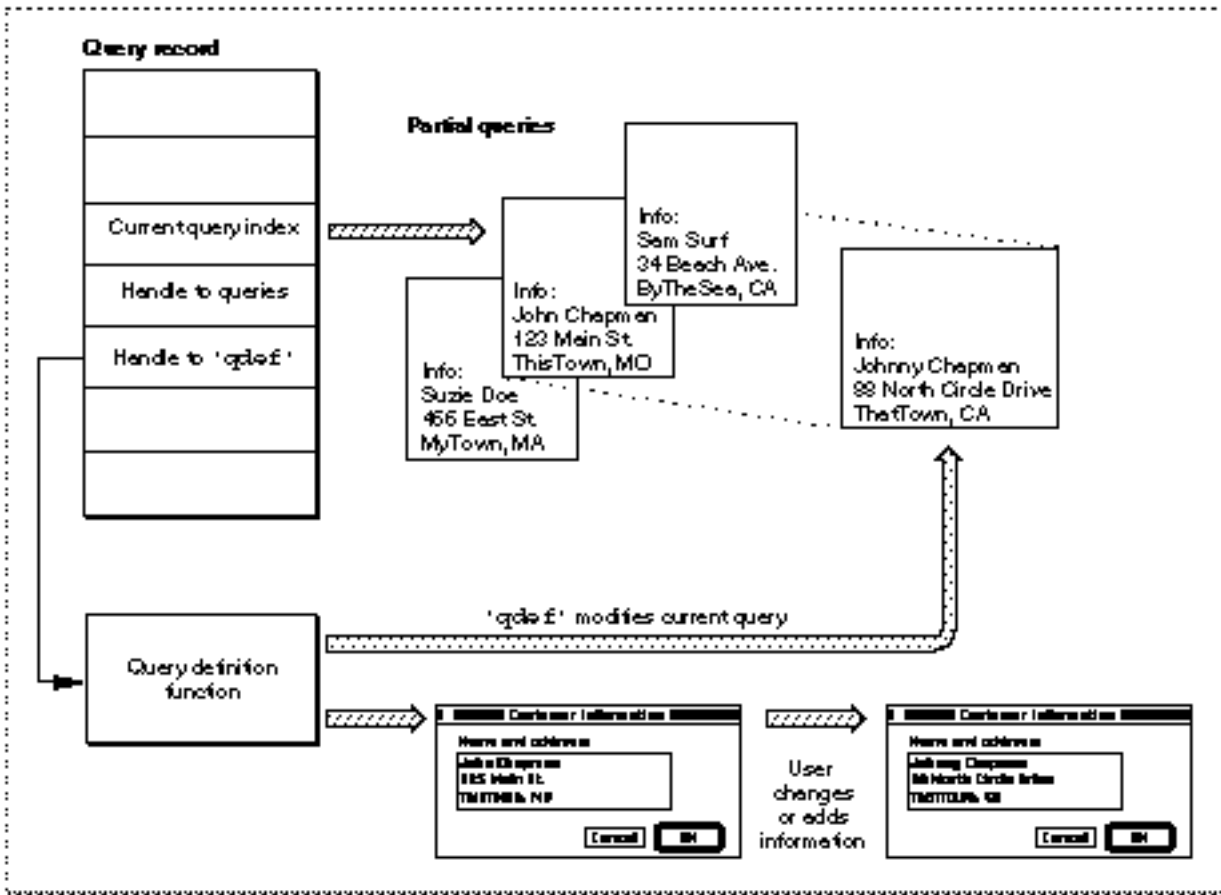
Query Records and Query Resources

The DBGetNewQuery function converts the 'qrsc' resource in the query document into a query record in memory. The query definition function can then modify the query record before the application sends the query to the data server. See "The Query Record" beginning on page 12-56 for a description of the query record. See "The Query Resource" beginning on page 12-90 for the format of a 'qrsc' resource. The next section provides information about query definition functions.

Writing a Query Definition Function

Before the DBStartQuery function sends a query to a data server, it calls the query definition function specified by the queryProc field in the query record. The purpose of the query definition function is to modify the query and the query record before the query is sent to the data server. The query definition function can use dialog boxes to request information from the user. Because a query document is most useful if it can be used by many different applications, no query definition function should depend on the presence of a particular application.

If you want to include a query definition function, you must make it the first piece of code in a resource of type 'qdef' in the query document.

Here is a function declaration for a query definition function.

```
FUNCTION MyQDef (VAR sessID: LongInt; query: QueryHandle): OSErr;
```

If the application has already initiated a session with the data server, the DBStartQuery function passes the session ID for that session in the sessID parameter to the query definition function. If the query definition function receives a 0 in this parameter, then the Data Access Manager has not initiated a session. In this case, the query definition function can return a 0 in the sessID parameter, or it can call the DBInit function to initiate a session and then return the session ID in this parameter.

If the query definition function returns a 0 in the sessID parameter, the DBStartQuery function calls the DBInit function and then calls the DBSend function to send a query to the data server. If the query definition function returns a session ID in this parameter, the DBStartQuery function calls the DBSend function immediately.

The query parameter to the query definition function specifies a handle to the query record. The query definition function can modify any of the fields in the query record, including the currQuery field that specifies which query is to be sent to the data server. In addition, the query definition function can modify an existing query or create a new query, adding the handle to the new query to the query list. Note that, because a query in memory consists only of a 2-byte length value followed by a character string, the query definition function has to know the exact contents and structure of a query in order to modify it.

The query definition function must return the noErr result code as the function result if
the function executed successfully. If it returns any other value, the DBStartQuery
function does not call the DBSend function. The query definition function can return any
result code, including noErr, userCanceledErr, or rcDBError.

When the DBStartQuery function calls the query definition function, the current
resource file is the file that contains the 'qrsc' resource from which the Data Access
Manager created the query record. When the query definition function returns control to
the Data Access Manager, the current resource file must be unchanged. See the chapter
"Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information on
the current resource file.

The query definition function can allocate memory and use the dataHandle field in the
query record to store a handle to it. The query definition function must free any memory
it allocates before terminating.

Listing 12-7 shows a query definition function that uses a dialog box to prompt the user
for a user name and password and then modifies the query record accordingly.

Listing 12-7   A query definition function

```
FUNCTION MyQDef(VAR sessID: LongInt; query: QueryHandle): OSErr;
CONST
   myNameItem        =   7;
   myPassWordItem    =   8;
VAR
   myNumRes:         Integer;
   myResList:        ResListHandle;
   myResLPtr:        ResListPtr;
   myIndex:          Integer;
   myDialog:         DialogPtr;
   myDlogID:         Integer;
   itemType:         Integer;
   itemHName:        Handle;
   itemHPasswd:      Handle;
   itemBox:          Rect;
   mySTR:            ARRAY[1..2] OF Str255;
   itemHit:          Integer;
   myQErr:           OsErr;
BEGIN
   {If sessID = 0 no session has been initiated. Your qdef may }
   { optionally initiate a session, or it can let DBStartQuery }
   { take care of this.  In this example, the qdef doesn't }
   { check the sessID parameter.}
   HLock(Handle(query));
   myNumRes := query^^.numRes;
```

```
myResList := query^^.resList;
HLock(Handle(myResList));
myResLPtr := myResList^;
myIndex := 0;
{look for a 'DLOG' resource}
WHILE (myIndex < myNumRes) AND
      (myResLPtr^[myIndex].theType <> 'DLOG') DO
BEGIN
   myIndex := myIndex + 1;
END;
IF (myIndex < myNumRes) THEN  {found the 'DLOG' resource}
   myDlogID := myResLPtr^[myIndex].id
ELSE
BEGIN
   {The 'DLOG' wasn't found; exit with no error.  This }
   { is probably OK; it just means that the query }
   { and the query record don't get modified.}
   MyQDEF := noErr;
   HUnlock(Handle(query));
   HUnlock(Handle(myResList));
   EXIT(MyQDef);
END;
{found the 'DLOG' and its ID; now display the dialog box}
myDialog := GetNewDialog(myDlogID, Ptr(NIL), WindowPtr(-1));
SetPort(GrafPtr(myDialog));
REPEAT
   ModalDialog(@MyEventFilter, itemHit);
UNTIL ((itemHit = kOK) OR (itemHit = kCancel));
IF itemHit = kOK THEN
BEGIN
   {The user clicked the OK button. Update the user }
   { and password fields of the query record.}
   GetDialogItem(myDialog, myNameItem, itemType, itemHName,
                 itemBox);
   GetDialogItemText(itemHName, mySTR[1]);
   GetDialogItem(myDialog, myPassWordItem, itemType,
               itemHPasswd, itemBox);
   GetDialogItemText(itemHPasswd, mySTR[2]);
   {Now you can change the query record or the query itself. }
   { What you change is entirely up to you. In this example, }
   { the qdef changes only the user and password fields }
   { of the query record.}
   query^^.user := mySTR[1];
```

```
        query^^.password := mySTR[2];
        MyQDef := noErr;
    END
    ELSE
        MyQDef := userCanceledErr;
    HUnlock(Handle(query));
    HUnlock(Handle(myResList));
    DisposDialog(myDialog);
END;
```

Data Access Manager Reference

This section describes the data structures, routines, and resources that are specific to the Data Access Manager. The "Data Structures" section shows the data structures for the asynchronous parameter block, the results record, the query record, and the data item record. The "Data Access Manager Routines" section beginning on page 12-59 describes routines for using the high-level and low-level interfaces, including initializing the Data Access Manager, handling query documents and results, controlling sessions, sending and executing queries, retrieving results, and installing and removing result handlers. The "Resources" section beginning on page 12-90 describes the query resource, the query string resource, and the query definition function resource.

Data Structures

This section describes the data structures that you use to provide information to the Data Access Manager or that the Data Access Manager uses to provide information to your application.

You provide a pointer to an asynchronous parameter block as a parameter to the DBStartQuery, DBGetQueryResults, DBInit, DBEnd, DBGetSessionNum, DBKill, DBSend, DBSendItem, DBExec, DBState, DBGetErr, DBBreak, DBGetItem, and DBUnGetItem functions.

The query record specifies connection information and contains a handle to an array of one or more complete queries or query templates. The DBGetNewQuery function returns a handle to a query record, and you provide a handle to a query record as a parameter to the DBStartQuery and DBDisposeQuery functions.

You use the results record to store the data that was returned by a data source in response to a query. The results record is a parameter to the DBGetQueryResults and DBResultsToText functions.

The Asynchronous Parameter Block

Each Data Access Manager routine that can be called asynchronously (that is, that can return control to your application before it has completed execution) takes as a parameter a pointer to a parameter block known as the *asynchronous parameter block.* If you specify NIL for this parameter, the routine does not return control to your application until it has completed execution.

Note

The asynchronous parameter block is passed on to the database extension, which is responsible for implementing the asynchronous routine. If the database extension does not support asynchronous routines, the Data Access Manager returns the rcDBAsyncNotSupp result code and terminates execution of the routine. u

The DBAsyncParamBlockRec data type defines the asynchronous parameter block.

```
TYPE DBAsyncParamBlockRec =
    RECORD
        completionProc:    ProcPtr;  {pointer to completion routine}
        result:            OSErr;    {result of call}
        userRef:           LongInt;  {reserved for use by application}
        ddevRef:           LongInt;  {reserved for use by database }
                                     { extension}
        reserved:          LongInt;  {reserved for use by Data }
                                     { Access Manager}
    END;


    DBAsyncParmBlkPtr = ^DBAsyncParamBlockRec;
```

Field descriptions

completionProc
: Points to a completion routine that the database extension calls when it has completed executing the asynchronous function. Before calling the completion routine, the Data Access Manager places a pointer to the asynchronous parameter block in the A0 register. If you do not want to use a completion routine, set this parameter to NIL.

result
: Returns the result code for the called routine. The database extension sets this field to 1 while the routine is executing and places the result code in it when the routine completes. Your application can poll this field to determine when an asynchronous routine has completed execution.

userRef
: Reserved for the application's use. Because the Data Access Manager passes a pointer to the parameter block to the completion routine, you can use this field to pass information to the completion routine.

`ddevRef`	Reserved for use by the database extension.
`reserved`	Reserved for use by the Data Access Manager.

The Query Record

The `DBGetNewQuery` function converts a `'qrsc'` resource in a query document into a query record in memory and returns a handle to the query record. The query record specifies connection information and also contains a handle to an array of queries; each query can be either a complete query or a template for a query. The `DBGetNewQuery` function creates the queries from the `'wstr'` resources stored in the query document.

The `QueryRecord` data type defines a query record.

```
TYPE QueryRecord =
   RECORD
      version:    Integer;         {query record format version}
      id:         Integer;         {resource ID of 'qrsc'}
      queryProc:  Handle;          {handle to qdef}
      ddevName:   Str63;           {name of database extension}
      host:       Str255;          {name of host computer}
      user:       Str255;          {name of user}
      password:   Str255;          {user's password}
      connStr:    Str255;          {connection string}
      currQuery:  Integer;         {index of current query}
      numQueries: Integer;         {number of queries in list}
      queryList:  QueryListHandle; {handle to array of }
                                   { handles to text}
      numRes:     Integer;         {number of resources in list}
      resList:    ResListHandle;   {handle to array of resource }
                                   { list elements}
      dataHandle: Handle;          {handle to memory for qdef}
      refCon:     LongInt;         {reserved for use by app}
   END;
QueryPtr    = ^QueryRecord;        {pointer to query record}
QueryHandle = ^QueryPtr;           {handle to query record}
```

Field descriptions

`version`	The version number of the query record format. For the Data Access Manager released with System 7, the version number is 0.
`id`	The resource ID of the `'qrsc'` resource from which the Data Access Manager created this query record.
`queryProc`	A handle to the query definition function that the `DBStartQuery` function calls. This handle is `NIL` if there is no query definition function—that is, if the `DBStartQuery` function should send the query specified by this query record to the data server without modifications.

ddevName	The database extension name used as a parameter to the DBInit function.
host	The name of the host computer system used as a parameter to the DBInit function.
user	The name of the user, used as a parameter to the DBInit function.
password	The user's password, used as a parameter to the DBInit function.
connStr	The connection string used as a parameter to the DBInit function.
currQuery	An index value from 1 through numQueries, indicating which element in the array of query handles represents the current query. The current query is the one actually sent to the data server. If the query document contains more than one 'wstr' resource, the query definition function can prompt the user to select a new current query and modify this field in the query record appropriately.
numQueries	The number of queries referred to by the queryList field.
queryList	A handle to an array of handles. Each handle in this array refers to a query. Each query is created from a 'wstr' resource in the query document and is stored in memory as a 2-byte length field followed by ASCII text. (The length does not include the 2 bytes of the length field.) The query definition function can create a new query. To add a new handle to the array of handles, use the Memory Manager's SetHandleSize function to increase the size of the array. Don't forget to change the value of the numQueries field as well.
numRes	The number of resources referred to by the resList field.
resList	A handle to an array of records of type ResListElem. Each record in the array contains the type and ID of a resource that is needed by the query definition function.

```
TYPE ResListElem =
   RECORD
      theType: ResType;     {resource type}
      id:      Integer;     {resource ID}
   END;
```

dataHandle	A handle to memory for use by the query definition function. When the Data Access Manager first creates the query record, it sets this field to NIL. The query definition function can allocate memory and place a handle to it in this field. The query definition function should dispose of any memory it allocates before it returns control to the Data Access Manager.
refCon	The query record's reference value. The application can use this field for any purpose.

The Results Record

The results record describes the data that was returned by a data source in response to a query. To get the results of a query, allocate a results record and pass this record to the `DBGetQueryResults` function. The Data Access Manager allocates the handles inside the results record. When your application is finished using the results record, you must deallocate both the results record and the handles inside the results record.

The results record is defined by the `ResultsRecord` data type.

```
TYPE ResultsRecord =
   RECORD
      numRows:    Integer;          {number of rows retrieved}
      numCols:    Integer;          {number of columns per row}
      colTypes:   ColTypesHandle;   {type of data in each column}
      colData:    Handle;           {array of data items}
      colInfo:    ColInfoHandle;    {info about each data item}
   END;
```

Field descriptions

numRows   The total number of rows retrieved. If the `DBGetQueryResults` function returns a result code other than `rcDBValue`, then not all of the data actually returned by the data source was retrieved. This could happen, for instance, if the user's computer does not have sufficient memory space to hold all the data. In this case, your application can make more space available (by writing the data in the data record to disk, for example) and then call the `DBGetQueryResults` function again to complete retrieval of the data.

Note

The `DBGetQueryResults` function retrieves whole rows only; if it runs out of space in the middle of a row, it stores the partial row in a private buffer so that the data in the results record ends with the last complete row. Because the last partial row is no longer available from the data server, you cannot start to retrieve data with the `DBGetQueryResults` function and then switch to the `DBGetItem` function to complete the data retrieval. u

numCols   The number of columns in each row of data.

colTypes   A handle to an array of data types, specifying the type of data in each column. The number of elements in the array is equal to the value in the `numCols` field. Table 12-1 beginning on page 12-38 shows the standard data types.

colData   A handle to the data retrieved by the `DBGetQueryResults` function.

colInfo   A handle to an array of records of type DBColInfoRecord, each of
                 which specifies the length, places, and flags for a data item. There
                 are as many records in the array as there are data items retrieved by
                 the DBGetQueryResults function. Here is the
                 DBColInfoRecord type definition:

```
TYPE DBColInfoRecord =
   RECORD
      len:      Integer;     {length of data item}
      places:   Integer;     {places for decimal }
                             { and money data items}
      flags:    Integer;     {flags for data item}
   END;
```

The len field indicates the length of the data item. The
DBGetQueryResults function returns a value in this field only for
those data types that do not have implied lengths; see Table 12-1 on
page 12-38 for a list of these data types.

The places field indicates the number of decimal places in data
items of types typeMoney and typeDecimal. For all other data
types, the places field returns 0.

The least significant bit of the flags field is set to 1 if the data item
is in the last column of the row. The third bit of the flags field is 1
if the data item is NULL. You can use the constants
kDBLastColFlag and kDBNullFlag to test for these flag bits.

Data Access Manager Routines

The Data Access Manager has high-level routines, low-level routines, and routines that
manipulate result handlers. This section describes all of the Data Access Manager
routines.

All of the low-level routines and some of the high-level routines accept a pointer to
an asynchronous parameter block as a parameter. For these routines, see "The
Asynchronous Parameter Block" beginning on page 12-55 for a description of the fields
in the parameter block.

If you specify a nonzero value for the pointer to the asynchronous parameter block, the
database extension executes the function asynchronously—that is, it returns control to
the Data Access Manager before the routine has completed execution, and the Data
Access Manager returns control to your application. If you specify NIL for this
parameter, the database extension does not return control to your application until the
routine has finished execution. Your application must call the Event Manager's
WaitNextEvent function periodically to allow an asynchronous routine to complete
execution. The WaitNextEvent function is described in the chapter "Event Manager"
in *Inside Macintosh: Macintosh Toolbox Essentials.*

You can tell when an asynchronous routine has completed execution and check the result code by looking at values in the asynchronous parameter block. You can use the `DBKill` function to cancel an asynchronous routine.

Note

A `noErr` result code returned by a routine that has been called asynchronously indicates only that the routine *began* execution successfully. You must check the `result` field of the asynchronous parameter block for the final result of the routine. u

Assembly-Language Note

You can invoke each of the Data Access Manager routines with a macro that has the same name as the routine, but preceded with an underscore; for example, the macro for the `DBInit` function is named `_DBInit`. Each of these macros places a routine selector in the D0 register and calls the trap `_Pack13`. The routine selectors are listed in each routine description and in "Assembly-Language Summary" beginning on page 12-103. u

Initializing the Data Access Manager

You must initialize the Data Access Manager before you can use it.

InitDBPack

Use the `InitDBPack` function to initialize the Data Access Manager.

```
FUNCTION InitDBPack: OSErr;
```

DESCRIPTION

The `InitDBPack` function initializes the Data Access Manager. You must call the `InitDBPack` function before you call any other Data Access Manager routines. If the Data Access Manager has already been initialized, the `InitDBPack` function returns the `noErr` result code but does nothing else.

The interface routine that implements the `InitDBPack` function includes a version number for the Data Access Manager. If the Data Access Manager is a different version from that specified by the interface routine, then the `InitDBPack` function returns the `rcDBWrongVersion` result code.

SPECIAL CONSIDERATIONS

The `InitDBPack` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

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

Trap macro   **Selector**

`_InitDBPack`   `$0100`

RESULT CODES

`noErr`	0	No error
`rcDBWrongVersion`	–812	Wrong version number

High-Level Interface: Handling Query Documents

The high-level interface to the Data Access Manager allows applications to manipulate query documents and to get the results of the query provided by a query document. The use and contents of query documents are discussed in "Creating a Query Document" beginning on page 12-46. The routines described in this section create query records, dispose of query records, and use query documents to establish communication with and send queries to a data server. For a general discussion of the high-level interface, see "The High-Level Interface" beginning on page 12-7 . For instructions on using the high-level interface, refer to "Using the High-Level Interface" beginning on page 12-14.

DBGetNewQuery

You can use the `DBGetNewQuery` function to create a query record.

```
FUNCTION DBGetNewQuery (queryID: Integer;
                        VAR query: QueryHandle): OSErr;
```

queryID   The resource ID of a `'qrsc'` resource.

query   Returns a handle to the query record.

DESCRIPTION

The `DBGetNewQuery` function creates a query record from the specified `'qrsc'` resource. The resource file that contains the `'qrsc'` resource must remain open until after the `DBStartQuery` function has completed execution. If you do not already know the resource ID of the `'qrsc'` resource (for example, if you call the `StandardGetFile` procedure to let the user select the query document), you can use Resource Manager routines to determine the resource ID.

SPECIAL CONSIDERATIONS

The DBGetNewQuery function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBGetNewQuery function are

Trap macro	Selector
_DBGetNewQuery	$030F

RESULT CODES

noErr	0	No error
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

See Listing 12-1 beginning on page 12-18 for an example of the use of the DBGetNewQuery function. For a description of the query record, see page 12-56. For a description of the 'qrsc' resource, see "The Query Resource" beginning on page 12-90. The StandardGetFile procedure is described in the chapter "Standard File Package" in *Inside Macintosh: Files,* and Resource Manager routines are described in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox.*

DBDisposeQuery

When you are finished using a query record, call DBDisposeQuery to dispose of the query record.

```
FUNCTION DBDisposeQuery (query: QueryHandle): OSErr;
```

query   A handle to the query record to dispose.

DESCRIPTION

The DBDisposeQuery function disposes of a query record and frees all the memory that the Data Access Manager allocated when it created the query record.

SPECIAL CONSIDERATIONS

The DBDisposeQuery function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBDisposeQuery function are

Trap macro	Selector
_DBDisposeQuery	$0210

RESULT CODES

noErr	0	No error
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

See Listing 12-1 beginning on page 12-18 for an example of the use of the DBDisposeQuery function in the high-level interface. For a description of the query record, see page 12-56.

DBStartQuery

Use the DBStartQuery function to initiate the process of sending a query to a data server.

```
FUNCTION DBStartQuery (VAR sessID: LongInt; query: QueryHandle;
                        statusProc: ProcPtr;
                        asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   A session ID that identifies a session with the data server. If you specify 0 for this parameter, then the DBStartQuery function initiates a session and returns the session ID in the sessID parameter.

query   A handle to a query record.

statusProc
                A pointer to a status routine that your application can use to update its windows after the query definition function has completed execution. (The DBStartQuery function does not attempt to update your application's windows.) The DBStartQuery function also calls your status routine before it initiates a session with a data server, after it calls the DBInit function, after it calls the DBSend function, and after it calls the DBExec function. Status routines are discussed in "Writing a Status Routine for High-Level Functions" beginning on page 12-22.

asyncPB   A pointer to an asynchronous parameter block. When specified, the DBStartQuery function calls the DBInit, DBSend, and DBExec functions asynchronously. As soon as the DBInit function has started execution, it returns control to your application. Your application must then call the Event Manager's WaitNextEvent function periodically to

allow these asynchronous routines to run, and it must check the `result` field of the asynchronous parameter block to determine when each routine has completed execution.

DESCRIPTION

The `DBStartQuery` function performs the following tasks, in the order specified:

1. It calls the query definition function (if any) pointed to by the query record. The query definition function modifies the query record and the query, usually by asking the user for input. The query definition function can display a dialog box that gives the user the option of canceling the query; if the user does cancel the query, the `DBStartQuery` function returns the `userCanceledErr` result code.

2. If you specify a nonzero value for the `statusProc` parameter, the `DBStartQuery` function calls your status routine with the `kDBUpdateWind` constant in the `message` parameter so that your application can update its windows.

3. If you specify a nonzero value for the `statusProc` parameter, the `DBStartQuery` function calls your status routine with the `kDBAboutToInit` constant in the `message` parameter so that your application can display a dialog box informing the user that a session is about to be initiated with a data server, and giving the user the option of canceling execution of the function.

4. If the `sessID` parameter is 0, the `DBStartQuery` function calls the `DBInit` function to initiate a session, and returns a session ID.

5. If you specify a nonzero value for the `statusProc` parameter and the `DBStartQuery` function calls the `DBInit` function, the `DBStartQuery` function calls your status routine with the `kDBInitComplete` constant in the `message` parameter and the result of the `DBInit` function in the function result.

6. The `DBStartQuery` function calls the `DBSend` function to send the query to the data server.

7. If you specify a nonzero value for the `statusProc` parameter, the `DBStartQuery` function calls your status routine with the `kDBSendComplete` constant in the `message` parameter and the result of the `DBSend` function in the `result` parameter.

8. The `DBStartQuery` function calls the `DBExec` function to execute the query.

9. If you specify a nonzero value for the `statusProc` parameter, the `DBStartQuery` function calls your status routine with the `kDBExecComplete` constant in the `message` parameter and the result of the `DBExec` function in the `result` parameter.

10. If you specify a nonzero value for the `statusProc` parameter, the `DBStartQuery` function calls your status routine with the `kDBStartQueryComplete` constant in the `message` parameter and the result of the `DBStartQuery` function in the `result` parameter.

SPECIAL CONSIDERATIONS

The `DBStartQuery` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBStartQuery function are

Trap macro	Selector
_DBStartQuery	$0811

RESULT CODES

noErr	0	No error
userCanceledErr	–128	User canceled the query
rcDBError	–802	Error initiating session, sending text, or executing query
rcDBBadSessID	–806	Session ID is invalid
rcDBBadDDev	–808	Couldn't find the specified database extension, or error occurred in opening database extension
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

See "Using the High-Level Interface" beginning on page 12-14 for a general description of how the DBStartQuery function works in conjunction with other Data Access Manager routines. See Listing 12-1 beginning on page 12-18 for an example of the use of the DBStartQuery function. For a description of the query record, see page 12-56. For information on how to write a query definition function or status routine, see "Writing a Query Definition Function" beginning on page 12-51 and "Writing a Status Routine for High-Level Functions" beginning on page 12-22, respectively. Descriptions of the DBInit, DBSend, and DBExec functions begin on page 12-68, page 12-76, and page 12-78, respectively.

High-Level Interface: Handling Query Results

The high-level interface to the Data Access Manager allows applications to manipulate query documents and to get the results of the query provided by a query document. The high-level routines in this section retrieve query results and convert them to text.

DBGetQueryResults

You can use the DBGetQueryResults function to retrieve the results of a query.

```
FUNCTION DBGetQueryResults (sessID: LongInt;
                            VAR results: ResultsRecord;
                            timeout: LongInt; statusProc: ProcPtr;
                            asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The session ID of the session from which you wish to retrieve results.

results   The results record, which contains handles to the retrieved data.

timeout   The value that the DBGetQueryResults function uses for the
                timeout parameter each time it calls the DBGetItem function.
                The timeout parameter specifies the maximum amount of time that the
                database extension should wait to receive results from the data server
                before canceling the DBGetItem function. Specify the timeout
                parameter in sixtieths of a second. To disable the timeout feature, set the
                timeout parameter to the kDBWaitForever constant. Some database
                extensions ignore the timeout parameter when you specify a nonzero
                value for the asyncPB parameter.

statusProc
                A pointer to a status routine that you provide. The DBGetQueryResults
                function calls your status routine after it calls the DBGetItem function to
                retrieve a data item. When it calls the status routine, the
                DBGetQueryResults function provides the result of the DBGetItem
                function, the data type, the data length, the number of decimal places, the
                flags associated with the data item, and a pointer to the data item.

asyncPB   A pointer to an asynchronous parameter block. If specified, the
                DBGetQueryResults function calls the DBGetItem function
                asynchronously for each data item. As soon as the DBGetItem function
                has started execution, it returns control to your application. Your
                application must then call the Event Manager's WaitNextEvent
                function periodically to allow this asynchronous routine to run, and it
                must check the result field of the asynchronous parameter block to
                determine when the routine has completed execution.

DESCRIPTION

The DBGetQueryResults function retrieves the results returned by a query and places
them in memory. If there is sufficient memory available, this function retrieves all of the
results at once. If the DBGetQueryResults function runs out of memory, it places as
much data as possible in memory, up to the last whole row. You can then make more
memory available and call the DBGetQueryResults function again to retrieve more
data.

You must allocate the results record and pass this record to the DBGetQueryResults
function. The Data Access Manager allocates the handles inside the results record. When
your application is finished using the results record, you must deallocate both the results
record and the handles inside the results record.

The DBGetQueryResults function can be used to retrieve the results of any query, not
only queries sent and executed by the DBStartQuery function.

SPECIAL CONSIDERATIONS

The DBGetQueryResults function may move or purge memory. You should not call
this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBGetQueryResults function are

Trap macro	Selector
_DBGetQueryResults	$0A12

RESULT CODES

noErr	0	No error
userCanceledErr	–128	Function canceled by status routine
rcDBValue	–801	Data available
rcDBError	–802	Query execution ended in an error
rcDBBreak	–804	Function timed out
rcDBExec	–805	Query currently executing
rcDBBadSessID	–806	Session ID is invalid
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

See Listing 12-1 beginning on page 12-18 for an example of the use of the DBGetQueryResults function. See page 12-55 for a description of the asynchronous parameter block. Descriptions of the DBStartQuery and DBGetItem functions begin on page 12-63 and page 12-83, respectively. For more information on results records, see "The Results Record" beginning on page 12-58 and "Getting Query Results" beginning on page 12-36. For more information on status routines, see "Writing a Status Routine for High-Level Functions" beginning on page 12-22.

DBResultsToText

After retrieving a results record from DBGetQueryResults, you can use the DBResultsToText function to convert the returned data to text.

```
FUNCTION DBResultsToText (results: ResultsRecord;
                         VAR theText: Handle): OSErr;
```

results   The results record returned by the DBGetQueryResults function.

theText   The DBResultsToText function returns a handle to the converted text in this parameter. This handle is allocated by the Data Access Manager.

DESCRIPTION

The DBResultsToText function calls result handlers to convert to text the data retrieved by the DBGetQueryResults function.

SPECIAL CONSIDERATIONS

The DBResultsToText function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBResultsToText function are

Trap macro	Selector
_DBResultsToText	$0413

RESULT CODES

noErr	0	No error
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

See Listing 12-1 begining on page 12-18 for an example of the use of the DBResultsToText function. See "Converting Query Results to Text" beginning on page 12-42 for a discussion of result handlers.

Low-Level Interface: Controlling the Session

The low-level interface to the Data Access Manager allows applications to open and close sessions with a data server, send and execute queries, retrieve query results, and obtain information about any current session.

DBInit

Use the DBInit function to initiate a session with a data server.

```
FUNCTION DBInit (VAR sessID: LongInt; ddevName: Str63;
                host: Str255; user: Str255; password: Str255;
                connStr: Str255;
                asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The DBInit function returns the session ID in this parameter. This session ID is unique; no other current session, for any database extension, has the same session ID. You must specify the session ID any time you want to send data to or retrieve data from this session. Depending on the database extension you are using, the DBInit function might return a

session ID of 0 if it fails to initiate a session, or it might return a nonzero session ID and a result code other than noErr. In the latter case, you can pass the session ID to the DBGetErr function to determine the cause of the error.

ddevName   A string of no more than 63 characters that specifies the name of the database extension. The name of the database extension is contained in the database extension file in a resource of type 'STR ' with a resource ID of 128. For the Data Access Language database extension provided by Apple, for example, this string is "DAL".

host   The name of the host system on which the data server is located. This name depends on the manner in which the database extension initiates communication with the data server and how the system administrator has set up the computer system.

user   The name of the user.

password   The password associated with the user name.

connStr   A string that is passed to the data server, which might pass it on to the database management software on the host computer. This string is necessary in some systems to complete log-on procedures.

asyncPB   A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

You must initiate a session before you call any Data Access Manager function that requires a session ID as an input parameter. If the DBInit function returns a nonzero session ID, you must call the DBEnd function to terminate the session, even if the DBInit function also returns a result code other than noErr.

Because the high-level function DBStartQuery can call the DBInit function, you do not have to call the DBInit function if you have called the DBStartQuery function.

SPECIAL CONSIDERATIONS

The DBInit function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBInit function are

Trap macro	Selector
_DBInit	$0E02

RESULT CODES

noErr	**0**	No error
rcDBError	**–802**	Error initiating session
rcDBBadDDev	**–808**	Couldn't find the specified database extension, or error occurred in opening database extension
rcDBAsyncNotSupp	**–809**	The database extension does not support asynchronous calls
rcDBPackNotInited	**–813**	The InitDBPack function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-55. See Listing 12-5 beginning on page 12-33 for an example of the use of the DBInit function. See page 12-63 for a description of the DBStartQuery function. The DBEnd function is described next.

DBEnd

You must call the DBEnd function to terminate a session.

```
FUNCTION DBEnd (sessID: LongInt;
                asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The session ID that was returned by the DBInit function.

asyncPB   A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

The DBEnd function terminates a session with a data server and terminates the network connection between the application and the host computer.

SPECIAL CONSIDERATIONS

The DBEnd function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBEnd function are

Trap macro	Selector
_DBEnd	$0403

RESULT CODES

noErr	0	No error
rcDBError	–802	Error ending session
rcDBBadSessID	–806	Session ID is invalid
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-55.

DBGetConnInfo

The DBGetConnInfo function returns information about the specified session, including

n the version of the database extension

n the name of the host system on which the session is running

n the user name

n the connection string that was used to initiate communication

n the name of the network

n the time at which the session started, in ticks (sixtieths of a second)

n the status of the session

```
FUNCTION DBGetConnInfo (sessID: LongInt; sessNum: Integer;
                        VAR returnedID: LongInt;
                        VAR version: LongInt;
                        VAR ddevName: Str63;
                        VAR host: Str255; VAR user: Str255;
                        VAR network: Str255; VAR connStr: Str255;
                        VAR start: LongInt; VAR state: OSErr;
                        asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The session ID that was returned by the DBInit function. If you include a nonzero value for the sessID parameter when you call the DBGetConnInfo function, the function returns the name of the database extension in the ddevName parameter. If you use 0 for the sessID parameter and specify the database extension and session number instead (in the ddevName and sessNum parameters), the function returns the session ID in the returnedID parameter.

sessNum   The session number of the session about which you want information. If you specify a nonzero session number, you must also provide the database extension in the ddevName parameter.

returnedID
: Returns the session ID if you specify the session number and the database extension.

version
: Returns the version number of the database extension that is currently in use.

ddevName
: A string of no more than 63 characters that specifies the name of the database extension. If you specify 0 for the session ID, you must include the name of the database extension as well as a session number. If you specify a valid session ID, then the DBGetConnInfo function returns the name of the database extension in the ddevName parameter. The name of the database extension is contained in the database extension file in a resource of type 'STR ' with a resource ID of 128. For the Data Access Language database extension provided by Apple, for example, this string is "DAL".

host
: Returns the host string used to initiate communication with the data server.

user
: Returns the user string used to initiate communication with the data server.

network
: Returns the name of the network through which the database extension is communicating with the data server. This parameter is an empty string if you are not communicating through a network.

connStr
: Returns the connection string used to initiate communication with the data server.

start
: Returns the time, in ticks (sixtieths of a second), at which this session was initiated.

state
: Returns one of the following values to provide information about the status of the session:

```
CONST noErr      =    0;   {no error--ready for more }
                          { text}
      rcDBValue  = -801;   {output data available}
      rcDBError  = -802;   {execution ended in an }
                          { error}
      rcDBExec   = -805;   {busy--currently executing }
                          { query}
```

asynchPB
: A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

You can use the DBGetConnInfo function to get information about a particular session, or you can call the function repeatedly, incrementing the session number each time, to get information about all of the sessions associated with a particular database extension.

The sessID parameter is the session ID that was returned by the DBInit function. The sessNum parameter is the session number of the session about which you want information. You can specify either the session ID or the session number when you call the DBGetConnInfo function. If you specify the sessID parameter, use 0 for the sessNum parameter. If you specify the sessNum parameter, then use 0 for the sessID parameter. If you specify the sessNum parameter, you must specify a value for the ddevName parameter as well. If you specify the session number and the database extension, then the DBGetConnInfo function returns the session ID in the returnedID parameter.

SPECIAL CONSIDERATIONS

The DBGetConnInfo function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBGetConnInfo function are

Trap macro	Selector
_DBGetConnInfo	$1704

RESULT CODES

noErr	0	No error
rcDBBadSessID	−806	Session ID is invalid or database extension name is invalid
rcDBBadSessNum	−807	Invalid session number
rcDBBadDDev	−808	Couldn't find the specified database extension, or error occurred in opening database extension
rcDBAsyncNotSupp	−809	The database extension does not support asynchronous calls
rcDBPackNotInited	−813	The InitDBPack function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-55. For more information on the use of the DBGetConnInfo function, see "Getting Information About Sessions in Progress" on page 12-35.

DBGetSessionNum

The DBGetSessionNum function returns a session number when you specify the session ID.

```
FUNCTION DBGetSessionNum (sessID: LongInt; VAR sessNum: Integer;
                          asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The session ID that was returned by the DBInit function.

sessNum   Returns the session number of the session you specify with the sessID parameter. The session number is unique for a particular database extension, but the same session number might be in use for different database extensions at the same time.

asyncPB   A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

You can use the DBGetSessionNum function to determine the session numbers for the sessions opened by your own application. You might want this information, for example, so you can distinguish your own sessions from those opened by other applications when you use the DBGetConnInfo function to get information about all open sessions.

SPECIAL CONSIDERATIONS

The DBGetSessionNum function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBGetSessionNum function are

Trap macro	Selector
_DBGetSessionNum	$0605

RESULT CODES

noErr	0	No error
rcDBBadSessID	–806	Session ID is invalid
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

A description of the asynchronous parameter block structure begins on page 12-55. The `DBInit` function description begins on page 12-68. A description of the `DBGetConnInfo` function begins on page 12-71.

DBKill

Use the `DBKill` function to cancel the execution of an asynchronous routine.

```
FUNCTION DBKill (asyncPB: DBAsyncParmBlkPtr): OSErr;
```

asyncPB   A pointer to an asynchronous parameter block.

DESCRIPTION

The `DBKill` function cancels the execution of the asynchronous call specified by the `asyncPB` parameter.

SPECIAL CONSIDERATIONS

The `DBKill` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
_DBKill	$020E

RESULT CODES

noErr	0	No error
rcDBError	–802	Error canceling routine
rcDBBadAsynchPB	–810	Invalid parameter block specified
rcDBPackNotInited	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-55.

Low-Level Interface: Sending and Executing Queries

The functions in this section send queries or portions of queries to the data server, execute queries that have been sent, return information about queries that have been sent, and halt execution of queries that are executing.

DBSend

You can use the DBSend function to send a query or a portion of a query to a data server.

```
FUNCTION DBSend (sessID: LongInt; text: Ptr; len: Integer;
                 asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The session ID that was returned by the DBInit function.

text   A pointer to the query or query fragment that you want to send to the data server. The query or query fragment must be a character string.

len   The length of the character string. If the len parameter has a value of –1, then the character string is assumed to be NULL terminated (that is, the string ends with a NULL byte); otherwise, the len parameter specifies the number of bytes in the string.

asyncPB   A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

The DBSend function sends a query or a portion of a query to the data server. The data server appends this portion of the query to any portion you sent previously. Because the Data Access Manager does not modify the string you send in any way, it does not insert any delimiter between fragments of queries that you send to the data server. If you want a blank or a semicolon to be included between query fragments, or if you want to use return characters to divide the query into lines of text, you must include them in the character string that you send with this function.

The data server does not execute the query until you call the DBExec function.

SPECIAL CONSIDERATIONS

The DBSend function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBSend function are

Trap macro   **Selector**

_DBSend $0706

RESULT CODES

noErr	0	No error
rcDBError	–802	Error trying to send text
rcDBBadSessID	–806	Session ID is invalid
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-55. See Listing 12-4 beginning on page 12-31 for an example of the use of the DBSend function in sending a query fragment. See page 12-78 for a description of the DBExec function.

DBSendItem

You can use the DBSendItem function to send to the data server the data that you wish to include in a query.

```
FUNCTION DBSendItem (sessID: LongInt; dataType: DBType;
                     len: Integer; places: Integer;
                     flags: Integer; buffer: Ptr;
                     asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The session ID that was returned by the DBInit function.

dataType   The data type for the data item that you are sending to the data server.

len   The length of the data item that you are sending to the data server. The database extension and data server ignore the len parameter if the data type has an implied length.

places   The number of decimal places for the data item that you are sending to the data server. The database extension and data server ignore the places parameter for all values of the dataType parameter except typeDecimal and typeMoney.

flags   Set the flags parameter to 0. There are no flags currently defined for the DBSendItem function.

buffer   A pointer to the memory location of the data item that you want to send. When you use the DBSendItem function to send an item of data to a data server, the database extension and data server format the data according to the data type, length, and decimal places you specify, convert it to a character string, and append the data to the query.

asyncPB   A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

The `DBSendItem` function sends a single data item to the data server. The database extension or the data server (depending on how the system is implemented) converts the data item to a character string and appends it to the query, just as the `DBSend` function appends a query program fragment to the query. The query is not executed until you call the `DBExec` function.

SPECIAL CONSIDERATIONS

The `DBSendItem` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
_DBSendItem	$0B07

RESULT CODES

noErr	0	No error
rcDBError	–802	Error trying to send item
rcDBBadSessID	–806	Session ID is invalid
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

For a discussion of data types, see "Getting Query Results" beginning on page 12-36. For a description of the asynchronous parameter block, see page 12-55. See Listing 12-4 beginning on page 12-31 for an example of the use of the `DBSendItem` function in sending a query fragment. See page 12-76 for a description of the `DBSend` function. The `DBExec` function is described next.

DBExec

The `DBExec` function initiates execution of a query that you have sent to a data server.

```
FUNCTION DBExec (sessID: LongInt;
                 asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The session ID that was returned by the `DBInit` function.

asyncPB   A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

The DBExec function initiates execution of a query that you have sent to a data server. You can use the DBState function to determine the status of a query after you have initiated execution.

SPECIAL CONSIDERATIONS

The DBExec function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBExec function are

Trap macro	Selector
_DBExec	$0408

RESULT CODES

noErr	0	No error
rcDBError	–802	Error trying to begin execution
rcDBBadSessID	–806	Session ID is invalid
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-55. See Listing 12-5 beginning on page 12-33 for an example of the use of the DBExec function. Descriptions of the DBSend and DBSendItem functions begin on page 12-76 and page 12-77, respectively. The DBState function is described next.

DBState

You can use the DBState function to determine whether the data server has successfully executed a query and whether it has data available for you to retrieve.

```
FUNCTION DBState (sessID: LongInt;
                  asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The session ID that was returned by the DBInit function.

asyncPB   A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

The DBState function returns a result code that indicates the status of the data server.

SPECIAL CONSIDERATIONS

The DBState function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBState function are

Trap macro	Selector
_DBState	$0409

RESULT CODES

noErr	0	No error
rcDBValue	–801	Output data available
rcDBError	–802	Error executing function
rcDBExec	–805	Query currently executing
rcDBBadSessID	–806	Session ID is invalid
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-55.

DBGetErr

The DBGetErr function retrieves error codes and error messages from a data server. You can use this function to obtain information when a low-level function returns the result code rcDBError.

```
FUNCTION DBGetErr (sessID: LongInt; VAR err1: LongInt;
                   VAR err2: LongInt; VAR item1: Str255;
                   VAR item2: Str255; VAR errorMsg: Str255;
                   asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The session ID that was returned by the DBInit function.

err1   Returns the primary error code.

err2   Returns the secondary error code.

item1   Returns a string that describes the object of the error message.

`item2`	Returns a string that describes the object of the error message.
`errorMsg`	Returns the error message.
`asyncPB`	A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to `NIL`.

DESCRIPTION

If the `DBState` function returns the `rcDBError` result code, indicating that execution of a query ended in an error, the error information retuned by `DBGetErr` can help you debug the query. The meaning of each error code and error message returned by this function depends on the data server with which you are communicating; see the documentation for that data server for more information.

SPECIAL CONSIDERATIONS

The `DBGetErr` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
`_DBGetErr`	`$0E0A`

RESULT CODES

`noErr`	0	No error
`rcDBError`	–802	Error retrieving error information
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-55.

DBBreak

You can use the `DBBreak` function to cancel a query—for example, if you determine that it is taking too long to complete execution.

```
FUNCTION DBBreak (sessID: LongInt; abort: Boolean;
                  asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The session ID that was returned by the DBInit function.

abort   A Boolean value that indicates how DBBreak should cancel the query.
                Specify TRUE (nonzero) to cause the data server to halt any query that is
                executing and terminate the current session. Specify FALSE (0) to cause
                the data server to halt any query that is executing and reinitialize itself.

asyncPB   A pointer to an asynchronous parameter block. If you do not want to call
                the function asynchronously, set this parameter to NIL.

DESCRIPTION

The DBBreak function can halt execution of a query and reinitialize the data server, or it
can unconditionally terminate a session with a data server.

SPECIAL CONSIDERATIONS

The DBBreak function may move or purge memory. You should not call this routine
from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBBreak function are

Trap macro	Selector
_DBBreak	$050B

RESULT CODES

noErr	0	No error
rcDBError	–802	Error executing function
rcDBBadSessID	–806	Session ID is invalid
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-55.

Low-Level Interface: Retrieving Results

The functions in this section allow you to retrieve a data item from the data server, to
obtain information about the next data item, and to retrieve the same data item more
than once.

DBGetItem

After you have executed a query and the DBState function returns the rcDBValue result code, indicating that data is available, you can use the DBGetItem function to retrieve the next data item. You can also use this function to obtain information about the next data item without retrieving the data.

```
FUNCTION DBGetItem (sessID: LongInt; timeout: LongInt;
                    VAR dataType: DBType; VAR len: Integer;
                    VAR places: Integer; VAR flags: Integer;
                    buffer: Ptr; asyncPB: DBAsyncParmBlkPtr)
                    : OSErr;
```

sessID   The session ID that was returned by the DBInit function.

timeout   The maximum amount of time that the database extension should wait to receive results from the data server before canceling the function. Specify the timeout parameter in ticks (sixtieths of a second). To disable the timeout feature, set the timeout parameter to the kDBWaitForever constant. If the timeout period expires, the DBGetItem function returns the rcDBBreak result code. The DBGetItem function ignores the timeout parameter if you call the function asynchronously.

One use for the timeout parameter is to call the DBGetItem function periodically with a short value set for this parameter in order to return control to your application while a query is executing. Your application can then retrieve the next data item as soon as execution of the query is complete without having to call the DBState function to determine when data is available.

dataType   The data type that you expect the next data item to be. If the item is not of the expected data type, the database extension returns the rcDBBadType result code. If you want to retrieve the next data item regardless of type, set the dataType parameter to the typeAnyType constant. To skip the next data item, set the dataType parameter to the typeDiscard constant. The data server sets the dataType parameter to the actual type of the data item when it retrieves the data item or returns information about the data item.

len   The length of the data buffer pointed to by the buffer parameter. If you use the DBGetItem function to obtain information only (by setting the buffer parameter to NIL), then the data server ignores the len parameter. The data server sets the len parameter to the actual length of the data item when it retrieves the data item or returns information about the data item.

places   Returns the number of decimal places in data items of types typeMoney and typeDecimal. For all other data types, the data server returns 0 for the places parameter.

flags   If the least significant bit of the `flags` parameter is set to 1, the data item
               is in the last column of the row. If the third bit of this parameter is set to 1,
               the data item is NULL. You can use the constants `kDBLastColFlag` and
               `kDBNullFlag` to test for these flag bits.

buffer   A pointer to the location where you want the retrieved data item to be
               stored. You must ensure that the location you specify contains enough
               space for the data item that will be returned. To determine the data type,
               length, and number of decimal places of the next data item without
               retrieving it, specify NIL for the `buffer` parameter.

asyncPB   A pointer to an asynchronous parameter block. If you do not want to call
               the function asynchronously, set this parameter to NIL.

DESCRIPTION

The `DBGetItem` function retrieves the next data item from the data server. You can
repeat the `DBGetItem` function as many times as is necessary to retrieve all of the data
returned by the data source in response to a query.

SPECIAL CONSIDERATIONS

The `DBGetItem` function may move or purge memory. You should not call this routine
from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
_DBGetItem	$100C

RESULT CODES

noErr	0	No error
rcDBNull	–800	The data item was NULL
rcDBValue	–801	Data available was successfully retrieved
rcDBError	–802	Error executing function
rcDBBadType	–803	Next data item not of requested data type
rcDBBreak	–804	Function timed out
rcDBBadSessID	–806	Session ID is invalid
rcDBAsyncNotSupp	–809	The database extension does not support asynchronous calls
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

For a discussion of data types, see "Getting Query Results" beginning on page 12-36. To retrieve all of a query's data items at once, use the high-level function DBGetQueryResults; a description of that function begins on page 12-65. For a description of the asynchronous parameter block, see page 12-55. See Listing 12-5 beginning on page 12-33 for an example that illustrates the use of the DBGetItem function.

DBUnGetItem

The DBUnGetItem function reverses the effect of the last call to the DBGetItem function, in the sense that the next time you call the DBGetItem function it retrieves the same item a second time.

```
FUNCTION DBUnGetItem (sessID: LongInt;
                          asyncPB: DBAsyncParmBlkPtr): OSErr;
```

sessID   The session ID that was returned by the DBInit function.

asyncPB   A pointer to an asynchronous parameter block. If you do not want to call the function asynchronously, set this parameter to NIL.

DESCRIPTION

The DBUnGetItem function does not remove the just-retrieved data item from the input buffer. This function can reverse the effect of only one call to the DBGetItem function; you cannot use it to step back through several previously retrieved data items.

SPECIAL CONSIDERATIONS

The DBUnGetItem function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBUnGetItem function are

Trap macro	Selector
_DBUnGetItem	$040D

`noErr`	0	No error
`rcDBError`	–802	Error executing function
`rcDBBadSessID`	–806	Session ID is invalid
`rcDBAsyncNotSupp`	–809	The database extension does not support asynchronous calls
`rcDBPackNotInited`	–813	The `InitDBPack` function has not yet been called

SEE ALSO

For a description of the asynchronous parameter block, see page 12-55. See page 12-83 for a description of the `DBGetItem` function.

Installing and Removing Result Handlers

The functions in this section install, remove, and return pointers to result handlers.

DBInstallResultHandler

The `DBInstallResultHandler` function installs a result handler for the data type specified by the `dataType` parameter. The result handler is then used by the `DBResultsToText` function to convert data of the specified type into a character string.

```
FUNCTION DBInstallResultHandler (dataType: DBType;
                                 theHandler: ProcPtr;
                                 isSysHandler: Boolean): OSErr;
```

`dataType`   The type of result handler to install.

`theHandler`
            A pointer to a result handler.

`isSysHandler`
            A Boolean value that specifies whether the result handler is an application result handler—to be used only when the `DBResultsToText` function is called by the application that installed the result handler—or a system result handler—to be used by every application running on the system. If the `isSysHandler` parameter is `TRUE`, the result handler is a system result handler.

DESCRIPTION

When you install an application result handler, it replaces any result handler with the same name previously installed by that application. Similarly, when you install a system result handler, it replaces any existing system result handler with the same name. Before you temporarily replace an existing result handler, use the `DBGetResultHandler`

function to obtain a pointer to the present handler, and save the present result handler in your application's private storage. Then you can reinstall the original result handler when you are finished using the temporary one.

Because an application result handler is used in preference to a system result handler if both are available, you can temporarily replace a system result handler for purposes of your application by installing an application result handler for the same data type. You can then use the `DBRemoveResultHandler` function to remove the application result handler and return to using the system result handler whenever you wish.

SPECIAL CONSIDERATIONS

The `DBInstallResultHandler` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
_DBInstallResultHandler	$0514

RESULT CODES

noErr	0	No error
rcDBPackNotInited	–813	The `InitDBPack` function has not yet been called

SEE ALSO

See page 12-67 for a description of the `DBResultsToText` function. For information on application and system result handlers, see "Converting Query Results to Text" beginning on page 12-42; that section also lists the data types for which Apple provides system result handlers. Listing 12-6 on page 12-45 shows a sample result handler. The `DBRemoveResultHandler` function is described on page 12-89, and the `DBGetResultHandler` function is described next.

DBGetResultHandler

The `DBGetResultHandler` function returns a pointer to a result handler for a specified data type.

```
FUNCTION DBGetResultHandler (dataType: DBType;
                             VAR theHandler: ProcPtr;
                             getSysHandler: Boolean): OSErr;
```

dataType   The data type for which to install a result handler.

`theHandler`
: Returns a pointer to the result handler.

`getSysHandler`
: If you set the `getSysHandler` parameter to `FALSE` (0), the function returns a pointer to the current application result handler for the specified data type, or it returns `NIL` if there is no application result handler for that data type. If you set the `getSysHandler` parameter to `TRUE` (nonzero), the function returns a pointer to the current system result handler for the specified data type, or it returns `NIL` if there is no system result handler for that data type.

DESCRIPTION

You can use the `DBGetResultHandler` function to obtain a pointer to a result handler so that you can use it to convert to text an individual data item retrieved by the `DBGetItem` function. The `DBGetQueryResults` function automatically converts to text all of the data pointed to by the results record.

SPECIAL CONSIDERATIONS

The `DBGetResultHandler` function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
_DBGetResultHandler	$0516

RESULT CODES

noErr	0	No error
rcDBNoHandler	–811	There is no handler for this data type installed for the current application
rcDBPackNotInited	–813	The `InitDBPack` function has not yet been called

SEE ALSO

The `DBGetQueryResults` function is described on page 12-65, and the `DBGetItem` function is described on page 12-83. See "Converting Query Results to Text" beginning on page 12-42 for a list of the data types for which Apple provides system result handlers. Listing 12-6 on page 12-45 shows a sample result handler.

DBRemoveResultHandler

You can use the DBRemoveResultHandler function to remove an application result handler.

```
FUNCTION DBRemoveResultHandler (dataType: DBType): OSErr;
```

dataType   The type of result handler to remove.

DESCRIPTION

The DBRemoveResultHandler function removes from memory the specified application result handler. This function cannot remove a system result handler.

SPECIAL CONSIDERATIONS

The DBRemoveResultHandler function may move or purge memory. You should not call this routine from within an interrupt, such as in a completion routine or a VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the DBRemoveResultHandler function are

Trap macro	Selector
_DBRemoveResultHandler	$0215

RESULT CODES

noErr	0	No error
rcDBNoHandler	–811	There is no handler for this data type installed for the current application
rcDBPackNotInited	–813	The InitDBPack function has not yet been called

SEE ALSO

For a discussion of result handlers, see "Converting Query Results to Text" beginning on page 12-42.

Application-Defined Routines

You can provide status functions, result handler functions, and query defintion functions for use with the Data Access Manager. For information on status functions, see "Writing a Status Routine for High-Level Functions" beginning on page 12-22. See "Processing Query Results" beginning on page 12-36 for information on result handlers. See "Writing a Query Definition Function" beginning on page 12-51 for information on query definition functions.

Resources

This section describes the query resource, the query string resource, and the query definition function resource. You use the query resource to provide information that the Data Access Manager uses to create a query record in memory. You use the query string resource to define individual queries. You use the query definition function to modify a query and the query record before the query is sent to the data server.

The Query Resource

Each query document should contain a single 'qrsc' resource. Figure 12-9 shows the format of the 'qrsc' resource.

Figure 12-9   Structure of a compiled query ('qrsc') resource
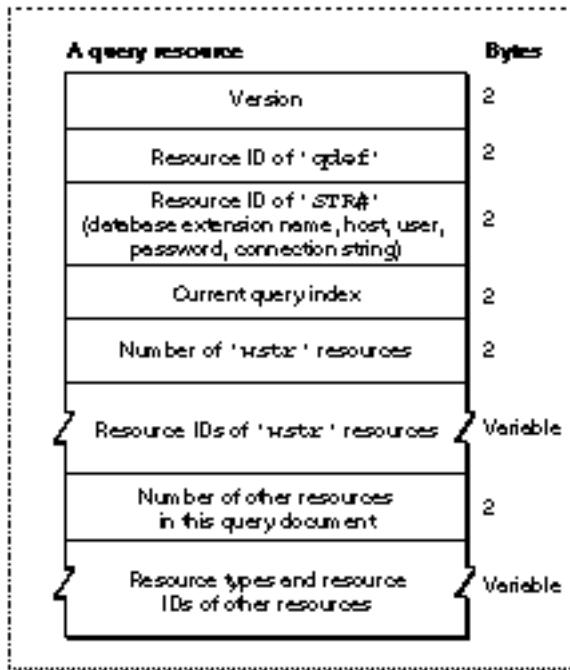
A 'qrsc' resource contains these elements:

n The version number of the 'qrsc' format. For the Data Access Manager released with System 7, the version number is 0.

n The resource ID of the 'qdef' resource containing the query definition function that the Data Access Manager is to call when it opens this 'qrsc' resource. Use an ID of 0 if there is no query definition function for this resource—that is, if the Data Access Manager should send the query in this resource to the data server without modifications.

n The resource ID of an 'STR#' resource that contains five Pascal strings corresponding to some of the parameters used by the DBInit function. If the query definition function is going to prompt the user for the values of these parameters before entering them in the query record, they should be zero-length strings in the 'STR#' resource.

n An index value indicating which element in the array of 'wstr' IDs represents the current query. The current query is the one actually sent to the data server.

n The number of 'wstr' resources in the query document.

n An array of resource IDs of the 'wstr' resources in the query document. (The array elements are numbered starting with 1.) If the query document contains more than one 'wstr' resource, the query definition function can prompt the user to select the query to use and modify the current query field in the query record appropriately.

n The number of other resources in this query document.

n An array listing the resource types and IDs of all the resources in the query document other than the standard resources included in all query documents. The resources listed in this final array are those used by the query definition function. This list should include resources embedded in other resources, such as a 'PICT' resource that is included in a 'DITL' resource.

The Query String Resource

A query document must contain one or more query string resources of type 'wstr'. These 'wstr' resources contain individual queries—that is, strings of commands and data that the DBSend function sends to the data server and that the DBExec function executes.

A 'wstr' resource consists of a 2-byte length field followed by a character string. (The *w* in 'wstr' refers to the length word as opposed to the length byte used in an 'STR ' resource.) Each 'wstr' resource contains one query (or one query template, to be modified by the query definition function before it is sent to the data server). Figure 12-10 shows the structure of the 'wstr' resource.

Figure 12-10 Structure of a compiled query string ('wstr') resource
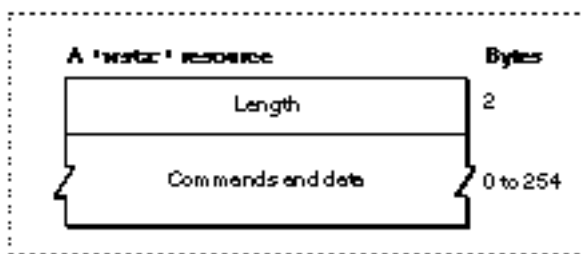
The 'qrsc' resource includes an array that lists the resource ID numbers of all of the 'wstr' resources in the query document and an index into the array that specifies which one of the 'wstr' resources should be sent to the data server.

The Query Definition Function Resource

A query document may contain a query definition function, which can modify the query record and, if necessary, fill in the query template to create a complete query.

If you want to include a query definition function, you must make it the first piece of code in a resource of type `'qdef'` in the query document.

Note that, because a query in memory consists only of a 2-byte length value followed by a character string, the query definition function has to know the exact contents and structure of a query in order to modify it. For a sample query definition function that uses a dialog box to prompt the user for a user name and password, see Listing 12-7 on page 12-52.

Summary of the Data Access Manager

Pascal Summary

Constants

```
CONST
      gestaltDBAccessMgrAttr      = 'dbac';     {Gestalt selector for }
                                                { Data Access Manager}
      {Gestalt selector response}
      gestaltDBAccessMgrPresent  = 0;           {TRUE if Data Access Manager }
                                                { is present}

      {DBStartQuery status messages}
      kDBUpdateWind               = 0;          {update windows}
      kDBAboutToInit              = 1;          {about to call DBInit}
      kDBInitComplete             = 2;          {DBInit has completed}
      kDBSendComplete             = 3;          {DBSend has completed}
      kDBExecComplete             = 4;          {DBExec has completed}
      kDBStartQueryComplete       = 5;          {DBStartQuery is about }
                                                { to complete}

      {DBGetQueryResults status messages}
      kDBGetItemComplete          = 6;          {DBGetItem has completed}
      kDBGetQueryResultsComplete = 7;           {DBGetQueryResults has }
                                                { completed data types}

      {data type codes}
      typeNone                    = 'none';     {no more data expected}
      typeDate                    = 'date';     {date}
      typeTime                    = 'time';     {time}
      typeTimeStamp               = 'tims';     {date and time}
      typeDecimal                 = 'deci';     {decimal number}
      typeMoney                   = 'mone';     {money value}
      typeVChar                   = 'vcha';     {variable character}
      typeVBin                    = 'vbin';     {variable binary}
      typeLChar                   = 'lcha';     {long character}
      typeLBin                    = 'lbin';     {long binary}
      typeDiscard                 = 'disc';     {discard next data item}
      typeBoolean                 = 'bool';     {Boolean}
      typeChar                    = 'TEXT';     {character}
```

```
typeSMInt               = 'shor';    {short integer}
typeInteger             = 'long';    {integer}
typeSMFloat             = 'sing';    {short floating point}
typeFloat               = 'doub';    {floating point}
{dummy data types for DBResultsToText}
typeUnknown             = 'unkn';    {result handler for unknown }
                                     { data type}
typeColBreak            = 'colb';    {result handler for column }
                                     { break}
typeRowBreak            = 'rowb';    {result handler for end of }
                                     { line}
{any data type in DBGetItem}
typeAnyType             = 0;         {any data type}
{infinite timeout value for DBGetItem}
kDBWaitForever          = -1;        {infinite timeout value for }
                                     { DBGetItem}
{flags for DBGetItem}
kDBLastColFlag          = $0001;     {data item is last column }
                                     { of the row}
kDBNullFlag             = $0004;     {data item is NULL}
```

Data Types

```
TYPE DBType =        OSType;                {data type}

  DBAsyncParamBlockRec =                    {asynchronous parameter block}
  RECORD
    completionProc:   ProcPtr;             {pointer to completion routine}
    result:           OSErr;               {result of call}
    userRef:          LongInt;             {reserved for use by }
                                           { application}
    ddevRef:          LongInt;             {reserved for use by database }
                                           { extension}
    reserved:         LongInt;             {reserved for use by }
                                           { Data Access Mgr}
  END;
  DBAsyncParmBlkPtr = ^DBAsyncParamBlockRec;

  ResListElem =                            {resource list in QueryRecord}
  RECORD
    theType:          ResType;             {resource type}
    id:               Integer;             {resource ID}
  END;
```

```
ResListArray   = ARRAY[0..255] OF ResListElem;
ResListPtr     = ^ResListArray;
ResListHandle  = ^ResListPtr;

QueryRecord =
RECORD
   version:      Integer;          {query record format version}
   id:           Integer;          {resource ID of 'qrsc'}
   queryProc:    Handle;           {handle to query def proc}
   ddevName:     Str63;            {name of database extension}
   host:         Str255;           {name of host computer}
   user:         Str255;           {name of user}
   password:     Str255;           {user's password}
   connStr:      Str255;           {connection string}
   currQuery:    Integer;          {index of current query}
   numQueries:   Integer;          {number of queries in list}
   queryList:    QueryListHandle;  {handle to array of handles to text}
   numRes:       Integer;          {number of resources in list}
   resList:      ResListHandle;    {handle to array of resource list }
                                   { elements}
   dataHandle:   Handle;           {handle to memory for query def proc}
   refCon:       LongInt;          {reserved for use by application}
END;
QueryPtr    = ^QueryRecord;        {pointer to query record}
QueryHandle = ^QueryPtr;           {handle to query record}
{query list in QueryRecord}
QueryArray       = ARRAY[0..255] OF Handle;
QueryListPtr     = ^QueryArray;
QueryListHandle  = ^QueryListPtr;

{column types array in ResultsRecord}
ColTypesArray  = ARRAY[0..255] OF DBType;
ColTypesPtr    = ^ColTypesArray;
ColTypesHandle = ^ColTypesPtr;

DBColInfoRecord =                      {column info in ResultsRecord}
RECORD
   len:       Integer;          {length of data item}
   places:    Integer;          {places for decimal and money }
                                { data items}
   flags:     Integer;          {flags for data item}
END;
```

```
ColInfoArray   = ARRAY[0..255] OF DBColInfoRecord;
ColInfoPtr     = ^ColInfoArray;
ColInfoHandle  = ^ColInfoPtr;

{structure of results returned by DBGetResults}
ResultsRecord =
RECORD
   numRows:      Integer;           {number of rows retrieved}
   numCols:      Integer;           {number of columns per row}
   colTypes:     ColTypesHandle;    {type of data in each column}
   colData:      Handle;            {array of data items}
   colInfo:      ColInfoHandle;     {DBColInfoRecord array--info about }
                                    { each data item}
END;
```

Data Access Manager Routines

Initializing the Data Access Manager

```
FUNCTION InitDBPack:          OSErr;
```

High-Level Interface: Handling Query Documents

```
FUNCTION DBGetNewQuery        (queryID: Integer; VAR query: QueryHandle)
                               : OSErr;
FUNCTION DBDisposeQuery       (query: QueryHandle): OSErr;
FUNCTION DBStartQuery         (VAR sessID: LongInt; query: QueryHandle;
                               statusProc: ProcPtr;
                               asyncPB: DBAsyncParmBlkPtr): OSErr;
```

High-Level Interface: Handling Query Results

```
FUNCTION DBGetQueryResults    (sessID: LongInt; VAR results: ResultsRecord;
                               timeout: LongInt; statusProc: ProcPtr;
                               asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBResultsToText      (results: ResultsRecord; VAR theText: Handle)
                               : OSErr;
```

Low-Level Interface: Controlling the Session

```
FUNCTION DBInit               (VAR sessID: LongInt; ddevName: Str63;
                               host: Str255; user: Str255; password: Str255;
                               connStr: Str255; asyncPB: DBAsyncParmBlkPtr)
                               : OSErr;
```

```
FUNCTION DBEnd             (sessID: LongInt;
                            asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBGetConnInfo     (sessID: LongInt; sessNum: Integer;
                            VAR returnedID: LongInt; VAR version: LongInt;
                            VAR ddevName: Str63; VAR host: Str255;
                            VAR user: Str255; VAR network: Str255;
                            VAR connStr: Str255; VAR start: LongInt;
                            VAR state: OSErr; asyncPB: DBAsyncParmBlkPtr)
                            : OSErr;
FUNCTION DBGetSessionNum   (sessID: LongInt; VAR sessNum: Integer;
                            asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBKill            (asyncPB: DBAsyncParmBlkPtr): OSErr;
```

Low-Level Interface: Sending and Executing Queries

```
FUNCTION DBSend            (sessID: LongInt; text: Ptr; len: Integer;
                            asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBSendItem        (sessID: LongInt; dataType: DBType;
                            len: Integer; places: Integer; flags: Integer;
                            buffer: Ptr; asyncPB: DBAsyncParmBlkPtr)
                            : OSErr;
FUNCTION DBExec            (sessID: LongInt; asyncPB: DBAsyncParmBlkPtr)
                            : OSErr;
FUNCTION DBState           (sessID: LongInt; asyncPB: DBAsyncParmBlkPtr)
                            : OSErr;
FUNCTION DBGetErr          (sessID: LongInt; VAR err1: LongInt;
                            VAR err2: LongInt; VAR item1: Str255;
                            VAR item2: Str255; VAR errorMsg: Str255;
                            asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBBreak           (sessID: LongInt; abort: Boolean;
                            asyncPB: DBAsyncParmBlkPtr): OSErr;
```

Low-Level Interface: Retrieving Results

```
FUNCTION DBGetItem         (sessID: LongInt; timeout: LongInt;
                            VAR dataType: DBType;
                            VAR len: Integer; VAR places: Integer;
                            VAR flags: Integer; buffer: Ptr;
                            asyncPB: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBUnGetItem       (sessID: LongInt;
                            asyncPB: DBAsyncParmBlkPtr): OSErr;
```

Installing and Removing Result Handlers

```
FUNCTION DBInstallResultHandler
                        (dataType: DBType; theHandler: ProcPtr;
                         isSysHandler: Boolean): OSErr;
FUNCTION DBGetResultHandler (dataType: DBType; VAR theHandler: ProcPtr;
                         getSysHandler: Boolean): OSErr;
FUNCTION DBRemoveResultHandler
                        (dataType: DBType): OSErr;
```

Application-Defined Routines

```
FUNCTION MyStatusFunc       (message: Integer; result: OSErr;
                         dataLen: Integer; dataPlaces: Integer;
                         dataFlags: Integer; dataType: DBType;
                         dataPtr: Ptr): Boolean;
FUNCTION MyResultHandler    (dataType: DBType; theLen: Integer;
                         thePlaces: Integer; theFlags: Integer;
                         theData: Ptr; theText: Handle): OSErr;
FUNCTION MyQDef             (VAR sessID: LongInt;
                         query: QueryHandle): OSErr;
```

C Summary

Constants

```
enum {
   #define gestaltDBAccessMgrAttr   'dbac'   /*Gestalt selector for */
                                             /* Data Access Manager*/
   /*Gestalt selector response*/
   gestaltDBAccessMgrPresent     = 0         /*TRUE if Data Access Manager */
                                             /* is present*/
};
enum {                                       /*DBStartQuery status messages*/
   kDBUpdateWind           = 0,       /*update windows*/
   kDBAboutToInit          = 1,       /*about to call DBInit*/
   kDBInitComplete         = 2,       /*DBInit has completed*/
   kDBSendComplete         = 3,       /*DBSend has completed*/
   kDBExecComplete         = 4,       /*DBExec has completed*/
   kDBStartQueryComplete   = 5        /*DBStartQuery is about */
                                      /* to complete*/
};
```

```
enum {
   /*DBGetQueryResults status messages*/
   kDBGetItemComplete            = 6,      /*DBGetItem has completed*/
   kDBGetQueryResultsComplete    = 7,      /*DBGetQueryResults has */
                                           /* completed data types*/
   /*data type codes*/
   #define typeNone               'none'   /*no more data expected*/
   #define typeDate               'date'   /*date*/
   #define typeTime               'time'   /*time*/
   #define typeTimeStamp          'tims'   /*date and time*/
   #define typeDecimal            'deci'   /*decimal number*/
   #define typeMoney              'mone'   /*money value*/
   #define typeVChar              'vcha'   /*variable character*/
   #define typeVBin               'vbin'   /*variable binary*/
   #define typeLChar              'lcha'   /*long character*/
   #define typeLBin               'lbin'   /*long binary*/
   #define typeDiscard            'disc'   /*discard next data item*/
   /*dummy data types for DBResultsToText*/
   #define typeUnknown            'unkn'   /*result handler for unknown */
                                           /* data type*/
   #define typeColBreak           'colb'   /*result handler for */
                                           /* column break*/
   #define typeRowBreak           'rowb'   /*result handler for */
                                           /* end of line*/
   /*any data type in DBGetItem*/
   #define typeAnyType       (DBType)0     /*any data type*/
   /*infinite timeout value for DBGetItem*/
   kDBWaitForever                 = -1,    /*infinite timeout value for */
                                           /* DBGetItem*/
   /*flags for DBGetItem*/
   kDBLastColFlag                 = 0x0001,/*data item is last column */
                                           /* of the row*/
   kDBNullFlag                    = 0x0004 /*data item is NULL*/
};
enum {
   /*more data type codes*/
   typeBoolean               = 'bool',   /*Boolean*/
   typeChar                  = 'TEXT',   /*character*/
   typeSMInt                 = 'shor',   /*short integer*/
   typeInteger               = 'long',   /*integer*/
   typeSMFloat               = 'sing',   /*short floating point*/
   typeFloat                 = 'doub'    /*floating point*/
};
```

Data Types

```
typedef OSType DBType;                /*data type*/

struct DBAsyncParamBlockRec {         /*asynchronous parameter block*/
   ProcPtr   completionProc;          /*pointer to completion routine*/
   OSErr     result;                  /*result of call*/
   long      userRef;                 /*reserved for use by application*/
   long      ddevRef;                 /*reserved for use by database */
                                      /* extension*/
   long      reserved;                /*reserved for use by */
                                      /* Data Access Manager*/
};
typedef struct DBAsyncParamBlockRec DBAsyncParamBlockRec;
typedef DBAsyncParamBlockRec *DBAsyncParmBlkPtr;

struct ResListElem {                  /*resource list in QueryRecord*/
   ResType   theType;                 /*resource type*/
   short     id;                      /*resource ID*/
};
typedef struct ResListElem ResListElem;
typedef ResListElem *ResLisPtr, **ResListHandle;

typedef Handle **QueryListHandle;
struct QueryRecord {                  /*query record*/
   short            version;          /*query record format version*/
   short            id;               /*resource ID of 'qrsc'*/
   Handle           queryProc;        /*handle to query def proc*/
   Str63            ddevName;         /*name of database extension*/
   Str255           host;            /*name of host computer*/
   Str255           user;            /*name of user*/
   Str255           password;        /*user's password*/
   Str255           connStr;         /*connection string*/
   short            currQuery;        /*index of current query*/
   short            numQueries;       /*number of queries in list*/
   QueryListHandle  queryList;        /*handle to array of handles to text*/
   short            numRes;           /*number of resources in list*/
   ResListHandle    resList;          /*handle to array of resource list */
                                      /* elements*/
   Handle           dataHandle;       /*handle to memory for query def proc*/
   long             refCon;           /*reserved for use by application*/
};
typedef struct QueryRecord QueryRecord;
```

```
typedef QueryRecord *QueryPtr, **QueryHandle;


/*column types array in ResultsRecord*/
typedef Handle ColTypesHandle;

struct DBColInfoRecord {               /*column info in ResultsRecord*/
   short len;                          /*length of data item*/
   short places;                       /*places for decimal and money */
                                       /* data items*/
   short flags;                        /*flags for data item*/
};
typedef struct DBColInfoRecord DBColInfoRecord;
typedef Handle ColInfoHandle;

struct ResultsRecord {                 /*results returned by DBGetResults*/
   short           numRows;            /*number of rows retrieved*/
   short           numCols;            /*number of columns per row*/
   ColTypesHandle  colTypes;           /*type of data in each column*/
   Handle          colData;            /*array of data items*/
   ColInfoHandle   colInfo;            /*DBColInfoRecord array--info about */
                                       /* each data item*/
};
typedef struct ResultsRecord ResultsRecord;
```

Data Access Manager Routines

Initializing the Data Access Manager

```
pascal OSErr InitDBPack      (void);
```

High-Level Interface: Handling Query Documents

```
pascal OSErr DBGetNewQuery   (short queryID, QueryHandle *query);
pascal OSErr DBDisposeQuery (QueryHandle query);
pascal OSErr DBStartQuery    (long *sessID, QueryHandle query,
                              ProcPtr statusProc, DBAsyncParmBlkPtr asyncPB);
```

High-Level Interface: Handling Query Results

```
pascal OSErr DBGetQueryResults
                             (long sessID, ResultsRecord *results,
                              long timeout, ProcPtr statusProc,
                              DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBResultsToText
                             (ResultsRecord *results, Handle *theText);
```

Low-Level Interface: Controlling the Session

```
pascal OSErr DBInit        (long *sessID, ConstStr63Param ddevName,
                            ConstStr255Param host, ConstStr255Param user,
                            ConstStr255Param passwd,
                            ConstStr255Param connStr,
                            DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBEnd         (long sessID, DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBGetConnInfo (long sessID, short sessNum, long *returnedID,
                            long *version, Str63 ddevName, Str255 host,
                            Str255 user, Str255 network, Str255 connStr,
                            long *start, OSErr *state,
                            DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBGetSessionNum
                           (long sessID, short *sessNum,
                            DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBKill        (DBAsyncParmBlkPtr asyncPB);
```

Low-Level Interface: Sending and Executing Queries

```
pascal OSErr DBSend        (long sessID, char *text, short len,
                            DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBSendItem    (long sessID, DBType dataType, short len,
                            short places, short flags, void *buffer,
                            DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBExec        (long sessID, DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBState       (long sessID, DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBGetErr      (long sessID, long *err1, long *err2,
                            Str255 item1, Str255 item2, Str255 errorMsg,
                            DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBBreak       (long sessID, Boolean abort,
                            DBAsyncParmBlkPtr asyncPB);
```

Low-Level Interface: Retrieving Results

```
pascal OSErr DBGetItem     (long sessID, long timeout, DBType *dataType,
                            short *len, short *places, short *flags,
                            void *buffer, DBAsyncParmBlkPtr asyncPB);
pascal OSErr DBUnGetItem   (long sessID, DBAsyncParmBlkPtr asyncPB);
```

Installing and Removing Result Handlers

```
pascal OSErr DBInstallResultHandler
                           (DBType dataType, ProcPtr theHandler,
                            Boolean isSysHandler);
```

```
pascal OSErr DBGetResultHandler
                        (DBType dataType, ProcPtr *theHandler,
                         Boolean getSysHandler);
pascal OSErr DBRemoveResultHandler
                        (DBType dataType);
```

Application-Defined Routines

```
pascal Boolean MyStatusFunc (short message, OSErr result, short dataLen,
                             short dataPlaces, short dataFlags,
                             DBType dataType, Ptr dataPtr);
pascal OSErr  MyResultHandler
                        (DBType dataType, short theLen,
                         short thePlaces, short theFlags, Ptr theData,
                         Handle theText);
pascal OSErr MyQDef          (long *sessID, QueryHandle query);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

`_Pack13`

Selector	Routine
$0100	InitDBPack
$020E	DBKill
$0210	DBDisposeQuery
$0215	DBRemoveResultHandler
$030F	DBGetNewQuery
$0403	DBEnd
$0408	DBExec
$0409	DBState
$040D	DBUnGetItem
$0413	DBResultsToText
$050B	DBBreak
$0514	DBInstallResultHandler
$0516	DBGetResultHandler
$0605	DBGetSessionNum

Selector	Routine
$0706	DBSend
$0811	DBStartQuery
$0A12	DBGetQueryResults
$0B07	DBSendItem
$0E02	DBInit
$0E0A	DBGetErr
$100C	DBGetItem
$1704	DBGetConnInfo

Result Codes

noErr	**0**	No error
userCanceledErr	**–128**	User canceled the query
rcDBNull	**–800**	The data item was NULL
rcDBValue	**–801**	Data available or successfully retrieved
rcDBError	**–802**	Error executing function
rcDBBadType	**–803**	Next data item not of requested data type
rcDBBreak	**–804**	Function timed out
rcDBExec	**–805**	Query currently executing
rcDBBadSessID	**–806**	Session ID is invalid
rcDBBadSessNum	**–807**	Invalid session number
rcDBBadDDev	**–808**	Couldn't find the specified database extension, or error occurred in opening database extension
rcDBAsyncNotSupp	**–809**	The database extension does not support asynchronous calls
rcDBBadAsynchPB	**–810**	Invalid parameter block specified
rcDBNoHandler	**–811**	There is no handler for this data type installed for the current application
rcDBWrongVersion	**–812**	Wrong version number
rcDBPackNotInited	**–813**	The InitDBPack function has not yet been called

Glossary

abstract superclass A superclass listed in the *Apple Event Registry: Standard Suites,* such as `cObject` or `cOpenableObject`, that is used only in definitions of object classes and not for real Apple event objects. See also **object class.**

active function A function called by a scripting component periodically during script compilation and execution. You must provide an alternative active function for the use of scripting components if you want your application to get time during script compilation and execution for tasks such as spinning the cursor or checking for system-level errors.

additional parameter A keyword-specified descriptor record that a server application uses in addition to the data specified in the direct parameter. For example, an Apple event for arithmetic operations may include additional parameters that specify operands in an equation. Additional parameters may be required, or they may be optional.

address descriptor record A descriptor record of data type `AEAddressDesc` that contains the address of the target or source of an Apple event.

AEIMP See **Apple Event Interprocess Messaging Protocol.**

AE record A descriptor record of data type `AERecord` that usually contains a list of parameters for an Apple event. See also **Apple event parameter.**

Apple event A high-level event that adheres to the Apple Event Interprocess Messaging Protocol. An Apple event consists of attributes (including the event class and event ID, which identify the event and its task) and, usually, parameters (which contain data used by the target application for the event). See also **Apple event attribute, Apple event parameter.**

Apple event array An array in a descriptor list. The data for an Apple event array is specified by an array data record, which is defined by the data type `AEArrayData`.

Apple event attribute A keyword-specified descriptor record that identifies the event class, event ID, target application, or some other characteristic of an Apple event. Taken together, the attributes of an Apple event identify the event and denote the task to be performed on the data specified in the Apple event's parameters. Unlike Apple event parameters (which contain data used only by the target application of the Apple event), Apple event attributes contain information that can be used by both the Apple Event Manager and the target application. See also **Apple event parameter.**

Apple event dispatch table A table in either the application heap or the system heap that the Apple Event Manager uses to map Apple events to the appropriate Apple event handlers.

Apple event handler An application-defined function that extracts pertinent data from an Apple event, performs the action requested by the Apple event, and returns a result.

Apple Event Interprocess Messaging Protocol (AEIMP) A standard defined by Apple Computer, Inc., for communication and data sharing among applications. High-level events that adhere to this protocol are called Apple events. See also **Apple event.**

Apple Event Manager The collection of routines that allows client applications to send Apple events to server applications for the purpose of requesting services or information.

Apple event object A distinct item in a target application or any of its documents that can be specified by an object specifier record in an Apple event sent by a source application. Apple event objects can be anything that an application can locate on the basis of such a description,

including items that a user can differentiate and manipulate while using an application, such as words, paragraphs, shapes, windows, or style formats. See also **object specifier record.**

Apple event object class   See **object class.**

Apple event parameter   A keyword-specified descriptor record containing data that the target application for an Apple event uses. Unlike Apple event attributes (which contain information that can be used by both the Apple Event Manager and the target application), Apple event parameters contain data used only by the target application of the Apple event. See also **Apple event attribute, direct parameter, optional parameter, required parameter.**

Apple event record   A descriptor record of data type `AppleEvent` that contains a list of keyword-specified descriptor records. These descriptor records describe—at least—the attributes necessary for an Apple event; they may also describe parameters for the Apple event. Apple Event Manager functions are used to add parameters to an Apple event record.

Apple event user terminology resources   Two resources with identical formats used by server applications to specify the Apple events and corresponding user terminology that the applications support. The `'aeut'` resource, which is provided by scripting components, contains terminology information for all the standard suites of Apple events defined in the *Apple Event Registry: Standard Suites.* An `'aete'` resource must be provided by every scriptable application; it describes which of the standard suites listed in the `'aeut'` resource the application supports and provides additional terminology information for extensions to the standard suites and custom Apple events supported by the application. See also **scripting component.**

AppleScript component   The scripting component that implements the AppleScript scripting language. See also **scripting component.**

AppleScript scripting language   The standard user scripting language defined by Apple Computer, Inc. The AppleScript scripting language is implemented by the AppleScript scripting component. See also **dialect.**

application result handler   A result handler that is associated with a particular application. Compare with **system result handler.**

asynchronous parameter block   In the Data Access Manager, the parameter block that allows a routine to return control to your application before the routine has completed execution.

authentication   The process of establishing the identity of a user. The authentication mechanism of the PPC Toolbox identifies each user through an assigned name and password.

boundary objects   The elements, specified in a range descriptor record, that identify the beginning and end of the range. See also **range descriptor record.**

client application   An application that uses Apple events to request a service (for example, printing a list of files, checking the spelling of a list of words, or performing a numeric calculation) from another application (called a server application). These applications can reside on the same local computer or on remote computers connected to a network.

coercion handler   A routine that coerces data from one descriptor type to another.

coercion handler dispatch table   A table in either the application heap or the system heap that the Apple Event Manager uses to map desired coercions to the appropriate coercion handler. See also **coercion handler.**

comparison descriptor record   A coerced AE record of type `typeCompDescriptor` that specifies an Apple event object and either another Apple event object or data for the Apple Event Manager to compare to the first object.

compiled script   Compiled code that a client application can decompile into source data or execute using the standard scripting component routines.

compiled script file   A script file with the file type `'scpt'` that contains script data as a resource of type `'scpt'`. Before executing the script in a compiled script file, a user must first open the script from the Finder or from an application such as Script Editor.

component-specific storage descriptor record   A descriptor record returned by `OSAStore`. The descriptor type for a component-specific storage descriptor record is the scripting component subtype value for the scripting component that created the script data.

container   An Apple event object that contains another Apple event object. A container is specified in an object specifier record by a keyword-specified descriptor record with the keyword `keyAEContainer`. The keyword-specified descriptor record is usually another object specifier record. It can also be a null descriptor record, or it can be used much like a variable when the Apple Event Manager determines a range or performs a series of tests. The objects a container contains can be either elements or properties. See also **Apple event object, element, object specifier record, property.**

container hierarchy   The chain of containers that determine the location of one or more Apple event objects. See also **container.**

core Apple event   An Apple event defined as part of the Core suite of Apple events in the *Apple Event Registry: Standard Suites.*

create function   A function called by a scripting component whenever it creates an Apple event during script execution. You must provide an alternative create function if you want to gain control over the creation and addressing of Apple events. If you don't provide an alternative create function, scripting components call the standard Apple Event Manager function `AECreateAppleEvent` with default parameters.

custom Apple event   An Apple event you define for use by your own applications. Instead of creating custom Apple events, you should try to use the standard Apple events and extend their definitions as necessary for your application. If you think you need to define custom Apple events, you should check with the Apple Event Registrar to find out whether Apple events that already exist or are under development can be adapted to the needs of your application.

database extension   The interface between the Data Access Manager and a data server.

data server   An application that acts as an interface between a database extension on a Macintosh computer and a data source, which can be on the Macintosh computer or on a remote host computer. A data server can be a database server program that can provide an interface to a variety of different databases, or it can be the data source itself, such as a Macintosh application.

default container   The outermost container in an application's container hierarchy; usually the application itself. See also **container hierarchy.**

default scripting component   The scripting component used by the generic scripting component when an application passes `kOSANullScript` rather than a valid script ID to `OSACompile` or `OSAStartRecording`.

descriptor list   A descriptor record of data type `AEDescList` whose data handle refers to a list of descriptor records.

descriptor record   A data structure of type `AEDesc` that consists of a handle to data and a descriptor type that identifies the type of the data referred to by the handle. Descriptor records are the fundamental data structures from which Apple events are constructed.

descriptor type   An identifier for the type of data referred to by the handle in a descriptor record.

dialect   A version of a scripting language that resembles a specific human language or programming language; for example, the AppleScript scripting language provides dialects that resemble English, Japanese, and other languages. See also **AppleScript scripting language.**

direct parameter   The parameter in an Apple event that contains the data or object specifier record to be used by the server application. For example, a list of documents to be opened is specified in the direct parameter of the Open Documents event. See also **Apple event parameter.**

edition   The data written to an edition container by a publisher. A publisher writes data to an edition whenever a user saves a document that contains a publisher, and subscribers in other

documents may read the data from the edition whenever it is updated. See also **publisher, subscriber.**

edition container   A file that holds edition data, represented on the desktop by an edition icon. An edition container obtains its data from a publisher within a document. See also **edition, publisher.**

Edition Manager   The collection of routines that allows applications to automate copy and paste operations between applications, so that data can be shared dynamically.

element   An Apple event object contained by another Apple event object specified as the element's container. An Apple event object can contain many elements of the same element class, whereas an Apple event object can have only one of each of its properties. See also **Apple event object, container, element classes, property.**

element classes   In the *Apple Event Registry: Standard Suites,* a list of the object classes for the elements that an Apple event object of a given object class can contain. See also **Apple event object, object class.**

error callback function   An object callback function that gives the Apple Event Manager an address. The Apple Event Manager writes to this address the descriptor record it is currently working with if an error occurs during the resolution of an object specifier record. See also **object callback function.**

event class   An attribute that identifies a group of related Apple events. The event class appears in the `message` field of the Apple event's event record. The event class and the event ID identify the action an Apple event performs. See also **Apple event attribute, event ID.**

event ID   An attribute that identifies a particular Apple event within a group of related Apple events. The event ID appears in the `where` field of the Apple event's event record. The event ID and the event class identify the action an Apple event performs. See also **Apple event attribute, event class.**

Event Manager   The collection of routines that an application can use to receive information about actions performed by the user, to receive

notice of changes in the processing status of the application, and to communicate with other applications.

extension   An object class that duplicates all the characteristics of an object class of the same name and adds some of its own. Like a word in a dictionary, a single object class ID can have several related definitions.

factoring   Using Apple events to separate the code that controls an application's user interface from the code that responds to the user's manipulation of the interface. In a fully factored application, any significant user actions generate Apple events that a scripting component can record as statements in a compiled script. See also **recordable application.**

functional-area Apple event   A standard Apple event supported by applications with related features; for example, an Apple event related to text manipulation for word-processing applications, or an Apple event related to graphics manipulation for drawing applications. Functional-area Apple events are defined by Apple Computer, Inc., in consultation with interested developers and are published in the *Apple Event Registry: Standard Suites.*

generic script ID   Special script IDs used by the generic scripting component to keep track of script IDs provided by multiple scripting components. The generic scripting component translates generic scripting IDs into the corresponding component-specific script IDs and vice versa when necessary.

generic scripting component   A special scripting component that establishes connections dynamically with the appropriate scripting component for each script that a client application attempts to manipulate or execute. The generic scripting component also provides routines that you can use to determine which scripting component created a particular script, get an instance of a specific scripting component, and perform other useful tasks when you are using multiple scripting components. See also **scripting component.**

generic storage descriptor record   A descriptor record of type `kOSAGenericStorage` that can be used by the generic scripting component or

any other scripting component to store script data. The script data in a generic storage descriptor record is followed by a trailer that contains the subtype for the scripting component that created the script data.

implied length   The definition of a specific length for a data type. An example of this is the Data Access Manager's `typeInteger` data type, which has a defined length of 4 bytes.

insertion location descriptor record   A record of type `typeInsertionLoc` that consists of two keyword-specified descriptor records. The first is an object specifier record, and the data for the second is a constant that specifies the insertion location in relation to the Apple event object described by the object specifier record.

interapplication communication (IAC) architecture   A standard and extensible mechanism for communication among Macintosh applications, including the Edition Manager, the Open Scripting Architecture, the Apple Event Manager, the Event Manager, and the PPC Toolbox.

key data   The data in an object specifier record that distinguishes one or more Apple event objects from other Apple event objects of the same object class in the same container. Key data is specified by a keyword-specified descriptor record with the keyword `keyAEKeyData`. The Apple Event Manager interprets key data according to the key form specified in the same object specifier record.

key form   The form taken by the key data in an object specifier record. The key form is specified by a keyword-specified descriptor record with the keyword `keyAEKeyForm`. The keyword-specified descriptor record contains a constant that determines how the Apple Event Manager and a target application use the key data to locate specific Apple event objects. For example, the key form constant `formName` indicates that the key data consists of a name, which should be compared to the names of Apple event objects in the container specified by the object specifier record.

keyword   A four-character code that uniquely identifies a descriptor record inside another descriptor record. In Apple Event Manager functions, constants are typically used to represent the four-character codes.

keyword-specified descriptor record   A record of data type `AEKeyDesc` that consists of a keyword and a descriptor record. Keyword-specified descriptor records are used to describe the attributes and parameters of an Apple event.

location name   An identifier for the network location of the computer on which a port resides. The PPC Toolbox provides the location name. It contains an object string, a type string, and a zone. An application can specify an alias for its location name by modifying its type string. See also **port.**

logical descriptor record   A coerced AE record of type `typeLogicalDescriptor` that specifies a logical expression—that is, an expression that the Apple Event Manager evaluates to either `TRUE` or `FALSE`. The logical expression is constructed from a logical operator (one of the Boolean operators `AND`, `OR`, or `NOT`) and a list of logical terms to which the operator is applied. Each logical term in the list can be either another logical descriptor record or a comparison descriptor record.

mark-adjusting function   A marking callback function that unmarks objects previously marked by a call to an application's marking function.

mark count   The number of times the Apple Event Manager has called the marking function for the current mark token. Applications that support marking callback functions should associate the mark count with each Apple event object they mark.

marking callback functions   Object callback functions that allow your application to use its own marking scheme rather than tokens when identifying large groups of Apple event objects. See also **mark-adjusting function, mark token function, object callback function,** and **object-marking function.**

mark token   A token returned by a mark token function. A mark token identifies the way an application marks Apple event objects during the current sessions while resolving a single test. A mark token does not identify a specific Apple event object; rather, it allows an application that supports marking callback functions to associate a group of objects with a marked set.

mark token function   A marking callback function that returns a mark token.

message block   A byte stream that an open application uses to send data to and receive data from another open application (which can be located on the same computer or across a network). The PPC Toolbox delivers message blocks to an application in the same sequence in which they were sent.

Name-Binding Protocol (NBP)   An AppleTalk protocol that maintains a table containing the internet address and name of each entity in the node that is visible to other entities on the internet (that is, each entity that has registered a name with NBP).

null descriptor record   A descriptor record whose descriptor type is `typeNull` and whose data handle is `NIL`.

object accessor dispatch table   A table in either the application heap or the system heap that the Apple Event Manager uses to map descriptions of objects in an object specifier record to object accessor functions that can locate those objects.

object accessor function   An application-defined function that locates an Apple event object of a specified object class in a container identified by a token of a specified descriptor type.

object callback function   An application-defined function used by the Apple Event Manager to resolve object specifier records. See also **error callback function, marking callback functions, object-comparison function, object-counting function,** and **token disposal function.**

object class   A category for Apple event objects that share specific characteristics listed in an object class definition in the *Apple Event Registry: Standard Suites.* Among these characteristics are

properties, element classes, and Apple events that can specify objects of that class. An object class is specified in an object specifier record by a keyword-specified descriptor record with the keyword `keyAEDesiredClass` whose data handle refers to either a constant or an object class ID.

object class ID   A four-character code, which can also be represented by a constant, that identifies an object class for an Apple event object. The object class ID for a primitive object class is the same as the four-character value of its descriptor type.

object class inheritance hierarchy   The hierarchy of subclasses and superclasses that determines which properties, elements, and Apple events object classes inherit from other object classes.

object-comparison function   An object callback function that compares an element to either another element or to a descriptor record and returns either `TRUE` or `FALSE`.

object-counting function   An object callback function that counts the number of elements of a specified class in a specified container, so that the Apple Event Manager can determine how many elements it must examine to find the element or elements that pass a test.

object-marking function   An object callback function called repeatedly by the Apple Event Manager to mark specific Apple event objects. See also **marking callback functions.**

object specifier record   A description of one or more Apple event objects based on the Apple Event Manager and the classification system defined in the *Apple Event Registry: Standard Suites.* An object specifier record consists of a descriptor record of descriptor type `typeObjectSpecifier` that comprises four keyword-specified descriptor records: the object class ID, the container for the Apple event object (which is usually another Apple event object, specified by another object specifier record), the key form, and the key data.

Open Application event   An Apple event that asks an application to perform the tasks—such as displaying untitled windows—associated with opening itself; one of the four required Apple events.

Open Documents event   An Apple event that asks an application to open one or more documents specified in a list; one of the four required Apple events.

Open Scripting Architecture (OSA)   A mechanism based on the Apple Event Manager and the *Apple Event Registry: Standard Suites* that allows users to control multiple applications by means of scripts. The scripts can be written in any scripting language that supports the OSA.

optional parameter   A supplemental parameter in an Apple event used to specify data that the server application can use in addition to the data specified in the direct parameter. Source applications list the keywords for parameters that they consider optional in the attribute identified by the `keyOptionalKeywordAttr` keyword. Target applications use this attribute to identify any parameters that they are required to understand. If a parameter's keyword is not listed in this attribute, the target application must understand that parameter to handle the event successfully. See also **Apple event attribute, Apple event parameter.**

port   (1) A portal through which an open application can exchange information with another open application using the PPC Toolbox. A port is designated by a port name and a location name. An application can open as many ports as it requires so long as each port name is unique within a particular computer. (2) A connection between the CPU and main memory or a device (such as a terminal) for transferring data. (3) A socket on the back panel of a computer where you plug in a cable for connection to a network or a peripheral device.

port name   A unique identifier for a particular application within a computer. The port name contains a name string, a type string, and a script code. An application can specify any number of port names for a single port so long as each name is unique. See also **port.**

primitive object class   An object class defined in the *Apple Event Registry: Standard Suites* for Apple event objects that contain a single value; for example, the `cBoolean`, `cLongInteger`, and `cAlias` object classes are all primitive object classes. An Apple event object that belongs to a primitive object class has no properties and contains only one element—the value of the data.

Print Documents event   An Apple event that requests that an application print a list of documents; one of the four required Apple events.

Program-to-Program Communications (PPC) Toolbox   The collection of routines that allows applications to exchange blocks of data with other applications by reading and writing low-level message blocks.

property   An Apple event object that defines some characteristic of another Apple event object, such as its font or point size, that can be uniquely identified by a constant. The definition of each object class in the *Apple Event Registry: Standard Suites* lists the constants and class IDs for properties of Apple event objects belonging to that object class. For example, the constants `pName` and `pBounds` identify the name and boundary properties of Apple event objects that belong to the object class `cWindow`. The `pName` property of a specific window is defined by an Apple event object of object class `cProperty`, such as the word "MyWindow," which defines the name of the window. An Apple event object can contain only one of each of its properties, whereas it can contain many elements of the same element class. See also **Apple event object, container, element classes.**

property ID   A four-character code, which can also be represented by a constant, that identifies a property.

publish   To make data available to other documents and applications through a publisher. When a user creates or edits the data in the publisher and then saves it, the current version of the data is stored in an edition. See also **edition, publisher, subscriber.**

publisher   A portion of a document that makes its data available to other documents or applications. A publisher stores its data in an

edition whenever a user creates or edits the data in the publisher and then saves it. See also **edition, section, subscriber.**

query   A string of commands and data sent to a database or other data source. A query does not necessarily extract data from a data source; it might only send data or commands to a database or other application.

query definition function   A function contained in a **query document** that prompts the user for information and modifies the query before the Data Access Manager sends it to the data server.

query document   A file of file type `'qery'` containing commands and data in a format appropriate for a database or other data source. An application uses high-level Data Access Manager routines to open a query document.

query record   A data structure in memory containing information provided by a `'qrsc'` resource. The query record includes a pointer to a query.

Quit Application event   An Apple event that requests that an application perform the tasks—such as releasing memory, asking the user to save documents, and so on—associated with quitting; one of the four required Apple events. The Finder sends this event to an application immediately after sending it a Print Documents event or if the user chooses Restart or Shut Down from the Finder's Special menu.

range descriptor record   A coerced AE record of type `typeRangeDescriptor` that identifies two Apple event objects marking the beginning and end of a range of elements. See also **boundary objects.**

recordable application   An application that uses Apple events to report user actions to the Apple Event Manager for recording purposes. When a user turns on recording (for example, by pressing the Record button in the Script Editor application), a scripting component translates the Apple events generated by the user's subsequent actions into statements in a scripting language and records them in a compiled script. See also **scriptable application.**

recordable event   Any Apple event that any recordable application sends to itself while recording is turned on for the local computer, with the exception of events that are sent with the `kAEDontRecord` flag set in the `sendMode` parameter of the `AESend` function.

recording process   Any process (for example, a script editor) that can turn Apple event recording on and off and receive and record recordable Apple events.

required Apple event   One of the four Apple events in the Required suite that the Finder sends to applications: Open Documents, Open Application, Print Documents, or Quit Application.

required parameter   An Apple event parameter that must be included in an Apple event. For example, a list of documents to open is a required parameter for the Open Documents event. Direct parameters are often required, and other additional parameters may be required. Optional parameters are never required.

resolve   To locate the Apple event object described by an object specifier record.

result handler   A routine that the Data Access Manager calls to convert a data item to a character string.

results record   A structure that the Data Access Manager uses to store the data retrieved by the `DBGetQueryResults` function. This data is returned by a data source in response to a query.

resume dispatch function   An application-defined function called by `OSADoEvent` or `OSAExecuteEvent` to dispatch an Apple event directly to an application's default handler for that event.

script   Any collection of data that, when executed by the appropriate program, causes a corresponding action or series of actions. When a scripting component that supports the OSA executes a script, it sends Apple events as necessary to trigger actions in server applications.

scriptable application   An application that can respond as a server application to Apple events sent to it by scripting components. To be scriptable, an application must respond to the

appropriate standard Apple events, and it must provide an `'aete'` resource that describes the nature of that support. See also **Apple event user terminology resources.**

script application A script file with the file type `'APPL'` that contains the script data as a resource of type `'scpt'`. If a script application has the creator signature `'aplt'`, a user can double-click its icon to trigger the script. If a script application has the creator signature `'dplt'`, a user can drag the icon for another file or a folder over the script application's icon to trigger its script. By default, when a user triggers the script in a script application, a splash screen appears that allows the user either to quit or to run the script. Users can also save a script application in a form that bypasses the splash screen, running the script immediately after the user double-clicks its icon.

script application component A component registered with the Component Manager at system startup. When a user opens a script application, the script application component loads the script and passes the resulting script ID to the appropriate scripting component for execution.

script comment A description, in a script editor window, of what the script displayed in that window does.

script context A form of script that maintains context information for the execution of other scripts. A script context can also be used to handle Apple events. Like a compiled script, a script context can be decompiled as source data. In the AppleScript scripting language, a script context is called a `script object`.

script data A compiled script, script value, script context, or any other representation of a script in memory used internally by a scripting component. See also **compiled script, script context, script value.**

script editor An application that allows users to record, edit, save, and execute scripts; for example, the Script Editor application provided with AppleScript.

script file A file in which a script is stored. A script file can be a compiled script file, a script application file, or a script text file.

script ID A data structure of type `OSAID`—that is, a long integer—used by scripting components to keep track of script data.

scripting Writing and executing scripts to control the behavior of multiple applications.

scripting component A component that responds appropriately to calls made to the standard scripting component routines. Most scripting components implement scripting languages; for example, the AppleScript component implements the AppleScript scripting language.

script object AppleScript term for script context. See also **script context.**

script text file Uncompiled statements in a scripting language saved by a script editor as a text file. A user must open a script text file in a script editor and successfully compile it before it will execute. See also **script editor.**

script value An integer, a string, a Boolean value, a constant, a `'PICT'`, or any other fixed data that a scripting component returns or uses in the course of executing a script.

section A document or portion of a document that shares its contents with other documents. The Edition Manager supports two types of sections: publishers and subscribers. A publisher makes its data available to share and a subscriber subscribes to available data. See also **publisher, subscriber.**

send function A function called by a scripting component whenever it sends an Apple event during script execution. You can provide an alternative send function if you want your application to perform some action instead of or in addition to sending Apple events. If you don't provide an alternative send function, scripting components call the standard Apple Event Manager function `AESend` with default parameters.

server application An application that responds to Apple events requesting a service or information sent by client applications or

scripting components (for example, by printing a list of files, checking the spelling of a list of words, or performing a numeric calculation). Apple event servers and clients can reside on the same local computer or on remote computers connected to a network.

session   (1) A logical (as opposed to physical) connection between two entities (such as a Macintosh program and a database server) that facilitates the transmission of information between the two entities. (2) In the PPC Toolbox, an exchange of information between one open application with a port and another open application with a port. Sessions can occur between applications that are located on the same computer or across a network. An application has the option to accept or reject a session request. Authentication of the requesting user may be required before a session can commence. See also **authentication, message block, port.**

session ID   A number that uniquely identifies a **session.**

source application   The application that sends a particular Apple event to another application or to itself. Typically, an Apple event client sends an Apple event requesting a service from an Apple event server; in this case, the client is the source application for the Apple event. The Apple event server may return a different Apple event as a reply; in this case, the server is the source for the reply Apple event.

source data   Statements in a scripting language that constitute an uncompiled script.

special handler dispatch table   A table in either the application heap or the system heap that the Apple Event Manager uses to keep track of various specialized handlers.

status routine   An application-defined routine that can update windows, check the results of the low-level calls made by the Data Access Manager's `DBStartQuery` and `DBGetQueryResults` functions, and cancel execution of these functions when appropriate to do so.

subclass   An object class that inherits properties, element classes, and Apple events from another object class—its superclass. A subclass can also include properties, element classes, or Apple events that are not inherited from its superclass. Every object class, with the exception of `cObject`, is a subclass of another object class. See also **object class, superclass.**

subscribe   To obtain data that a publisher makes available in an edition. A user subscribes to a publisher by choosing Subscribe To from the Edit menu and selecting the desired edition. See also **edition, publish.**

subscriber   A portion of a document that automatically obtains current data from other documents and applications. A subscriber reads data from an edition. See also **edition, section.**

suite   In the *Apple Event Registry: Standard Suites,* a group of definitions for Apple events, object classes, primitive object classes, descriptor types, and constants that are all used for a set of related activities. For example, the Text suite includes definitions of Apple events, object classes, and so on that are used for text processing.

superclass   The object class from which a subclass inherits properties, elements, and Apple events. See also **object class, subclass.**

system Apple event dispatch table   See **Apple event dispatch table.**

system coercion dispatch table   See **coercion handler dispatch table.**

system object accessor dispatch table   See **object accessor dispatch table.**

system result handler   A result handler that is available to all applications that use the system. Compare with **application result handler.**

target address   An application signature, a process serial number, a session ID, a target ID record, or some other application-defined type that identifies the target of an Apple event.

target application   The application addressed to receive an Apple event. Typically, an Apple event client sends an Apple event requesting a service from a server application; in this case, the server is the target application of the Apple event. The

server application may return a different Apple event as a reply; in this case, the client is the target of the reply Apple event.

token A descriptor record returned by an object accessor function that identifies a requested Apple event object in a specified container.

token disposal function An object callback function that disposes of a token.

transaction A sequence of Apple events sent back and forth between the client and server applications, beginning with the client's initial request for a service. All Apple events that are part of one transaction must have the same transaction ID.

whose descriptor record A coerced AE record of descriptor type `typeWhoseDescriptor`. The Apple Event Manager creates whose descriptor records when it resolves object specifier records that specify `formTest`.

whose range descriptor record A coerced AE record of type `typeWhoseRange`. **Under certain conditions, the Apple Event Manager coerces a range descriptor record to a whose range descriptor record when it resolves** object specifier records that specify `formTest`.

Index

E

F