Apple Events Programming Guide

(Legacy)

Developer

Contents

Figures, Tables, and Listings

Introduction to Apple Events Programming Guide

Important: This document may not represent best practices for current development. Links to downloads and other resources may no longer be valid.

Apple Events Programming Guide provides conceptual information and programming examples for working with Apple events.

An **Apple event** is a type of interprocess message that can specify complex operations and data. Apple events allow you to gather all the data necessary to accomplish a high level task into a single package that can be passed across process boundaries, evaluated, and returned with results. The Mac OS uses Apple events to communicate with applications. Apple events are also an essential part of the AppleScript scripting system, which allows users to automate actions using **scriptable applications**—applications that can respond to a variety of Apple events by performing operations or supplying data.

> **Note:** Mac OS X offers other mechanisms for communicating between processes. These mechanisms are described in "IPC and Notification Mechanisms" in "Darwin and Core Technologies" in *Mac OS X Technology Overview*.

Apple Events Programming Guide assumes that you are familiar with the information in *AppleScript Overview*.

The information in this document applies primarily to Carbon applications. While Cocoa applications can take advantage of most of the described features, in many cases they won't need to. For more information, see "Framework and Language Support" (page 16).

Who Should Read This Document

You should read this document if you want to:

- Make your Carbon application respond to the Apple events sent by the Mac OS (for launching and quitting applications, opening documents, and so on).
- Work with Apple events as part of writing a scriptable Carbon application.
- Use Apple events to communicate with other applications.

- Gain background information about Apple events for your work with scriptable Cocoa applications, AppleScript Studio applications, Automator workflows, or AppleScript scripts.

Important: An Apple event can contain object specifiers that identify objects within the application that receives the event. Object specifiers are mentioned only briefly in this document. For information on how to work with object specifiers, see "Resolving and Creating Object Specifier Records" in Inside Macintosh: Interapplication Communication.

Do not rely on the API descriptions in Interapplication Communication—*Open Scripting Architecture Reference* and *Apple Event Manager Reference* provide the current API documentation.

Organization of This Document

This document is organized into the following chapters:

- "About Apple Events" (page 11) defines Apple events, explains when they're useful, and provides a quick overview of common tasks for working with them. It also provides a brief description of the framework and language support available in Mac OS X and provides links to additional information in *Apple Events Programming Guide* and in other documents.
- "Building an Apple Event" (page 18) provides an overview of Apple event data structures and describes how to build an Apple event.
- "Apple Event Dispatching" (page 29) shows how your application works with the Apple Event Manager to register the Apple events it can handle and dispatch those events to the code that should handle them.
- "Working With the Data in an Apple Event" (page 39) describes how to extract data from Apple events and the data structures that comprise them.
- "Responding to Apple Events" (page 48) describes how to respond to an Apple event by examining the event, performing the requested action, interacting with the user (if necessary), and returning a reply event. It also provides an overview of how to respond to Apple events sent by the Mac OS.
- "Creating and Sending Apple Events" (page 63) provides information and sample code that will help you create and send Apple events and respond to reply Apple events.
- "Writing and Installing Coercion Handlers" (page 79) describes how to write coercion handlers that convert between various types of data and how to install them so that they are available to your application.
- "Testing and Debugging Apple Event Code" (page 86) provides tips for displaying and debugging Apple events in your application.
- "Selected Apple Event Manager Functions" (page 92) provides information about some commonly used functions.
- "Selected Apple Event Constants" (page 95) provides information about some commonly used constants.

- "Default Coercion Handlers" (page 102) lists the type conversions performed by the default coercion handlers provided by the Mac OS.

See Also

The following documents provide related information.

- *AppleScript Overview* provides information that is useful for working with AppleScript and Apple events, including a description of the Open Scripting Architecture, on which both rely.
- *Apple Event Manager Reference* describes the API for sending and receiving Apple events and working with the information they contain.
- Technical Note TN2106, Scripting Interface Guidelines, describes how to design the scripting interface for a scriptable application.
- AppleScript Language Guide describes the features and terminology of the AppleScript scripting language.
- For information on sending Apple events to web services, see *XML-RPC and SOAP Programming Guide*.

About Apple Events

This chapter provides an overview of when and how applications use Apple events, with links to more detailed information on those topics. It also provides a brief description of the framework and language support available for working with Apple events in Mac OS X.

An **Apple event** is a type of interprocess message that can encapsulate commands and data of arbitrary complexity. Apple events provide a data transport and event dispatching mechanism that can be used within a single application, between applications on the same computer, and between applications on different computers connected to a network. The Mac OS uses Apple events to communicate with applications.

Apple events are part of the **Open Scripting Architecture (OSA)**, which provides a standard and extensible mechanism for interapplication communication in Mac OS X. The OSA is described in *AppleScript Overview*.

> **Note:** Apple events are not always the most efficient or appropriate method for communicating between processes. Mac OS X offers other mechanisms, including distributed objects, notifications, sockets, ports, streams, shared memory, and Mach messaging. These mechanisms are described in "Interprocess Communication" in System-Level Technologies in *Mac OS X Technology Overview*.

A Quick Look at Working With Apple Events

Apple events are designed to provide a flexible mechanism for interprocess communication. An Apple event specifies a target application (or other process) and provides a detailed description of an operation to perform. The operating system locates the target, delivers the event and, if necessary, delivers a reply Apple event back to the sender. While simple in concept, this mechanism provides a basis for powerful interaction between processes and for automating tasks that use multiple applications.

When Applications Use Apple Events

An application is most likely to work with Apple events for the following reasons:

- To respond to Apple events received from the Mac OS.

  For applications that present a graphical user interface, Mac OS X sends Apple events to initiate certain operations, such as launching or quitting the application.

For Cocoa applications, most of the work of responding to these events happens automatically. Carbon applications need to provide more of their own implementation—for details, see "Handling Apple Events Sent by the Mac OS" (page 58).

- To make its services or data available to other processes.

  Most applications that provide services through Apple events are scriptable applications—they provide a scripting terminology that lets users write AppleScript scripts to access the application's operations and data. When a script is executed, some of its statements result in Apple events being sent to the application. Other applications can also send Apple events directly to scriptable applications.

  Scriptable applications make it possible for users to automate their work. And starting in Mac OS X version 10.4, your scriptable application can also provide users with Automator actions. (Automator is an application that lets users work in a graphical interface to put together complex, automated workflows, made up of actions that perform discrete operations.)

  Scriptable Carbon applications can use the techniques for working with Apple events that are described throughout this document. For information on scriptable Cocoa applications, see "Framework and Language Support" (page 16).

  For information on designing and creating scriptable applications, and on creating Automator actions, see the learning paths in *Getting Started with AppleScript*.

- To communicate directly with other applications.

  An application can send an Apple event to ask another application to perform an operation or return data. For example, you might create a scriptable server application, running locally or remotely. Your other applications create Apple events and send them to the scriptable server to access its services. This is, in effect, another way to factor your code, with the shared functionality made available through Apple events.

- To support recording in a scriptable application.

  Recording refers to the assembling of Apple events that represent a user's actions into a script. Users can turn on recording in the Script Editor application, then perform actions with scriptable applications that support recording. Any Apple events generated by those actions are recorded into an AppleScript script. To support recording as fully as possible, you can take these steps:

  - Factor code that implements the user interface in your application from code that actually performs operations—this is a standard approach for applications that follow the model-view-controller design paradigm.
  - Send Apple events within the application to connect these two parts of your application. The Apple Event Manager provides a mechanism for doing this with a minimum of overhead, described in "Addressing an Apple Event for Direct Dispatching" (page 65).
  - Make sure that any significant action within your application generates an Apple event that can be recorded in a script.

Recording is beyond the scope of this document, but you can read more about it in the sections "Recordable Applications" and "Making Your Application Recordable" in Inside Macintosh: Interapplication Communication.

Apple Event Terminology

A scriptable application specifies the terminology that can be used in scripts that target the application. There are currently three formats for this information:

- **aete:** This is the original dictionary format and is still used in Carbon applications. The name comes from the Resource Manager resource type in which the information is stored (`'aete'`).
- **script suite:** This is the original format used by Cocoa applications. A script suite contains a pair of information property list (plist) files.
- **sdef:** "sdef" is short for "scripting definition." This XML-based format is a superset of the other two formats and supports additional features.

For more information on these formats, including pointers to additional documentation, see "Scriptable Applications" in Open Scripting Architecture in *AppleScript Overview*.

Sending and Receiving Apple Events

Applications typically use Apple events to request services and information from other applications or to provide services and information in response to such requests. In client-server terms, the **client application** sends an Apple event to request a service or information from the **server application**. The recipient of an Apple event is also known as the **target application** because it is the target of the event. A client application must know which kinds of Apple events the server supports.

Figure 1-1 Client and server applications communicating with Apple events

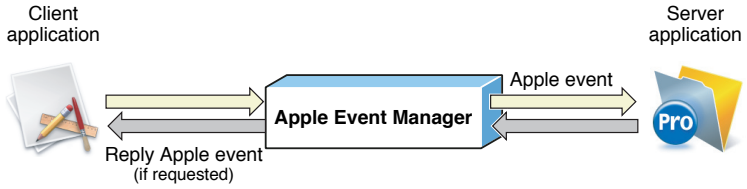


Figure 1-1 shows two applications communicating with Apple events. The client uses Apple Event Manager functions to create and send an Apple event to the server, the FileMaker Pro database application. The event might, for example, request employee information from a payroll database. FileMaker Pro uses other Apple

Event Manager functions to extract information from the event and identify the requested operation. Depending on the event, FileMaker Pro may need to add a record, delete a record, or return specified information in a reply Apple event.

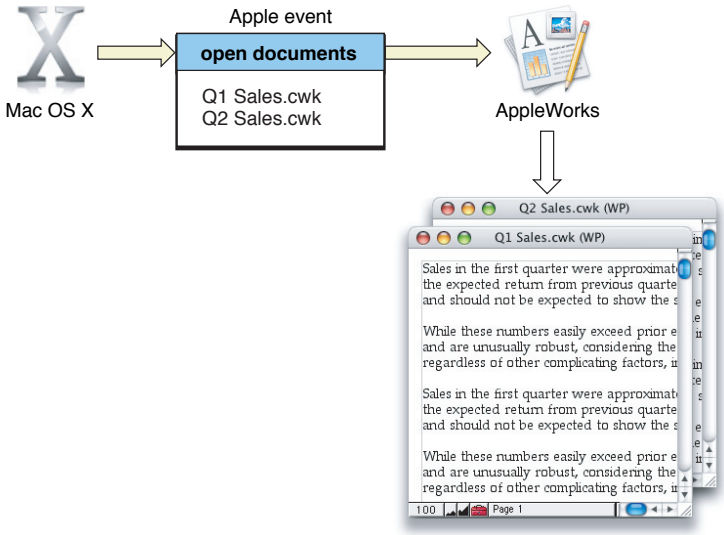
The most common Apple event client is a script editor application executing an AppleScript script. Statements in a script that target an application may result in Apple events being sent to the application. Another common client is the Mac OS, which sends Apple events to applications to open documents and perform other operations.

The most common servers are scriptable applications and scriptable parts of the Mac OS, such as the Finder and the System Events application (located in `/System/Library/CoreServices`). You can read more about script editors and scriptable applications in *AppleScript Overview*.

Responding to Apple Events

Your application should be prepared to respond to Apple events sent by the Mac OS, as well as to other Apple events the application supports. An Apple event typically contains information that specifies the target application, the action to perform, and optionally the objects on which to operate. For example, Figure 1-2 shows an `open documents` Apple event sent by Mac OS X to the AppleWorks application. This type of Apple event provides a list of files for the target application to open.

Figure 1-2 The Mac OS sending an open documents Apple event



For an application to handle a specific Apple event such as the `open documents` event, it must register with the Apple Event Manager a function that handles events of that type. The Apple Event Manager dispatches a received Apple event to the handler registered for it. An **Apple event handler** is an application-defined function that extracts pertinent data from an Apple event, performs the requested action, and if necessary, returns a result.

How to Use Apple Events

The steps your application takes in working with Apple events will differ, depending on whether it is responding to Apple events or creating and sending them.

Steps for Responding to Apple Events

To respond to Apple events in your application, you perform steps like the following:

- Determine which Apple events your application will support.
- Make sure your application can receive Apple events and dispatch them to a function that can handle them.
  - Write functions that handle the Apple events you support.
  - Register the functions with the Apple Event Manager so it can dispatch events to them.
- Call Apple Event Manager functions to extract information from received Apple events and locate specified items in your application.

  Note: Locating objects in your application is not covered in this document—for details, see "Resolving and Creating Object Specifier Records" in Inside Macintosh: Interapplication Communication.

- Perform the actions specified by received Apple events.
- If necessary, add information to a reply Apple event (sent to you as part of a received Apple event).

  If an error occurs, return an error code; you can also add error information to the reply event.

For guidelines and sample code for performing these steps, see "Apple Event Dispatching" (page 29) and "Responding to Apple Events" (page 48).

Steps for Creating and Sending Apple Events

To create and send Apple events in your application, you perform steps like the following:

- Create an Apple event.

  The Apple Event Manager provides functions for building an Apple event in one step and for creating an Apple event and adding information to it as a sequence of steps.

- Send the Apple event.

  The Apple Event Manager provides functions for sending an event with more options and more overhead, or with less options and less overhead.

  You will have to specify information such as

  - the target application to send the Apple event to
  - how to handle a timeout (in case the target doesn't respond)
  - whether to allow interaction with the user (for example, if the Apple event might result in showing a dialog)

For guidelines and sample code for performing these steps, see "Two Approaches to Creating an Apple Event" (page 25) and "Creating and Sending Apple Events" (page 63).

Framework and Language Support

You work with Apple events primarily through the API defined by the Apple Event Manager, which is documented in *Apple Event Manager Reference*. This API is implemented by the AE framework, a subframework of the Application Services framework, and is made available through headers written in the C programming language.

> **Note:** The Apple Event Manager is part of the Open Scripting Architecture. Some Apple-event related functions and constants are not defined in the AE framework—they are defined in other frameworks, as described in Open Scripting Architecture" in *AppleScript Overview*.

Carbon applications that work with Apple events, whether written in C or C++, typically call Apple Event Manager functions directly. Apple also provides C sample code, such as *MoreOSL*, to help in the implementation of scriptable Carbon applications.

Cocoa applications are written in Objective-C or Java. The Cocoa application framework provides built-in support for AppleScript scripting that allows many applications to be scriptable without working directly with Apple Event Manager functions. The Cocoa application framework also includes classes such as `NSAppleEventDescriptor`, for working with underlying Apple event data structures, and `NSAppleEventManager`, for accessing certain Apple Event Manager functions.

However, because Objective-C is a super set of the C language, Cocoa applications written in Objective-C can call Apple Event Manager functions directly and use any of the mechanisms described in this document. For example, a Cocoa application might use Apple Event Manager functions to perform operations that are not currently supported by the Cocoa framework, such as directly sending an Apple event. For details on writing a scriptable Cocoa application, see *Cocoa Scripting Guide*.

Building an Apple Event

This chapter provides an overview of Apple event data structures and describes how to build an Apple event.

An Apple event is capable of describing complex commands and the data necessary to carry them out. For example, an Apple event might request that a database application return data from records that meet certain criteria. An event sent by the Mac OS might request that the receiving application print a specified list of documents. The Apple Event Manager provides a relatively small number of Apple event data structures that together can be used to represent commands of great complexity.

Your application typically works with Apple events and the data they contain when:

- It receives an Apple event and must extract information to figure out what to do with the event.
- In response to a received Apple event, it must add information to a reply event to return to the sender.
- It creates an Apple event from scratch for internal communication or to request data or services from another application.

Working effectively with Apple events in these cases requires some knowledge of the data structures and organization of an Apple event, as well as familiarity with the Apple Event Manager functions you use to create Apple events and manipulate their data.

About Apple Event Data Structures

Understanding a few key concepts can help things go smoothly in creating an Apple event or working with its data:

- Each piece of information in an Apple event is associated with a four-character code (or in some cases, two such codes).
- A descriptor is a data structure that stores data and an accompanying four-character code. All the information you work with in an Apple event is stored in descriptors and lists of descriptors.
- The content of an Apple event is conceptually divided into two kinds of items, both constructed from descriptors:
  - Attributes identify characteristics of the task to be performed by the Apple event.
  - Parameters provide additional data to be used in performing the task.

- To create an Apple event, extract data from an event, or add data to an event, an application calls Apple Event Manager functions and passes the appropriate four-character codes and other information.
- To operate effectively with Apple events, you just need to find the right function for the task at hand.

Apple Event Building Blocks

This section describes the constants and data structures used to construct an Apple event.

Apple Event Constants

The Apple Event Manager uses four-character codes (also referred to as Apple event codes) to identify the data within an Apple event. A four-character code is just four bytes of data that can be expressed as a string of four characters in the Mac OS Roman encoding. For example, `'capp'` is the four-character code that specifies an application.

The Apple Event Manager defines four-character-code constants for many common commands (or verbs) and data objects (or nouns) that can be used in Apple events. These constants are defined primarily in the header files `AppleEvents.h` and `AERegistry.h` in the AE framework. They are documented in *Apple Event Manager Reference*. A subset of these constants is described in "Selected Apple Event Constants" (page 95) in this document.

Listing 2-1 (page 19) shows some constants from `AERegistry.h`. Each constant definition includes a comment showing the actual numeric value as a hex number.

Listing 2-1 Some four-character codes from AERegistry.h

```
enum {
  cApplication                  = 'capp', /*  0x63617070  */
  cArc                          = 'carc', /*  0x63617263  */
  cBoolean                      = 'bool', /*  0x626f6f6c  */
  cCell                         = 'ccel', /*  0x6363656c  */
  cChar                         = 'cha ', /*  0x63686120  */
  cDocument                     = 'docu', /*  0x646f6375  */
  cGraphicLine                  = 'glin', /*  0x676c696e  */
...
};
```


For the Apple event support in your application, you should use existing constants wherever they make sense, rather than defining new constants. For example, if your application supports an Apple event to get the name of a document, you can use the constant `cDocument` to denote a document.

Apple reserves all values that consist entirely of lowercase letters and spaces. You can generally avoid conflicts with Apple-defined constants by including at least one uppercase letter when defining a four-character code.

Event Class and Event ID

An Apple event is uniquely identified by its **event class** and **event ID** attributes, each of which is an arbitrary four-character code (described in "Apple Event Constants" (page 19)). The Apple Event Manager uses these values in dispatching Apple events to code in your application (described in "Apple Event Dispatching" (page 29)).

Apple defines event class and event ID values for standard Apple events, including those that it sends. For example, a `delete` Apple event has an event class value of `'core'` (represented by the constant `kAECoreSuite`) and an event ID value of `'delo'` (`kAEDelete`). For examples and descriptions of Apple-defined event class and event ID values, see "Event Class Constants" (page 95), "Event ID Constants for Apple Events Sent by the Mac OS" (page 96), and "Event ID Constants for Standard AppleScript Commands" (page 96).

You define the event class and event ID values for application-specific Apple events your application supports. While these values are arbitrary, you should follow the simple guidelines described in "Apple Event Constants" (page 19) in choosing values.

You can use a common event class for multiple Apple events as a way to group related events that your application supports. For example, many Apple-defined Apple events share a common event class. This can be useful in organizing the name space for your Apple events and may simplify your coding, but it doesn't result in any special treatment by the Apple Event Manager.

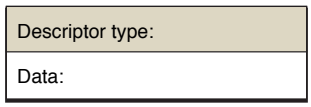
If you want other applications to be able to send Apple events to your application, you must publish event class and event ID values for those events. You should also describe the contents your application expects to find in each type of Apple event.

Similarly, if you want to send Apple events to other applications, you are dependent on those applications to provide the event class, event ID, and any other information you need to construct an Apple event the application can understand. One exception is that most applications, including yours, should be able to handle the Apple events described in "Common Apple Events Sent by the Mac OS" (page 58). So, for example, you might construct a `quit` Apple event that targets almost any Mac OS X application, and expect the application to handle it.

Descriptors, Descriptor Lists, and Apple Events

Descriptors and descriptor lists are the basic structural elements used in Apple events. A **descriptor** stores data and an accompanying descriptor type to form the basic building block of all Apple Events. The **descriptor type** is a four-character code that identifies the type of data associated with the descriptor. Table B-4 (page 97) lists constants for some of the main descriptor types—for a complete list, see *Apple Event Manager Reference*. Figure 2-1 shows the format of a descriptor.

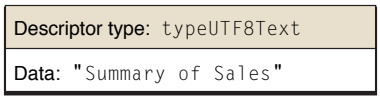
Figure 2-1 A descriptor



The data field of a descriptor is opaque—you should not attempt to access it directly. Table A-1 (page 93) lists functions provided by the Apple Event Manager for accessing the data in a descriptor (and related data types).

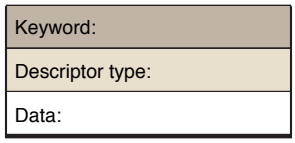
Figure 2-2 shows a descriptor with a descriptor type of `typeUTF8Text`, which specifies that the descriptor's data is text in UTF-8 encoding—in this case, the text is "Summary of Sales".

Figure 2-2 A descriptor whose data is a Unicode text string



A **keyword** is a four-character code used by the Apple Event Manager to identify a specific descriptor within an Apple event. A **keyword-specified descriptor** combines a keyword with a descriptor. This is the basic type used to specify attributes and parameters, which are described in detail in "Apple Event Attributes and Parameters" (page 23). Figure 2-3 shows the format of a keyword-specified descriptor.

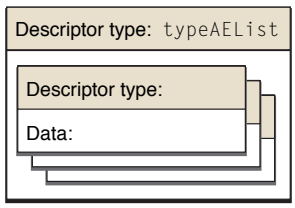
Figure 2-3 A keyword-specified descriptor



An **address descriptor** is a descriptor that specifies a target address for an Apple event—that is, it specifies the application or other process to send the event to. The descriptor type can be specified by one of the constants shown in Table B-5 (page 98).

A **descriptor list** is a descriptor whose data consists of a list of zero or more descriptors (it can be an empty list). A descriptor list can contain other lists, which allows for the construction of complex descriptors, and hence complex Apple events. Figure 2-4 shows the format of a descriptor list.

Figure 2-4 A descriptor list

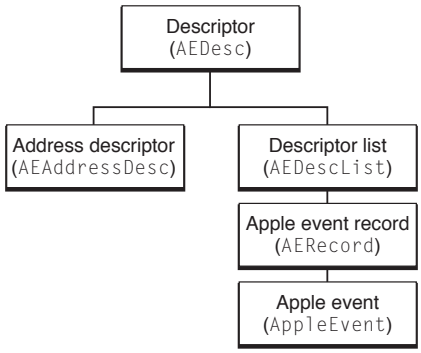


An **Apple event record** is a descriptor list whose data is a set of keyword-specified descriptors that describe Apple event parameters.

An **Apple event** is an Apple event record whose contents are conceptually divided into two parts, one for attributes and one for parameters, as shown in Figure 2-6 (page 23).

Figure 2-5 shows the inheritance for descriptors and related data structures, with the corresponding data types shown in parentheses.

Figure 2-5 Hierarchy of Apple event data structures

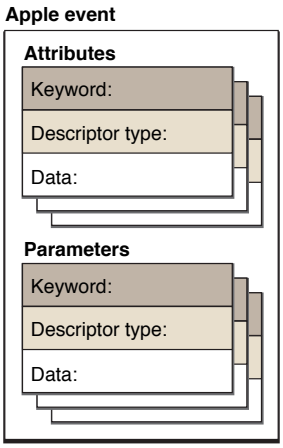


An Apple Event Manager function that operates on one of these data structures can also operate on any type that inherits from it. For example, Apple events inherit from Apple event records, which inherit from descriptor lists. As a result, you can pass an Apple event to any Apple Event Manager function that expects an Apple event record or a descriptor list. Similarly, you can pass Apple events and Apple event records, as well as descriptor lists and descriptors, to any Apple Event Manager function that expects a descriptor. See Table A-1 (page 93) for a list of functions for working with the various data types.

Apple Event Attributes and Parameters

Every Apple event consists of attributes and, often, parameters, as shown in Figure 2-6. Taken together, the attributes of an Apple event denote the task to be performed, while the parameters provide additional information to be used in performing the task.

Figure 2-6 An Apple event, with attributes and parameters



You use Apple Event Manager functions to create an Apple event, to add attributes or parameters to an Apple event, and to extract and examine the attributes or parameters from an Apple event.

Apple Event Attributes

An **Apple event attribute** is a keyword-specified descriptor that identifies a characteristic of an Apple event. For example, every Apple event must include attributes for event class, event ID, and target address:

- The event class and event ID attributes provide a pair of arbitrary four-character codes that together uniquely identify an Apple event. For more information, see "Event Class and Event ID" (page 20).
- The **target address attribute** specifies the process to send the Apple event to.

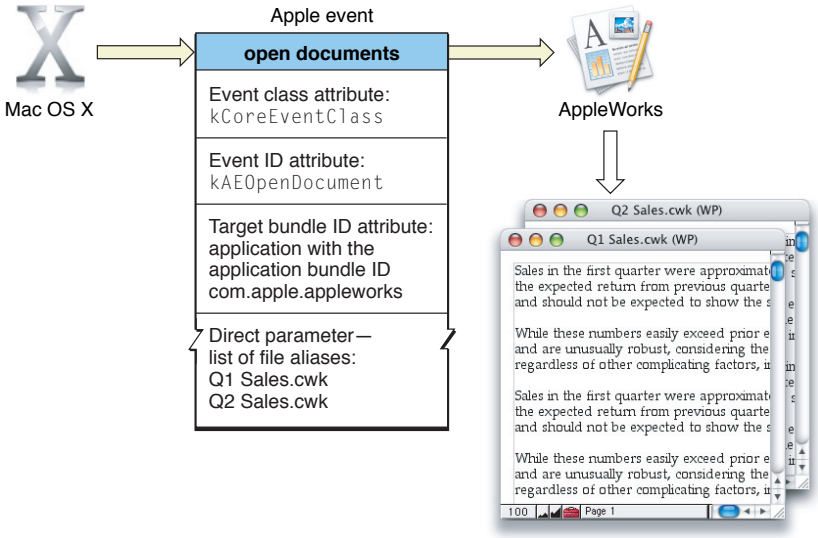
Apple events can include other kinds of attributes—see Table B-6 (page 99) for a list of keyword constants for Apple event attributes.

Apple Event Parameters

An **Apple event parameter** is a keyword-specified descriptor that contains additional data for the command. Keywords for common Apple event parameters are shown in Table B-7 (page 100).

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, Figure 2-7 shows the main Apple event attributes and the direct parameter for an `open documents` event that targets the AppleWorks application. The direct parameter specifies a descriptor list containing file aliases to the documents to open. An Apple event has at most one direct parameter.

Figure 2-7 Main attributes and direct parameter of an open documents event



Apple event parameters can contain standard data types, such as text strings, integers of various lengths, Boolean values, and others, listed in Table B-4 (page 97).

An Apple event can include other parameters, in addition to the direct parameter. For example, an Apple event that represents an arithmetic operation might contain a direct parameter that specifies a pair of values to operate on, as well as an additional parameter that specifies the operator. A reply Apple event may contain an error number parameter and an error string parameter, added by your application when an error occurs, as described in "Returning Error Information" (page 53).

Accessing Attributes and Parameters

Your application cannot examine the attributes and parameters of an assembled Apple event directly. Instead, it calls Apple Event Manager functions such as `AEGetAttributeDesc` and `AEGetParamDesc` to request an attribute or parameter by keyword. Because attributes and parameters are descriptors, you can also operate on them by index, using the functions `AEGetNthDesc` or `AEGetNthPtr`. However, that only makes sense if, for example, you are iterating over every descriptor—you should not assume that the parameters or attributes in an Apple event are in any particular order.

Apple event parameters often contain descriptions of Apple event objects within the target application. For example, a `get data` Apple event contains a parameter that describes the Apple event object that contains the requested data. Thus, an event might request, for example, the first paragraph of text from a named document. Apple event objects are described in "Resolving and Creating Object Specifier Records" in Inside Macintosh: Interapplication Communication.

For more information on accessing attributes and parameters in an Apple event, see "Working With the Data in an Apple Event" (page 39).

Optional Parameters

An Apple event may be defined so that it has optional parameters. An **optional parameter** is one that the sending application may or may not include. A target application must be prepared to handle the event whether or not the optional parameter is present. However, it can choose to ignore an optional parameter, even if present. This allows optional parameters to be added to an Apple event retroactively, without breaking existing code.

If an optional parameter is not present, or if your application chooses to ignore it, you should provide the default behavior for the Apple event. To determine if an optional parameter is present, you call a function such as `AEGetParamDesc`, specifying the keyword for the parameter. If the function returns successfully, you can extract information from the parameter and respond accordingly. If the function returns an error, you can assume the parameter is not present and provide the default behavior.

Two Approaches to Creating an Apple Event

How do you put together the data structures described in this chapter to create an Apple event? The Apple Event Manager provides functions that lend themselves to two main approaches, which you can combine as needed:

- You can create an Apple event with one call, passing all the information needed for a complete Apple event.

  Related functions allow you to create a complex descriptor, attribute, or parameter and add it to an existing Apple event in one step.

- You can create an Apple event sequentially, by first creating a potentially incomplete event, then adding information to it with subsequent calls.

  With this approach, you build descriptors into more complex data structures from the bottom up.

In either case, your application relies on the Apple Event Manager to construct Apple event data structures based on the arguments you pass.

> **Note:** There is a third way, not shown in this document, to create Apple event records and Apple event descriptors using stream-oriented calling conventions. The stream APIs are documented in *Apple Event Manager Reference*, while Technical Note 2046 AEStream and Friends provides conceptual overview and examples. You can use the stream functions independently, or combine them with the other mechanisms.

Creating an Apple Event in One Step

You can call the `AEBuildAppleEvent`function to create an Apple event in one step. To do so, you pass event class, event ID, and other information that is used to create the Apple event's attributes. The `AEBuildAppleEvent` function includes parameters for specifying the target address—you don't have to separately create an address descriptor. You may need to prepare data you'll pass to `AEBuildAppleEvent`—for example, you may need to create aliases to files you will insert into the Apple event as a descriptor list.

> **Note:** In this discussion, the word "parameter" is overloaded. When you call `AEBuildAppleEvent`, you pass information in *function parameters* to specify how to create *Apple event parameters* (and attributes), which are just Apple event data structures.

In addition, you provide a specially formatted string, similar to the string you might pass to a `printf` function, along with parameters that specify the data that corresponds to items in the format string. As a result, `AEBuildAppleEvent` can also create the parameters for your Apple event, resulting in a full-fledged Apple event.

The Apple Event Manager also provides the `AEBuildDesc` function as a one-step mechanism for adding potentially complex descriptors to an existing Apple event, and the `AEBuildParameters` function for adding parameters or attributes.

The one-step functions are most useful in situations where your application knows in advance all the information needed to create an Apple event or other data structure. They also make it easier to do parameterized substitution of values. For an example of this approach, see "Creating an Apple Event With AEBuildAppleEvent" (page 67).

Creating an Apple Event Sequentially

You can create the basic structure for an Apple event by calling the `AECreateAppleEvent` function. You pass information specifying event class, event ID, target address, return ID, and transaction ID; the function creates an Apple event containing the corresponding attributes. However, before you can call `AECreateAppleEvent`, you have to create a target address descriptor to pass to it, using a function such as `AECreateDesc`.

After calling `AECreateAppleEvent`, the resulting Apple event contains attributes for the event but no parameters. To add parameters to the event, you can use the `AEBuildDesc` and `AEBuildParameters` functions described in the previous section, or you can continue to work sequentially. For example, if the direct object of the event you are creating is a list of file aliases, you could create it sequentially by performing the following steps:

1. Call `AECreateDesc` once for each file alias to create a descriptor for it.
2. Call `AECreateList` to create a descriptor list.
3. Call `AEPutDesc` once for each descriptor to add it to the descriptor list.
4. Call `AEPutParamDesc` to add the descriptor list to the Apple event as a parameter.

This sequential approach is most useful for creating simple Apple events, or in situations where your application must factor the creation of an Apple event across several layers of code—for example, where you create an event, then pass it to various subsystems to add data to it.

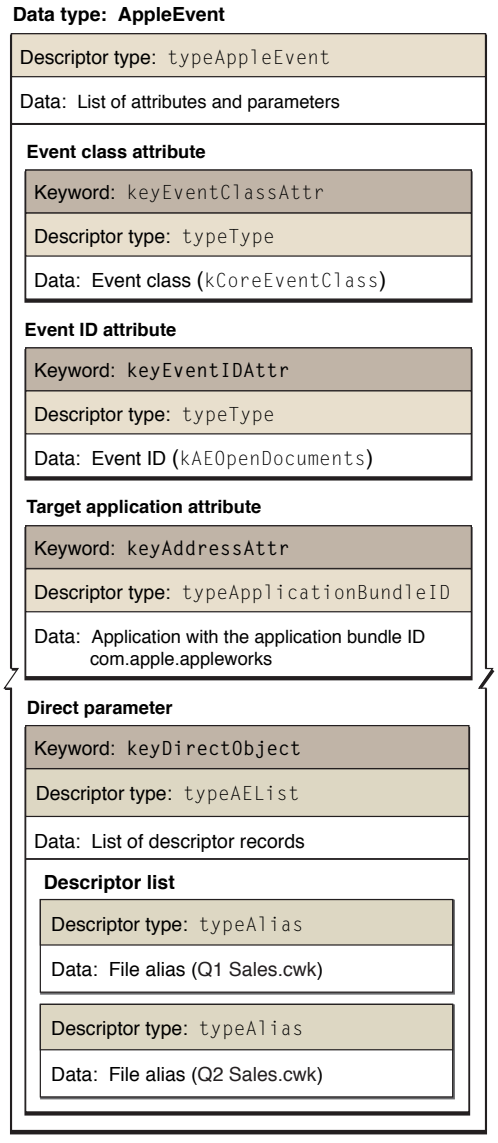
For an example of how to create an Apple event step by step, see "Creating an Apple Event With AECreateAppleEvent" (page 70).

The Completed Apple Event

Figure 2-8 shows the main data structures of a complete Apple event, containing a list of keyword-specified descriptors that specify the attributes and parameters of an `open documents` Apple event. Although this is an event sent to your application by the Mac OS, events you create have a similar structure.

The figure includes attributes for the event class, event ID, and target address. It also shows the direct parameter—a keyword-specified descriptor with the keyword `keyDirectObject`. The entire figure corresponds to the `open documents` event shown in Figure 2-7 (page 24).

Figure 2-8 Structure of an open documents Apple event



Apple Event Dispatching

This chapter shows how your application works with the Apple Event Manager to register the Apple events it supports and dispatch those events to the appropriate Apple event handlers. An **Apple event handler** is an application-defined function that extracts pertinent data from an Apple event, performs the requested action, returns a result code, and if necessary, returns information in a reply Apple event.

> **Note:** For information on how Apple events are dispatched in Cocoa applications, see How Cocoa Applications Handle Apple Events in *Cocoa Scripting Guide*.

How Apple Event Dispatching Works

Apple event dispatching works by a simple process:

- Your application registers Apple event handler functions with the Apple Event Manager for all the types of Apple events it can work with. A handler may handle just one type of Apple event or multiple types.
- The Apple Event Manager creates an Apple event dispatch table for your application. A **dispatch table** maps the Apple events your application supports to the Apple event handlers it provides.
- When your application receives an Apple event, it calls an Apple Event Manager function to process the event. If your dispatch table contains an entry for the event, the Apple Event Manager calls the associated handler.

> **Note:** For your application to respond to Apple events sent from remote computers, a user must open System Preferences, go to Sharing preferences, and select Remote Apple Events in the Services pane.

Dispatch Tables

You build an Apple event dispatch table by making calls to the function `AEInstallEventHandler` to register (or install) the events your application can handle and the functions that handle them. An Apple event is uniquely identified by the four-character codes specified by its event class and event ID attributes (described in "Event Class and Event ID" (page 20)). These codes are the values your application uses to register event

handler functions with the Apple Event Manager, and they are the values stored in the dispatch table. For example, a `delete` Apple event has an event class value of `'core'` (represented by the constant `kAECoreSuite`) and an event ID value of `'delo'` (`kAEDelete`).

When you install a handler function for an Apple event, you provide the following information:

- The event class and event ID of the Apple event; you can also specify wildcard values to register a handler for more than one event class or event ID.
- The address of the Apple event handler for the Apple event.
- An optional reference constant for passing additional information to the event handler.
- A Boolean value indicating whether the handler should be installed in the application dispatch table or the system dispatch table (you typically install handlers in the application dispatch table).

If you install a handler and there is already an entry in the dispatch table for the same event class and event ID, the entry is replaced. You can also specifically remove a handler with the function `AERemoveEventHandler` or get a handler (if it is present) with the function `AEGetEventHandler`.

An Apple event handler should return the value `errAEEventNotHandled` if it does not handle an event that is dispatched to it but wants to allow the event to be redispatched to another handler, if one is available.

You have several options in how you install event handlers:

- You can individually register, by event class and event ID, each of the Apple events that your application supports.
    - If you provide a separate event handler for each event, each handler will always knows exactly which type of event was dispatched to it.
    - If you provide the same handler for more than one event, a handler will need to do some work to determine which event it has received.

      For example, you might supply a different refcon value for each registered event type so that the event handler can examine its refcon parameter to determine how to respond.

      Or you might have your handler examine the event ID attribute of the passed Apple event (which requires an additional call to an Apple Event Manager function).
- You can use the `typeWildCard` constant for either the event class or the event ID (or for both), allowing multiple Apple events to be dispatched to a single handler, while minimizing the number of calls to `AEInstallEventHandler`. A handler can then examine the actual event class or event ID attribute of a received event to determine how to respond.

There are several reasons why you might choose to use a wildcard handler. For example, early in the development process, you may want to combine many events in one handler, then add (and register) more specific handlers at a later time. Or your application may support operations on a large number of objects that are nearly identical—rather than install many handlers that duplicate some of their code, you may prefer to install a wildcard handler.

If an Apple event dispatch table contains one entry for an event class and a specific event ID, and another that is identical except that it specifies a wildcard value for the event class or event ID, the Apple Event Manager dispatches to the more specific entry. For example, suppose the dispatch table includes one entry with event class `kAECoreSuite` and event ID `kAEDelete`, and another with event class `kAECoreSuite` and event ID `typeWildCard`. If the application receives a `delete` Apple event, it is dispatched to the handler function associated with the event ID `kAEDelete`.

The System Dispatch Table

When you call `AEInstallEventHandler`, you have the option of installing an Apple event handler in the application dispatch table or in the system dispatch table. When an event is dispatched, the Apple Event Manager checks first for a matching handler in the application dispatch table, then in the system dispatch table.

In Mac OS X, you should generally install all handlers in the application dispatch table. For Carbon applications running in Mac OS 8 or Mac OS 9, a handler in the system dispatch table could reside in the system heap, where it would be available to other applications. However, this won't work in Mac OS X.

Warning: If your application installs a handler in the system heap in Mac OS 8 or Mac OS 9, then quits or crashes without uninstalling the handler, the system is likely to crash the next time another application tries to call that handler.

Dispatching Apple Events in Your Application

To dispatch Apple events in your application, you perform the following steps:

1. Determine which Apple events your application will respond to and write event handler functions for those events.
2. Call the Apple Event Manager function `AEInstallEventHandler` to install your handlers.
3. Make sure your application processes Apple events correctly so that they are dispatched to the appropriate handler.

   The process for this step varies depending on how your application handles events in general.

These steps are described in the following sections.

Determining Which Apple Events to Respond To

All applications that present a graphical user interface through the Human Interface Toolbox should respond to certain events sent by the Mac OS, such as the `open application`, `open documents`, and `quit` events. These are described in "Handling Apple Events Sent by the Mac OS" (page 58). These events can also be sent by other applications.

Your application also responds to any Apple events it has specified for working with its commands and data. To handle Apple events that your application defines, it registers handlers for the event class and event ID values you have chosen for those events. As noted, you can register wildcard values to dispatch multiple events to one or more common handlers.

Installing Apple Event Handlers

Listing 3-1 shows how your application calls `AEInstallEventHandler` to install an Apple event handler function. The listing assumes that you have defined the function `InstallMacOSEventHandlers` to install handlers for Apple events that are sent by the Mac OS—that function is shown in Listing 5-6 (page 61).

Listing 3-1 also assumes you have defined the functions `HandleGraphicAE` and `HandleSpecialGraphicAE` to handle Apple events that operate on graphic objects used by your application. That function is not shown, but other event handlers are described in "Responding to Apple Events" (page 48).

Listing 3-1 Installing event handlers for various Apple events

```
static  OSErr   InstallAppleEventHandlers(void)
{
    OSErr   err;

    err = InstallMacOSEventHandlers();
    require_noerr(err, CantInstallAppleEventHandler);

    err = AEInstallEventHandler(kMyGraphicEventClass, kSpecialID,
            NewAEEventHandlerUPP(HandleSpecialGraphicAE), 0, false);            // 1
    require_noerr(err, CantInstallAppleEventHandler);

    err = AEInstallEventHandler(kMyGraphicEventClass, typeWildCard,
                NewAEEventHandlerUPP(HandleGraphicAE), 0, false);               // 2
```



```
    require_noerr(err, CantInstallAppleEventHandler);

CantInstallAppleEventHandler:
    return err;
}
```

In Listing 3-1, the application-defined function `InstallAppleEventHandlers` uses the macro `require_noerr` (defined in `AssertMacros.h`) to check the return value of each function. If an error occurs, it jumps to an error label. The function always returns an error value, which can be `noErr` if no error occurred.

The following descriptions apply to the numbered lines in Listing 3-1:

1. The application-defined function `HandleSpecialGraphicAE` handles Apple events that deal with one specific type of graphic object supported by the application, identified by the event class `kMyGraphicEventClass` and event ID `kSpecialID`.
2. The application-defined function `HandleGraphicAE` handles Apple events that deal with the application's other graphic objects. It is installed with an event ID of `typeWildCard`, an Apple Event Manager constant that matches any event ID. As a result, the `HandleGraphicAE` function will be called to handle any received Apple event with event class `kMyGraphicEventClass`, except those with the event ID `kSpecialID`.

Both of the numbered calls to `AEInstallEventHandler` provide the event class, event ID, and address of an Apple event handler. The `NewAEEventHandlerUPP` function creates a universal procedure pointer to a handler function. The value of `0` passed for the reference constant indicates the application does not need to pass additional information to the event handler function.

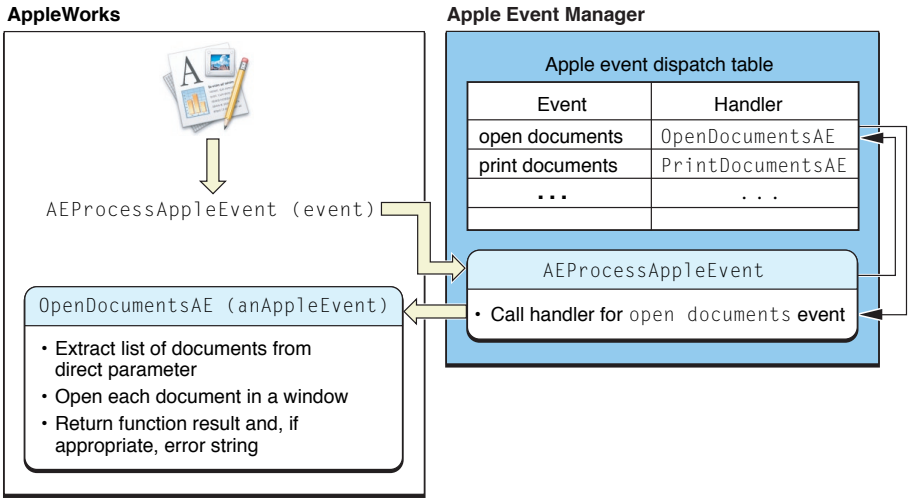
Each call to `AEInstallEventHandler` also passes a Boolean value of `false`, indicating the handler should be installed in the application dispatch table, not the system dispatch table.

Processing Apple Events

How your application processes Apple events depends on how it processes events in general—whether it uses the modern Carbon event model (described in *Carbon Event Manager Programming Guide*), or relies on the older `WaitNextEvent` function. In either case, your application receives events from the operating system, some of which represent Apple events. The application dispatches these events to its Apple event handlers by calling the Apple Event Manager function `AEProcessAppleEvent`.

Figure 3-1 provides a conceptual view of the dispatching mechanism. It shows the flow of control between the AppleWorks application and the Apple Event Manager when the application receives an `open documents` event and calls `AEProcessAppleEvent`.

Figure 3-1 An application working with the Apple Event Manager to dispatch an open documents Apple event



The `AEProcessAppleEvent` function takes an event of type `EventRecord` and looks up the Apple event it refers to in the application's dispatch table. If it finds a handler function for the Apple event, it calls the function, passing the Apple event.

The following sections describe in detail how an application calls the `AEProcessAppleEvent` function.

Processing Apple Events With the Carbon Event Model

An application that uses the Carbon event model receives each Apple event as a Carbon event of type `{kEventClassAppleEvent, kEventAppleEvent}`. For applications that call the `RunApplicationEventLoop` function to process Carbon events, the `AEProcessAppleEvent` function is called automatically, and dispatches received Apple events as shown in Figure 3-1.

Applications that use the Carbon event model but do not call `RunApplicationEventLoop` must install a Carbon event handler to process Apple events. Listing 3-2 shows how you might install such a handler—in this case, named `AEHandler`.

Listing 3-2 Installing a Carbon event handler to handle Apple events

```
const EventTypeSpec kEvents[] = {{kEventClassAppleEvent, kEventAppleEvent}};
```



```
InstallApplicationEventHandler(NewEventHandlerUPP(AEHandler),
    GetEventTypeCount(kEvents), kEvents, 0, NULL);
```

Your Carbon event handler for Apple events must perform these steps:

1. Remove the Carbon event of type `{kEventClassAppleEvent, kEventAppleEvent}` from the event queue before dispatching the Apple Event inside it.

   The process of removing the event from the queue triggers a synchronization with the Apple Event Manager that allows the next call to `AEProcessAppleEvent` to dispatch the Apple event properly. If the handler doesn't remove the Carbon event from the queue, the application will end up dispatching the wrong Apple event.
2. Call `ConvertEventRefToEventRecord` to get an `EventRecord` to pass to `AEProcessAppleEvent`.
3. Call `AEProcessAppleEvent`.
4. Return `noErr` to indicate the Carbon event has been handled (which does not depend on whether the dispatched Apple event is handled by the application).

Listing 3-3 shows a version of the `AEHandler` function.

Listing 3-3 A handler for a Carbon event that represents an Apple event

```
OSStatus AEHandler(EventHandlerCallRef inCaller, EventRef inEvent, void* inRefcon)
{
    Boolean     release = false;
    EventRecord eventRecord;
    OSErr       ignoreErrForThisSample;

    // Events of type kEventAppleEvent must be removed from the queue
    //  before being passed to AEProcessAppleEvent.
    if (IsEventInQueue(GetMainEventQueue(), inEvent))
    {
        // RemoveEventFromQueue will release the event, which will
        //  destroy it if we don't retain it first.
        RetainEvent(inEvent);
        release = true;
        RemoveEventFromQueue(GetMainEventQueue(), inEvent);
    }
```



```
    // Convert the event ref to the type AEProcessAppleEvent expects.
    ConvertEventRefToEventRecord(inEvent, &eventRecord);
    ignoreErrForThisSample = AEProcessAppleEvent(&eventRecord);

    if (release)
        ReleaseEvent(inEvent);

    // This Carbon event has been handled, even if no AppleEvent handlers
    //  were installed for the Apple event.
    return noErr;
}
```

If your application has not installed a handler for a received Apple event, the event may be handled by a handler installed by the system. For example, if your application calls `RunApplicationEventLoop`, a simple `quit application` handler is installed automatically. But if you call `AEProcessAppleEvent` for an event for which there is no handler installed, the event is ignored.

Processing Apple Events With WaitNextEvent

A Carbon application that uses the `WaitNextEvent` function, rather than the Carbon event model, receives an Apple event in its event loop as a standard event record (type `EventRecord`), identified by the constant `kHighLevelEvent`. The application passes these events to `AEProcessAppleEvent`, as shown in the following code listings.

Listing 3-4 shows a simplified main event loop function which continually loops, getting events and calling the `HandleEvent` function (in Listing 3-5) to process them.

Listing 3-4 A main event loop

```
static void MainEventLoop()
{
    RgnHandle       cursorRgn;
    Boolean         gotEvent;
    EventRecord     event;

    cursorRgn = NULL;
```



```
    while(!gQuit)
    {
        gotEvent = WaitNextEvent(everyEvent, &event, 32767L, cursorRgn);

        if (gotEvent)
        {
            HandleEvent(&event);
        }
    }
}
```

Listing 3-5 shows a standard approach for processing event records. When the value in the `what` field of the event record is `kHighLevelEvent`, the function calls `AEProcessAppleEvent`, which dispatches the event as shown in Figure 3-1.

Listing 3-5 A function that processes Apple events

```
static void HandleEvent(EventRecord *event)
{
    switch (event->what)
    {
        case mouseDown:
            HandleMouseDown(event);
            break;

        case keyDown:
        case autoKey:
            HandleKeyPress(event);
            break;

        case kHighLevelEvent:
            AEProcessAppleEvent(event);
            break;
    }
}
```


Listing 3-4 and Listing 3-5 are based on functions in the Xcode project `AppearanceSample`, located in `/Developer/Examples/Carbon`.

Working With the Data in an Apple Event

This chapter describes how your application gets various kinds of data from Apple events and the data structures that comprise them.

Your application responds to Apple events in the Apple event handlers it registers, or in routines your handlers call. Within a handler, you know that the passed Apple event matches the expected event class and event ID, although there can be some variation if the handler is registered with one or more wildcards. In either case, your handler has at least a general idea of what information the Apple event should contain. In the case of a wildcard handler, it can obtain information from the Apple event to identify the type more closely.

About Extracting Data From an Apple Event

The parameters and attributes of an Apple event, as well as the data within an individual descriptor, are stored in a format that is opaque to your application. You use Apple Event Manager functions to extract the data you need from a received Apple event. To obtain data in a format your application can use, you typically follow a common series of steps:

1. You call a function that returns the descriptor for a high-level data structure you are interested in, such as an Apple event attribute or parameter. For example, you can call the `AEGetAttributeDesc` function to obtain the descriptor for an attribute, specifying the attribute by its keyword.
2. If the returned descriptor is a list, you use another function to iterate over the items in the list. For example, you can use `AEGetNthDesc` to get descriptors from a list by index.
3. Once you have obtained an individual descriptor, you use other functions to extract its data. For example, you can call `AEGetDescDataSize` to determine the size of the data in a descriptor and `AEGetDescData` to get the data.

Many of the functions for getting data from an Apple event are available in two forms:

- One that returns a copy of the descriptor for the data.
- One that returns a copy of the data in a buffer you have supplied.

For example, `AEGetParamDesc` returns a copy of the descriptor for a parameter, while `AEGetParamPtr` returns the data from an Apple event parameter in a specified buffer. You typically use the buffer form to extract data of fixed length or known maximum length, such as a result code. You use the descriptor form to extract data of variable length, such as a list of unknown length.

You can also use Apple Event Manager functions to get data from descriptors, descriptor lists, and Apple events. For example, you can use the `AESizeOfAttribute` function to get the size and descriptor type of an Apple event attribute from an Apple event. And you can use the `AESizeOfParam` function to get the size and descriptor type of an Apple event parameter from an Apple event or an Apple event record (type `AERecord`).

Where performance is critical, you can use `AECreateDescFromExternalPtr` to efficiently create a descriptor containing large amounts of data, and call `AEGetDescDataRange` to efficiently get data from a descriptor.

Other functions allow you to count the number of items in a descriptor list (`AECountItems`) or iterate through those descriptors (`AEGetNthPtr`or `AEGetNthDesc`).

This chapter provides examples of how to work with some of these functions, while "Functions for Working With Apple Event Data Structures" (page 92) lists these and other Apple Event Manager functions. For complete reference, see *Apple Event Manager Reference* .

Coercing Data From an Apple Event

Coercion is the process of converting a descriptor and, if necessary, the data it contains, from one type to another. When your handler receives an Apple event, you typically use one or more of the functions `AEGetParamPtr`, `AEGetAttributePtr`, `AEGetParamDesc`, `AEGetAttributeDesc`, `AEGetNthPtr`, and `AEGetNthDesc` to get data from the Apple event. Each of these Apple Event Manager functions allows your application to specify a desired descriptor type for the resulting data. If the original data is of a different type, the Apple Event Manager attempts to coerce the data to the requested descriptor type. To prevent coercion and ensure that the descriptor type of the result is of the same type as the original, you specify `typeWildCard`for the desired type.

The following code snippet shows how to specify a desired descriptor type when calling the function `AEGetParamPtr`.

Listing 4-1 Getting and coercing an Apple event parameter

```
DescType        returnedType;
long            multResult;
Size            actualSize;
OSErr           err;
```

```
err = AEGetParamPtr(
        theAppleEvent,                                  // 1
        keyMultResult,                                  // 2
        typeSInt32,                                     // 3
        &returnedType,                                  // 4
        &multResult,                                    // 5
        sizeof(multResult),                             // 6
        &actualSize);                                   // 7
```

Here's a description of the parameters used in this call:

1. A pointer to the Apple event to get the parameter data from.
2. A keyword constant specifying the parameter to get the data from. In this example, the keyword is defined by the application and indicates a parameter containing the result of a multiplication operation.
3. A constant specifying the type to coerce the returned value to (if it isn't already that type).
4. The address of a variable in which the function stores the actual type of the returned value, which may not match the requested type.
5. The address of a variable in which the function stores the returned data.
6. The maximum size of the returned data. The `AEGetParamPtr` function won't return more data than you specify in this parameter.
7. The address of a variable in which the function stores the actual size of the requested data. If the returned value is greater than the amount your application allocated to store the returned data, you can increase the size of your buffer to this amount and call the function again. You can also choose to use the `AEGetParamDesc` function when your application doesn't know the size of the data.

If the coercion fails, the `AEGetParamPtr` function returns the result code `errAECoercionFail`.

By default, the Apple Event Manager can coerce between many different data types, listed in Table C-2 (page 103). To perform other coercions, such as those involving data types you have defined, you can provide your own coercion handlers. See "Writing and Installing Coercion Handlers" (page 79) for more information on working with coercion handlers.

Getting Data From an Apple Event Parameter

Apple event parameters are keyword-specified descriptors. You can use the `AEGetParamDesc` function to get the descriptor for a parameter or to extract the descriptor list from a parameter; you can use `AEGetParamPtr` to get the data from the descriptor for a parameter. In general, use the former to extract data of variable length, and the latter to extract data of fixed length or known maximum length.

Listing 4-2 shows how to call `AEGetParamDesc` to extract a parameter descriptor from an Apple event such as an `open documents` event.

Listing 4-2 Getting a parameter as a descriptor

```
AEDescList  docList;
OSErr       myErr;

myErr = AEGetParamDesc( theAppleEvent,                  // 1
                        keyDirectObject,                // 2
                        typeAEList,                     // 3
                        &docList);                      // 4

// Check the returned value from AEGetParamDesc for any error.
// (Not shown.)
```

Here's a description of the parameters used in this call:

1. A pointer to the Apple event to get the parameter descriptor from (obtained previously).
2. A constant specifying that the function should get the descriptor for the direct parameter.
3. A constant specifying that the descriptor should be returned as a descriptor list, coercing it if necessary. If the coercion fails, the function returns the result code `errAECoercionFail`.
4. The address of a variable in which the function stores the returned descriptor list.

See "Getting Data From a Descriptor List" (page 45) for details on working with a descriptor list.

The `AEGetParamDesc` function supplies a copy of the descriptor for the parameter, so you must dispose of it when you're finished with it by calling the `AEDisposeDesc` function. For an example, see Listing 4-6 (page 47).

If an Apple event parameter contains an object specifier, your handler should use the `AEResolve` function, other Apple Event Manager functions, and your own application-defined functions to resolve the object specifier—that is, to locate the Apple event object in your application that the specifier describes. For more information, see "Resolving and Creating Object Specifier Records" in Inside Macintosh: Interapplication Communication.

Getting Data From an Apple Event Attribute

To get the descriptor for an attribute or to get the data from an attribute you use routines that are similar to those you use with parameters: the `AEGetAttributePtr` and `AEGetAttributeDesc` functions.

For example, Listing 4-3 shows how to use `AEGetAttributePtr` to get the data from the `keyEventSourceAttr` attribute of an Apple event.

Listing 4-3 Getting a value from an attribute

```
    DescType        returnedType;
    AEEventSource   sourceOfAE;
    Size            actualSize;
    OSErr           myErr;

myErr = AEGetAttributePtr( theAppleEvent,                          // 1
                           keyEventSourceAttr,                     // 2
                           typeShortInteger,                       // 3
                           &returnedType,                          // 4
                           (void *) &sourceOfAE,                   // 5
                           sizeof (sourceOfAE),                    // 6
                           &actualSize);                           // 7

    // Check the returned value from AEGetParamDesc for any error.
    // (Not shown.)
```

Here's a description of the parameters used in this call:

1. A pointer to the Apple event to get the attribute data from (obtained previously).
2. A constant specifying the attribute from which to get the data.
3. A constant specifying that the data should be returned as a short integer.

4. The address of a variable in which the function stores the type of the actual descriptor returned.
5. The address of a variable in which the function stores the requested data. If the data is not already a short integer, the Apple Event Manager coerces it as necessary. The value should equal one of the event source constant values described in "Event Source Constants" (page 100).

   If you allocate a buffer for this parameter, it's up to you to free it when you are finished with it.
6. The size of the return buffer.
7. The address of a variable in which the function stores 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.

Getting Data From a Descriptor

Because the data within a descriptor is opaque, you use Apple Event Manager functions to extract it. In some cases, such as the example shown in Listing 4-3, the data is of known type and size, or can be coerced to a known type, and you can store it in a variable of that type.

It is common, however, for a descriptor to contain data, such as text or an image, of unknown size. In situations of that type, you can call the `AEGetDescDataSize` function to find out how much memory you will need to store the data, allocate a buffer of that size, then call `AEGetDescData` to get the actual data from the descriptor. The code snippet in Listing 4-4 shows how you might use these functions to get data into a buffer.

Listing 4-4 Determining the size, then obtaining descriptor data

```
Size dataSize = AEGetDescDataSize(desc);                    // 1
UInt8* buffer = malloc(dataSize);                           // 2
if (buffer)
{
    OSErr err = AEGetDescData(desc, buffer, dataSize);      // 3

    // If no error, use the data.                           // 4

    free(buffer);                                           // 5
}
```

Here's a description of what this code does:

1. Calls an Apple Event Manager function to get the data size for the descriptor.

2. Creates a buffer of that size.
3. Calls an Apple Event Manager function to get the data from the descriptor into the buffer.
4. If no error occurred in getting the data, your application can use it as needed.
5. Frees the allocated buffer.

Alternatively, you can do the following:

1. Call the `AEGetParamPtr` function (shown in Listing 4-1 (page 40)), passing a size of zero for the maximum size (line 6 in the listing).
2. Get the actual size value returned by `AEGetParamPtr`.
3. Allocate a buffer of that size.
4. Call `AEGetParamPtr` again to get the data into the buffer.
5. If you allocated memory for the buffer, free it when you are finished with it.

Getting Data From a Descriptor List

To get descriptors and their data from a descriptor list, you can call the `AECountItems` function to get the number of descriptors in the list, then set up a loop that calls `AEGetNthDesc` or `AEGetNthPtr` to get the data from each descriptor.

For example, an `open documents` event contains a direct parameter that specifies a list of documents to open. The parameter contains a descriptor list in which each descriptor specifies an alias to a file to open. Listing 4-5 shows how you can extract the descriptor list from the parameter, determine the number of items it contains, and extract each descriptor from the list.

Listing 4-5 Getting a descriptor list and its items

```
AEDescList  docList;
FSRef       theFSRef;
long        index;
long        count = 0;
OSErr       err;

err = AEGetParamDesc(theAppleEvent, keyDirectObject,
                     typeAEList, &docList);                    // 1
```



```
err = AECountItems(&docList, &count);                                    // 2

for(index = 1; index <= count; index++)                                  // 3
{
    err = AEGetNthPtr(&docList, index, typeFSRef,
                      NULL, NULL, &theFSRef,
                      sizeof(theFSRef), NULL);                           // 4

    // Call routine to open document with current reference.             // 5
}
```

Here's a description of what this code does:

1. Calls `AEGetParamDesc` to obtain, in the variable `docList`, a copy of the descriptor list from the direct parameter of the Apple event. Passes the constant `keyDirectObject` to identify the direct parameter and the constant `typeAEList` to indicate the desired descriptor type. (`theAppleEvent` is a pointer to the Apple event, obtained previously, to get the parameter descriptor from).
2. Calls `AECountItems`, passing the previously obtained descriptor list, to get the number of items in the list.
3. Sets up a loop based on the count.
4. Calls `AEGetNthPtr` to obtain the indexed descriptor from the list. Passes `typeFSRef` to indicate the descriptor should be coerced to a file system reference, if necessary (otherwise the returned value will be a file alias). Passes the address of the variable `theFSRef` as a buffer to store the reference.

   Passes `NULL` for the fourth and fifth parameters, indicating the keyword and descriptor type of the returned item aren't needed.

   Also passes `NULL` for the last parameter, indicating the actual size of the returned data isn't required. (If the returned 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.)
5. Here you would call your own routine to open a document specified by the extracted file system reference.

For a more complete version of this code, including simple error handling, see Listing 5-5 (page 56).

Disposing of Apple Event Data Structures

If you create a descriptor, you must dispose of it when you are finished with it to prevent memory leaks. For example, when you extract a descriptor using the `AEGetParamDesc`, `AEGetAttributeDesc`, `AEGetNthDesc`, or `AEGetKeyDesc` function, you get a copy of the descriptor. You call the `AEDisposeDesc` function to dispose of your copy, thereby deallocating the memory used by its data.

Listing 4-6 shows how to dispose of a descriptor list returned by `AEGetParamDesc`.

Listing 4-6 Disposing of a descriptor list obtained from the direct object of an Apple event

```
AEDescList  docList;
OSErr       err;
err = AEGetParamDesc(theAppleEvent, keyDirectObject, typeAEList, &docList);

// Check for error, then perform operations on the descriptor list here.

AEDisposeDesc(&docList);
```

You can safely call `AEDisposeDesc` on a null descriptor (but not on a null pointer!) A **null descriptor** is one that has been initialized as shown in the following code snippet.

```
AEDesc      someDesc = { typeNull, 0L };

// Code to obtain a descriptor, which may fail.

// Safe to dispose, whether or not previous code succeeded.
AEDisposeDesc(&someDesc);
```

You can perform the same initialization using the `AEInitializeDesc` function, as shown in this code snippet.

```
    AEDesc      someDesc;

    AEInitializeDesc(&someDesc);
```

When you obtain a copy of a descriptor with one of the buffer-based functions, such as `AEGetAttributePtr` or `AEGetNthPtr`, the data is copied into a buffer provided by your application. You must then free any allocated memory when finished with the buffer.

Responding to Apple Events

Your application must be able to respond to certain Apple events sent by the Mac OS, such as the `open application` and `quit` events. If your application has defined additional Apple events that it supports, either to supply services to other applications or to make itself scriptable, it must be ready to respond to those events as well.

This chapter describes how you write handlers to respond to the Apple events your application receives. It also provides an overview of how Carbon applications work with Apple events sent by the Mac OS.

About Apple Event Handlers

To respond to Apple events, your application registers handler functions with the Apple Event Manager for the events it can handle, as described in "Apple Event Dispatching" (page 29). Your application will only receive Apple events that target it—that is, events that specify your application in their target address descriptor. These include the events described in "Handling Apple Events Sent by the Mac OS" (page 58). Should your application receive an Apple event it is not expecting, the Apple Event Manager will not find it in your dispatch table, so the event will not be dispatched to one of your handlers, unless you have installed a wildcard handler.

> **Note:** One exception is the case where an event handler is installed automatically for your application. For example, "Processing Apple Events With the Carbon Event Model" (page 34) describes a case in which a system `quit` event handler is installed automatically.

As a result, your event handlers should generally be called only for events they understand. If a problem occurs, it is most commonly due to an error in constructing the event that was sent to your application. Because of this possibility, you may want to test the error handling in your Apple event code by intentionally sending your application Apple events containing unexpected information.

Definition of an Apple Event Handler

When you declare an Apple event handler, the syntax must match the `AEEventHandlerProcPtr` data type, which is described in detail in *Apple Event Manager Reference*. Listing 5-1 shows the declaration for a function that handles the `open documents` Apple event—all your handler declarations use a similar declaration.

Listing 5-1 Definition for an Apple event handler

```
static pascal OSErr HandleOpenDocAE (                                    // 1
    const AppleEvent * theAppleEvent,                                    // 2
    AppleEvent * reply,                                                  // 3
    SInt32 handlerRefcon);                                               // 4
```

Here's a description of this function declaration:

1. The `pascal` keyword ensures proper ordering of parameters on the stack.
2. The Apple event for the function to handle. You'll often extract information from it to help you process the Apple event.
3. The default reply (provided by the Apple Event Manager). It contains no parameters. If no reply is requested, the reply event is a null descriptor.
4. The 32-bit reference constant you supplied for the handler when you first registered it by calling `AEInstallEventHandler`. You can use the reference constant to provide information about the Apple event, or for any other purpose you want.

   For example, you can install, for multiple Apple events, dispatch table entries that specify the same handler but different reference constants. Your handler can then use the reference constant to distinguish between the different Apple events it receives.

Interacting With the User

When your application handles an Apple event, it may need to interact with the user. For example, your handler for the `print documents` event may need to display a print dialog before printing. Your application should not just assume it is okay to interact with the user in response to an Apple event. It should always call the `AEInteractWithUser` function to make sure it is in the foreground before it interacts with the user.

If `AEInteractWithUser` returns the value `noErr`, then your application is currently in front and is free to interact with the user. If `AEInteractWithUser` returns the value `errAENoUserInteraction`, your application should not attempt to interact with the user.

You can specify flags to the `AESetInteractionAllowed` function to allow the following types of interaction with your application:

- Only when your application is sending an Apple event to itself.
- Only when the client application is on the same computer as your application.
- For any event sent by any client application on any computer.

Both client and server applications specify their preferences for user interaction: the client specifies whether the server should be allowed to interact with the user, and the server (your application) specifies when it allows user interaction while processing an Apple event. For interaction to take place, these two conditions must be met:

- The client application must set flags in the `sendMode` parameter of the `AESend` function indicating that user interaction is allowed.
- The server application must either set no user interaction preferences, in which case it is assumed that only interaction with a client on the local computer is allowed; or it must use the `AESetInteractionAllowed` function to indicate 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 a dialog 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.

> **Important:** Calling `AEInteractWithUser` ensures that your application will get correct notification in terms of being activated. There is no guarantee that other applications are setting an interaction level, but if they are, your application can respect it.
>
> Interaction is not currently guaranteed to work correctly in the context of fast user switching. For example, your application could ask the user whether it should present UI, but the user might never respond because your application has been switched out.

To track when your Carbon or Cocoa application has been switched in or out, you can sign up for notification of the `kEventSystemUserSessionActivated` and `kEventSystemUserSessionDeactivated` events, as described in User Switch Notifications in *Multiple User Environments*.

To determine whether your application is currently running in the active user session, you can use the `CGSessionCopyCurrentDictionary` function, together with the `kCGSessionOnConsoleKey`, as described in Supporting Fast User Switching in *Multiple User Environments*.

For more information on working with user interaction, see "Specifying Send and Reply Options" (page 73) in this document, as well as the following in *Apple Event Manager Reference*:

- The section "User Interaction Level Constants."
- The constants listed under "AESendMode."
- The functions `AEInteractWithUser`, `AESetInteractionAllowed`, `AEGetInteractionAllowed`, and `AESend`.

Writing an Apple Event Handler

Your Apple event handler typically performs the following operations:

- It extracts information from the received Apple event.
- It performs any requested actions, checking that interaction is allowed before attempting to interact with a user.
- It adds information to the reply Apple event if appropriate.
- It returns a result code. If an error occurs in extracting information from the event or if an error occurs in any other operation:
  - It may add error information to the reply Apple event.
  - It returns an appropriate error code.

Extracting Information From the Apple Event

For nearly all standard Apple events, your handler extracts data from the event to help determine how to process the event. The following are some kinds of information you are likely to extract:

- The event class and/or the event ID. For example, if you registered your handler with one or more wildcards, you may need to get the event class or event ID to determine which kind of event you actually received. You can obtain this information by calling a function such as `AEGetAttributePtr`, and passing `keyEventClassAttr` or `keyEventIDAttr` for the parameter that specifies the keyword of the attribute to access.
- The direct parameter, which usually specifies the data to be acted upon. Listing 4-5 (page 45) shows how to get a parameter that contains a list of files from an `open documents` Apple event.
- Additional parameters. These can be any kind of data stored in an Apple event. They might include object specifiers, which identify objects within your application on which the Apple event operates. For information on how to work with object specifiers, see "Resolving and Creating Object Specifier Records" in Inside Macintosh: Interapplication Communication.

"Working With the Data in an Apple Event" (page 39) provides additional information and examples of how you extract data from an Apple event.

Performing the Requested Action

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 `print documents` event by printing the specified documents.

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.

Adding Information to a Reply Apple Event

If the application that sent the Apple event requested a reply, the Apple Event Manager passes a default reply Apple event to your handler. The reply event has event class `kCoreEventClass` and event ID `kAEAnswer`. If the client application does not request a reply, the Apple Event Manager passes a null descriptor instead—that is, a descriptor of type `typeNull` whose data handle has the value `NULL`.

The default reply Apple event has no parameters, but your handler can add parameters or attributes 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 to it. Attempting to add to a null descriptor generates an error.

Your handler may need to add error information to the reply event, as described in subsequent sections.

Returning a Result Code

When your handler finishes processing an Apple event, it should dispose of any Apple event descriptors or other data it has acquired. It should then return a result code to the Apple Event Manager. Your handler should return `noErr` if it successfully handles the Apple event or a nonzero result code if an error occurs.

If an error occurs because your application cannot understand the event, it should return `errAEEventNotHandled`. This allows the Apple Event Manager to look for a handler in the system dispatch table 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, but it cannot handle such an event. In this situation, you can return `errAEEventNotHandled` in case another available handler can handle the `get data` event. However, this strategy is only useful if your application has installed such a system handler, or the event is one of the few for which a system handler may be installed automatically. (For information on handlers that are installed automatically, see "Handling Apple Events Sent by the Mac OS" (page 58).)

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.

In addition to returning a result code, your handler can also return an error string by adding a `keyErrorString` parameter to the reply Apple event. The client can use this string in an error message to the user. For more information, see "Returning Error Information" (page 53).

For scriptable applications, the client is often the Script Editor, or some other application, executing a script. When you return an error code that the Apple Event Manager understands, the Script Editor automatically displays an appropriate error message to the user. You can find tables of error codes documented in *Apple Event Manager Reference* and *Open Scripting Architecture Reference*.

Returning Error Information

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. This can be useful because it associates the error number directly with the return event, which the client application may pass around without the result code.

Rather than just returning an error code, your handler itself can add error information to the reply Apple event. It can add both an error number and an error string. As noted previously, in many cases returning an Apple Event Manager error code will result in an appropriate error message. You should only add your own error text in cases where you can provide information the Apple Event Manager cannot, such as information specific to the operation of your application.

> **Note:** Handlers can return several additional types of error information. For details, see the `OSAScriptError` function and the constant section "OSAScriptError Selectors" in *Open Scripting Architecture Reference*.

To directly add an error number parameter to a reply Apple event, your handler can call a function like the one in Listing 5-2.

Listing 5-2 A function to add an error number to a reply Apple event

```
static void AddErrNumberToEvent(OSStatus err, AppleEvent* reply)                // 1
{
    OSStatus returnVal = errAEEventFailed;                                      // 2

    if (reply->descriptorType != typeNull)                                      // 3
    {
        returnVal = AESizeOfParam(reply, keyErrorNumber, NULL, NULL);
        if (returnVal != noErr))                                                // 4
```



```
        {
            AEPutParamPtr(reply, keyErrorNumber,
                        typeSInt32, &err, sizeof(err));                    // 5
        }
    }
    return returnVal;
}
```

Here's a description of how this function works:

1. It receives an error number and a pointer to the reply Apple event to modify.
2. It declares a variable for the return value and sets it to the Apple Event Manager error constant for "Apple event handler failed." If the function is successful, this value is changed.
3. It checks the descriptor type of the reply Apple event to make sure it is not a null event.
4. It verifies that the event doesn't already contain an error number parameter (`AESizeOfParam` returns an error if it doesn't find the parameter).
5. It calls `AEPutParamPtr` to add an error number parameter to the reply event:
    - `keyErrorNumber` specifies the keyword for the added parameter.
    - `typeSInt32` specifies the descriptor type for the added parameter.
    - `err` specifies the error number to store in the added parameter.

In addition to returning a result code, your handler can 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 5-3 shows a function that adds a parameter, from a string reference that refers to Unicode text, to a passed descriptor. You pass the desired keyword to identify the parameter to add. The `AEPutParamString` function calls `AEPutParamPtr`, which adds a parameter to an Apple event record or an Apple event, so those are the types of descriptors you should pass to it. ("Descriptors, Descriptor Lists, and Apple Events" (page 21) describes the inheritance relationship between common Apple event data types.)

Listing 5-3 A function that adds a string parameter to a descriptor

```
OSErr AEPutParamString(AppleEvent *event,
                    AEKeyword keyword, CFStringRef stringRef)              // 1
{
```



```
    UInt8 *textBuf;
    CFIndex length, maxBytes, actualBytes;

    length = CFStringGetLength(stringRef);                                  // 2

    maxBytes = CFStringGetMaximumSizeForEncoding(length,
                                    kCFStringEncodingUTF8);                 // 3
    textBuf = malloc(maxBytes);                                             // 4
    if (textBuf)
    {
        CFStringGetBytes(stringRef, CFRangeMake(0, length),
                kCFStringEncodingUTF8, 0, true,
                (UInt8 *) textBuf, maxBytes, &actualBytes);                 // 5

        OSErr err = AEPutParamPtr(event, keyword,
                typeUTF8Text, textBuf, actualBytes);                        // 6

        free(textBuf);                                                      // 7
        return err;
    }
    else
        return memFullErr;
}
```

Here's a description of how this function works:

1. It receives a pointer to a descriptor to add a parameter to, a key word to identify the parameter, and a string reference from which to obtain the text for the parameter.
2. It gets the length of the string from the passed reference.
3. It determines the maximum number of bytes needed to store the text, based on its encoding.
4. It allocates a buffer of that size.
5. It gets the text from the string reference.
6. It adds the text as a parameter to the passed descriptor. (If called with the keyword `keyErrorString`, for example, it adds an error string parameter.)
7. It frees its local buffer.

Listing 5-4 shows how you could call `AEPutParamString` to add an error string to a reply Apple event.

Listing 5-4 Adding an error string to an Apple event with AEPutParamString

```
CFStringRef errStringRef = getLocalizedErrMsgForErrNumber(err);
OSErr anErr = AEPutParamString(reply, keyErrorString, errStringRef);
```

Here's a description of how this code snippet works:

1. It calls an application-defined function to obtain a localized error message as a string reference, based on a passed error number. (For information on localized strings, see Working With Localized Strings in *Bundle Programming Guide*.)
2. It calls `AEPutParamString`, passing the event (obtained previously) to add the error text to, the error string keyword, and the string reference for the error message text.

A Handler for the Open Documents Apple Event

Listing 5-5 shows a handler that responds to the `open documents` Apple event.

Listing 5-5 An Apple event handler for the open documents event

```
static pascal OSErrOpenDocumentsAE(const AppleEvent *theAppleEvent, AppleEvent
*reply, long handlerRefcon)
{
    AEDescList  docList;
    FSRef       theFSRef;
    long        index;
    long        count = 0;
    OSErr       err = AEGetParamDesc(theAppleEvent,
                            keyDirectObject, typeAEList, &docList);              // 1
    require_noerr(err, CantGetDocList);                                          // 2

    err = AECountItems(&docList, &count);                                        // 3
    require_noerr(err, CantGetCount);

    for(index = 1; index <= count; index++)                                      // 4
    {
        err = AEGetNthPtr(&docList, index, typeFSRef,
```



```
                        NULL, NULL, &theFSRef, sizeof(FSRef), NULL);            // 5
        require_noerr(err, CantGetDocDescPtr);

        err = OpenDocument(&theFSRef);                                          // 6
    }
    AEDisposeDesc(&docList);                                                    // 7

CantGetDocList:
CantGetCount:
CantGetDocDescPtr:
    if (err != noErr)                                                           // 8
    {
        // For handlers that expect a reply, add error information here.
    }
    return(err);                                                                // 9
}
```

Here's a description of how this `open documents` handler works:

1. It calls an Apple Event Manager function (`AEGetParamDesc`) to obtain a descriptor list from the direct object of the received Apple event. This is a list of file aliases, one for each document to open.
2. It uses the `require_noerr` macro (defined in `AssertMacros.h`) to jump to a labeled location as a simple form of error handling.
3. It calls an Apple Event Manager function (`AECountItems`) to obtain the number of items in the descriptor list.
4. It sets up a loop over the items in the descriptor list.
5. It calls an Apple Event Manager function (`AEGetNthPtr`) to get an item from the list, by index, asking for a type of `typeFSRef`. This causes the function to coerce the retrieved item (a file alias) to the requested type.
6. It calls a function you have written (`OpenDocument`) to open the current document from the list.

   A `print documents` event handler would be very similar to this one, but in this step could call your `PrintDocument` function.
7. It disposes of the document list descriptor.

8. The `open documents` event sent by the Mac OS doesn't expect a reply, so the `reply` parameter is a null event. For event handlers that expect a reply, this is where you can use the techniques described earlier in this chapter to add an error number or an error string. Some examples of events that expect a reply (and thus supply a non-null reply event) include `get data` events and `quit` events.
9. It returns an error code (`noErr` if no error has occurred).

As an alternative to code shown in this handler, you might use an approach that extracts the document list in the event handler, then passes it to a separate function, `OpenDocuments`, to iterate over the items in the list. This has the advantage that you can also call the `OpenDocuments` function with the document list obtained from `NavDialogGetReply` when you are working with Navigation Services. The `selection` field of the returned `NavReplyRecord` is a descriptor list like the one described above.

Handling Apple Events Sent by the Mac OS

Mac OS X sends Apple events to communicate with applications in certain common situations, such as when launching the application or asking it to open a list of documents. These events are sometimes referred to as the "required" events, although applications are not required to support all of them. For example, if your application isn't document-based, it won't support the `open documents` or `print documents` events. However, any application that presents a graphical user interface through the Human Interface Toolbox or the Cocoa application framework should respond to as many of these events as it makes sense for that application to support.

Carbon applications typically install Apple event handlers to handle these events, though in some cases a default handler may be installed automatically (as described in "Common Apple Events Sent by the Mac OS" (page 58)).

For Cocoa applications, most of the work of responding to events sent by the Mac OS happens automatically, though in some cases applications may want to step in and modify the default behavior. For more details, including some information of general interest, see How Cocoa Applications Handle Apple Events in *Cocoa Scripting Guide*.

Common Apple Events Sent by the Mac OS

The following are Apple events your application is likely to receive from the Mac OS. For information on the constants associated with these events, see "Event ID Constants for Apple Events Sent by the Mac OS" (page 96).

- `open application` (or `launch`)

Received when your application is launched. The application should perform any tasks required when a user launches it without opening or printing any existing documents, though it may of course open a new, untitled document.

Starting in Mac OS X version 10.4, an `open application` Apple event may contain information about whether the application was launched as a login item or as a service item. For details on working with this kind of information, see "Launch Apple Event Constants" in "Apple Event Manager Constants" in *Apple Event Manager Reference*.

- `reopen application`

  Received when the application is reopened—for example, when the application is open but not frontmost, and the user clicks its icon in the Dock. The application should perform appropriate tasks—for example, it might create a new document if none is open.

- `open documents`

  Received with a list of documents for the application to open. Listing 5-5 (page 56) shows a complete Apple event handler for this event.

  Starting in Mac OS X version 10.4, the `open documents` Apple event may contain an optional parameter identified by the key `keyAESearchText`. If present, the parameter contains the search text from the Spotlight search that identified the documents to be opened. The application should make a reasonable effort to display an occurrence of the search text in each opened document—for example by scrolling text into view that contains all or part of the search text.

- `print documents`

  Received with a list of documents for the application to print. You can obtain file system references to the items in the list using code like that you use when handling an `open documents` event. See "Interacting With the User" (page 49) for information on when it is appropriate to display a dialog or perform other interaction.

  Starting in Mac OS X version 10.3, the print documents event was extended to include an optional `print settings` parameter that defines the settings for the print job or jobs, and an optional `print dialog` parameter that specifies whether the application should display the Print dialog. By default, the application should not show the Print dialog if the caller doesn't specify a print dialog parameter. For details, see Technical Note TN2082, The Enhanced Print Apple Event.

- `open contents`

  Available starting in Mac OS X version 10.4. Received when content, such as text or an image, is dropped on the application icon—for example, when a dragged image is dropped on the icon in the Dock.

  The structure of this event is very similar to the `open documents` event. The direct object parameter consists of a list of content data items to be opened. The `descriptorType` for each item in the list indicates the type of the content (`'PICT'`, `'TIFF'`, `'utf8'`, and so on).

The application should use the content in an appropriate way—for example, if a document is open, it might insert the content at the current insertion point; if no document is open, it might open a new document and insert the provided text or image.

If your Carbon application provides a service that can accept the type of data in an `open contents` Apple event it receives, you can handle the event without even installing an event handler for it—a default handler will be installed, if one isn't present, and invoked automatically. As a result, your service provider will receive a `{kEventClassService, kEventServicePerform}` Carbon event.

Because there is no way to return data from an `open contents` Apple event, the default handler only invokes a service that accepts the given data type but returns nothing.

If your application doesn't support services, or if you want to provide special handling for the `open contents` Apple event, your application can install a handler for the event.

For information on working with services, see *Setting Up Your Carbon Application to Use the Services Menu*.

- `quit application`

  Received when the application should quit. The application can perform any special operations required at that time.

  If your application calls `RunApplicationEventLoop`, a simple `quit` handler is installed automatically. If you install your own handler, it can perform any required operations, then return `errAEEventNotHandled` so that the default handler continues quitting the application. If you return another error value, the application will not quit. However, your `quit` handler can call `QuitApplicationEventLoop`, which causes `RunApplicationEventLoop` to quit and return back to `main()`. In that case you can return `noErr` from your handler instead of `errAEEventNotHandled`.

  If your application does not call `RunApplicationEventLoop`, you should supply your own `quit` handler.

  You should not terminate your application from within a `quit` event handler. Instead, you should set a flag that you check outside the handler.

- `show preferences`

  Received when the user chooses the Preferences menu item. Carbon applications that handle the Preferences command can install an Apple event handler for this event, but they more commonly install a Carbon event handler for `kEventCommandProcess` and check for the `kHICommandPreferences` command ID.

- `application died`

  Received by an application that launched another application when the launched application quits or terminates. Your application can respond to the information that the other application is no longer running.

Installing Handlers for Apple Events Sent by the Mac OS

Listing 5-6 shows how your application installs handlers for various Apple events that are sent by the Mac OS. The listing assumes that you have defined the functions `OpenApplicationAE`, `ReopenApplicationAE`, `OpenDocumentsAE`, and `PrintDocumentsAE` to handle the Apple events `open application`, `reopen`, `open documents`, and `print documents`, respectively.

This function doesn't install handlers for the `open contents` and `quit` Apple events, so the application will rely on the default handlers for those events (described previously in "Common Apple Events Sent by the Mac OS" (page 58)).

Listing 5-6 Installing event handlers for Apple events from the Mac OS

```
static  OSErr   InstallMacOSEventHandlers(void)
{
    OSErr       err;

    err     = AEInstallEventHandler(kCoreEventClass, kAEOpenApplication,
                NewAEEventHandlerUPP(OpenApplicationAE), 0, false);
    require_noerr(err, CantInstallAppleEventHandler);

    err     = AEInstallEventHandler(kCoreEventClass, kAEReopenApplication,
                NewAEEventHandlerUPP(ReopenApplicationAE), 0, false);
    require_noerr(err, CantInstallAppleEventHandler);

    err     = AEInstallEventHandler(kCoreEventClass, kAEOpenDocuments,
                NewAEEventHandlerUPP(OpenDocumentsAE), 0, false);
    require_noerr(err, CantInstallAppleEventHandler);

    err     = AEInstallEventHandler(kCoreEventClass, kAEPrintDocuments,
                NewAEEventHandlerUPP(PrintDocumentsAE), 0, false);
    require_noerr(err, CantInstallAppleEventHandler);


CantInstallAppleEventHandler:
    return err;
}
```


For a description of the parameters you pass to the `AEInstallEventHandler` function, see "Installing Apple Event Handlers" (page 32).

Creating and Sending Apple Events

This chapter provides information and sample code that will help you create and send Apple events and handle Apple events you receive in response to those you send. Before reading this chapter, you should be familiar with the information in "Building an Apple Event" (page 18).

Applications most commonly create and send Apple events for one of the reasons described in "When Applications Use Apple Events" (page 11): to communicate directly with other applications or to support recording in a scriptable application. And as described in "Two Approaches to Creating an Apple Event" (page 25), you can create an Apple event in one step with the `AEBuildAppleEvent` function, or you can create a possibly incomplete Apple event with `AECreateAppleEvent`, then add attributes and parameters to complete the event.

> **Note:** "Two Approaches to Creating an Apple Event" (page 25) also briefly describes a third approach, using stream-oriented calling conventions, and points to documentation for that approach.

Functions for Creating Apple Event Data Structures

In addition to the `AEBuildAppleEvent` and `AECreateAppleEvent` functions for creating Apple events, you use the following functions for creating other data structures you use with Apple events:

- For creating descriptor records and lists and adding items to lists:

  `AEBuildDesc`, `AECreateDesc`, `AECreateList`, `AEPutPtr`, `AEPutDesc`

- For adding attributes and parameters to Apple events and Apple event records:

  `AEBuildParameters`, `AEPutParameter`, `AEPutParamDesc`, `AEPutAttributePtr`, `AEPutAttributeDesc`

Table A-1 (page 93) lists these and other Apple Event Manager functions. For complete function descriptions, see *Apple Event Manager Reference*.

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 send the Apple event to. You can send Apple events to applications on the local computer or on remote computers on the network. You specify a target address with a **target address descriptor**.

The preferred descriptor types for identifying the address in an address descriptor are shown in "Address Descriptor Type Constants" (page 98). You can also use another type if your application provides a coercion handler that coerces that type into one of the address types that the Apple Event Manager recognizes. See "Writing and Installing Coercion Handlers" (page 79) for more information on coercion handlers.

To address an Apple event to a target on a remote computer on the network, you generally use `typeApplicationURL` for the address descriptor type. To address an Apple event to a target on the local computer you can use any of the address types.

The fastest way for your application to send an Apple event to itself is to use a target address with a process serial number of `kCurrentProcess`. For more information, see "Addressing an Apple Event for Direct Dispatching" (page 65).

Creating a Target Address Descriptor

If you want to create an Apple event with the `AECreateAppleEvent` function, you need to create a target address descriptor to pass to the function. You can do so by calling the `AECreateDesc` function. Listing 6-1 shows how to create an address descriptor for a process serial number (`typeProcessSerialNumber`). Other target address descriptor types are shown in "Address Descriptor Type Constants" (page 98).

Listing 6-1 Creating an address descriptor using a process serial number

```
OSErr               err;
ProcessSerialNumber thePSN;
AEAddressDesc       addressDesc;

err = GetProcessNumber(&thePSN);                                    // 1
if (err == noErr)
{
    err = AECreateDesc(typeProcessSerialNumber,
                       &thePSN, sizeof(thePSN), &addressDesc);      // 2
}
```


Here's a description of how this code snippet works:

1. It calls an application-defined function, `GetProcessNumber`, to obtain the process serial number of another process.

   To create a target address descriptor for the current process, see Listing 6-2 (page 66).

2. It calls `AECreateDesc` to create a target address based on the specified process serial number.

   Your application should call `AEDisposeDesc` to dispose of the address descriptor when it is finished with it.

When you create an Apple event with the `AEBuildAppleEvent` function, it uses information from the following three parameters to create a target address descriptor:

`DescType addressType`
: The address type for the address information described in the next two parameters: for example, `typeProcessSerialNumber` or `typeKernelProcessID`.

`const void * addressData`
: A pointer to the address information.

`long addressLength`
: The number of bytes pointed to by the `addressData` parameter.

For a code example that uses `AEBuildAppleEvent`, see Listing 6-4 (page 68).

Addressing an Apple Event for Direct Dispatching

Applications typically send Apple events to themselves to support recordability, discussed briefly in "When Applications Use Apple Events" (page 11). The best way for your application to send Apple events to itself is to use an address descriptor of `typeProcessSerialNumber` with the `lowLongOfPSN` field set to `kCurrentProcess` and the `highLongOfPSN` field set to 0. Listing 6-2 shows how to do this.

When you send an Apple event with this type of target address descriptor, the Apple Event Manager jumps directly to the appropriate Apple event handler without going through the normal event-processing sequence. This is not only more efficient, it avoids the situation 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` Apple event arrives in the queue after the window activate event, a selection in the wrong window might be cut.

Listing 6-2 Creating an address descriptor that specifies the current application

```
AEAddressDesc addressDesc;
ProcessSerialNumber selfPSN = { 0, kCurrentProcess };                    // 1

OSErr err = AECreateDesc(typeProcessSerialNumber, &selfPSN,
                         sizeof(selfPSN), &addressDesc);                 // 2
```

Here's a description of how this code snippet works:

1. Sets up a process serial number for the current process, as described above.
2. Calls `AECreateDesc` to create a target address that specifies the current application by its process serial number.

   Your application should call `AEDisposeDesc` to dispose of the address descriptor when it is finished with it.

Your application can send events to itself using other forms of target addressing. However, this can lead to the event dispatching sequence problems just described.

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` or `AESendMessage` function.

Obtaining the Addresses of Remote Processes

You can use `AECreateRemoteProcessResolver` and related functions to obtain the addresses of other processes on the network. To do so you follow these steps:

1. Call `AECreateRemoteProcessResolver` to get a process resolver.
2. Call `AERemoteProcessResolverGetProcesses` to obtain a dictionary containing, for each remote application, the URL of the application and its human readable name (and possibly other information).
3. Call `AEDisposeRemoteProcessResolver` to dispose of the process resolver.
4. Create a target address descriptor based on a specified URL.
   a. Extract the URL from the dictionary entry as a `CFURLRef`.
   b. Convert it to a `CFStringRef` (for example, with `CFURLGetString`).
   c. Extract a text string in UTF-8 encoding. (For an example of code that extracts unicode text from a `CFStringRef`, see Listing 5-3 (page 54).)

d. Put the text into an address descriptor using the type `typeApplicationURL`. The format of the URL associated with this type is described in the Discussion section of "Descriptor Type Constants" in *Apple Event Manager Reference*.

5. Alternatively, if the dictionary entry contains a process ID for the remote process, you can create an address descriptor based on the type `typeKernelProcessID`.

You can use the remote process resolver technology, for example, to present an interface to allow a user to select a remote application. For more information on these functions, see the section "Locating Processes on Remote Computers" in *Apple Event Manager Reference*.

Creating an Apple Event

This section provides examples of how to create an Apple event with `AECreateAppleEvent` and with `AEBuildAppleEvent`. These functions are introduced in "Two Approaches to Creating an Apple Event" (page 25).

Creating an Apple Event With AEBuildAppleEvent

The `AEBuildAppleEvent` function provides a mechanism for converting a specially formatted string into a complete Apple event. This function is similar to `AECreateAppleEvent`, but contains additional parameters it uses in creating the Apple event and constructing parameters for it.

The `AEBuildAppleEvent` function is similar to the `printf` family of routines in the standard C library. The syntax for the format string defines an Apple event as a sequence of name-value pairs, with optional parameters preceded with a tilde (~) character. For details, see Technical Note TN2106, AEBuild*, AEPrint*, and Friends.

The next two code listings show how you might use `AEBuildAppleEvent` to create an Apple event that tells the Finder to reveal the startup disk (make it visible on the desktop).

Listing 6-3 Constants used in creating a reveal Apple event for the Finder

```
const CFStringRef startupDiskPath = CFSTR("/");                    // 1
const OSType finderSignature = 'MACS';                             // 2
```

Here's a description of these constants, which are defined in `AERegistry.h` and used by the Finder in its Apple event support:

1. Defines a string reference for the POSIX-style path of the startup disk.
2. Defines the application signature for the Mac OS Finder application.

Because the Finder is always running in Mac OS X, it is generally safe to send it an Apple event without first making sure it has been launched.

Listing 6-4 shows a function that creates an Apple event to reveal the startup disk in the Finder.

Listing 6-4 Creating a reveal Apple event with AEBuildAppleEvent

```
OSErr BuildRevealStartupDiskAE (AppleEvent * revealEvent)                    // 1
{
    FSRef startupDiskFSRef;
    AliasHandle startupDiskAlias;
    OSErr err = noErr;

    CFURLRef startupURLRef =
        CFURLCreateWithFileSystemPath(kCFAllocatorDefault,
            startupDiskPath, kCFURLPOSIXPathStyle, true);                    // 2

    if (CFURLGetFSRef(startupURLRef, &startupDiskFSRef))                     // 3
    {
        err = FSNewAlias(NULL, &startupDiskFSRef, &startupDiskAlias);        // 4
        if (err == noErr)
        {
            err = AEBuildAppleEvent(                                         // 5
                    kAEMiscStandards,                                        // 6
                    kAEMakeObjectsVisible,                                   // 7
                    typeApplSignature,                                       // 8
                    &finderSignature,                                        // 9
                    sizeof(finderSignature),                                 // 10
                    kAutoGenerateReturnID,                                   // 11
                    kAnyTransactionID,                                       // 12
                    revealEvent,                                             // 13
                    NULL,                                                    // 14
                    "'----':[alis(@@)]",                                     // 15
                    startupDiskAlias);                                       // 16
        }
    }
```



```
    else
        err = memFullErr;                                                   // 17

    return err;                                                             // 18
}
```

Here's a description of how the `BuildRevealStartupDiskAE` function works:

1. It is passed a pointer to an Apple event data structure for the event to be created.
2. It calls `CFURLCreateWithFileSystemPath` to create a `CFURLRef` for the path to the startup disk, passing the constant `startupDiskPath` declared in Listing 6-3.
3. It calls `CFURLGetFSRef` to get an `FSRef` file reference to the startup disk from the `CFURLRef`.
4. It calls `FSNewAlias` to convert the `FSRef` to an alias handle for the startup disk, to use in creating the Apple event.
5. It calls `AEBuildAppleEvent` to create the Apple event. The next several items describe the parameters you pass to that function.
6. Specifies `kAEMiscStandards`, a constant defined in `AERegistry.h`, for the event class.
7. Specifies `kAEMakeObjectsVisible`, also defined in `AERegistry.h`, for the event ID.
8. Passes `typeApplSignature` to specify a target address type.
9. Passes `finderSignature`, defined in Listing 6-3, to specify the application signature for the Finder.
10. Passes the size of the application signature.
11. Passes the Apple Event Manager constant `kAutoGenerateReturnID`, indicating the Apple Event Manager should set a return ID for the event. Your application can specify its own return ID, if needed.
12. Passes the Apple Event Manager constant `kAnyTransactionID`, indicating the event is not part of a series of interdependent transactions.
13. Passes a pointer to the Apple event to be built.
14. Passes a value of `NULL`, indicating no error information is required.

    See Technical Note TN2106, AEBuild*, AEPrint*, and Friends for information on working with error information when using `AEBuildAppleEvent`.
15. Passes a format string containing information for any attributes and parameters to add to the Apple event. In this case, there is just one parameter, an alias (`'----'`).

    The identifier for the direct parameter in an Apple event is specified by the constant `keyDirectObject` (`'----'`). The minus sign has special meaning in AEBuild strings, so it should always be enclosed in single quotes when it is used to identify the direct parameter for an Apple event.

16. Passes the previously created alias handle as the data corresponding to the entry in the format string.
17. In the case where the function could not create an `FSRef`, it sets a return error value.
18. It returns a value indicating whether an error occurred.

To see how the `BuildRevealStartupDiskAE` function is called, and how you can send the resulting Apple event, see Listing 6-7 (page 75).

Creating an Apple Event With AECreateAppleEvent

To create an Apple event with `AECreateAppleEvent`, you perform these steps:

- Prepare a target address descriptor, as described in "Creating a Target Address Descriptor" (page 64).
- Call `AECreateAppleEvent` to create the Apple event, passing the event class, event ID, and other information.
- If necessary, call other Apple Event Manager functions to add additional information to the event, until it contains all the information required for the task you want to perform.

For example, suppose your application wants to send a `quit` Apple event to another application. It might do this to terminate an application it launched previously, or perhaps to make sure an application is not running so it can perform an update. Listing 6-5 shows how to create a `quit` application Apple event.

Listing 6-5 Creating a quit Apple event with AECreateAppleEvent

```
AppleEvent someAE;

err = AECreateAppleEvent(                                    // 1
        kCoreEventClass,                                     // 2
        kAEQuitApplication,                                  // 3
        &theTarget,                                          // 4
        kAutoGenerateReturnID,                               // 5
        kAnyTransactionID,                                   // 6
        &someAE);                                            // 7
```

Here's how the code in Listing 6-5 works:

1. It calls `AECreateAppleEvent` to create the Apple event. The following items describe the parameters you pass to that function.
2. Specifies `kCoreEventClass`, a constant defined in `AppleEvents.h`, for the event class.

3. Specifies `kAEQuitApplication`, also defined in `AppleEvents.h`, for the event ID.
4. Passes the address of the previously constructed target address descriptor, which identifies the application to send the `quit` event to.
5. Passes the Apple Event Manager constant `kAutoGenerateReturnID`, indicating the Apple Event Manager should set a return ID for the event. Your application can specify its own return ID, if needed.
6. Passes the Apple Event Manager constant `kAnyTransactionID`, indicating the event is not part of a series of interdependent transactions.
7. It passes the address of an Apple event data structure for the event to be created.

That's all you need to do to create a `quit` Apple event. However, to create a more complicated Apple event, you typically need to add attributes or parameters to the event. For example, your application might receive a `get data` Apple event for which it returns some specified text as the direct parameter of the reply Apple event. Listing 6-6 shows how to add such a direct parameter, using a previously defined function.

Listing 6-6 Adding a direct parameter to an Apple event

```
OSErr anErr = AEPutParamString(                                    // 1
                    reply,                                         // 2
                    keyDirectObject,                               // 3
                    textStringRef);                                // 4
```

Here's how the code in Listing 6-6 works:

1. It calls the application-defined function `AEPutParamString`, shown in Listing 5-3 (page 54). The following items describe the parameters you pass to add a direct parameter to the Apple event.
2. A pointer to the Apple event to add the parameter to.
3. An Apple Event Manager keyword constant specifying that the parameter is to be added as the direct parameter of the Apple event.
4. A `CFStringRef`, obtained prior to the call, that specifies the text for the direct parameter.

You can also use the `AEBuildParameters` function, introduced in "Two Approaches to Creating an Apple Event" (page 25), to add more complicated attributes or parameters to an existing Apple event.

Disposing of Apple Events You Create

Regardless of how you create an Apple event, your application is responsible for disposing of it with the `AEDisposeDesc` function when you are finished with it. Your application must also dispose of all the descriptor records it creates when adding parameters and attributes to an Apple event—remember, many Apple Event Manager functions make copies of descriptors and of their associated data, as noted in "Disposing of Apple Event Data Structures" (page 47).

You normally dispose of your Apple event and its reply, if any, after you send the event and receive a result (either from `AESend` or `AESendMessage`). You should dispose of the data structures you created even if the sending function returns an error. If you are sending the event asynchronously, you need not wait for the reply Apple event before disposing of the sent Apple event.

Sending an Apple Event

Once you have created an Apple event, the Apple Event Manager provides two choices for sending it. The `AESend` function provides more options but higher overhead, while the `AESendMessage` function provides fewer options but less overhead.

When to Use AESend

The `AESend` function has parameters for specifying how to handle timeouts, idle processing, and event filtering. It requires your application to link with the entire Carbon framework, which brings in the HIToolbox framework and requires a connection to the window server.

Using `AESend` is appropriate only for applications that require the use of idle processing and event filtering. This type of processing is not generally needed, or recommended, for applications that handle events with Carbon event handlers. Therefore, only Carbon applications that use older style event handling are likely to need to use `AESend`.

> **Note:** If you call `AESend` and pass `NULL` for the filter and idle functions, `AESend` will call through to the less expensive `AESendMessage` function.

The `AESend` function provides these parameters that are not available to `AESendMessage`:

`AESendPriority sendPriority`
: This parameter is deprecated in Mac OS X.

`AEIdleUPP idleProc`

A universal procedure 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 idle function can also perform other tasks (such as displaying a delay cursor) while waiting for a reply or a return receipt.

If your application specifies the `kAEWaitReply` flag in the `sendMode` parameter and you wish your application to get periodic time while waiting for the reply to return, you must provide an idle function. Otherwise, you can pass a value of `NULL` for this parameter.

For advice on whether to use idle functions, see "Specifying Send and Reply Options" (page 73).

`AEFilterUPP filterProc`

A universal procedure pointer to a function that determines which incoming Apple events should be received while the handler waits for a reply or a return receipt. If your application doesn't need to filter Apple events, you can pass a value of `NULL` for this parameter. If you do so, no application-oriented Apple events are processed while waiting.

When to Use AESendMessage

The `AESendMessage` function requires less overhead than the `AESend` function. It also allows you to send Apple events by linking with just the Application Services framework, and not the entire Carbon framework (and window server), as required by `AESend`.

Using the `AESendMessage` function is appropriate for applications that don't require idle processing and event filtering (most modern Carbon applications), and where any of the following applies:

- Efficiency is a key concern.
- The application has no user interface.
- The application should not link with Carbon.

Specifying Send and Reply Options

Regardless of which function you use to send Apple events, you'll need to specify certain options, such as how to interact with the user, whether you want a reply Apple event, and if so, how to receive it. You do this by setting flags in the `sendMode` parameter, present in both `AESend` and `AESendMessage`. Here's a brief description of that parameter:

`AESendMode sendMode`

Specifies options for how the server application should handle the Apple event. To obtain a value for this parameter, you add together constants to set bits that specify the reply mode, the interaction level, and possibly flags for other options.

You can read a full description of all the constants for setting send mode flags in the constant section "AESendMode" in *Apple Event Manager Reference*. But here are some of the more common choices you might make for these flags.

Specifying a Reply in the Send Mode Parameter

If you do not want a reply to the Apple event you are sending, you specify the `kAENoReply` flag in the `sendMode` parameter.

If you're willing to wait a certain amount of time for a reply, specify `kAEWaitReply`—the 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 contains the reply Apple event.

If you specify the `kAEWaitReply` flag to `AESend`, you should provide an idle function to 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.

You don't provide an idle function or a filter function when you call the `AESendMessage` function because it doesn't provide parameters for them. If you specify `kAEWaitReply` to `AESendMessage`, it executes in such a way that any timers or event sources added to your run loop are not called while inside `AESendMessage`.

Rather than wait for a reply, you can send Apple events asynchronously and receive reply events when they're ready through normal event processing. To do this, you specify the `kAEQueueReply` flag and register a handler for reply events with the event class `kCoreEventClass` and event ID `kAEAnswer`. Your application processes reply events in the same way it does other Apple events. This approach is recommended because it is far more efficient than waiting in an idle routine. In addition, it avoids the possibility of getting into odd states, such as loops within loops while waiting for replies as the idle routine processes events.

If you specify `kAEWaitReply` or `kAEQueueReply`, the Apple Event Manager automatically passes 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.

If you specify the `kAENoReply` flag, the reply Apple event passed to the server application consists of a null descriptor record.

Specifying User Interaction in the Send Mode Parameter

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. 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 "Interacting With the User" (page 49).

Sending an Apple Event With AESend

Listing 6-7 shows how you can build an Apple event with `AEBuildAppleEvent` and send the event with `AESend`.

Listing 6-7 Sending an Apple event with the AESend function

```
OSErr err = noErr;
AppleEvent revealEvent;

err = BuildRevealStartupDiskAE(&revealEvent);                          // 1

if (err == noErr)
{
    err = AESend(&revealEvent,                                         // 2
                 NULL,       // No reply event needed.                // 3
                 kAENoReply | kAECanInteract,                          // 4
                 kAENormalPriority,// Normal priority.                // 5
                 kAEDefaultTimeout,// Let AE Mgr decide on timeout.   // 6
                 NULL,           // No idle function.                 // 7
                 NULL);          // No filter function.               // 8

    err = AEDisposeDesc(&revealEvent);                                 // 9
}
```

Here's how the code in Listing 6-7 works:

1. Calls the application-defined function `BuildRevealStartupDiskAE`, shown in Listing 6-4 (page 68), to build an Apple event that tells the Finder to reveal the startup disk.

2. Calls the function `AESend`, passing the Apple event obtained by the previous function call. The next several items describe the remaining parameters you pass to the `AESend` function.
3. Passes `NULL` for the reply event because no reply is expected.
4. Passes a logical combination of constants indicating that no reply is expected and that interaction with the user is allowed (although none is likely for this event).

   In the case where your application wants an asynchronous reply and does not allow interaction, you would pass `kAEQueueReply | kAENeverInteract`.
5. Passes a constant indicating the event should have normal priority. However, priority is ignored on Mac OS X.
6. Passes a constant indicating the Apple Event Manager should use the default timeout length (usually about thirty seconds). You can, instead, pass a specific value for the number of ticks (sixtieths of a second) to delay.
7. Passes `NULL`, indicating it will not use an idle function (needed only in specific cases where your application waits for a reply).
8. Passes `NULL`, indicating it will not use a filter function (needed, in conjunction with an idle function, only in specific cases where your application waits for a reply).
9. Calls `AEDisposeDesc` to dispose of the Apple event.

   In cases where you expect a reply Apple event, your application should dispose of that event as well, once it is finished with it. If the reply is handled asynchronously, dispose of the reply event in the reply handler.

In this example, the call to `AESend` does not supply an idle function or a filter function. As a result, `AESend` will fall through and call the `AESendMessage` function, which has less overhead. When you don't require idle and filter functions, you can, of course, call `AESendMessage` directly, as described in the next section.

As noted in "Specifying a Reply in the Send Mode Parameter" (page 74), even in the case where your application expects a delayed reply to an Apple event it sends, the most efficient mechanism is to ask for queued replies and register a handler to receive them.

Sending an Apple Event With AESendMessage

Listing 6-8 shows how you can send an Apple event with `AESendMessage`.

Listing 6-8 Sending an Apple event with the AESendMessage function

```
OSErr err = noErr;
AppleEvent revealEvent;
```



```
err = BuildRevealStartupDiskAE(&revealEvent);                    // 1

if (err == noErr)
{
    err = AESendMessage(&revealEvent,                            // 2
                NULL,                                            // 3
                kAENoReply | kAENeverInteract,                   // 4
                kAEDefaultTimeout);                              // 5

    err = AEDisposeDesc(&revealEvent);                           // 6
}
```

Here's how the code in Listing 6-7 works:

1. Calls the application-defined function `BuildRevealStartupDiskAE`, shown in Listing 6-4 (page 68), to build an Apple event that tells the Finder to reveal the startup disk.
2. Calls the function `AESendMessage`, passing the Apple event obtained by the previous function call. The next several items describe the remaining parameters you pass to the `AESendMessage` function.
3. Passes `NULL` for the reply event because no reply is expected.
4. Passes a logical combination of constants indicating that no reply is expected and that interaction with the user is not allowed.
5. Passes a constant indicating the Apple Event Manager should use the default timeout length (usually about thirty seconds). You can, instead, pass a specific value for the number of ticks (sixtieths of a second) to delay.
6. Calls `AEDisposeDesc` to dispose of the Apple event.

Handling a Reply Apple Event

When your application receives a reply event, either as a return parameter from the sending routine or in a return event handler, it uses Apple Event Manager functions to extract the information it needs from the event. This process is the same as the one described in "Responding to Apple Events" (page 48).

Whenever a server application provides an error string parameter in a reply event, it should also provide an error number. However, you can't count on all server applications to do so. A client application should therefore check for both the `keyErrorNumber` (for example, see Listing 5-2 (page 53)) and `keyErrorString` parameters before assuming that no error has occurred.

When your application has finished using the reply Apple event, it should dispose of it by calling the `AEDisposeDesc` function. The Apple Event Manager takes care of disposing of both the Apple event and the reply Apple event after a server application's handler returns, but the server is responsible for disposing of any data structures it creates while extracting data from the event.

Writing and Installing Coercion Handlers

Coercion is the process of converting a descriptor and the data it contains from one type to another. When coercing between types, by definition the descriptor type is changed to the new type. However, if the underlying data representation is the same, data conversion is not required.

Functions that perform coercions are referred to as **coercion handlers**. The Mac OS provides default coercion handlers to convert between many different descriptor types. Default handlers can, for example, convert aliases to file system specifications, integers to Boolean data types, and characters to numeric data types. These handlers may be implemented by the Apple Event Manager, the Open Scripting framework, or other frameworks. Table C-2 (page 103) lists descriptor types and the available default coercions.

You can also provide your own coercion handlers to perform coercions that the default handlers don't support. This chapter describes how to write coercion handlers and how to install them so that they are available to your application.

For many of the Apple Event Manager functions that your application uses to extract data from an Apple event, you can specify a desired descriptor type for the returned data. If the original data is of a different type, the Apple Event Manager attempts to coerce the data to the requested descriptor type. For more information on functions that let you specify a desired descriptor type, see "Coercing Data From an Apple Event" (page 40).

How Coercion Handlers are Installed and Dispatched

Coercion handling works by a process that is similar to the one described previously for dispatching Apple events to Apple event handlers:

- You write coercion handler functions to coerce data between types that your application works with but that are not handled by a default coercion handler.
- Your application registers coercion handlers with the Apple Event Manager for all the descriptor data types it can convert between. A handler may handle just one type of coercion or multiple types. To specify multiple types, you use the constant `typeWildCard`.
- The Apple Event Manager creates an application coercion handler dispatch table for your application. The dispatch table maps the descriptor types your application can coerce to the coercion handlers it provides.

  The Apple Event Manager also creates a system coercion dispatch handler, described below, for the default coercion handlers.

- When the Apple Event Manager needs to perform a coercion in your application, it goes through these steps until the coercion is handled:
  - It looks for a coercion handler in your application coercion dispatch table.
  - It looks for a coercion handler in your system coercion dispatch table, which includes the default coercions described in Table C-2 (page 103).

    Note: Default coercion handlers may be installed lazily, as they are needed.

  - If it can't find an appropriate coercion handler, it returns the result code `errAECoercionFail`.

Although your application can have both a system coercion dispatch table and an application dispatch table, you should generally install all coercion handlers in the application coercion dispatch table. For Carbon applications running in Mac OS 8 or Mac OS 9, a handler in the system coercion dispatch table could reside in the system heap, where it would be available to other applications. However, your application should not install a handler in a system coercion dispatch table with the goal that the handler will get called when other applications perform coercions—this won't necessarily work in Mac OS 8 or Mac OS 9 and will not work in Mac OS X.

Writing a Coercion Handler

When you write a coercion handler, it must do the following things:

- Validate the information passed to it so that it has a reasonable chance of success.
- Gain access to the information to be coerced.
- Convert the information to the specified type.
- Create a new descriptor of the specified type.

Warning: This coercion handler uses `typeChar`, which should be avoided starting in Mac OS X version 10.3, as noted in "Default Coercion Handlers" (page 102).

There are two types of coercion handlers. The first, which matches the format defined for the `AECoerceDescProcPtr` data type, expects the caller to pass a descriptor containing the data to be coerced. The second, which matches the format defined for the `AECoercePtrProcPtr` data type, expects the caller to pass a a pointer to the data to be coerced. These data types are described in *Apple Event Manager Reference*.

The examples in this chapter show how to work with a coercion handler that uses descriptors. However the differences in working with the pointer-based type are fairly straight-forward.

To write a a coercion handler named `CoerceApplesToOranges`, based on the `AECoerceDescProcPtr` data type, you would declare the handler as shown in Listing 7-1.

Listing 7-1 Declaring a coercion handler

```
OSErr CoerceApplesToOranges (
    const AEDesc * fromDesc,                                    // 1
    DescType toType,                                            // 2
    long handlerRefcon,                                         // 3
    AEDesc * toDesc                                             // 4
);
```

The following are descriptions of the numbered parameters:

1. A pointer to the descriptor to be coerced.
2. The type to coerce to.
3. A reference variable that will be passed to your handler—you can use it for any purpose.
4. A descriptor pointer where the handler will store the coerced descriptor.

   This routine creates a new descriptor, so it is up to the calling routine to dispose of the descriptor.

Suppose that you want to write a coercion handler to convert text strings into your internal “bar” data type. The `typeBar` type is defined as shown in Listing 7-2.

Listing 7-2 An application-defined data type

```
enum {
  typeBar = 'bar!'
};
```

Listing 7-3 provides a slightly simplified version of the handler you might write.

Listing 7-3 A simple coercion handler

```
static OSErr TextToBarCoercionHandler(const AEDesc* fromDesc, DescType toType,
long handlerRefcon, AEDesc* toDesc)
{
```



```
    require_noerr((fromDesc->descriptorType != typeChar),CoercionFailed);        // 1
    require_noerr((toType != typeBar), CoercionFailed);
    require_noerr((handlerRefcon != 1234), CoercionFailed);                      // 2

    long dataSize = AEGetDescDataSize(fromDesc);                                 // 3
    char dataPtr[dataSize];
    OSErr err = AEGetDescData(fromDesc, dataPtr, dataSize);
    require_noerr(err, CoercionFailed);

    long result = 0;
    const char* pChar = (const char*) dataPtr;
    while (dataSize-- > 0)                                                       // 4
        result += *pChar++;

    err = AECreateDesc(typeBar, &result, sizeof(result), toDesc);                // 5

    if (err != noErr)                                                            // 6
        err = errAECoercionFail;

CoercionFailed:                                                                  // 7
        return err;
}
```

Here's what the code in `TextToBarCoercionHandler` does:

1. Checks the passed parameters to validate that it can handle the requested coercion, using the macro `require_noerr`, which jumps to the error label `CoercionFailed` if a value isn't supported.

   Coercion handlers that support multiple types will have additional work to do here.

2. Checks that the passed reference constant matches the expected value. You previously set the reference constant when you installed the coercion handler (shown in "Installing a Coercion Handler" (page 83)).
3. Gets the size of the data in the descriptor to be coerced and uses it to get the actual data from the descriptor.
4. In a while loop, converts the data from type text to type bar (by summing the bytes).
5. Creates a new descriptor of type bar, using the coerced data.
6. If it was unable to create the coerced descriptor, returns an error indicating the coercion failed.

7. Whether it jumped to the error label or fell through on success, returns an error code indicating whether the coercion succeeded or failed.

These are the main differences if you want to write a coercion handler based on the `AECoercePtrProcPtr` data type, which works with a pointer to data:

- The coercion handler has two additional parameters: one specifies the descriptor type of the data the pointer parameter points to; the other specifies the size of the data.
- When obtaining the data to convert, you use the size and type information to extract the data from the pointer, rather than getting the type and the data from a passed descriptor.

Installing a Coercion Handler

To install a coercion handler, you use the `AEInstallCoercionHandler` function, which is declared as shown in Listing 7-4.

Listing 7-4 Declaration of AEInstallCoercionHandler

```
OSErr AEInstallCoercionHandler (
    DescType fromType,                          // 1
    DescType toType,                            // 2
    AECoercionHandlerUPP handler,               // 3
    long handlerRefcon,                         // 4
    Boolean fromTypeIsDesc,                     // 5
    Boolean isSysHandler                        // 6
);
```

You specify as parameters to this function:

1. The descriptor type of the data coerced by the handler. You can pass `typeWildCard`to accept all types.
2. The descriptor type of the resulting data. You can pass `typeWildCard` to accept all types.
3. The address of the coercion handler for this descriptor type; the handler is declared as described in "Writing a Coercion Handler" (page 80).
4. A reference constant to pass to the handler when it is called. You can use the reference constant for any purpose you want.
5. A Boolean value that indicates whether your coercion handler expects the data to be specified as a descriptor or as a pointer to the actual data.

6. 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.

To call the `TextToBarCoercionHandler` handler defined in Listing 7-3 (page 81), you can use code like that shown in Listing 7-5.

Listing 7-5 Installing a coercion handler

```
OSErr err = AEInstallCoercionHandler(
                typeChar, // Coerce from this type
                typeBar, // to this type,
                TextToBarCoercionHandler, //using this handler,
                1234,   //  passing this reference constant.
                true,   //  The handler operates on descriptors,
                false); //  and resides in the application table.
```

Here's what the parameters in this call specify:

1. Coerce from type `'TEXT'`.
2. Coerce to type `typeBar`.
3. Use the handler `TextToBarCoercionHandler` to perform the coercion.
4. Pass the reference constant 1234.
5. The handler operates on descriptors.
6. The handler resides in the application's coercion dispatch table.

Testing a Coercion Handler

You can use code like that shown in Listing 7-6 to test the `TextToBarCoercionHandler` handler defined in Listing 7-3 (page 81).

Listing 7-6 Testing a coercion handler

```
    AEDesc textDesc;
    const char* kText = "1234";
    OSErr err = AECreateDesc(typeChar, kText, strlen(kText), &textDesc);         // 1
```



```
AEDesc barDesc;
err = AECoerceDesc(&textDesc, typeBar, &barDesc);                    // 2
if (err == noErr)
{
    // Use the descriptor as needed.

    // Dispose of the descriptor when finished with it.
    err = AEDisposeDesc(&barDesc);                                   // 3
}
err = AEDisposeDesc(&textDesc);                                      // 4
```

Here is what the code in this snippet does:

1. Creates a descriptor containing the text string "1234".
2. Calls the Apple Event Manager function `AECoerceDesc`, passing the text descriptor, the desired type (type bar), and the address of a descriptor in which to store the coerced descriptor.
3. If the coercion is successful, disposes of the coerced descriptor. Because a coercion returns a new descriptor, your application must dispose of the descriptor when it is finished with it. (This code snippet does not include full error handling.)
4. Disposes of the text descriptor.

If the coercion fails, you can set a breakpoint in your coercion handler to determine if it is ever called. Possible problems include:

- You didn't install the coercion handler.
- The type you passed to `AECoerceDesc` was different than the type you registered for your coercion handler.
- The text descriptor you created has a different descriptor type than you registered for your coercion handler.

You can install a single coercion handler that specifies the constant `typeWildCard` for both the to and from types. Then any time the Apple Event Manager attempts to dispatch a coercion to the application, it will invoke that handler. This provides a convenient bottleneck for debugging.

Using wildcards can also be convenient as a general way to handle coercions, with one or a small number of handlers converting between supported types and the application's private data types.

Testing and Debugging Apple Event Code

To test and debug your Apple event code you can generally use the same techniques and tools you use with any code. For example, Xcode contains a full-featured source-level debugger that lets you set breakpoints and step through your code line by line. For C, C++, and Objective-C code, Xcode uses GDB, the debugger from the GNU Project. You can use Xcode's graphical interface to GDB or you can enter commands directly into the GDB console. For more information, see "Debugging" in Xcode Help and Debugging With GDB.

The remainder of this chapter provides tips that are specific to debugging code that works with Apple events.

Determining If an Application Is Receiving Apple Events

The easiest way to determine if your application is receiving a particular event is to set a breakpoint in the event handler for that event, then send an event of that type to your application. There are several ways to send Apple events to your application:

- You can create and send events that target your application from within the application itself or from a test application. For more information, see "Creating and Sending Apple Events" (page 63). This approach can be useful if you're setting up a test suite and want to thoroughly test how your application responds to a variety of events.
- You can use Script Editor to send Apple events to your application.

  For example, you can execute this one-line script to send your application a `quit` event:

Listing 8-1 A simple script to quit an application

```
tell app "MyApplication" to quit
```

  If your application is scriptable, you may prefer to set up a test suite of scripts to thoroughly exercise the events you support, rather than perform the same task in a test application.

  For additional information on sending Apple events with Script Editor, see "Script Editor is an Apple Event Test Tool" (page 87).
- You can use the Mac OS to send Apple events to your application.

For example, you can select an application document in the Finder and double-click it to send an `open documents` event to the application. See "Handling Apple Events Sent by the Mac OS" (page 58) for a list of the events the Mac OS sends to applications.

What if you're sending events to your application, but the debugger never reaches the breakpoints you set in your Apple event code? One simple approach is to install a single Apple event handler with the event type and event class set to `typeWildCard`. Any Apple event your application receives will be dispatched to this handler, so a breakpoint in the handler should allow you to verify whether the application is actually receiving events.

Once you know your application is receiving events, you can take a closer look at the events using the information provided in "Examining Apple Events" (page 88). That information is also useful if you never hit the breakpoint in your wildcard handler. For example, you can examine the test events you are sending to see if they correctly target your application.

Script Editor is an Apple Event Test Tool

The Script Editor application, located in `/Applications/AppleScript`, is a useful tool for working with Apple events. If your application is scriptable, you can use Script Editor to write and execute scripts that target your application, resulting in Apple events being sent to the application.

Even if your application is not fully scriptable, you can use Script Editor to send it events such as the `open application` and `quit` events that your application usually receives from the Mac OS. Listing 8-1 shows an example of this.

In addition, you can use Script Editor to construct and send Apple events using raw format, in which you enclose actual four-character codes in special characters to specify an event. For example, Listing 8-2 shows how to use the raw format to send an `open location` command to the Safari application and open the specified web page.

Listing 8-2 Sending a raw event to open an URL

```
tell application "Safari"
    «event GURLGURL» "http://www.apple.com/"
end tell
```

When you compile this script, Script Editor examines Safari's scripting dictionary and converts the second line to this:


```
open location "http://www.apple.com/"
```

However, for an application that doesn't have a scripting dictionary, the raw code is not replaced by an equivalent term, but the Apple event can still be sent and understood (if the application supports it).

You enter the special characters that surround the raw code (called double angle brackets or guillemets) by typing Option-\ and Option-Shift-\. For additional information, see Double Angle Brackets in Results and Scripts in AppleScript Language Guide.

"Turning on Apple Event Logging" (page 88) shows how you can examine the Apple events Script Editor sends and receives.

> **Note:** The following features can also be useful in debugging your Apple event code:
>
> You can execute AppleScript scripts from shell scripts and shell scripts from AppleScript scripts, which may come in handy for automating your testing. For more information, see "Using AppleScript With Other Scripting Systems" in *AppleScript Overview*.
>
> In addition, Xcode provides a build phase in which you can execute AppleScript scripts, and Xcode itself is scriptable. For additional information, see Xcode Help.

Examining Apple Events

There are several available mechanisms for examining the contents of Apple events that your application sends and receives.

Turning on Apple Event Logging

You can set environment variables in a Terminal window so that any Apple events sent or received by an application launched in that window are logged to the window in a human-readable format. Listing 8-4 shows how you would do this if you're working with the C shell.

Listing 8-3 Turning on logging for sent and received Apple events in the C shell

```
%setenv AEDebugSends 1; setenv AEDebugReceives 1
```

If you are using the `bash` shell you, you can use the form shown in Listing 8-4.

Listing 8-4 Turning on Apple event logging in the Bash shell

```
%export AEDebugSends=1; export AEDebugReceives=1
```

To see which Apple events an application sends and receives, you set these environment variables, then launch the application in a Terminal window. For example, to see what events the Script Editor application sends, you can execute the line in Listing 8-5. Once the Script Editor launches, you can compile and execute scripts and examine, in the Terminal window, the Apple events that are generated.

Listing 8-5 Launching Script Editor in Terminal

```
% /Applications/AppleScript/Script\ Editor.app/Contents/MacOS/Script\ Editor
```

Important: To launch an application in Terminal, you provide the full path to its executable, which is typically located inside the application bundle, rather than the path to the application itself.

Listing 8-6 shows how to perform the same task with the Finder, a scriptable application that may send Apple events to your application.

Listing 8-6 Launching Finder in Terminal

```
% /System/Library/CoreServices/Finder.app/Contents/MacOS/Finder
```

This technique for examining Apple events works for both Carbon and Cocoa applications. For example, Listing 8-7 shows the output for a `reopen` Apple event sent to a Carbon application when you click on its icon in the Dock.

Listing 8-7 Output of a reopen Apple event in Terminal

```
AE2000 (968): Received an event:
------oo start of event oo------
{ 1 } 'aevt':  aevt/rapp (ppc ){
          return id: 22609967 (0x159002f)
     transaction id: 0 (0x0)
  interaction level: 112 (0x70)
     reply required: 0 (0x0)
             remote: 0 (0x0)
  target:
    { 1 } 'psn ':  8 bytes {
```



```
      { 0x0, 0x60001 } (Dock)
    }
  optional attributes:
    { 1 } 'reco':  - 1 items {
      key 'optk' -
        { 1 } 'list':  - 1 elements {
          { 1 } 'keyw':  4 bytes {
            'frnt'
          }
        }
    }

  event data:
    { 1 } 'aevt':  - 1 items {
      key 'frnt' -
        { 1 } 'bool':  1 bytes {
          false
        }
    }
}
------oo  end of event  oo------
```

From the formatted output in Listing 8-7, you can identify various information in the Apple event. For example, `aevt/rapp` is the event class/event ID pair for the event. You can look up these values in the Apple Event Manager headers and see that `'rapp'` is the value of the constant `kAEReopenApplication`, defined in `AERegistry.h`. For this event, no reply is required (`reply required: 0`), but if a reply were required, the target would be the Dock (`target: { 1 } 'psn ': 8 bytes { { 0x0, 0x60001 } (Dock) }`), which sent the Apple event.

Although Apple event log information can be somewhat cryptic, you can see that the event contains an optional attribute containing boolean data with the value false and the key `'frnt'`. This indicates that the application was not frontmost at the time the `reopen` event was sent (when you clicked the application icon in the Dock). If the application is in front, the event data will contain the value `false`.

Note: For Cocoa applications, you can display additional formatted output for Apple events using the mechanism described in "Turn On Debugging Output for Scripting" in Key Steps for Creating Scriptable Applications in *Cocoa Scripting Guide*.

Observing Apple Events for Multiple Applications

You can open multiple Terminal windows, turn on debugging output in each, and debug your own application and other applications that it sends Apple events to or receives Apple events from at the same time.

Setting Breakpoints and Printing Apple Events in GDB

The GDB debugger provides a `call` command that you can use to call routines such as Apple Event Manager functions. The section "Using AEPrint* with GDB" in Technical Note TN2106, AEBuild*, AEPrint*, and Friends, shows how you can set breakpoints in GDB and print out the contents of Apple events in a readable format. While this approach is a little more complex, it provides a good example of setting breakpoints on Apple Event Manager routines and examining Apple events.

Third Party Options

There are a number of third-party tools, some of them quite powerful, for monitoring and debugging Apple events and scriptable applications. You can find some of them listed at AppleScript Resources.

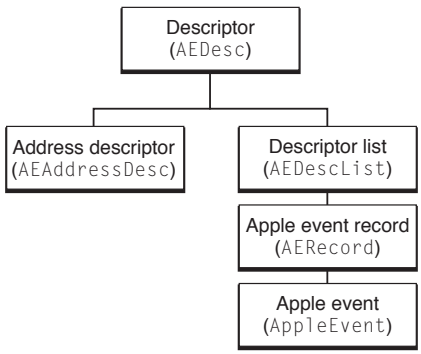
Selected Apple Event Manager Functions

This appendix provides a table of functions that you use to work with the data in Apple event data structures. For complete descriptions of these functions (and the underlying data types), see *Apple Event Manager Reference* .

Functions for Working With Apple Event Data Structures

Figure A-1 shows the hierarchy for key Apple event data structures. An Apple Event Manager function that operates on one of these data structures can also operate on any type that inherits from it. For example, the same function that works on a descriptor (`AEDesc`) also works on a descriptor list (`AEDescList`).

Figure A-1 Hierarchy of Apple event data structures



However, if there is a specific function for working with a data type, you should use that function. For example, because `AppleEvent` inherits from `AEDesc`, it is theoretically possible to create an Apple event with the `AECreateDesc` function (assuming you have access to the raw data for the Apple event). However, you should instead use the `AEBuild` function or the `AECreateAppleEvent` function, which are designed specifically for creating Apple events. For information on using these functions, see "Two Approaches to Creating an Apple Event" (page 25).

Table A-1 shows some of the functions the Apple Event Manager provides for working with Apple event data structures. Functions listed for structure types higher in the table also work for types lower in the table, though as noted, you should generally use the most specific function available for a type. For structures that are stored by key value (such as Apple event attributes and parameters), you typically use a function that accesses data by key, rather than by index.

Some functions have both a pointer and a descriptor version (for example, `AECoerceDesc` and `AECoercePtr`). The pointer version works with data in a pointed to buffer, while the descriptor version works with data that is in a descriptor.

Table A-1 Functions for working with Apple event data structures

Data type	Operation	Function
Descriptor (`AEdesc`)	create	`AECreateDesc`, `AEBuildDesc`
	dispose of	`AEDisposeDesc`
	get data	`AEGetDescData`, `AEGetDescDataSize`, `AEGetDescDataRange` (valid for `AEDesc` types only)
	set data	`AEReplaceDescData`
	coerce	`AECoerceDesc`, `AECoercePtr`
Descriptor list (`AEDescList`)	create	`AECreateList`
	count	`AECountItems`
	get by index (base 1)	`AEGetNthDesc`, `AEGetNthPtr`
	delete by index (base 1)	`AEDeleteItem`
	put	`AEPutDesc`, `AEPutPtr`, `AEBuildDesc`
Apple event record (`AERecord`)	get by key	`AEGetKeyDesc`, `AEGetKeyPtr`
	delete by key	`AEDeleteKeyDesc`
	put by key	`AEPutKeyDesc`, `AEPutKeyPtr`
Apple event (`AppleEvent`)	create	`AECreateAppleEvent`, `AEBuildAppleEvent`
	get by parameter	`AEGetParamDesc`, `AEGetParamPtr`
	get by attribute	`AEGetAttributeDesc`, `AEGetAttributePtr`

Data type	Operation	Function
	delete by parameter	`AEDeleteParam`
	put by parameter	`AEPutParamDesc`, `AEPutParamPtr`, `AEBuildParameters`
	put by attribute	`AEPutAttributeDesc`, `AEPutAttributePtr`, `AEBuildParameters`

For the following functions, you can specify the descriptor type of the resulting data; if the actual descriptor type of the attribute or parameter is different from the specified type, the Apple Event Manager attempts to coerce it to the specified type:

- `AEGetParamPtr`
- `AEGetParamDesc`
- `AEGetAttributePtr`
- `AEGetAttributeDesc`
- `AEGetNthPtr`
- `AEGetNthDesc`

Selected Apple Event Constants

This appendix describes some commonly used Apple event constants. For more complete documentation, see *Apple Event Manager Reference*.

Event Class Constants

Table B-1 lists several event class constants. For related information, see "Event Class and Event ID" (page 20).

Table B-1 Examples of Apple event class constants

Event class	Value	Description
`kCoreEventClass`	`'aevt'`	An Apple event sent by the Mac OS that your application should support if appropriate (sometimes called "required" events)
`kAECoreSuite`	`'core'`	A core Apple event; events with this type or with the `kCoreEventClass` type make up the "standard" events—basic events that scriptable applications should support if applicable
`kAEFinderSuite`	`'fndr'`	An event that the Finder accepts
`kAEFinderEvents`	`'FNDR'`	Deprecated constant for Finder event
`kAERPCClass`	`'rpc '`	Remote procedure call event
`kAETextSuite`	`'TEXT'`	Text suite event
`kFASuiteCode`	`'faco'`	Folder actions event
`kAEInternetSuite`	`'gurl'`	Internet suite event
`kAETableSuite`	`'tbls'`	Table suite event

Event ID Constants for Apple Events Sent by the Mac OS

Table B-2 shows event IDs for various Apple events that may be sent to your application by the Mac OS. For more information, see "Handling Apple Events Sent by the Mac OS" (page 58). These events are sometimes referred to as the "required" events, and have the event class value `'aevt'`, defined by the constant `kCoreEventClass`. For related information, see "Event Class and Event ID" (page 20).

Table B-2 Event ID constants for Apple events sent by the Mac OS

Event ID	Value	Description
`kAEOpenApplication`	`'oapp'`	Sent when a user opens your application without opening or printing any documents.
`kAEReopenApplication`	`'rapp'`	Sent when the application is reopened.
`kAEOpenDocuments`	`'odoc'`	Sent with a list of documents to be opened.
`kAEPrintDocuments`	`'pdoc'`	Sent with a list of documents to be printed.
`kAEOpenContents`	`'ocon'`	Sent with content to be displayed (such as when dragged content is dropped on an application icon in the Dock).
`kAEQuitApplication`	`'quit'`	Sent when the application is quitting.
`kAEShowPreferences`	`'pref'`	Sent when the user chooses the Preferences menu item.
`kAEApplicationDied`	`'obit'`	Sent to an application that launched another application when the launched application quits or terminates.

Event ID Constants for Standard AppleScript Commands

Table B-3 shows event IDs for Apple events that represent various standard AppleScript commands. Each scriptable application should support as many of these commands as make sense for that particular application. These events are sometimes referred to as "standard" or "core" events, and have the event class value `'core'`, defined by the constant `kAECoreSuite`. For related information, see "Event Class and Event ID" (page 20).

Table B-3 Event ID constants for Apple events for AppleScript commands

Event ID	Value	Description
`kAEClone`	`'clon'`	Duplicate the specified AppleScript object or objects.

Event ID	Value	Description
kAEClose	'clos'	Close the specified object or objects, usually consisting of windows or documents.
kAECountElements	'cnte'	Return the number of objects of a particular class contained by the specified object or objects.
kAECreateElement	'crel'	Create a new object.
kAEDelete	'delo'	Delete the specified object or objects.
kAEDoObjectsExist	'doex'	Return a boolean value indicating whether the specified object or objects exist.
kAEGetData	'getd'	Return the specified data from an object or set of objects.
kAEMove	'move'	Move an object or set of objects.
kAESave	'save'	Save an object or objects, often consisting of windows or documents.
kAESetData	'setd'	Set the data of an object or objects.

Descriptor Type Constants

In a descriptor, the `descriptorType` structure member stores a value that is a four-character code. Table B-4 lists constants for some of the main descriptor types, along with their four-character code values and a description of the kinds of data they identify. For a complete list of the basic descriptor types, see *Apple Event Manager Reference*.

Table B-4 Common descriptor type constants

Descriptor type	Value	Description of data
typeBoolean	'bool'	1-byte Boolean value
typeSInt32	'long'	32-bit integer
typeUTF8Text	'utf8'	Unicode text (UTF-8 encoding)
typeUTF16ExternalRepresentation	'ut16'	Unicode text (UTF-16 encoding)
typeSInt16	'shor'	16-bit integer

Descriptor type	Value	Description of data
typeAEList	'list'	List of descriptors
typeAERecord	'reco'	List of keyword-specified descriptors
typeAppleEvent	'aevt'	Apple event
typeEnumerated	'enum'	Enumerated data
typeType	'type'	Four-character code
typeFSRef	'fsrf'	File-system reference
typeNull	'null'	Nonexistent data

Address Descriptor Type Constants

The descriptor type in an address descriptor can be specified by one of the type constants shown in Table B-5 (or by any type you define for which you provide a coercion to one of these types):

Table B-5 Descriptor type constants for address descriptors

Descriptor type	Value	Description
typeApplSignature	'sign'	Application signature
typeTargetID	'targ'	Deprecated; do not use in Mac OS X
typeProcessSerialNumber	'psn '	Process serial number
typeKernelProcessID	'kpid'	Kernel process ID
typeApplicationBundleID	'bund'	Application bundle ID (Mac OS X version 10.3 and later)
typeApplicationURL	'aprl'	Application URL, possibly for a remote application (Mac OS X version 10.1 and later)
typeMachPort	'port'	Mach port.

For information on how to create a target descriptor, see "Specifying a Target Address" (page 64).

Attribute Keyword Constants

Your application cannot examine the attributes and parameters of an assembled Apple event directly. Instead, it calls Apple Event Manager functions to request an attribute or parameter by keyword. Keywords are arbitrary values used to keep track of various descriptors.

See "Keyword Parameter Constants" in *Apple Event Manager Reference* for descriptions of keyword constants for additional Apple event parameters.

Table B-6 lists keyword constants for Apple event attributes:

Table B-6 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 whether to allow bringing a server application to the foreground, if necessary, to interact with the user
`keyOriginalAddressAttr`	`'from'`	Address of original source of Apple event if the event has been forwarded
`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

Parameter Keyword Constants

Table B-7 lists keyword constants for commonly used Apple event parameters:

Table B-7 Keyword constants for common Apple event parameters

Parameter keyword	Value	Description
`keyDirectObject`	`'----'`	Direct parameter
`keyErrorNumber`	`'errn'`	Error number parameter (used only in reply events)
`keyErrorString`	`'errs'`	Error string parameter (used only in reply events)
`keyProcessSerialNumber`	`'psn '`	Process serial number
`keyPreDispatch`	`'phac'`	Dispatch event to predispatch handler
`keyAEVersion`	`'vers'`	AppleScript version

See "Keyword Parameter Constants" in *Apple Event Manager Reference* for descriptions of keyword constants for additional Apple event parameters.

Event Source Constants

You use the constants in Table B-8 to check the value of the `keyEventSourceAttr` attribute, which specifies the source of an Apple event. Listing 4-3 (page 43) shows how to obtain that attribute from an Apple event.

Table B-8 Event source type constants

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

Send Mode Constants for the AESend Function

When you send an Apple event with `AESend`, you use one of the constants in Table B-9 in the `sendMode` parameter to specify how to deal with replies.

Table B-9 Send mode constants for 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.

Default Coercion Handlers

This appendix lists the type conversions performed by the default coercion handlers provided by the Mac OS. These handlers may be implemented by the Apple Event Manager, the Open Scripting framework, or other frameworks.

Support for the following coercions was added in Mac OS X version 10.4:

- Between these types:

  `typeStyledText`, `typeUnicodeText`, `typeUTF8Text`, and `typeUTF16ExternalRepresentation`

  and these types:

  `typeType`, `typeEnumerated`, and the numeric types `typeSInt16`, `typeSInt32`, `typeUInt32`, `typeSInt64`, `typeIEEE32BitFloatingPoint`, `typeIEEE64BitFloatingPoint`, and `typeExtended`.

  > **Important:** See the description for the `typePixelMap` enum in *Apple Event Manager Reference* for important information about restrictions on coercions involving typeStyledText.

  See table Table C-1 for a mapping between the preferred and non-preferred numeric constant names.

- From `typeChar` to `typeStyledText`.

  > **Note:** Starting in Mac OS X version 10.3, `typeChar` is deprecated in favor of one of the Unicode string types. For details, see the descriptions for `typeChar` and for the Unicode string types (`typeUnicodeText`) in *Apple Event Manager Reference*.

Coercion Handler Tables

Table C-1 shows some older, non-preferred numeric constant names and their preferred equivalents. To find available coercions for a non-preferred name from Table C-1, look up its equivalent preferred name in Table C-2.

Table C-1 Older, non-preferred numeric types and their preferred types

Non-preferred numeric constant name	Preferred numeric constant name
typeSMInt, typeShortInteger	typeSInt16
typeInteger, typeLongInteger	typeSInt32
typeMagnitude	typeUInt32
typeComp	typeSInt64
typeSMFloat, typeShortFloat	typeIEEE32BitFloatingPoint
typeFloat, typeLongFloat	typeIEEE64BitFloatingPoint

Table C-2 lists the default coercions handled by the Apple Event Manager.

Table C-2 Default coercions provided by the Apple Event Manager

Coerce from descriptor type	To descriptor type	Comment
typeChar typeStyledText typeUnicodeText typeUTF8Text typeUTF16External-Representation	typeSInt16 typeSInt32 typeSInt64 typeIEEE32Bit-FloatingPoint typeIEEE64Bit-FloatingPoint typeExtended	Any string that is a valid representation of a number can be coerced into an equivalent numeric value.
typeSInt16 typeSInt32 typeSInt64 typeIEEE32Bit-FloatingPoint typeIEEE64Bit-FloatingPoint typeExtended	typeChar typeStyledText typeUnicodeText typeUTF8Text typeUTF16External-Representation	Any numeric descriptor type can be coerced into the equivalent text string.

Coerce from descriptor type	To descriptor type	Comment
typeSInt16 typeSInt32 typeSInt64 typeIEEE32Bit-FloatingPoint typeIEEE64Bit-FloatingPoint typeExtended	typeSInt16 typeSInt32 typeSInt64 typeIEEE32Bit-FloatingPoint typeIEEE64Bit-FloatingPoint typeExtended	Any numeric descriptor type can be coerced into any other numeric descriptor type.
typeChar typeIntlText typeStyledText typeUnicodeText typeUTF8Text typeUTF16External-Representation	typeChar typeIntlText typeStyledText typeUnicodeText typeUTF8Text typeUTF16External-Representation	If the destination encoding cannot represent a character in the source text, the result is undefined.
typeChar typeStyledText typeUnicodeText typeUTF8Text typeUTF16External-Representation	typeType typeEnumerated typeKeyword typeProperty	Any four-character string can be coerced to one of these descriptor types.
typeEnumerated typeKeyword typeProperty typeType	typeChar typeStyledText typeUnicodeText typeUTF8Text typeUTF16External-Representation	Any of these descriptor types can be coerced to the equivalent text string.

Coerce from descriptor type	To descriptor type	Comment
typeIntlText	typeChar typeStyledText typeUnicodeText typeUTF8Text typeUTF16External-Representation	The result contains text only, without the script code or language code from the original descriptor.
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'.
typeSInt16	typeBoolean	A value of 1 becomes the Boolean value true. A value of 0 becomes the Boolean value false.
typeBoolean	typeSInt16	A value of false becomes 0. A value of true becomes 1.
typeAlias	typeFSS	An alias is coerced into a file system specification. Not recommended—use typeFSRef instead.
typeAlias	typeFSRef	An alias is coerced into a file system reference.
typeAppleEvent	typeAppParameters	An Apple event is coerced into a list of application parameters for the LaunchParamBlockRec parameter block.
any descriptor type	typeAEList	A descriptor is coerced into a descriptor list containing a single item.

Coerce from descriptor type	To descriptor type	Comment
`typeAEList`	type of list item	A descriptor list containing a single descriptor is coerced into a descriptor.

Document Revision History

This table describes the changes to *Apple Events Programming Guide*.

Date	Notes
2007-10-31	Updated document links.
2007-07-11	Added information on handling open application Apple event. For details, see "Common Apple Events Sent by the Mac OS" (page 58).
2006-04-04	Clarified information on event class and event ID and removed extraneous entries from coercions table. Moved section "Event Class and Event ID" (page 20) to "Building an Apple Event" (page 18). Removed instances of non-preferred types (`typeSMInt` and `typeShortInteger`) from Table C-2 (page 103). For equivalent coercions, look for `typeSInt16` in that table. Added the section "Event ID Constants for Standard AppleScript Commands" (page 96). Added the section "Apple Event Terminology" (page 13), which briefly describes terminology formats and points to additional documentation. In "Specifying a Reply in the Send Mode Parameter" (page 74), added detail on the behavior of the `AESendMessage` function with respect to timers and run loops.
2005-04-29	Fixed links to other documents and made corrections to the index. New document that describes how to work with Apple events. It replaces several Apple event chapters in "Inside Macintosh: Interapplication Communication."

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010