Responding to Apple Events

This chapter describes how your application can use the Apple Event Manager to respond to Apple events. Your application must be able to respond to the four required Apple events to take advantage of the launching and terminating mechanisms that are part of System 7 and later versions of system software. If your application provides publish and subscribe capabilities, it should also handle the events sent by the Edition Manager. To be scriptable, or capable of responding to Apple events sent by scripting components, your application should handle the appropriate core and functional-area Apple events.

Before you read this chapter, you should be familiar with the chapters "Introduction to Interapplication Communication" and "Introduction to Apple Events" in this book. You should also have a copy of the *Apple Event Registry: Standard Suites* available for reference.

Although the Apple events used by the Edition Manager are discussed in this chapter, you must refer to the chapter "Edition Manager" in this book for a full discussion of how to implement the Edition Manager's publish and subscribe features.

This chapter provides the basic information you need to make your application capable of responding to Apple events. To respond to core and functional-area Apple events, your application must also be able to resolve object specifier records. You should read the chapter "Resolving and Creating Object Specifier Records" before you write Apple event handlers for events that can contain object specifier records.

The section "Handling Apple Events," which begins on page 4-4, describes how to

- accept and process Apple events
- install entries in the Apple event dispatch tables
- handle the required events
- handle events sent by the Edition Manager
- get data out of an Apple event
- write handlers that perform the action requested by an Apple event
- reply to an Apple event
- dispose of Apple event data structures
- write and install coercion handlers

The section "Interacting With the User," which begins on page 4-45, describes

- how a server application can interact with the user when processing an Apple event
- how client applications set user interaction preferences
- how the client application's preferences and the server application's preferences affect user interaction

Handling Apple Events

You do not need to implement all Apple events at once. If you want to begin by supporting only the required Apple events, you must

- set bits in the `'SIZE'` resource to indicate that your application supports high-level events
- include code to handle high-level events in your main event loop
- write routines that handle the required events
- install entries for the required Apple events in your application's Apple event dispatch table

The following sections explain how to perform these tasks: "Accepting an Apple Event," which begins on page 4-5, "Installing Entries in the Apple Event Dispatch Tables," which begins on page 4-7, and "Handling the Required Apple Events," which begins on page 4-11.

To respond to the Apple events sent by the Edition Manager in addition to the required events, you must install entries for the Section Read, Section Write, Section Scroll, and Create Publisher events in your application's Apple event dispatch table and write the corresponding handlers, as described in "Handling Apple Events Sent by the Edition Manager" on page 4-20.

To respond to core and functional-area Apple events, you must install entries and write handlers for those events. You must also make sure that your application can locate Apple event objects with the aid of the Apple Event Manager routines described in the chapter "Resolving and Creating Object Specifier Records." These routines are currently available as the Object Support Library (OSL), which you must link with your application when you build it.

The Apple Event Manager (excluding the OSL) is available only in System 7 and later versions of system software. Use the `Gestalt` function with the `gestaltAppleEventsAttr` selector to determine whether the Apple Event Manager is available. In the `response` parameter, the bit defined by the constant `gestaltAppleEventsPresent` is set if the Apple Event Manager is available.

```
CONST gestaltAppleEventsAttr  = 'evnt';   {Gestalt selector}
      gestaltAppleEventsPresent = 0;      {if this bit is set, }
                                          { then the Apple Event }
                                          { Manager is available}
```

To find out which version of the Apple Event Manager is available, you can use the `AEManagerInfo` function; for more information, see page 4-104.

Accepting an Apple Event

To accept or send Apple events (or any other high-level events), you must set the appropriate flags in your application's `'SIZE'` resource and include code to handle high-level events in your application's main event loop.

Two flags in the `'SIZE'` resource determine whether an application receives high-level events:

- The `isHighLevelEventAware` flag must be set for your application to receive any high-level events.
- The `localAndRemoteHLEvents` flag must be set for your application to receive high-level events sent from another computer on the network.

Note that in order for your application to respond to Apple events sent from remote computers, the user of your application must also allow network users to link to your application. The user does this by selecting your application in the Finder, choosing Sharing from the File menu, and clicking the Allow Remote Program Linking checkbox. If the user has not yet started program linking, the Sharing command offers to display the Sharing Setup control panel so that the user can start program linking. The user must also authorize remote users for program linking by using the Users & Groups control panel. Program linking and setting up authenticated sessions are described in the chapter "Program-to-Program Communications Toolbox" in this book.

For a complete description of the `'SIZE'` resource, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

Apple events (and other high-level events) are identified by a message class of `kHighLevelEvent` in the `what` field of the event record. You can test the `what` field of the event record to determine whether the event is a high-level event.

Listing 4-1 is an example of a procedure called from an application's main event loop to handle events, including high-level events. The procedure determines the type of event received and then calls another routine to take the appropriate action.

Listing 4-1 A `DoEvent` procedure

```
PROCEDURE DoEvent (event: EventRecord);
BEGIN
   CASE event.what OF         {determine the type of event}
      mouseDown:
         DoMouseDown (event);
      .
      . {handle other kinds of events}
      .
      {handle high-level events, including Apple events}
      kHighLevelEvent: DoHighLevelEvent (event);
   END;
END;
```

Listing 4-2 is an example of a procedure that handles both Apple events and the high-level event identified by the event class `mySpecialHLEventClass` and the event ID `mySpecialHLEventID`. Note that, in most cases, you should use Apple events to communicate with other applications.

Listing 4-2 A `DoHighLevelEvent` procedure for handling Apple events and other high-level events

```
PROCEDURE DoHighLevelEvent (event: EventRecord);
VAR
   myErr: OSErr;
BEGIN
   IF (event.message = LongInt(mySpecialHLEventClass)) AND
      (LongInt(event.where) = LongInt(mySpecialHLEventID))
   THEN
      {it's a high-level event that doesn't use AEIMP}
      myErr := HandleMySpecialHLEvent(event)
   ELSE
      {otherwise, assume that the event is an Apple event}
      myErr := AEProcessAppleEvent(event);

   {check and handle error}
   IF myErr <> noErr THEN DoError(myErr);
END;
```

If your application accepts high-level events that do not follow the Apple Event Interprocess Messaging Protocol (AEIMP), you must dispatch these high-level events before calling `AEProcessAppleEvent`. To dispatch high-level events that do not follow AEIMP, you should check the event class, the event ID, or both for each event to see whether your application can handle the event.

After receiving a high-level event (and, if appropriate, checking whether it is a high-level event other than an Apple event), your application typically calls the `AEProcessAppleEvent` function. The `AEProcessAppleEvent` function determines the type of Apple event received, gets the event buffer that contains the parameters and attributes of the Apple event, and calls the corresponding Apple event handler in your application.

You should provide an Apple event handler for each Apple event that your application supports. This handler is responsible for performing the action requested by the Apple event and if necessary can return data in the reply Apple event.

If the client application requests a reply, the Apple Event Manager passes a default reply Apple event to your handler. If the client application does not request a reply, the Apple Event Manager passes a null descriptor record (a descriptor record of descriptor type `typeNull` and a data handle whose value is `NIL`) to your handler instead of a default reply Apple event.

After your handler finishes processing the Apple event and adds any parameters to the reply Apple event, it must return a result code to `AEProcessAppleEvent`. If the client application is waiting for a reply, the Apple Event Manager returns the reply Apple event to the client.

Installing Entries in the Apple Event Dispatch Tables

When your application receives an Apple event, use the `AEProcessAppleEvent` function to retrieve the data buffer of the event and to route the Apple event to the appropriate Apple event handler in your application. Your application supplies an Apple event dispatch table to map the Apple events your application supports to the Apple event handlers provided by your application.

To install entries in your application's Apple event dispatch table, use the `AEInstallEventHandler` function. You usually install entries for all of the Apple events that your application accepts into your application's Apple event dispatch table.

To install an Apple event handler in your Apple event dispatch table, you must specify

- the event class of the Apple event
- the event ID of the Apple event
- the address of the Apple event handler for the Apple event
- a reference constant

You provide this information to the `AEInstallEventHandler` function. In addition, you indicate whether the entry should be added to your application's Apple event dispatch table or to the system Apple event dispatch table.

The ***system Apple event dispatch table*** is a table in the system heap that contains system Apple event handlers—handlers that are available to all applications and processes running on the same computer. The handlers in your application's Apple event dispatch table are available only to your application. If `AEProcessAppleEvent` cannot find a handler for the Apple event in your application's Apple event dispatch table, it looks in the system Apple event dispatch table for a handler (see "How Apple Event Dispatching Works" on page 4-9 for details). If it doesn't find a handler for the event, it returns the `errAEEventNotHandled` result code.

If you add a handler to the system Apple event dispatch table, the handler should reside in the system heap. If there was already an entry in the system Apple event dispatch table for the same event class and event ID, it is replaced unless you chain it to your system handler. See "Creating and Managing the Apple Event Dispatch Tables" on page 4-61 for details.

Installing Entries for the Required Apple Events

Listing 4-3 illustrates how to add entries for the required Apple events to your application's Apple event dispatch table.

Listing 4-3 Adding entries for the required Apple events to an application's Apple event dispatch table

```
myErr := AEInstallEventHandler(kCoreEventClass,
                               kAEOpenApplication,
                               @MyHandleOApp, 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallEventHandler(kCoreEventClass,
                               kAEOpenDocuments,
                               @MyHandleODoc,0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallEventHandler(kCoreEventClass,
                               kAEPrintDocuments,
                               @MyHandlePDoc, 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallEventHandler(kCoreEventClass,
                               kAEQuitApplication,
                               @MyHandleQuit, 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

The code in Listing 4-3 creates entries for all four required Apple events in the Apple event dispatch table. (For examples of handlers that correspond to these entries, see "Handling the Required Apple Events," which begins on page 4-11.) The first entry creates an entry for the Open Application event. The entry indicates the event class and event ID of the Open Application event, supplies the address of the handler for that event, and specifies 0 as the reference constant.

The Apple Event Manager passes the reference constant to your handler each time your handler is called. Your application can use this reference constant for any purpose. If your application doesn't use the reference constant, use 0 as the value.

The last parameter to the `AEInstallEventHandler` function is a Boolean value that determines whether the entry is added to the system Apple event dispatch table or to your application's Apple event dispatch table. To add the entry to your application's Apple event dispatch table, use `FALSE` as the value of this parameter. If you specify `TRUE`, the entry is added to the system Apple event dispatch table. The code shown in Listing 4-3 adds entries to the application's Apple event dispatch table.

Installing Entries for Apple Events Sent by the Edition Manager

If your application supports the Edition Manager, you should also add entries to your application's Apple event dispatch table for the Apple events that your application receives from the Edition Manager. Listing 4-4 shows how to add these entries.

Listing 4-4 Adding entries for Apple events sent by the Edition Manager to an application's Apple event dispatch table

```
myErr := AEInstallEventHandler(sectionEventMsgClass,
                               sectionReadMsgID,
                               @MyHandleSectionReadEvent,
                               0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallEventHandler(sectionEventMsgClass,
                               sectionWriteMsgID,
                               @MyHandleSectionWriteEvent,
                               0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallEventHandler(sectionEventMsgClass,
                               sectionScrollMsgID,
                               @MyHandleSectionScrollEvent,
                               0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

See "Handling Apple Events Sent by the Edition Manager" on page 4-20 for the parameters associated with these events. See the chapter "Edition Manager" in this book for information on how your application should respond to the Apple events sent by the Edition Manager.

How Apple Event Dispatching Works

In addition to the Apple event handler dispatch tables, applications can add entries to a ***special handler dispatch table*** in either the application heap or the system heap. These dispatch tables are used for various specialized handlers; for more information, see "Creating and Managing the Special Handler Dispatch Tables," which begins on page 4-99.

When an application calls `AEProcessAppleEvent`, the function looks first in the application's special handler dispatch table for an entry that was installed with the constant `keyPreDispatch`. If the application's special handler dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the function looks in the application's Apple event dispatch table for an entry that matches the event class and event ID of the specified Apple event.

If the application's Apple event dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the `AEProcessAppleEvent` function looks in the system special handler dispatch table for an entry that was installed with the constant `keyPreDispatch`. If the system special handler dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the function looks in the system Apple event dispatch table for an entry that matches the event class and event ID of the specified Apple event.

If the system Apple event dispatch table does not include such a handler, the Apple Event Manager returns the result code `errAEEventNotHandled` to the server application and, if the client application is waiting for a reply, to the client application.

▲ **WARNING**
Before an application calls a system Apple event handler, system software has set up the A5 register for the calling application. For this reason, if you provide a system Apple event handler, it should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system handler may crash. ▲

For any entry in your Apple event dispatch table, you can specify a wildcard value for the event class, event ID, or both. You specify a wildcard by supplying the `typeWildCard` constant when installing an entry into the Apple event dispatch table. A wildcard value matches all possible values. Wildcards make it possible to supply one Apple event handler that dispatches several related Apple events.

For example, if you specify an entry with the `typeWildCard` event class and the `kAEOpenDocuments` event ID, the Apple Event Manager dispatches Apple events of any event class with an event ID of `kAEOpenDocuments` to the handler for that entry.

If you specify an entry with the `kCoreEventClass` event class and the `typeWildCard` event ID, the Apple Event Manager dispatches Apple events of the `kCoreEventClass` event class with any event ID to the handler for that entry.

If you specify an entry with the `typeWildCard` event class and the `typeWildCard` event ID, the Apple Event Manager dispatches all Apple events of any event class and any event ID to the handler for that entry.

If an Apple event dispatch table contains one entry for an event class and a specific event ID, and also contains another entry that is identical except that it specifies a wildcard value for either the event class or the event ID, the Apple Event Manager dispatches the more specific entry. For example, if an Apple event dispatch table includes one entry that specifies the event class as `kAECoreSuite` and the event ID as `kAEDelete`, and another entry that specifies the event class as `kAECoreSuite` and the event ID as `typeWildCard`, the Apple Event Manager will dispatch the Apple event handler associated with the entry that specifies the event ID as `kAEDelete`.

IMPORTANT

If your application sends Apple events to itself using a `typeProcessSerialNumber` address descriptor record with the `lowLongOfPSN` field set to `kCurrentProcess`, the Apple Event Manager jumps directly to the appropriate Apple event handler without going through the normal event-processing sequence. For this reason, your application will not appear to run more slowly when it sends Apple events to itself. For more information, see "Addressing an Apple Event for Direct Dispatching" on page 5-13. ▲

Handling the Required Apple Events

This section describes the required Apple events—the Apple events your application must support to be compatible with System 7 and later versions of system software—and the descriptor types for all parameters of the required Apple events. It also describes how to write the handlers for these events, and it provides sample code.

To support the required Apple events, you must set the necessary flags in the `'SIZE'` resource of your application, install entries in your application's Apple event dispatch table, add code to the event loop of your application to recognize high-level events, and call the `AEProcessAppleEvent` function, as described in "Accepting an Apple Event," which begins on page 4-5, and "Installing Entries in the Apple Event Dispatch Tables," which begins on page 4-7. You must also write handlers for each Apple event; this section describes how to write these handlers.

Required Apple Events

When a user opens or prints a file from the Finder, the Finder sets up the information your application uses to determine which files to open or print. In System 7 and later versions, if your application supports high-level events, the Finder communicates this information to your application through the required Apple events.

The Finder sends these required Apple events to your application to request the corresponding actions:

Apple event	Requested action
Open Application	Perform tasks your application normally performs when a user opens your application without opening or printing any documents
Open Documents	Open the specified documents
Print Documents	Print the specified documents
Quit Application	Perform tasks—such as releasing memory, requesting the user to save documents, and so on—associated with quitting before the Finder terminates your application

In System 7 and later versions, the Finder uses these events as part of the mechanisms for launching and terminating applications. When the Finder launches your application, the application receives the Open Application, Open Documents, or Print Documents event. When the Finder terminates your application, the application receives the Quit Application event. This method of communicating Finder information to your application replaces the mechanisms used in earlier versions of system software.

Applications that do not support high-level events can still use the `CountAppFiles`, `GetAppFiles`, and `ClrAppFiles` procedures (or the `GetAppParms` procedure) to get the Finder information. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for information on these routines. To make your application compatible with System 7 and with earlier and later versions, you must support both the old and new mechanisms.

Use the `Gestalt` function to determine whether the Apple Event Manager is present. If it is and the `isHighLevelEventAware` flag is set in your application's `'SIZE'` resource, your application receives the Finder information through the required Apple events.

If your application accepts high-level events, it must be able to process the four required Apple events. Your application receives the required Apple events from the Finder in these situations:

- If your application is not open and the user opens your application from the Finder without opening or printing any documents, the Finder launches your application and sends it the Open Application event.
- If your application is not open and the user opens one of your application's documents from the Finder, the Finder launches your application and sends it the Open Documents event.
- If your application is not open and the user prints one of your application's documents from the Finder, the Finder launches your application and sends it the Print Documents event. Your application should print the selected documents and remain open until it receives a Quit Application event from the Finder.
- If your application is open and the user opens or prints any of your application's documents from the Finder, the Finder sends your application the Open Documents or Print Documents event.
- If your application is open and the user chooses Restart or Shut Down from the Finder's Special menu, the Finder sends your application the Quit Application event.

Upon receiving any of the required Apple events, your application should perform the action requested by the event. Here is a summary of the contents of the required events and the actions they request applications to perform:

Open Application—perform tasks associated with opening an application

Event class	`kCoreEventClass`
Event ID	`kAEOpenApplication`
Parameters	None
Requested action	Perform any tasks—such as opening an untitled document window—that you would normally perform when a user opens your application without opening or printing any documents.

Open Documents—open the specified documents

Event class	`kCoreEventClass`
Event ID	`kAEOpenDocuments`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeAEList`
Data:	A list of alias records for the documents to be opened
Requested action	Open the documents specified in the `keyDirectObject` parameter.

Print Documents—print the specified documents

Event class	`kCoreEventClass`
Event ID	`kAEPrintDocuments`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeAEList`
Data:	A list of alias records for the documents to be printed
Requested action	Print the documents specified in the `keyDirectObject` parameter without opening windows for the documents.

Quit Application—perform tasks associated with quitting

Event class	`kCoreEventClass`
Event ID	`kAEQuitApplication`
Parameters	None

Quit Application—perform tasks associated with quitting (continued)

Requested action	Perform any tasks that your application would normally perform when the user chooses Quit. Such tasks typically include asking the user whether to save documents that have been changed. When appropriate, the Finder sends this event to an application immediately after sending it a Print Documents event (unless the application was already open) or if the user chooses Restart or Shut Down from the Finder's Special menu.

Your application needs to recognize only two descriptor types to handle the required Apple events: descriptor lists and alias records. The Open Documents event and Print Documents event use descriptor lists to store a list of documents to open. Each document is specified as an alias record in the descriptor list.

You can retrieve the data that specifies the document to open as an alias record, or you can request that the Apple Event Manager coerce the alias record to a file system specification (`FSSpec`) record. The file system specification record provides a standard method of identifying files in System 7 and later versions. See *Inside Macintosh: Files* for a complete description of how to specify files using file system specification records.

Handling the Open Application Event

When the user opens your application, the Finder uses the Process Manager to launch your application. On startup, your application typically performs any needed initialization, and then begins to process events. If your application supports high-level events, and if the user opens your application without selecting any documents to open or print, your application receives the Open Application event.

To handle the Open Application event, your application should do just what the user expects it to do when it is opened. For example, your application might open a new untitled window in response to an Open Application event.

Listing 4-5 shows a handler that processes the Open Application event. This handler first calls an application-defined function called `MyGotRequiredParams`, which checks whether the Apple event contains any required parameters. If so, the handler returns an error, because by definition, the Open Application event should not contain any required parameters. Otherwise, the handler opens a new document window.

Listing 4-5 A handler for the Open Application event

```
FUNCTION MyHandleOApp (theAppleEvent, reply: AppleEvent;
                       handlerRefcon: LongInt): OSErr;
VAR
   myErr: OSErr;
BEGIN
   myErr := MyGotRequiredParams(theAppleEvent);
   IF myErr = noErr THEN
      DoNew;
   MyHandleOApp := myErr;
END;
```

For a description of the `MyGotRequiredParams` function, see Listing 4-11 on page 4-35. For information about the `reply` and `handlerRefcon` parameters for an Apple event handler, see "Writing Apple Event Handlers" on page 4-33.

Handling the Open Documents Event

To handle the Open Documents event, your application should open the documents that the Open Documents event specifies in its direct parameter. Your application extracts this information and then opens the specified documents. Listing 4-6 shows a handler for the Open Documents event.

Listing 4-6 A handler for the Open Documents event

```
FUNCTION MyHandleODoc (theAppleEvent, reply: AppleEvent;
                       handlerRefcon: LongInt): OSErr;
VAR
   myFSS:               FSSpec;
   docList:             AEDescList;
   myErr, ignoreErr:    OSErr;
   index, itemsInList:  LongInt;
   actualSize:          Size;
   keywd:               AEKeyword;
   returnedType:        DescType;
BEGIN
   {get the direct parameter--a descriptor list--and put it }
   { into docList}
   myErr := AEGetParamDesc(theAppleEvent, keyDirectObject,
                           typeAEList, docList);
```

```
   IF myErr = noErr THEN
   BEGIN
      {check for missing required parameters}
      myErr := MyGotRequiredParams(theAppleEvent);
      IF myErr = noErr THEN
      BEGIN
         {count the number of descriptor records in the list}
         myErr := AECountItems (docList, itemsInList);
         IF myErr = noErr THEN
            {now get each descriptor record from the list, }
            { coerce the returned data to an FSSpec record, and }
            { open the associated file}
            FOR index := 1 TO itemsInList DO
            BEGIN
               myErr := AEGetNthPtr(docList, index, typeFSS,
                                    keywd, returnedType, @myFSS,
                                    Sizeof(myFSS),actualSize);
               IF myErr = noErr THEN
               BEGIN
                  myErr := MyOpenFile(@myFSS);
                  IF myErr <> noErr THEN
                     ; {handle error from MyOpenFile}
               END
               ELSE
                  ; {handle error from AEGetNthPtr}
            END; {of For index Do}
      END
      ELSE
         ; {handle error from MyGotRequiredParams}
      ignoreErr := AEDisposeDesc(docList);
   END
   ELSE
      ; {failed to get direct parameter, handle error}
   MyHandleODoc := myErr;
END;
```

The handler in Listing 4-6 first uses the `AEGetParamDesc` function to get the direct parameter (specified by the `keyDirectObject` keyword) out of the Apple event. The handler requests that `AEGetParamDesc` return a descriptor list in the `docList` variable. The handler then checks that it has retrieved all of the required parameters by calling the `MyGotRequiredParams` function. (See Listing 4-11 on page 4-35 for a description of this function.)

Once the handler has retrieved the descriptor list from the Apple event, it uses `AECountItems` to count the number of descriptors in the list. Using the returned number as an index, the handler can get the data of each descriptor record in the list. This handler requests that the `AEGetNthPtr` function coerce the data in the descriptor record to a file system specification record. The handler can then use the file system specification record as a parameter to its own routine for opening files.

For more information on the `AEGetParamDesc` function, see page 4-69. For more information on the `AEGetNthPtr` and `AECountItems` functions, see "Getting Data Out of a Descriptor List" on page 4-31.

After extracting the file system specification record that describes the document to open, your application can use this record to open the file. For example, in Listing 4-6, the code passes the file system specification record to its routine for opening files, the `MyOpenFile` function.

The `MyOpenFile` function should be designed so that it can be called in response to both the Open Documents event and to events generated by the user. For example, when the user chooses Open from the File menu, the code that handles the mouse-down event uses the `StandardGetFile` procedure to let the user choose a file; it then calls `MyOpenFile`, passing the file system specification record returned by `StandardGetFile`. By isolating code that performs a requested action from code that interacts with the user, you can easily adapt your application to handle Apple events that request the same action.

Note the use of the `AEDisposeDesc` function to dispose of the descriptor list when your handler no longer requires the data in it. Your handler should also return a result code.

Handling the Print Documents Event

To handle the Print Documents event, your application should extract information about the documents to be printed from the direct parameter, then print the specified documents.

If your application can interact with the user, it should open windows for the documents, display a Print dialog box for the first document, and use the settings entered by the user for the first document to print all the documents. If user interaction is not allowed, your application may either return the error `errAENoUserInteraction` or print the documents using default settings. See "Interacting With the User," which begins on page 4-45, for information about using the `AEInteractWithUser` function to interact with the user.

Note that your application can remain open after processing the Print Documents event; when appropriate, the Finder sends your application a Quit Application event immediately after sending it a Print Documents event.

The handler for the Print Documents event shown in Listing 4-7 is similar to the handler for the Open Documents event, except that it prints the documents referred to in the direct parameter.

Listing 4-7 A handler for the Print Documents event

```
FUNCTION MyHandlePDoc (theAppleEvent, reply: AppleEvent;
                       handlerRefcon: LongInt): OSErr;
VAR
   myFSS:               FSSpec;
   docList:             AEDescList;
   myErr, ignoreErr:    OSErr;
   index, itemsInList:  LongInt;
   actualSize:          Size;
   keywd:               AEKeyword;
   returnedType:        DescType;
BEGIN
   {get the direct parameter--a descriptor list--and put it }
   { into docList}
   myErr := AEGetParamDesc(theAppleEvent, keyDirectObject,
                           typeAEList, docList);
   IF myErr = noErr THEN
   BEGIN
      {check for missing required parameters}
      myErr := MyGotRequiredParams(theAppleEvent);
      IF myErr = noErr THEN
      BEGIN
         {count the number of descriptor records in the list}
         myErr := AECountItems (docList, itemsInList);
         IF myErr = noErr THEN
            {now get each descriptor record from the list, }
            { coerce the returned data to an FSSpec record, and }
            { print the associated file}
            FOR index := 1 TO itemsInList DO
            BEGIN
               myErr := AEGetNthPtr(docList, index, typeFSS,
                                    keywd, returnedType, @myFSS,
                                    Sizeof(myFSS), actualSize);
               IF myErr = noErr THEN
               BEGIN
                  myErr := MyPrintFile(@myFSS);
                  IF myErr <> noErr THEN
                     ; {handle error from MyOpenFile}
               END
               ELSE
                  ; {handle error from AEGetNthPtr}
            END; {of For index Do}
      END
```

```
      ELSE
         ; {handle error from MyGotRequiredParams}
      ignoreErr := AEDisposeDesc(docList);
   END
   ELSE
      ; {failed to get direct parameter, handle error}
   MyHandlePDoc := myErr;
END;
```

Handling the Quit Application Event

To handle the Quit Application event, your application should take any actions that are necessary before it is terminated (such as saving any open documents). Listing 4-8 shows an example of a handler for the Quit Application event.

When appropriate, the Finder sends your application a Quit Application event immediately after a Print Documents event. The Finder also sends your application a Quit Application event if the user chooses Restart or Shut Down from the Finder's Special menu.

Listing 4-8 A handler for the Quit Application event

```
FUNCTION MyHandleQuit (theAppleEvent, reply: AppleEvent;
                       handlerRefcon: LongInt): OSErr;
VAR
   myErr:         OSErr;
   userCanceled:  Boolean;
BEGIN
   {check for missing required parameters}
   myErr := MyGotRequiredParams(theAppleEvent);
   IF myErr = noErr THEN
   BEGIN
      userCanceled := MyPrepareToTerminate;
      IF userCanceled THEN
         MyHandleQuit := kUserCanceled
      ELSE
         MyHandleQuit := noErr;
   END
   ELSE
      MyHandleQuit := myErr;
END;
```

The handler in Listing 4-8 calls another function supplied by the application, the `MyPrepareToTerminate` function. This function saves the documents for any open windows and returns a Boolean value that indicates whether the user canceled the Quit operation. This is another example of isolating code for interacting with the user from the code that performs the requested action. By structuring your application in this way, you can use the same routine to respond to a user action (such as choosing the Quit command from the File menu) or to the corresponding Apple event. (For a description of the `MyGotRequiredParams` function, see "Writing Apple Event Handlers" on page 4-33.)

IMPORTANT

When your application is ready to quit, it should call the `ExitToShell` procedure from the main event loop, not from your handler for the Quit Application event. Your application should quit only after the handler returns `noErr` as its function result. ▲

Handling Apple Events Sent by the Edition Manager

If your application provides publish and subscribe capabilities, it should handle the Apple events sent by the Edition Manager in addition to the required Apple events. Your application should also handle the Create Publisher event, which is described in the "Handling the Create Publisher Event" section on page 4-22.

The Edition Manager sends your application Apple events to communicate information about the publishers and subscribers in your application's documents. Specifically, the Edition Manager uses Apple events to notify your application

- when the information in an edition is updated
- when your application needs to write the data from a publisher to an edition
- when your application should locate a particular publisher and scroll through the document to that location

The Section Read, Section Write, and Section Scroll Events

The following descriptions identify the three Apple events sent by the Edition Manager—Section Read, Section Write, and Section Scroll—and the actions they tell applications to perform.

Section Read—read information into the specified section

Event class	`SectionEventMsgClass`
Event ID	`SectionReadMsgID`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeSectionH`
Data:	A handle to the section record of the subscriber whose edition contains updated information
Requested action	Update the subscriber with the new information from the edition.

Section Write—write the specified section to an edition

Event class	`SectionEventMsgClass`
Event ID	`SectionWriteMsgID`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeSectionH`
Data:	A handle to the section record of the publisher
Requested action	Write the publisher's data to its edition.

Section Scroll—scroll through the document to the specified section

Event class	`SectionEventMsgClass`
Event ID	`SectionScrollMsgID`
Required parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeSectionH`
Data:	A handle to the section record of the publisher to scroll to
Requested action	Scroll through the document to the publisher identified by the specified section record.

See the chapter "Edition Manager" in this book for details on how your application should respond to these events.

Handling the Create Publisher Event

If your application supports publish and subscribe capabilities, it should also handle the Create Publisher event.

Create Publisher—create a publisher

Event class	`kAEMiscStdSuite`
Event ID	`kAECreatePublisher`
Required parameter	None
Optional parameter	
Keyword:	`keyDirectObject`
Descriptor type:	`typeObjectSpecifier`
Data:	An object specifier record that specifies the Apple event object or objects to publish. If this parameter is omitted, publish the current selection.
Optional parameter	
Keyword:	`keyAEEditionFileLoc`
Descriptor type:	`typeAlias`
Data:	An alias record that contains the location of the edition container to create. If this parameter is omitted, use the default edition container.
Requested action	Create a publisher for the specified data using the specified location for the edition container. If the data isn't specified, publish the current selection. If the location of the edition isn't specified, use the default location.

When your application receives the Create Publisher event, it should create a publisher and write the publisher's data to an edition. The data of the publisher, and the location and name of the edition, are defined by the Apple event. If the Create Publisher event includes a `keyDirectObject` parameter, then your application should publish the data contained in the parameter. If the `keyDirectObject` parameter is missing, then your application should publish the current selection. If the document doesn't have a current selection, your handler for the event should return a nonzero result code.

If the Create Publisher event includes a `keyAEEditionFileLoc` parameter, your application should use the location and name contained in the parameter as the default location and name of the edition. If the `keyAEEditionFileLoc` parameter is missing, your application should use the default location and name your application normally uses to specify the edition container.

Listing 4-9 shows a handler for the Create Publisher event. This handler checks for the `keyDirectObject` parameter and the `keyAEEditionFileLoc` parameter. If either of these is not specified, the handler uses default values. The handler uses the application-defined function `DoNewPublisher` to create the publisher and its edition, create a section record, and update other data structures associated with the document. See the chapter "Edition Manager" in this book for an example of the `DoNewPublisher` function.

Listing 4-9 A handler for the Create Publisher event

```
FUNCTION MyHandleCreatePublisherEvent (theAppleEvent,
                                       reply: AppleEvent;
                                       handlerRefcon: LongInt)
                                       : OSErr;
VAR
   myErr:                  OSErr;
   returnedType:           DescType;
   thePublisherDataDesc:   AEDesc;
   actualSize:             LongInt;
   promptForDialog:        Boolean;
   thisDocument:           MyDocumentInfoPtr;
   preview:                Handle;
   previewFormat:          FormatType;
   defaultLocation:        EditionContainerSpec;
BEGIN
   MyGetDocumentPtr(thisDocument);
   myErr := AEGetParamDesc(theAppleEvent, keyDirectObject,
                           typeObjectSpecifier,
                           thePublisherDataDesc);
   CASE myErr OF
      errAEDescNotFound:
      BEGIN
         {use the current selection as the publisher and set up }
         { info for later when DoNewPublisher displays preview}
         preview := MyGetPreviewForSelection(thisDocument);
         previewFormat := 'TEXT';
      END;
      noErr:
         {use the data in keyDirectObject parameter as the }
         { publisher (which is returned in the }
         { thePublisherDataDesc variable), and set up info for }
         { later when DoNewPublisher displays preview}
         MySetInfoForPreview(thePublisherDataDesc, thisDocument,
                             preview, previewFormat);
      OTHERWISE
      BEGIN
         MyHandleCreatePublisherEvent := myErr;
         Exit(MyHandleCreatePublisherEvent);
      END;
   END;
   myErr := AEDisposeDesc(thePublisherDataDesc);
```

```
   myErr := AEGetParamPtr(theAppleEvent, keyAEEditionFileLoc,
                          typeFSS, returnedType,
                          @defaultLocation.theFile,
                          SizeOf(FSSpec), actualSize);
   CASE myErr OF
      errAEDescNotFound:
         {use the default location as the edition container}
            myErr := MyGetDefaultEditionSpec(thisDocument,
                                             defaultLocation);
      noErr:
      BEGIN             {the keyAEEditionFileLoc parameter }
                        { contains a default location}
         defaultLocation.thePart := kPartsNotUsed;
         defaultLocation.theFileScript := smSystemScript;
      END;
      OTHERWISE
      BEGIN
         MyHandleCreatePublisherEvent := myErr;
         Exit(MyHandleCreatePublisherEvent);
      END;
   END;
   myErr := MyGotRequiredParams(theAppleEvent);
   IF myErr <> noErr THEN
   BEGIN
      MyHandleCreatePublisherEvent := myErr;
      Exit(MyHandleCreatePublisherEvent);
   END;
   myErr := AEInteractWithUser(kAEDefaultTimeout, gMyNotifyRecPtr,
                               @MyIdleFunction);
   IF myErr = noErr THEN promptForDialog := TRUE
                    ELSE promptForDialog := FALSE;
   myErr := DoNewPublisher(thisDocument, promptForDialog,
                           preview, previewFormat,
                           defaultLocation);
   {add keyErrorNumber and keyErrorString parameters if desired}
END;
```

Note that the `MyHandleCreatePublisherEvent` handler in Listing 4-9 uses the `AEInteractWithUser` function to determine whether user interaction is allowed. If so, the handler sets the `promptForDialog` variable to `TRUE`, indicating that the `DoNewPublisher` function should display the publisher dialog box. If not,

the handler sets the `promptForDialog` variable to `FALSE`, and the `DoNewPublisher` function does not prompt the user for the location or name of the edition. For more information about `AEInteractWithUser`, see "Interacting With the User," which begins on page 4-45.

Getting Data Out of an Apple Event

The Apple Event Manager stores the parameters and attributes of an Apple event in a format that is internal to the Apple Event Manager. You use Apple Event Manager functions to retrieve the data from an Apple event and return it to your application in a format your application can use.

Most of the functions that retrieve data from Apple event parameters and attributes are available in two forms: one that returns the desired data in a specified buffer and one that returns a descriptor record containing the same data. For example, the `AEGetParamPtr` function uses a specified buffer to return the data contained in an Apple event parameter, and the `AEGetParamDesc` function returns the descriptor record for a specified parameter.

You can also use Apple Event Manager functions to get data out of descriptor records, descriptor lists, and AE records. You use similar functions to put data into descriptor records, descriptor lists, and AE records.

When your handler receives an Apple event, you typically use the `AEGetParamPtr`, `AEGetAttributePtr`, `AEGetParamDesc`, or `AEGetAttributeDesc` function to get the data out of the Apple event.

Some Apple Event Manager functions let your application request that the data be returned using any descriptor type, even if it is different from the original descriptor type. If the original data is of a different descriptor type, the Apple Event Manager attempts to coerce the data to the requested descriptor type.

For example, the `AEGetParamPtr` function lets you specify the desired descriptor type of the resulting data as follows:

```
VAR
   theAppleEvent:    AppleEvent;
   returnedType:     DescType;
   multResult:       LongInt;
   actualSize:       Size;
   myErr:            OSErr;

myErr := AEGetParamPtr(theAppleEvent, keyMultResult,
                       typeLongInteger, returnedType,
                       @multResult, SizeOf(multResult),
                       actualSize);
```

In this example, the desired type is specified in the third parameter by the `typeLongInteger` descriptor type. This requests that the Apple Event Manager coerce the data to a long integer if it is not already of this type. To prevent coercion and ensure that the descriptor type of the result is of the same type as the original, specify `typeWildCard` for the third parameter.

The Apple Event Manager returns, in the `returnedType` parameter, the descriptor type of the resulting data. This is useful information when you specify `typeWildCard` as the desired descriptor type; you can determine the descriptor type of the resulting data by examining this parameter.

The Apple Event Manager can coerce many different types of data. For example, the Apple Event Manager can convert alias records to file system specification records, integers to Boolean data types, and characters to numeric data types, in addition to other data type conversions. For a complete list of the data types for which the Apple Event Manager provides coercion handling, see Table 4-1 on page 4-43.

To perform data coercions that the Apple Event Manager doesn't perform, you can provide your own coercion handlers. See "Writing and Installing Coercion Handlers," which begins on page 4-41, for information on providing your own coercion handlers.

Apple event parameters are keyword-specified descriptor records. You can use `AEGetParamDesc` to get the descriptor record of a parameter, or you can use `AEGetParamPtr` to get the data out of the descriptor record of a parameter. If an Apple event parameter consists of an object specifier record, you can use `AEResolve` and your own object accessor functions to resolve the object specifier record—that is, to locate the Apple event object it describes. For more information about `AEResolve` and object accessor functions, see "Writing Object Accessor Functions," which begins on page 6-28. Attributes are also keyword-specified descriptor records, and you can use similar routines to get the descriptor record of an attribute or to get the data out of an attribute.

The following sections show how to use the `AEGetParamPtr`, `AEGetAttributePtr`, `AEGetParamDesc`, or `AEGetAttributeDesc` function to get the data out of an Apple event.

Getting Data Out of an Apple Event Parameter

You can use the `AEGetParamPtr` or `AEGetParamDesc` function to get the data out of an Apple event parameter. Use the `AEGetParamPtr` function (or the `AEGetKeyPtr` function, which works the same way) to return the data contained in a parameter. Use the `AEGetParamDesc` function when you need to get the descriptor record of a parameter or to extract the descriptor list from a parameter.

For example, suppose you need to get the data out of a Section Read event. The Edition Manager sends your application a Section Read event to tell your application to read updated information from an edition into the specified subscriber. The direct parameter of the Apple event contains a handle to the section record of the subscriber. You can use the `AEGetParamPtr` function to get the data out of the Apple event.

You specify the Apple event that contains the desired parameter, the keyword of the desired parameter, the descriptor type the function should use to return the data, a buffer to store the data, and the size of this buffer as parameters to the `AEGetParamPtr` function. The `AEGetParamPtr` function returns the descriptor type of the resulting data and the actual size of the data, and it places the requested data in the specified buffer.

```
VAR
   sectionH:         SectionHandle;
   theAppleEvent:    AppleEvent;
   returnedType:     DescType;
   actualSize:       Size;
   myErr:            OSErr;

myErr := AEGetParamPtr(theAppleEvent, keyDirectObject,
                       typeSectionH, returnedType, @sectionH,
                       SizeOf(sectionH), actualSize);
```

In this example, the `keyDirectObject` keyword specifies that the `AEGetParamPtr` function should extract information from the direct parameter; `AEGetParamPtr` returns the data in the buffer specified by the `sectionH` variable.

You can request that the Apple Event Manager return the data using the descriptor type of the original data or you can request that the Apple Event Manager coerce the data into a descriptor type that is different from the original. To prevent coercion, specify the desired descriptor type as `typeWildCard`.

The `typeSectionH` descriptor type specifies that the returned data should be coerced to a handle to a section record. You can use the information returned in the `sectionH` variable to identify the subscriber and read in the information from the edition.

In this example, the `AEGetParamPtr` function returns, in the `returnedType` variable, the descriptor type of the resulting data. The descriptor type of the resulting data matches the requested descriptor type unless the Apple Event Manager wasn't able to coerce the data to the specified descriptor type or you specified the desired descriptor type as `typeWildCard`. If the coercion fails, the Apple Event Manager returns the `errAECoercionFail` result code.

The `AEGetParamPtr` function returns, in the `actualSize` variable, the actual size of the data (that is, the size of coerced data, if any coercion was performed). If the value returned in this variable is greater than the amount your application allocated for the buffer to hold the returned data, your application can increase the size of its buffer to this amount, and get the data again. You can also choose to use the `AEGetParamDesc` function when your application doesn't know the size of the data.

In general, use the `AEGetParamPtr` function to extract data that is of fixed length or known maximum length, and the `AEGetParamDesc` function to extract data that is of variable length. The `AEGetParamDesc` function returns the descriptor record for an Apple event parameter. This function is useful, for example, for extracting a descriptor list from a parameter.

You specify, as parameters to `AEGetParamDesc`, the Apple event that contains the desired parameter, the keyword of the desired parameter, the descriptor type the function should use to return the descriptor record, and a buffer to store the returned descriptor record. The `AEGetParamDesc` function returns the descriptor record using the specified descriptor type.

For example, the direct parameter of the Open Documents event contains a descriptor list that specifies the documents to open. You can use the `AEGetParamDesc` function to get the descriptor list out of the direct parameter.

```
VAR
   docList:          AEDescList;
   theAppleEvent:    AppleEvent;
   myErr:            OSErr;

myErr := AEGetParamDesc(theAppleEvent, keyDirectObject,
                        typeAEList, docList);
```

In this example, the Apple event specified by the variable `theAppleEvent` contains the desired parameter. The `keyDirectObject` keyword specifies that the `AEGetParamDesc` function should get the descriptor record of the direct parameter. The `typeAEList` descriptor type specifies that the descriptor record should be returned as a descriptor list. In this example, the `AEGetParamDesc` function returns a descriptor list in the `docList` variable.

The descriptor list contains a list of descriptor records. To get the descriptor records and their data out of a descriptor list, use the `AECountItems` function to find the number of descriptor records in the list and then make repetitive calls to the `AEGetNthPtr` function to get the data out of each descriptor record. See “Getting Data Out of a Descriptor List” on page 4-31 for more information.

Note that the `AEGetParamDesc` function copies the descriptor record from the parameter. When you’re done with a descriptor record that you obtained from `AEGetParamDesc`, you must dispose of it by calling the `AEDisposeDesc` function.

If an Apple event parameter consists of an object specifier record, you can use `AEResolve` to resolve the object specifier record (that is, locate the Apple event object it describes), as explained in “Finding Apple Event Objects” on page 3-46.

Getting Data Out of an Attribute

You can use the `AEGetAttributePtr` or `AEGetAttributeDesc` function to get the data out of the attributes of an Apple event.

You specify, as parameters to `AEGetAttributePtr`, the Apple event that contains the desired attribute, the keyword of the desired attribute, the descriptor type the function should use to return the data, a buffer to store the data, and the size of this buffer. The `AEGetAttributePtr` function returns the descriptor type of the returned data and the actual size of the data and places the requested data in the specified buffer.

For example, this code gets the data out of the `keyEventSourceAttr` attribute of an Apple event.

```
VAR
   theAppleEvent:    AppleEvent;
   returnedType:     DescType;
   sourceOfAE:       Integer;
   actualSize:       Size;
   myErr:            OSErr;

myErr := AEGetAttributePtr(theAppleEvent, keyEventSourceAttr,
                           typeShortInteger, returnedType,
                           @sourceOfAE, SizeOf(sourceOfAE),
                           actualSize);
```

The `keyEventSourceAttr` keyword specifies the attribute from which to get the data. The `typeShortInteger` descriptor type specifies that the data should be returned as a short integer; the `returnedType` variable contains the actual descriptor type that is returned. You also must specify a buffer to hold the returned data and specify the size of this buffer. If the data is not already a short integer, the Apple Event Manager coerces it as necessary before returning it. The `AEGetAttributePtr` function returns, in the `actualSize` variable, the actual size of the returned data after coercion has taken place. You can check this value to make sure you got all the data.

As with the `AEGetParamPtr` function, you can request that `AEGetAttributePtr` return the data using the descriptor type of the original data, or you can request that the Apple Event Manager coerce the data into a descriptor type that is different from the original.

In this example, the `AEGetAttributePtr` function returns the requested data as a short integer in the `sourceOfAE` variable, and you can get information about the source of the Apple event by examining this value. You can test the returned value against the values defined by the data type `AEEventSource`.

```
TYPE AEEventSource = (kAEUnknownSource, kAEDirectCall,
                      kAESameProcess, kAELocalProcess,
                      kAERemoteProcess);
```

The constants defined by the data type `AEEventSource` have the following meanings:

Constant	Meaning
`kAEUnknownSource`	Source of Apple event unknown
`kAEDirectCall`	A direct call that bypassed the PPC Toolbox
`kAESameProcess`	Target application is also the source application
`kAELocalProcess`	Source application is another process on the same computer as the target application
`kAERemoteProcess`	Source application is a process on a remote computer on the network

The next example shows how to use the `AEGetAttributePtr` function to get data out of the `keyMissedKeywordAttr` attribute. After your handler extracts all known parameters from an Apple event, it should check whether the `keyMissedKeywordAttr` attribute exists. If it does, then your handler did not get all of the required parameters.

Note that if `AEGetAttributePtr` returns the `errAEDescNotFound` result code, then the `keyMissedKeywordAttr` attribute does not exist—that is, your application has extracted all of the required parameters. If `AEGetAttributePtr` returns `noErr`, then the `keyMissedKeywordAttr` attribute does exist—that is, your handler did not get all of the required parameters.

```
myErr := AEGetAttributePtr(theAppleEvent, keyMissedKeywordAttr,
                           typeWildCard, returnedType, NIL, 0,
                           actualSize);
```

The data in the `keyMissedKeywordAttr` attribute contains the keyword of the first required parameter, if any, that your handler didn't retrieve. If you want this data returned, specify a buffer to hold it and specify the buffer size. Otherwise, as in this example, specify `NIL` as the buffer and 0 as the size of the buffer.

This example shows how to use the `AEGetAttributePtr` function to get the address of the sender of an Apple event from the `keyAddressAttr` attribute of the Apple event:

```
VAR
   theAppleEvent: AppleEvent;
   returnedType:  DescType;
   addressOfAE:   TargetID;
   actualSize:    Size;
   myErr:         OSErr;

myErr := AEGetAttributePtr(theAppleEvent, keyAddressAttr,
                           typeTargetID, returnedType,
                           @addressOfAE, SizeOf(addressOfAE),
                           actualSize);
```

The `keyAddressAttr` keyword specifies the attribute to get the data from. The `typeTargetID` descriptor type specifies that the data should be returned as a target ID record; the `returnedType` variable contains the actual descriptor type that is returned. You can examine the address returned in the `addressOfAE` variable to determine the sender of the Apple event.

The target ID record returned in the `addressOfAE` variable contains the sender's port name, port location, and session reference number. To get the process serial number for a process on the local machine, pass the port name returned in the target ID record to the `GetProcessSerialNumberFromPortName` function. You can then pass the process serial number to the `GetProcessInformation` function to find the creator signature for a given process. (For more information about these functions, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.)

For more information about target addresses, see "Specifying a Target Address" on page 5-10.

Getting Data Out of a Descriptor List

You can use the `AECountItems` function to count the number of items in a descriptor list, and you can use `AEGetNthDesc` or `AEGetNthPtr` to get a descriptor record or its data out of a descriptor list.

The Open Documents event contains a direct parameter that specifies the list of documents to open. The list of documents is contained in a descriptor list. After extracting the descriptor list from the parameter, you can determine the number of items in the list and then extract each descriptor record from the descriptor list. See Figure 3-9 on page 3-19 for a depiction of the Open Documents event.

For example, when your handler receives an Open Documents event, you can use the `AEGetParamDesc` function to return the direct parameter as a descriptor list. You can then use `AECountItems` to return the number of descriptor records in the list.

```
VAR
   theAppleEvent:    AppleEvent;
   docList:          AEDescList;
   itemsInList:      LongInt;
   myErr:            OSErr;

myErr := AEGetParamDesc(theAppleEvent, keyDirectObject,
                        typeAEList, docList);
myErr := AECountItems(docList, itemsInList);
```

The `AEGetParamDesc` function returns, in the `docList` variable, a copy of the descriptor list from the direct parameter of the Open Documents event. You specify this list to the `AECountItems` function.

You specify the descriptor list whose items you want to count in the first parameter to `AECountItems`. The Apple Event Manager returns, in the second parameter, the number of items in the list. When extracting the descriptor records from a list, you often use the number of items as a loop index. Here's an example:

```
FOR index := 1 TO itemsInList DO
   BEGIN
   {for each descriptor record in the list, get its data}
   END;
```

The format of the descriptor records in a descriptor list is private to the Apple Event Manager. You must use the `AEGetNthPtr` or `AEGetNthDesc` function to extract descriptor records from a descriptor list.

You specify the descriptor list that contains the desired descriptor records and an index as parameters to the `AEGetNthPtr` function. The index represents a specific descriptor record in the descriptor list. The `AEGetNthPtr` function returns the data for the descriptor record represented by the specified index.

You also specify the descriptor type the function should use to return the data, a buffer to store the data, and the size of this buffer. If the specified descriptor record exists, the `AEGetNthPtr` function returns the keyword of the parameter, the descriptor type of the returned data, and the actual size of the data, and it places the requested data in the specified buffer.

Here's an example that uses the `AEGetNthPtr` function to extract an item from the descriptor list in the direct parameter of the Open Documents event:

```
myErr := AEGetNthPtr(docList, index, typeFSS, keywd,
                     returnedType, @myFSS, Sizeof(myFSS),
                     actualSize);
```

The `docList` variable specifies the descriptor list from the direct parameter of the Open Documents event. The `index` variable specifies the index of the descriptor record to extract. You can use the `typeFSS` descriptor type, as in this example, to specify that the data be returned as a file system specification record. The Apple Event Manager automatically coerces the original data type of the descriptor record from an alias record to a file system specification record. The `AEGetNthPtr` function returns the keyword of the parameter and the descriptor type of the resulting data in the `keywd` and `returnedType` variables, respectively.

You also specify a buffer to hold the desired data and the size (in bytes) of the buffer. In this example, the `myFSS` variable specifies the buffer. The function returns the actual size of the data in the `actualSize` variable. If this size is larger than the size of the buffer you provided, you know that you didn't get all of the data for the descriptor record.

Listing 4-10 shows a more complete example of extracting the items from a descriptor list in the Open Documents event.

Listing 4-10 Extracting items from a descriptor list

```
VAR
   index:            LongInt;
   itemsInList:      LongInt;
   docList:          AEDescList;
   keywd:            AEKeyword;
   returnedType:     DescType;
   myFSS:            FSSpec;
   actualSize:       Size;
   myErr:            OSErr;

FOR index := 1 TO itemsInList DO
   BEGIN
      myErr := AEGetNthPtr(docList, index, typeFSS, keywd,
                           returnedType, @myFSS, Sizeof(myFSS),
                           actualSize);
      IF myErr <> noErr THEN DoError(myErr);
      myErr := MyOpenFile(@myFSS);
      IF myErr <> noErr THEN DoError(myErr);
   END;
myErr := AEDisposeDesc(docList);
```

Writing Apple Event Handlers

For each Apple event your application supports, you must provide a function called an Apple event handler. The `AEProcessAppleEvent` function calls one of your Apple event handlers when it processes an Apple event. Your Apple event handlers should perform any action requested by the Apple event, add parameters to the reply Apple event if appropriate, and return a result code.

The Apple Event Manager uses dispatch tables to route Apple events to the appropriate Apple event handler. You must supply an Apple event handler for each entry in your application's Apple event dispatch table. Each handler must be a function that uses this syntax:

```
FUNCTION MyEventHandler (theAppleEvent: AppleEvent;
                         reply: AppleEvent;
                         handlerRefcon: LongInt): OSErr;
```

The parameter `theAppleEvent` is the Apple event to handle. Your handler uses Apple Event Manager functions to extract any parameters and attributes from the Apple event and then performs the necessary processing. If any of the parameters include object specifier records, your handler should call `AEResolve` to resolve them—that is, to locate the Apple event objects they describe. For more information, see the chapter "Resolving and Creating Object Specifier Records" in this book.

The `reply` parameter is the default reply provided by the Apple Event Manager. ("Replying to an Apple Event," which begins on page 4-36, describes how to add parameters to the default reply.) The `handlerRefcon` parameter is the reference constant stored in the Apple event dispatch table entry for the Apple event. Your handler can check the reference constant, if necessary, for information about the Apple event.

You can use the reference constant for anything you wish. For example, if you want to use the same handler for several Apple events, you can install entries for each event in your application's Apple event dispatch table that specify the same handler but different reference constants. Your handler can then use the reference constant to distinguish the different Apple events it handles.

To provide an Apple event handler in C, be sure to include the Pascal declaration before the handler declaration. This is the syntax for an Apple event handler in C:

```
pascal OSErr MyEventHandler (const AppleEvent *theAppleEvent,
                             const AppleEvent *reply,
                             long handlerRefcon);
```

After extracting all known parameters from the Apple event, every handler should determine whether the Apple event contains any further required parameters. Your handler can determine whether it retrieved all the required parameters by checking whether the `keyMissedKeywordAttr` attribute exists. If the attribute exists, then your handler has not retrieved all the required parameters and should immediately return an error. If the attribute does not exist, then the Apple event does not contain any more required parameters, although it may contain additional optional parameters.

The Apple Event Manager determines which parameters are optional according to the keywords listed in the `keyOptionalKeywordAttr` attribute. The source application is responsible for adding these keywords to the `keyOptionalKeywordAttr` attribute, but is not required to do so, even if that parameter is listed in the *Apple Event Registry: Standard Suites* as an optional parameter. If the source application does not add the necessary keyword to the `keyOptionalKeywordAttr` attribute, the target application treats the parameter as required for that Apple event. If the target application supports the parameter, it should handle the Apple event as the source application expects. If the target application does not support the parameter and checks whether it has received all the required parameters, it finds that there's another parameter that the client application considered required, and should return the result code `errAEParamMissed` without attempting to handle the event.

Listing 4-11 shows a function that checks for a `keyMissedKeywordAttr` attribute. A handler calls this function after getting all the required parameters it knows about from an Apple event.

Listing 4-11 A function that checks for a `keyMissedKeywordAttr` attribute

```
FUNCTION MyGotRequiredParams (theAppleEvent: AppleEvent): OSErr;
VAR
   myErr:           OSErr;
   returnedType:    DescType;
   actualSize:      Size;
BEGIN
   myErr := AEGetAttributePtr(theAppleEvent,
                              keyMissedKeywordAttr,
                              typeWildCard, returnedType,
                              NIL, 0, actualSize);
   IF myErr = errAEDescNotFound THEN
      {you got all the required parameters}
      MyGotRequiredParams := noErr
   ELSE IF myErr = noErr THEN
      {you missed a required parameter}
      MyGotRequiredParams := errAEParamMissed;
END;
```

The code in Listing 4-11 uses the `AEGetAttributePtr` function to get the `keyMissedKeywordAttr` attribute. This attribute contains the first required parameter, if any, that your handler didn't retrieve. If `AEGetAttributePtr` returns the `errAEDescNotFound` result code, the Apple event doesn't contain a `keyMissedKeywordAttr` attribute. If the Apple event doesn't contain this attribute, then your handler has extracted all of the parameters that the client application considered required.

If the `AEGetAttributePtr` function returns `noErr` as the result code, then the attribute does exist, meaning that your handler has not extracted all of the required parameters. In this case, your handler should return an error and not process the Apple event.

The first remaining required parameter is specified by the data of the `keyMissedKeywordAttr` attribute. If you want this data returned, specify a buffer to hold the data. Otherwise, specify `NIL` as the buffer and 0 as the size of the buffer. If you specify a buffer to hold the data, you can check the value of the `actualSize` parameter to see if the data is larger than the buffer you allocated.

For more information about specifying Apple event parameters as optional or required, see "Specifying Optional Parameters for an Apple Event" beginning on page 5-7.

Replying to an Apple Event

Your handler routine for a particular Apple event is responsible for performing the action requested by the Apple event, and can optionally return data in a reply Apple event. The Apple Event Manager passes a default reply Apple event to your handler. The default reply Apple event has no parameters when it is passed to your handler. Your handler can add parameters to the reply Apple event. If the client application requested a reply, the Apple Event Manager returns the reply Apple event to the client.

The reply Apple event is identified by the `kCoreEventClass` event class and by the `kAEAnswer` event ID. If the client application specified the `kAENoReply` flag in the `reply` parameter of the `AESend` function, the Apple Event Manager passes a null descriptor record (a descriptor record of type `typeNull` whose data handle has the value `NIL`) to your handler instead of a default reply Apple event. Your handler should check the descriptor type of the reply Apple event before attempting to add any attributes or parameters to it. An attempt to add an Apple event attribute or parameter to a null descriptor record generates an error.

If the client application requests a reply, the Apple Event Manager prepares a reply Apple event for the client by passing a default reply Apple event to your handler. The default reply Apple event has no parameters when it is passed to your handler. Your handler can add any parameters to the reply Apple event. If your application is a spelling checker, for example, you can return a list of misspelled words in a parameter.

When your handler finishes processing an Apple event, it returns a result code to `AEProcessAppleEvent`, which returns this result code as its function result. If your handler returns a nonzero result code, and if you have not added your own `keyErrorNumber` parameter, the Apple Event Manager also returns this result code to the client application by putting the result code into a `keyErrorNumber` parameter for the reply Apple event. The client can check for the existence of this parameter to determine whether the handler performed the requested action.

The client application specifies whether it wants a reply Apple event or not by specifying flags (represented by constants) in the `sendMode` parameter of the `AESend` function.

If the client specifies the `kAEWaitReply` flag in the `sendMode` parameter, the `AESend` function does not return until the timeout specified by the `timeoutInTicks` parameter expires or the server application returns a reply. When the server application returns a reply, the `reply` parameter to `AESend` contains the reply Apple event that your handler returned to the `AEProcessAppleEvent` function. When the client application no longer needs the original Apple event and the reply event, it must dispose of them, but the Apple Event Manager disposes of both the Apple event and the reply event for the server application when the server's handler returns to `AEProcessAppleEvent`.

If the client specified the `kAEQueueReply` flag, the client receives the reply event at a later time during its normal processing of other events.

Your handler should always set its function result to `noErr` if it successfully handles the Apple event. If an error occurs, your handler should return either `errAEEventNotHandled` or some other nonzero result code. If the error occurs because your application cannot understand the event, return `errAEEventNotHandled`. This allows the Apple Event Manager to look for a handler in the system special handler or system Apple event dispatch tables that might be able to handle the event. If the error occurs because the event is impossible to handle as specified, return the result code returned by whatever function caused the failure, or whatever other result code is appropriate.

For example, suppose your application receives a Get Data event requesting the name of the current printer, and your application cannot handle such an event. In this situation, you should return `errAEEventNotHandled` in case another handler available to the Apple Event Manager can handle the Get Data event. This strategy allows users to take advantage of system capabilities from within your application via system handlers.

However, if your application cannot handle a Get Data event that requests the fifth paragraph in a document because the document contains only four paragraphs, you should return some other nonzero error, because further attempts to handle the event are pointless.

If your Apple event handler calls the `AEResolve` function and `AEResolve` calls an object accessor function in the system object accessor dispatch table, your Apple event handler may not recognize the descriptor type of the token returned by the function. In this case, your handler should return the result code `errAEUnknownObjectType`. When your handler returns this result code, the Apple Event Manager attempts to locate a system Apple event handler that can recognize the token. For more information, see "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

The Apple Event Manager automatically adds any nonzero result code that your handler returns to a `keyErrorNumber` parameter in the reply Apple event. In addition to returning a result code, your handler can also return an error string in the `keyErrorString` parameter of the reply Apple event. Your handler should provide meaningful text in the `keyErrorString` parameter, so that the client can display this string to the user if desired.

Listing 4-12 shows how to add the `keyErrorString` parameter to the reply Apple event. See "Adding Parameters to an Apple Event" on page 5-5 for a description of the `AEPutParamPtr` function.

Listing 4-12 Adding the `keyErrorString` parameter to the reply Apple event

```
FUNCTION MyHandler (theAppleEvent: AppleEvent; reply: AppleEvent;
                    handlerRefcon: LongInt): OSErr;
VAR
   myErr:      OSErr;
   errStr:     Str255;
BEGIN
   {handle your Apple event here}

   {if an error occurs when handling an Apple event, set the }
   { function result and error string accordingly}
   IF myErr <> noErr THEN
   BEGIN
      MyHandler := myErr; {result code to be returned--the }
                          { Apple Event Manager adds this }
                          { result code to the reply Apple }
                          { event as the keyErrorNumber }
                          { parameter}
      IF (reply.dataHandle <> NIL) THEN
      {add error string parameter to the default reply}
      BEGIN
         {strings should normally be stored in resources}
         errStr := 'Why error occurred';
         myErr := AEPutParamPtr(reply, keyErrorString,
                                typeIntlText, @errStr[1],
                                length(errStr));
      END;
   END
   ELSE
      MyHandler := noErr;
END;
```

If your handler needs to return data to the client, it can add parameters to the reply Apple event. Listing 4-13 shows how a handler for the Multiply event (an imaginary Apple event that asks the server to multiply two numbers) might return the results of the multiplication to the client.

Listing 4-13 Adding parameters to the reply Apple event

```
FUNCTION MyMultHandler (theAppleEvent: AppleEvent;
                        reply: AppleEvent;
                        handlerRefcon: LongInt): OSErr;
VAR
   myErr:            OSErr;
   number1,number2:  LongInt;
   replyResult:      LongInt;
   actualSize:       Size;
   returnedType:     DescType;
BEGIN
   {get the numbers to multiply from the parameters of the }
   { Apple event; put the numbers in the number1 and number2 }
   { variables and then perform the requested multiplication}
   myErr := MyDoMultiply(theAppleEvent, number1,
                         number2, replyResult);
   IF myErr = noErr THEN
      IF (reply.dataHandle <> NIL) THEN
         {return result of the multiplication in the reply Apple }
         { event}
         myErr := AEPutParamPtr(reply, keyDirectObject,
                                typeLongInteger, @replyResult,
                                SizeOf(replyResult));
   MyMultHandler := myErr;
   {if an error occurs, set the error string }
   { accordingly, as shown in Listing 4-12}
END;
```

Disposing of Apple Event Data Structures

Whenever a client application uses Apple Event Manager functions to create a descriptor record, descriptor list, or Apple event record, the Apple Event Manager allocates memory for these data structures in the client's application heap. Likewise, when a server application extracts a descriptor record from an Apple event by using Apple Event Manager functions, the Apple Event Manager creates a copy of the descriptor record, including the data to which its handle refers, in the server's application heap.

Whenever you finish using a descriptor record or descriptor list that you have created or extracted from an Apple event, you should dispose of the descriptor record—and thereby deallocate the memory it uses—by calling the `AEDisposeDesc` function. If the descriptor record you pass to `AEDisposeDesc` (such as an Apple event record or an AE record) includes other nested descriptor records, one call to `AEDisposeDesc` will dispose of them all.

When a client application adds a descriptor record to an Apple event (for example, when it creates a descriptor record by calling `AECreateDesc` and then puts a copy of it into a parameter of an Apple event by calling `AEPutParamDesc`), it is still responsible for disposing of the original descriptor record. After a client application has finished using both the Apple event specified in the `AESend` function and the reply Apple event, it should dispose of their descriptor records by calling `AEDisposeDesc`. The client application should dispose of them even if `AESend` returns a nonzero result code.

The Apple event that a server application's handler receives is a copy of the original event created by the client application. When a server application's handler returns to `AEProcessAppleEvent`, the Apple Event Manager disposes of the server's copy (in the server's application heap) of both the Apple event and the reply event. The server application is responsible for disposing of any descriptor records created while extracting data from the Apple event or adding data to the reply event.

In general, outputs from Apple Event Manager functions are your application's responsibility. Once you finish using them, you should use `AEDisposeDesc` to dispose of any Apple event data structures created or returned by these functions:

```
AECoerceDesc          AEDuplicateDesc
AECoercePtr           AEGetAttributeDesc
AECreateAppleEvent    AEGetKeyDesc
AECreateDesc          AEGetNthDesc
AECreateList          AEGetParamDesc
```

If you attempt to dispose of descriptor records returned by successful calls to these functions without using `AEDisposeDesc`, your application may not be compatible with future versions of the Apple Event Manager. However, if any of these functions return a nonzero result code, they return a null descriptor record, which does not need to be disposed of.

Outputs from functions, such as `AEGetKeyPtr`, that use a buffer rather than a descriptor record to return data do not require the use of `AEDisposeDesc`. It is therefore preferable to use these functions for any data that is not identified by a handle.

Some of the functions described in the chapter "Resolving and Creating Object Specifier Records" in this book also create descriptor records. If you set the `disposeInputs` parameter to `FALSE` for any of the following functions, you should dispose of any Apple event data structures that they create or return:

```
CreateCompDescriptor      CreateObjSpecifier
CreateLogicalDescriptor   CreateRangeDescriptor
```

Your application is also responsible for disposing of some of the tokens it creates in the process of resolving an object specifier record. For information about token disposal, see "Defining Tokens" on page 6-39.

Writing and Installing Coercion Handlers

When your application extracts data from a parameter, it can request that the Apple Event Manager return the data using a descriptor type that is different from the original descriptor type. For example, when extracting data from the direct parameter of the Open Documents event, you can request that the alias records be returned as file system specification records. The Apple Event Manager can automatically coerce many different types of data from one to another. Table 4-1 on page 4-43 shows descriptor types and the kinds of coercion that the Apple Event Manager can perform.

You can also provide your own routines, referred to as ***coercion handlers,*** to coerce data into any other descriptor type. To install your own coercion handlers, use the `AEInstallCoercionHandler` function. You specify as parameters to this function

- the descriptor type of the data coerced by the handler
- the descriptor type of the resulting data
- the address of the coercion handler for this descriptor type
- a reference constant
- a Boolean value that indicates whether your coercion handler expects the data to be specified as a descriptor record or as a pointer to the actual data
- a Boolean value that indicates whether your coercion handler should be added to your application's coercion dispatch table or the system coercion dispatch table

The ***system coercion dispatch table*** is a table in the system heap that contains coercion handlers available to all applications and processes running on the same computer. The coercion handlers in your application's coercion dispatch table are available only to your application. When attempting to coerce data, the Apple Event Manager first looks for a coercion handler in your application's coercion dispatch table. If it cannot find a handler for the descriptor type, it looks in the system coercion dispatch table for a handler. If it doesn't find a handler there, it attempts to use the default coercion handling described by Table 4-1 on page 4-43. If it can't find an appropriate default coercion handler, it returns the `errAECoercionFail` result code.

Any handler that you add to the system coercion dispatch table should reside in the system heap. If there was already an entry in the system coercion dispatch table for the same descriptor type, it is replaced. Therefore, if there is an entry in the system coercion dispatch table for the same descriptor type, you should chain it to your system coercion handler as explained in "Creating and Managing the Coercion Handler Dispatch Tables," which begins on page 4-96.

▲ **WARNING**
Before an application calls a system coercion handler, system software has set up the A5 register for the calling application. For this reason, if you provide a system coercion handler, it should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system coercion handler may crash. ▲

You can provide a coercion handler that expects to receive the data in a descriptor record or a buffer referred to by a pointer. When you install your coercion handler, you specify how your handler wishes to receive the data. Whenever possible, you should write your coercion handler so that it can accept a pointer to the data, because it's more efficient for the Apple Event Manager to provide your coercion handler with a pointer to the data.

A coercion handler that accepts a pointer to data must be a function with the following syntax:

```
FUNCTION MyCoercePtr (typeCode: DescType; dataPtr: Ptr;
                      dataSize: Size; toType: DescType;
                      handlerRefcon: LongInt;
                      VAR result: AEDesc): OSErr;
```

The `typeCode` parameter is the descriptor type of the original data. The `dataPtr` parameter is a pointer to the data to coerce; the `dataSize` parameter is the length, in bytes, of the data. The `toType` parameter is the desired descriptor type of the resulting data. The `handlerRefcon` parameter is a reference constant stored in the coercion table entry for the handler and passed to the handler by the Apple Event Manager whenever the handler is called. The `result` parameter is the descriptor record returned by your coercion handler.

Your coercion handler should coerce the data to the desired descriptor type and return the data in the descriptor record specified by the `result` parameter. If your handler successfully performs the coercion, it should return the `noErr` result code; otherwise, it should return a nonzero result code.

A coercion handler that accepts a descriptor record must be a function with the following syntax:

```
FUNCTION MyCoerceDesc (theAEDesc: AEDesc; toType: DescType;
                       handlerRefcon: LongInt;
                       VAR result: AEDesc): OSErr;
```

The parameter `theAEDesc` is the descriptor record that contains the data to be coerced. The `toType` parameter is the descriptor type of the resulting data. The `handlerRefcon` parameter is a reference constant stored in the coercion table entry for the handler and passed to the handler by the Apple Event Manager whenever the handler is called. The `result` parameter is the resulting descriptor record.

Your coercion handler should coerce the data in the descriptor record to the desired descriptor type and return the data in the descriptor record specified by the `result` parameter. Your handler should return an appropriate result code.

Note

To ensure that no coercion is performed and that the descriptor type of the result is of the same descriptor type as the original, specify `typeWildCard` for the desired type. ◆

Table 4-1 lists the descriptor types for which the Apple Event Manager provides coercion.

Table 4-1 Coercion handling provided by the Apple Event Manager

Original descriptor type of data to be coerced	Desired descriptor type	Description

`typeChar`	`typeInteger`
	`typeLongInteger`
	`typeSMInt`
	`typeSMFloat`
	`typeShortInteger`
	`typeFloat`
	`typeLongFloat`
	`typeShortFloat`
	`typeExtended`
	`typeComp`
	`typeMagnitude`
		Any string that is a valid representation of a number can be coerced into an equivalent numeric value.

`typeInteger`
`typeLongInteger`
`typeSMInt`
`typeSMFloat`
`typeShortInteger`
`typeFloat`
`typeLongFloat`
`typeShortFloat`
`typeExtended`
`typeComp`
`typeMagnitude`
	`typeChar`	Any numeric descriptor type can be coerced into the equivalent text string.

`typeInteger`
`typeLongInteger`
`typeSMInt`
`typeSMFloat`
`typeShortInteger`
`typeFloat`
`typeLongFloat`
`typeShortFloat`
`typeExtended`
`typeComp`
`typeMagnitude`
	`typeInteger`
	`typeLongInteger`
	`typeSMInt`
	`typeSMFloat`
	`typeShortInteger`
	`typeFloat`
	`typeLongFloat`
	`typeShortFloat`
	`typeExtended`
	`typeComp`
	`typeMagnitude`
		Any numeric descriptor type can be coerced into any other numeric descriptor type.

`typeChar`	`typeType`
	`typeEnumerated`
	`typeKeyword`
	`typeProperty`
		Any four-character string can be coerced to one of these descriptor types.

`typeEnumerated`
`typeKeyword`
`typeProperty`
`typeType`
	`typeChar`	Any of these descriptor types can be coerced to the equivalent text string.

continued

Table 4-1 Coercion handling provided by the Apple Event Manager (continued)

Original descriptor type of data to be coerced	Desired descriptor type	Description
typeIntlText	typeChar	The result contains text only, without the script code or language code from the original descriptor record.
typeTrue	typeBoolean	The result is the Boolean value TRUE.
typeFalse	typeBoolean	The result is the Boolean value FALSE.
typeEnumerated	typeBoolean	The enumerated value 'true' becomes the Boolean value TRUE. The enumerated value 'fals' becomes the Boolean value FALSE.
typeBoolean	typeEnumerated	The Boolean value FALSE becomes the enumerated value 'fals'. The Boolean value TRUE becomes the enumerated value 'true'.
typeShortInteger typeSMInt	typeBoolean	A value of 1 becomes the Boolean value TRUE. A value of 0 becomes the Boolean value FALSE.
typeBoolean	typeShortInteger typeSMInt	A value of FALSE becomes 0. A value of TRUE becomes 1.
typeAlias	typeFSS	An alias record is coerced into a file system specification record.
typeAppleEvent	typeAppParameters	An Apple event is coerced into a list of application parameters for the LaunchParamBlockRec parameter block.
any descriptor type	typeAEList	A descriptor record is coerced into a descriptor list containing a single item.
typeAEList	*type of list item*	A descriptor list containing a single descriptor record is coerced into a descriptor record.

NOTE Some of the descriptor types listed in this table are synonyms; for example, the constants typeSMInt and typeShortInteger have the same four-character code, 'shor'.

Interacting With the User

When your application receives an Apple event, it may need to interact with the user. For example, it may need to display a dialog box asking the user for additional information or confirmation. You must use the `AEInteractWithUser` function to make sure your application is in the foreground before it actually interacts with the user.

Both the client application and the server application specify their preferences for user interaction. The `AEInteractWithUser` function checks the user interaction preferences set by each application. If both the client and the server allow user interaction, `AEInteractWithUser` usually posts a notification request, and the Notification Manager brings the server to the foreground after the user responds to the notification request.

The `AEInteractWithUser` function can also bring the server application directly to the foreground, but only if the client application is the active application on the same computer and has set two flags in the `sendMode` parameter of the `AESend` function: the `kAEWaitReply` flag, which indicates that it is waiting for a reply, and the `kAECanSwitchLayer` flag, which indicates that it wants the server application to come directly to the foreground rather than posting a notification request.

To specify its preferences for how the server application should interact with the user, the client application sets various flags in the `sendMode` parameter to `AESend`. The Apple Event Manager sets the corresponding flags in the `keyInteractLevelAttr` attribute of the Apple event.

The server application sets its preferences with the `AESetInteractionAllowed` function. This function lets your application specify whether it allows interaction with the user as a result of receiving an Apple event from itself; from itself and other processes on the local computer; or from itself, local processes, and processes from another computer on the network.

Your application calls the `AEInteractWithUser` function before interacting with the user. If `AEInteractWithUser` returns the `noErr` result code, then your application is currently in the front and free to interact with the user. If `AEInteractWithUser` returns the `errAENoUserInteraction` result code, the conditions didn't allow user interaction and your application should not interact with the user.

The rest of this section explains how to set user interactions for the client and server applications and the practical effect these settings have when a server needs to interact with a user.

Setting the Client Application's User Interaction Preferences

The client application sets its user interaction preferences by setting flags in the `sendMode` parameter to the `AESend` function. The Apple Event Manager automatically adds the specified flags to the `keyInteractLevelAttr` attribute of the Apple event. These flags are represented by the following constants:

Flag	Description
`kAENeverInteract`	The server application should never interact with the user in response to the Apple event. If this flag is set, `AEInteractWithUser` returns the `errAENoUserInteraction` result code. This flag is the default when an Apple event is sent to a remote application.
`kAECanInteract`	The server application can interact with the user in response to the Apple event—by convention, if the user needs to supply information to the server. If this flag is set and the server allows interaction, `AEInteractWithUser` either brings the server application to the foreground or posts a notification request. This flag is the default when an Apple event is sent to a local application.
`kAEAlwaysInteract`	The server application can interact with the user in response to the Apple event—by convention, whenever the server application normally asks a user to confirm a decision or interact in any other way, even if no additional information is needed from the user. If this flag is set and the server allows interaction, `AEInteractWithUser` either brings the server application to the foreground or posts a notification request.

For example, suppose a client application sends a Set Data event to a database application to change a customer's address. The database application is configured to request user confirmation of changes to a customer's record. In this case the client sets the `kAECanInteract` flag before sending the event. Thus, the database application attempts to interact with the user if interaction is allowed. If interaction is not allowed, the database makes the correction anyway without consulting the user. However, if the client application sends a Delete event to delete a customer's record entirely and sets the `kAEAlwaysInteract` flag, the database application deletes the specified record only if it can interact with the user first and receives confirmation of the decision to delete a record. If interaction with the user is not allowed, the database application returns an error. By setting the `kAEAlwaysInteract` flag, the client application ensures that the entire record won't be lost if the user sends the Delete event by mistake.

If the client application doesn't specify any of the three user interaction flags, the Apple Event Manager sets either the `kAENeverInteract` or the `kAECanInteract` flag in the `keyInteractLevelAttr` attribute of the Apple event, depending on the location of the server application. If the server application is on a remote computer,

the Apple Event Manager sets the `kAENeverInteract` flag as the default. If the server application is on the local computer, the Apple Event Manager sets the `kAECanInteract` flag as the default.

In addition to the three user interaction flags, the client application can set another flag in the `sendMode` parameter to `AESend` to request that the Apple Event Manager immediately bring the server application directly to the foreground instead of posting a notification request:

Flag	Description
`kAECanSwitchLayer`	If both the client and server allow interaction, and if the client application is the active application on the local computer and is waiting for a reply (that is, it has set the `kAEWaitReply` flag), `AEInteractWithUser` brings the server directly to the foreground. Otherwise, `AEInteractWithUser` uses the Notification Manager to request that the user bring the server application to the foreground.

Note that although the `kAECanSwitchLayer` flag must be set for the Apple Event Manager to bring the server application directly to the foreground, setting it does not guarantee that the Apple Event Manager will bypass the notification request if user interaction is permitted. Another flag, the `kAEWaitReply` flag, must also be set in the `sendMode` parameter, and the client application must provide an idle function.

The `kAEWaitReply` flag is one of three flags in the `sendMode` parameter that a client application can set to specify whether and how the client should wait for a reply. (For a description of these flags, see "Sending an Apple Event and Handling the Reply" on page 3-30.) If the client application is not waiting for a reply, the user may have continued with other work. An application switch at this point might be unexpected and would thus violate the principle of user control as described in *Macintosh Human Interface Guidelines.*

If the client application sets the `kAEWaitReply` flag, it should also provide an idle function when it calls `AESend` so that it can handle events such as update events that it receives while waiting for the reply. Idle functions are described in "Writing an Idle Function," which begins on page 5-22.

When a server application calls `AEInteractWithUser`, the function first checks whether the `kAENeverInteract` flag in the `keyInteractLevelAttr` attribute of the Apple event is set. (The Apple Event Manager sets this attribute according to the flags specified in the `sendMode` parameter of `AESend`.) If the `kAENeverInteract` flag is set, `AEInteractWithUser` immediately returns the `errAENoUserInteraction` result code. If the client specified `kAECanInteract` or `kAEAlwaysInteract`, `AEInteractWithUser` checks the server's preferences for user interaction.

Setting the Server Application's User Interaction Preferences

The server sets its user interaction preferences by using the `AESetInteractionAllowed` function. This function specifies the conditions under which your application is willing to interact with the user.

```
myErr := AESetInteractionAllowed(level);
```

The `level` parameter is of type `AEInteractAllowed`.

```
TYPE  AEInteractAllowed = (kAEInteractWithSelf,
                           kAEInteractWithLocal,
                           kAEInteractWithAll);
```

You can specify one of these values for the interaction level:

Flag	Description
`kAEInteractWithSelf`	Your server application can interact with the user in response to an Apple event only when your application is also the client application—that is, only when your application is sending the Apple event to itself.
`kAEInteractWithLocal`	Your server application can interact with the user in response to an Apple event only if the client application is on the same computer as your application. This is the default if the server application does not call the function `AESetInteractionAllowed`.
`kAEInteractWithAll`	Your server application can interact with the user in response to an Apple event sent by any client application on any computer.

If the server application does not set the user interaction level, `AEInteractWithUser` uses `kAEInteractWithLocal` as the value.

If the application sends itself an Apple event (that is, if the application is both the client and the server) without setting the `kAENeverInteract` flag, `AEInteractWithUser` always allows user interaction. If the client application is a process on the local computer and specifies `kAECanInteract` or `kAEAlwaysInteract`, and if the server has set the interaction level to `kAEInteractWithLocal` or `kAEInteractWithAll`, then `AEInteractWithUser` allows user interaction. If the client is a process on a remote computer on the network and specifies `kAECanInteract` or `kAEAlwaysInteract`, `AEInteractWithUser` allows user interaction only if the server specified the `kAEInteractWithAll` flag for the interaction level. In all other cases, `AEInteractWithUser` does not allow user interaction.

Requesting User Interaction

If your server application needs to interact with the user for any reason, it must call the `AEInteractWithUser` function to make sure it is in the foreground before it actually interacts with the user. When `AEInteractWithUser` allows user interaction (based on the client's and server's preferences), `AEInteractWithUser` brings the server application to the foreground—either directly or after the user responds to a notification request—and then returns a `noErr` result code. If `AEInteractWithUser` brings the server to the foreground directly, the client returns to the foreground immediately after the server has finished interacting with the user. If `AEInteractWithUser` brings the server to the foreground after the user responds to a notification request, the server remains in the foreground after completing the user interaction.

The `AEInteractWithUser` function specifies how long your handler is willing to wait for a response from the user. For example, if the timeout value is 900 ticks (15 seconds) and the Apple Event Manager posts a notification request, the Notification Manager begins to display a blinking icon in the upper-right corner of the screen, then removes the notification request (and the blinking icon) if the user does not respond within 15 seconds. (The discussion that follows describes some restrictions on the icons that can be displayed in this situation.)

Note that the timeout value passed to the `AEInteractWithUser` function is separate from the timeout value passed to the `AESend` function, which specifies how long the client application is willing to wait for the reply or return receipt from the server application. If `AEInteractWithUser` does not receive a response from the user within the specified time, `AEInteractWithUser` returns `errAETimeout`.

You may want to give the user a method of setting the interaction level. For example, some users may not want to be interrupted while background processing of an Apple event occurs, or they may not want to respond to dialog boxes when your application is handling Apple events sent from another computer.

Listing 4-14 illustrates the use of the `AEInteractWithUser` function. You call this function before your application displays a dialog box or otherwise interacts with the user when processing an Apple event. You specify a timeout value, a pointer to a Notification Manager record, and the address of an idle function as parameters to `AEInteractWithUser`.

Listing 4-14 Using the `AEInteractWithUser` function

```
myErr := AEInteractWithUser(kAEDefaultTimeout, gMyNotifyRecPtr,
                            @MyIdleFunction);
IF myErr <> noErr THEN
   {the attempt to interact failed; do any error handling}
   DoError(myErr)
ELSE
   {interact with the user by displaying a dialog box }
   { or by interacting in any other way that is necessary}
   DisplayMyDialogBox;
```

You can set a timeout value, in ticks, in the first parameter to `AEInteractWithUser`. Use the `kAEDefaultTimeout` constant if you want the Apple Event Manager to use a default value for the timeout value. The Apple Event Manager uses a timeout value of about one minute if you specify this constant. You can also specify the `kNoTimeOut` constant if your application is willing to wait an indefinite amount of time for a response from the user. Usually you should provide a timeout value, so that your application can complete processing of the Apple event in a reasonable amount of time.

If you specify `NIL` instead of a Notification Manager record in the second parameter of `AEInteractWithUser`, the Apple Event Manager looks for an application icon with the ID specified by the application's bundle (`'BNDL'`) resource and the application's file reference (`'FREF'`) resource. The Apple Event Manager first looks for an `'SICN'` resource with the specified ID; if it can't find an `'SICN'` resource, it looks for the `'ICN#'` resource and compresses the icon to fit in the menu bar. The Apple Event Manager won't look for any members of an icon family other than the icon specified in the `'ICN#'` resource.

If the application doesn't have `'SICN'` or `'ICN#'` resources, or if it doesn't have a file reference resource, the Apple Event Manager passes `NIL` to the Notification Manager, and no icon appears in the upper-right corner of the screen. Therefore, if you want to display any icon other than those of type `'SICN'` or `'ICN#'`, you must specify a notification record as the second parameter to the `AEInteractWithUser` function.

Note
If you want the Notification Manager to use a color icon when it posts a notification request, you should provide a Notification Manager record that specifies a `'cicn'` resource. ◆

The `AEInteractWithUser` function posts a notification request only when user interaction is allowed and the `kAECanSwitchLayer` flag in the `keyInteractLevelAttr` attribute is not set.

The last parameter to `AEInteractWithUser` specifies an idle function provided by your application. Your idle function should handle any update events, null events, operating-system events, or activate events while your application is waiting to be brought to the front. See "Writing an Idle Function" on page 5-22 for more information.

Figure 4-1 illustrates a situation in which a client application (a forms application) might request a service from a server application (a database application). To perform this service, the server application must interact with the user.

Figure 4-1 A document with a button that triggers a Get Data event

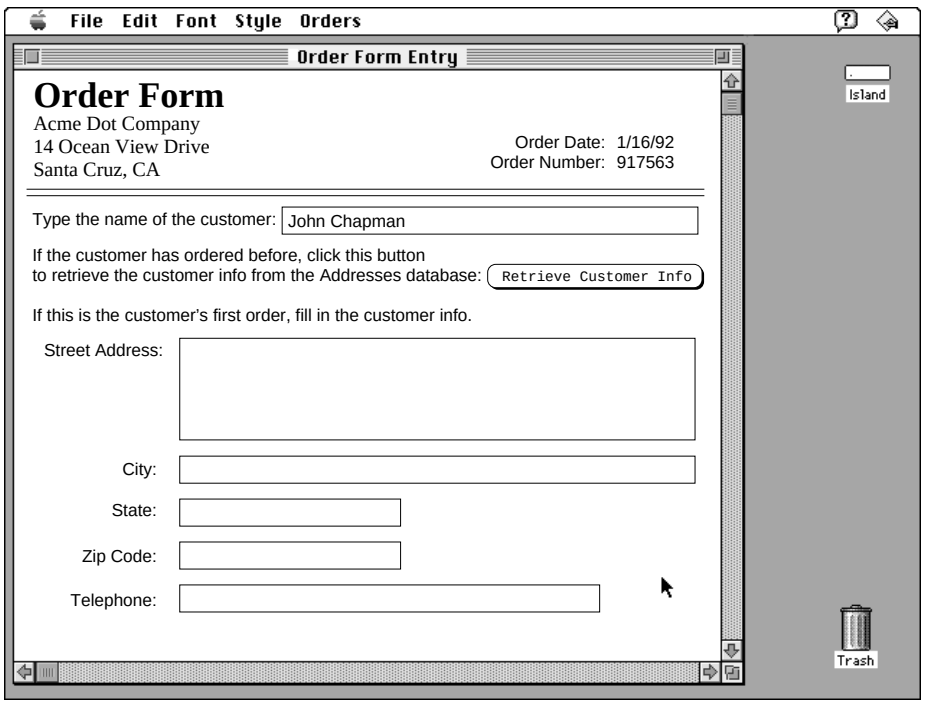


Figure 4-1 shows part of an electronic form used to enter information about an order received by telephone. If the customer has ordered from the company before, the user can quickly retrieve the customer's address and telephone number by clicking the Retrieve Customer Info button. In response, the forms application sends a Get Data event to a database application (SurfDB) currently open on the same computer. The Get Data event sent by the forms application (the client application for the ensuing transaction) asks SurfDB (the server) to locate the customer's name in a table of addresses and return the customer's address. When the forms application receives the reply Apple event, it can add the address data to the appropriate fields in the order form.

If SurfDB, as the server application, locates more than one entry for the specified customer name, it needs to interact with the user to determine which data to return in the reply Apple event. To interact with the user, the server application must be in the foreground, so that it can display a dialog box like the one shown in Figure 4-2.

Figure 4-2 A server application displaying a dialog box that requests information from the user

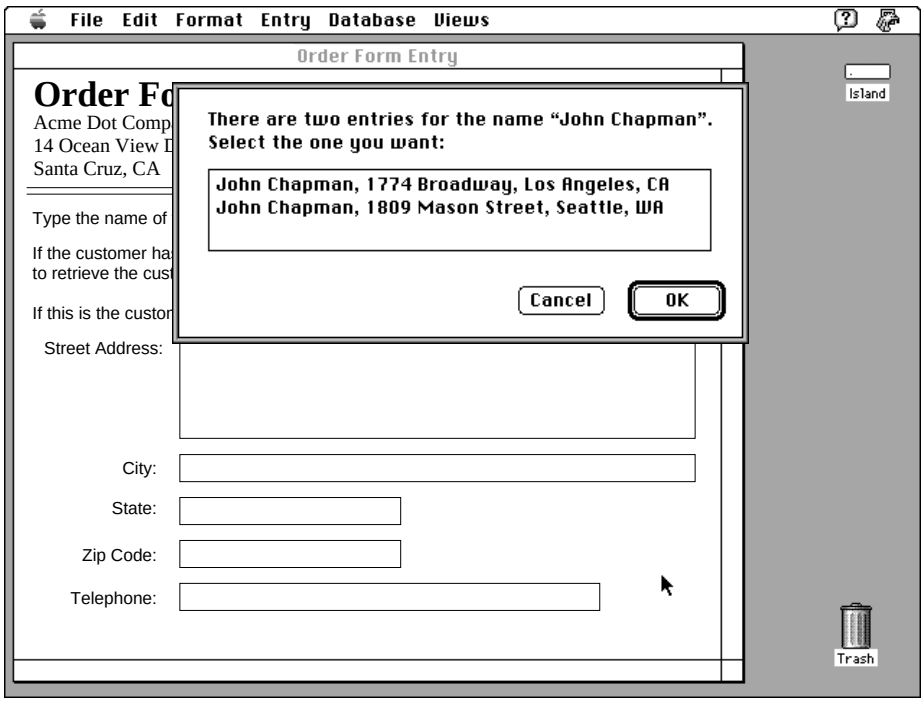


Figure 4-3, Figure 4-4, and Figure 4-5 illustrate two methods of dealing with this situation. Figure 4-3 shows the behavior of the server application that is common to both methods. In both cases, the server uses `AESetInteractionAllowed` to set its own interaction level to `kAEInteractWithLocal`. After calling `AEResolve` to locate the requested data, the server application discovers that two addresses match the name the user typed into the electronic form. The server then calls `AEInteractWithUser` with a timeout value of `kAEDefaultTimeout` so it can find out which address the user wants.

Figure 4-3 Handling user interaction

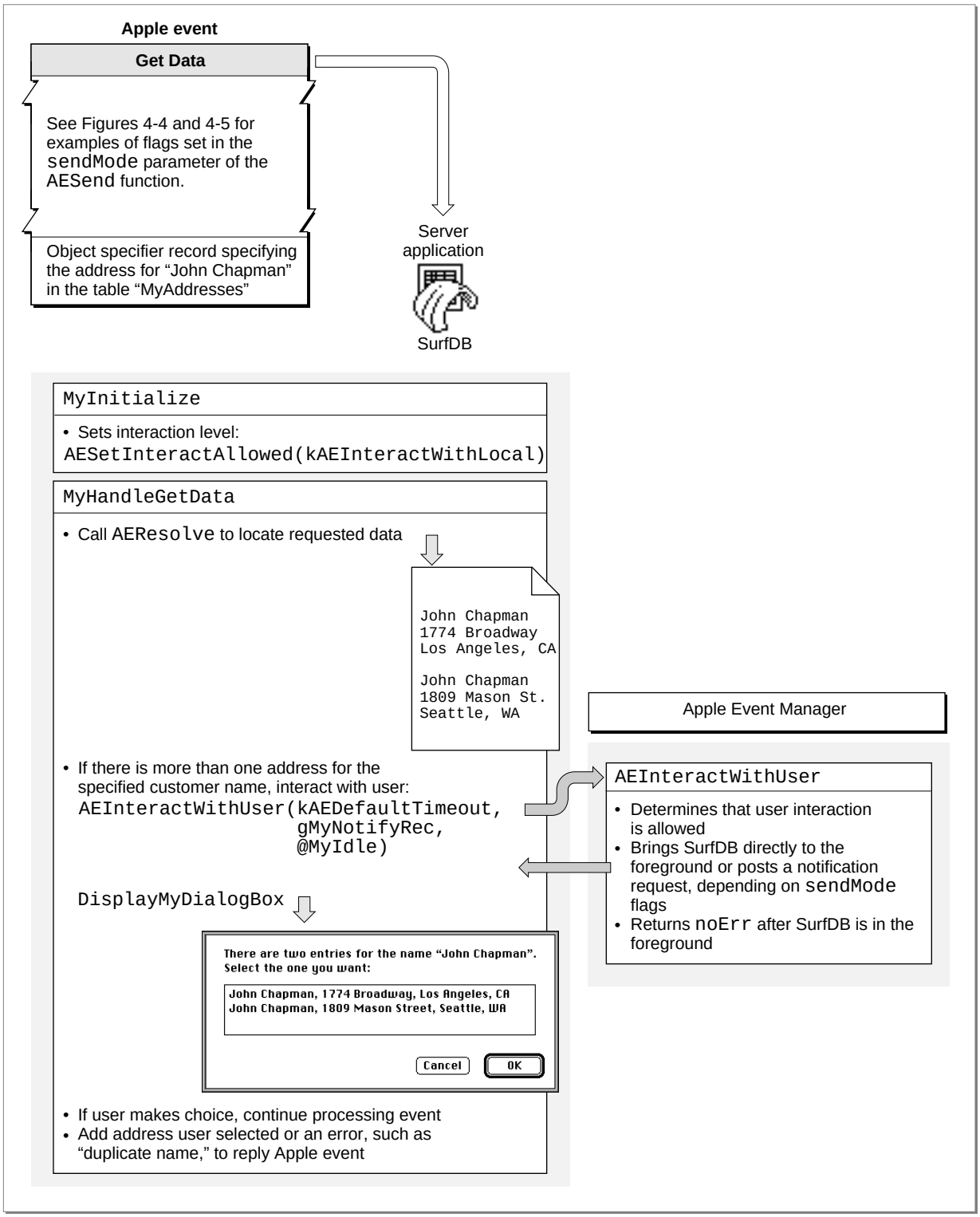


Figure 4-4 shows the circumstances in which the server application's call to `AEInteractWithUser` shown in Figure 4-3 will cause the Apple Event Manager to bring the server application directly to the foreground. The client application sets the `kAECanInteract`, `kAECanSwitchLayer`, and `kAEWaitReply` flags in the `sendMode` parameter of the `AESend` function when it sends the Get Data event shown in the figure. These flags indicate that the client application expects the user to wait until the address appears in the appropriate fields of the electronic form before continuing with any other work. In this case, an automatic layer switch will not surprise the user and will avoid the additional user action required to respond to a notification request, so `AEInteractWithUser` brings the server application directly to the foreground and returns a `noErr` result code. The server application then displays the dialog box requesting that the user select the desired customer.

After the user selects the desired customer and clicks OK, the server application's Get Data event handler returns. The Apple Event Manager immediately brings the client application to the foreground, and the client application displays the requested customer information in the appropriate fields.

Figure 4-4 Handling user interaction with the `kAEWaitReply` flag set

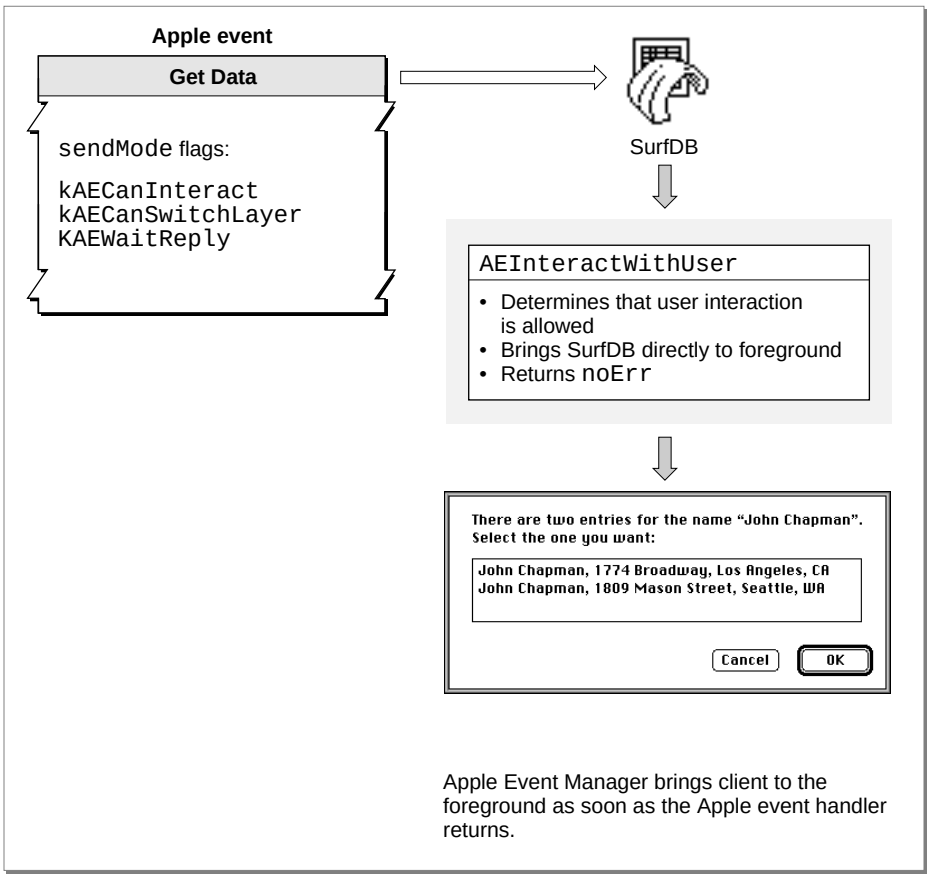
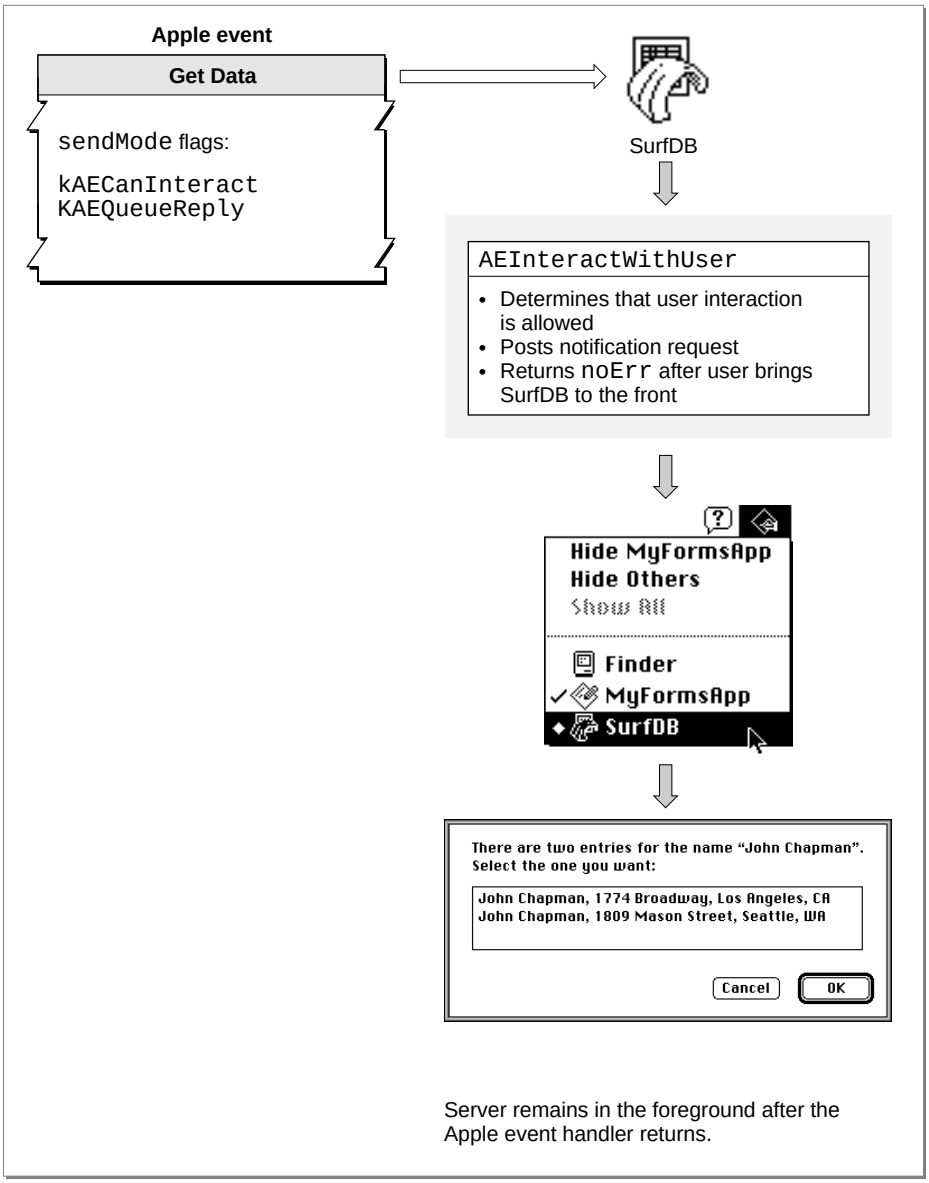


Figure 4-5 shows the circumstances in which the server application's call to `AEInteractWithUser` in Figure 4-3 will cause the Apple Event Manager to post a notification request rather than bringing the server application directly to the foreground.

Figure 4-5 Handling user interaction with the `kAEQueueReply` flag set



The only difference between the Get Data event shown in Figure 4-4 and the Get Data event shown in Figure 4-5 is that the client application has set the `kAEQueueReply` flag instead of the `kAEWaitReply` flag in the `sendMode` parameter of `AESend` and has not set the `kAECanSwitchLayer` flag. This combination of flags indicates that the client application expects the user to continue filling in other parts of the form, such as the items being ordered; the address will just appear after a while, provided there is no duplicate name. In this case, an automatic layer switch would disrupt the user's work. Instead of bringing the server application directly to the foreground, `AEInteractWithUser` uses the Notification Manager to post a notification request.

After the user has responded to the request and has brought the server application to the foreground, `AEInteractWithUser` returns a `noErr` result code, and the server application displays the dialog box requesting that the user select the desired customer. When the user selects a customer and clicks OK, the server application's Get Data event handler returns. Because the user brought the server to the foreground manually, the server remains in the foreground after the handler returns.

Reference to Responding to Apple Events

This section describes the basic Apple Event Manager data structures and routines that your application can use to respond to Apple events. It also describes the syntax for application-defined Apple event handlers and coercion handlers that your application can provide for use by the Apple Event Manager.

For information about routines used to create and send Apple events, see the chapter "Creating and Sending Apple Events" in this book. For information about routines and data structures used with object specifier records, see the chapter "Resolving and Creating Object Specifier Records" in this book.

Data Structures Used by the Apple Event Manager

This section summarizes the major data structures used by the Apple Event Manager. For an overview of the relationships among these data structures, see "Data Structures Within Apple Events," which begins on page 3-12.

Descriptor Records and Related Data Structures

Descriptor records are the fundamental data structures from which Apple events are constructed. A descriptor record is a data structure of type `AEDesc`.

```
TYPE AEDesc =
   RECORD                              {descriptor record}
      descriptorType:   DescType;      {type of data being passed}
      dataHandle:       Handle;        {handle to data being passed}
   END;
```

Field descriptions

`descriptorType` A four-character string of type `DescType` that indicates the type of data being passed.

`dataHandle` A handle to the data being passed.

The descriptor type is a structure of type `DescType`, which in turn is of data type `ResType`—that is, a four-character code. Constants, rather than these four-character codes, are usually used to refer to descriptor types. Table 4-2 lists the constants for the basic descriptor types used by the Apple Event Manager.

Table 4-2 Descriptor types used by the Apple Event Manager (excluding those used with object specifier records)

Descriptor type	Value	Description
`typeBoolean`	`'bool'`	Boolean value
`typeChar`	`'TEXT'`	Unterminated string
`typeLongInteger`	`'long'`	32-bit integer
`typeInteger`	`'long'`	32-bit integer
`typeShortInteger`	`'shor'`	16-bit integer
`typeSMInt`	`'shor'`	16-bit integer
`typeLongFloat`	`'doub'`	SANE double
`typeFloat`	`'doub'`	SANE double
`typeShortFloat`	`'sing'`	SANE single
`typeSMFloat`	`'sing'`	SANE single
`typeExtended`	`'exte'`	SANE extended
`typeComp`	`'comp'`	SANE comp
`typeMagnitude`	`'magn'`	Unsigned 32-bit integer
`typeAEList`	`'list'`	List of descriptor records
`typeAERecord`	`'reco'`	List of keyword-specified descriptor records
`typeAppleEvent`	`'aevt'`	Apple event record
`typeTrue`	`'true'`	`TRUE` Boolean value

continued

Table 4-2 Descriptor types used by the Apple Event Manager (excluding those used with object specifier records) (continued)

Descriptor type	Value	Description
typeFalse	'fals'	FALSE Boolean value
typeAlias	'alis'	Alias record
typeEnumerated	'enum'	Enumerated data
typeType	'type'	Four-character code for event class or event ID
typeAppParameters	'appa'	Process Manager launch parameters
typeProperty	'prop'	Apple event property
typeFSS	'fss '	File system specification
typeKeyword	'keyw'	Apple event keyword
typeSectionH	'sect'	Handle to a section record
typeWildCard	'****'	Matches any type
typeApplSignature	'sign'	Application signature
typeSessionID	'ssid'	Session reference number
typeTargetID	'targ'	Target ID record
typeProcessSerialNumber	'psn '	Process serial number
typeNull	'null'	Nonexistent data (data handle is NIL)

For information about descriptor records and descriptor types used with object specifier records, see the chapter "Resolving and Creating Object Specifier Records" in this book.

Apple event attributes, Apple event parameters, object specifier records, tokens, and most of the other data structures used by the Apple Event Manager are constructed from one or more descriptor records. The Apple Event Manager identifies the various parts of an Apple event by means of keywords associated with the corresponding descriptor records. The AEKeyword data type is defined as a four-character code.

```
TYPE  AEKeyword = PACKED ARRAY[1..4] OF Char;
```

Constants are typically used for keywords. A keyword combined with a descriptor record forms a keyword-specified descriptor record, which is defined by a data structure of type AEKeyDesc.

```
TYPE  AEKeyDesc =
      RECORD
         descKey:       AEKeyword;     {keyword}
         descContent:   AEDesc;        {descriptor record}
      END;
```

Field descriptions

`descKey`	A four-character code of type `AEKeyword` that identifies the data in the `descContent` field.
`descContent`	A descriptor record of type `AEDesc`.

Every Apple event includes an attribute that contains the address of the target application. A descriptor record that contains an application's address is called an address descriptor record.

```
TYPE  AEAddressDesc = AEDesc;       {address descriptor record}
```

Many Apple Event Manager functions take or return lists of descriptor records in a special descriptor record called a descriptor list. A descriptor list is a structure of data type `AEDescList` whose data consists of a list of other descriptor records.

```
TYPE  AEDescList = AEDesc;          {list of descriptor records}
```

Other Apple Event Manager functions take or return lists of keyword-specified descriptor records in the form of an AE record. An AE record is a structure of data type `AERecord` whose data handle refers to a list of keyword-specified descriptor records.

```
TYPE  AERecord = AEDescList;        {list of keyword-specified }
                                    { descriptor records}
```

The handle for a descriptor list of data type `AERecord` refers to a list of keyword-specified descriptor records that specify Apple event parameters; they cannot specify Apple event attributes.

Finally, a full-fledged Apple event, including both attributes and parameters, is an Apple event record, which is a structure of data type `AppleEvent`.

```
TYPE  AppleEvent = AERecord;        {list of attributes and }
                                    { parameters for an Apple }
                                    { event}
```

The event class and event ID of an Apple event are specified in Apple Event Manager routines by structures of data types `AEEventClass` and `AEEventID`, respectively.

```
TYPE AEEventClass = PACKED ARRAY[1..4] OF Char;

TYPE AEEventID = PACKED ARRAY[1..4] OF Char;
```

For more information about descriptor records and the other data structures described in this section, see "Data Structures Within Apple Events," which begins on page 3-12.

With the exception of array data records, which are described in the next section, the other Apple Event Manager data structures used in responding to Apple events are described in "Routines for Responding to Apple Events," beginning on page 4-61, under the descriptions of the routines that use them.

Apple Event Array Data Types

The `AEGetArray` function (see page 4-77) creates a Pascal or C array that corresponds to an ***Apple event array*** in a descriptor list, and the `AEPutArray` function (see page 5-32) adds data specified in a buffer to a descriptor list as an Apple event array.

You can use the data type `AEArrayType` to define the type of Apple event array you want to add to or obtain from a descriptor list.

```
TYPE AEArrayType = (kAEDataArray, kAEPackedArray, kAEHandleArray,
                    kAEDescArray, kAEKeyDescArray);
```

When your application adds an Apple event array to a descriptor list, it provides the data for an Apple event array in an array data record, which is defined by the data type `AEArrayData`.

```
TYPE AEArrayData =
   RECORD                          {data for an Apple event array}
      CASE AEArrayType OF
      kAEDataArray:
         (AEDataArray:     ARRAY[0..0] OF Integer);
      kAEPackedArray:
         (AEPackedArray:   PACKED ARRAY[0..0] OF Char);
      kAEHandleArray:
         (AEHandleArray:   ARRAY[0..0] OF Handle);
      kAEDescArray:
         (AEDescArray:     ARRAY[0..0] OF AEDesc);
      kAEKeyDescArray:
         (AEKeyDescArray:  ARRAY[0..0] OF AEKeyDesc);
   END;
```

The type of array depends on the data for the array:

Array type	Description of Apple event array
`kAEDataArray`	Array items consist of data of the same size and same type, and are aligned on word boundaries.
`kAEPackedArray`	Array items consist of data of the same size and same type, and are packed without regard for word boundaries.
`kAEHandleArray`	Array items consist of handles to data of variable size and the same type.
`kAEDescArray`	Array items consist of descriptor records of different descriptor types with data of variable size.
`kAEKeyDescArray`	Array items consist of keyword-specified descriptor records with different keywords, different descriptor types, and data of variable size.

Array items in Apple event arrays of type `kAEDataArray`, `kAEPackedArray`, or `kAEHandleArray` must be factored—that is, contained in a factored descriptor list. Before adding array items to a factored descriptor list, you should provide both a pointer to the data that is common to all array items and the size of that common data when you first call `AECreateList` to create a factored descriptor list. When you call `AEPutArray` to add the array data to such a descriptor list, the Apple Event Manager automatically isolates the common data you specified in the call to `AECreateList`.

When you call `AEGetArray` or `AEPutArray`, you specify a pointer of data type `AEArrayDataPointer` that points to a buffer containing the data for the array.

```
TYPE AEArrayDataPointer = ^AEArrayData;
```

For more information about using `AECreateList` to create factored descriptor lists for arrays, see page 5-29. For information about using `AEGetArray` and `AEPutArray`, see page 4-77 and page 5-32, respectively.

Routines for Responding to Apple Events

This section describes the Apple Event Manager routines you can use to create and manage the Apple event dispatch tables, dispatch Apple events, extract information from Apple events, request user interaction, request more time to respond to Apple events, suspend and resume Apple event handling, delete descriptor records, deallocate memory for descriptor records, create and manage the coercion handler and special handler dispatch tables, and get information about the Apple Event Manager.

Because the Apple Event Manager uses the services of the Event Manager, which in turn uses the services of the PPC Toolbox, the routines described in this section may return Event Manager and PPC Toolbox result codes in addition to the Apple Event Manager result codes listed.

Creating and Managing the Apple Event Dispatch Tables

An Apple event dispatch table contains entries that specify the event class and event ID that refer to one or more Apple events, the address of the handler routine that handles those Apple events, and a reference constant. You can use the `AEInstallEventHandler` function to add entries to the Apple event dispatch table. This function sets up the initial mapping between the handlers in your application and the Apple events that they handle.

To get the address of a handler currently in the Apple event dispatch table, use the `AEGetEventHandler` function. If you need to remove any of your Apple event handlers after the mapping between handlers and Apple events is established, you can use the `AERemoveEventHandler` function.

AEInstallEventHandler

You can use the `AEInstallEventHandler` function to add an entry to either your application's Apple event dispatch table or the system Apple event dispatch table.

```
FUNCTION AEInstallEventHandler (theAEEventClass: AEEventClass;
                                theAEEventID: AEEventID;
                                handler: EventHandlerProcPtr;
                                handlerRefcon: LongInt;
                                isSysHandler: Boolean): OSErr;
```

`theAEEventClass`
The event class for the Apple event or events to be dispatched for this entry. The `AEEventClass` data type is defined as a four-character code:

```
TYPE AEEventClass = PACKED ARRAY[1..4] OF Char;
```

`theAEEventID`
The event ID for the Apple event or events to be dispatched for this entry. The `AEEventID` data type is defined as a four-character code:

```
TYPE AEEventID = PACKED ARRAY[1..4] OF Char;
```

`handler` A pointer to an Apple event handler for this dispatch table entry. Note that a handler in the system dispatch table must reside in the system heap; this means that if the value of the `isSysHandler` parameter is `TRUE`, the handler parameter should point to a location in the system heap. Otherwise, if you put your system handler code in your application heap, you must use `AERemoveEventHandler` to remove the handler before your application terminates.

`handlerRefcon`
A reference constant that is passed by the Apple Event Manager to the handler each time the handler is called. If your handler doesn't use a reference constant, use 0 as the value of this parameter.

`isSysHandler`
Specifies the dispatch table to which you want to add the handler. If the value of `isSysHandler` is `TRUE`, the Apple Event Manager adds the handler to the system Apple event dispatch table. Entries in the system dispatch table are available to all applications. If the value of `isSysHandler` is `FALSE`, the Apple Event Manager adds the handler to your application's Apple event dispatch table. The application's dispatch table is searched first; the system dispatch table is searched only if the necessary handler is not found in your application's dispatch table.

DESCRIPTION

The `AEInstallEventHandler` function creates an entry in the Apple event dispatch table. You must supply parameters that specify the event class, the event ID, the address of the handler that handles Apple events of the specified event class and event ID, and whether the handler is to be added to the system Apple event dispatch table or your application's Apple event dispatch table. You can also specify a reference constant that the Apple Event Manager passes to your handler whenever your handler processes an Apple event.

The parameters `theAEEventClass` and `theAEEventID` specify the event class and event ID of the Apple events to be handled by the handler for this dispatch table entry. For these parameters, you must provide one of the following combinations:

- the event class and event ID of a single Apple event to be dispatched to the handler
- the `typeWildCard` constant for `theAEEventClass` and an event ID for `theAEEventID`, which indicate that Apple events from all event classes whose event IDs match `theAEEventID` should be dispatched to the handler
- an event class for `theAEEventClass` and the `typeWildCard` constant for `theAEEventID`, which indicate that all events from the specified event class should be dispatched to the handler
- the `typeWildCard` constant for both the `theAEEventClass` and `theAEEventID` parameters, which indicates that all Apple events should be dispatched to the handler

IMPORTANT

If you use the `typeWildCard` constant for either the `theAEEventClass` or the `theAEEventID` parameter (or for both parameters), the corresponding handler must return the error `errAEEventNotHandled` if it does not handle a particular event. ▲

If there was already an entry in the specified dispatch table for the same event class and event ID, it is replaced. Therefore, before installing a handler for a particular Apple event in the system dispatch table, use the `AEGetEventHandler` function (described next) to determine whether the table already contains a handler for that event. If an entry exists, `AEGetEventHandler` returns a reference constant and a pointer to that event handler. Chain the existing handler to your handler by providing pointers to the previous handler and its reference constant in the `handlerRefcon` parameter of `AEInstallEventHandler`. When your handler is done, use these pointers to call the previous handler. If you remove your system Apple event handler, be sure to reinstall the chained handler.

SPECIAL CONSIDERATIONS

Before an application calls a system Apple event handler, system software has set up the A5 register for the calling application. For this reason, if you provide a system Apple event handler, it should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system handler may crash.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Parameter error (handler pointer is `NIL` or odd)
`memFullErr`	–108	Not enough room in heap zone

SEE ALSO

For more information about installing Apple event handlers, see "Installing Entries in the Apple Event Dispatch Tables," which begins on page 4-7.

AEGetEventHandler

You can use the `AEGetEventHandler` function to get an entry from an Apple event dispatch table.

```
FUNCTION AEGetEventHandler (theAEEventClass: AEEventClass;
                            theAEEventID: AEEventID;
                            VAR handler: EventHandlerProcPtr;
                            VAR handlerRefcon: LongInt;
                            isSysHandler: Boolean): OSErr;
```

`theAEEventClass`
The value of the event class field of the dispatch table entry for the desired handler.

`theAEEventID`
The value of the event ID field of the dispatch table entry for the desired handler.

`handler` The `AEGetEventHandler` function returns, in this parameter, a pointer to the specified handler.

`handlerRefcon`
The `AEGetEventHandler` function returns, in this parameter, the reference constant from the dispatch table entry for the specified handler.

`isSysHandler`
Specifies the Apple event dispatch table from which to get the handler. If the value of `isSysHandler` is `TRUE`, the `AEGetEventHandler` function returns the handler from the system dispatch table. If the value is `FALSE`, `AEGetEventHandler` returns the handler from your application's dispatch table.

DESCRIPTION

The `AEGetEventHandler` function returns, in the `handler` parameter, a pointer to the handler for the Apple event dispatch table entry you specify in the parameters `theAEEventClass` and `theAEEventID`. You can use the `typeWildCard` constant for either or both of these parameters; however, `AEGetEventHandler` returns an error unless an entry exists that specifies `typeWildCard` in exactly the same way. For example, if you specify `typeWildCard` in both the `theAEEventClass` parameter and the `theAEEventID` parameter, the Apple Event Manager will not return the first handler for any event class and event ID in the dispatch table; instead, the dispatch table must contain an entry that specifies type `typeWildCard` for both the event class and the event ID.

RESULT CODES

`noErr`	0	No error
`errAEHandlerNotFound`	–1717	No handler found for an Apple event

SEE ALSO

For an explanation of wildcard values, see the description of the `AEInstallEventHandler` function on page 4-62.

AERemoveEventHandler

You can use the `AERemoveEventHandler` function to remove an entry from an Apple event dispatch table.

```
FUNCTION AERemoveEventHandler (theAEEventClass: AEEventClass;
                               theAEEventID: AEEventID;
                               handler: EventHandlerProcPtr;
                               isSysHandler: Boolean): OSErr;
```

`theAEEventClass`
The event class for the handler whose entry you want to remove from the dispatch table.

`theAEEventID`
: The event ID for the handler whose entry you want to remove from the Apple event dispatch table.

`handler`
: A pointer to the handler to be removed. Although the parameters `theAEEventClass` and `theAEEventID` would be sufficient to identify the handler to be removed, providing the handler parameter is a recommended safeguard that ensures that you remove the correct handler. If the value of this parameter is `NIL`, the Apple Event Manager relies solely on the event class and event ID to identify the handler to be removed.

`isSysHandler`
: Specifies the dispatch table from which to remove the handler. If the value of `isSysHandler` is `TRUE`, `AERemoveEventHandler` removes the handler from the system dispatch table. If the value is `FALSE`, `AERemoveEventHandler` removes the handler from your application's dispatch table.

DESCRIPTION

The `AERemoveEventHandler` function removes the Apple event dispatch table entry you specify in the parameters `theAEEventClass`, `theAEEventID`, and `handler`. You can use the `typeWildCard` constant for the `theAEEventClass` or the `theAEEventID` parameter, or for both parameters; however, `AERemoveEventHandler` returns an error unless an entry exists that specifies `typeWildCard` in exactly the same way. For example, if you specify `typeWildCard` in both the `theAEEventClass` parameter and the `theAEEventID` parameter, the Apple Event Manager will not remove the first handler for any event class and event ID in the dispatch table; instead, the dispatch table must contain an entry that specifies type `typeWildCard` for both the event class and the event ID.

RESULT CODES

`noErr`	0	No error
`errAEHandlerNotFound`	–1717	No handler found for an Apple event

SEE ALSO

For an explanation of wildcard values, see the description of the `AEInstallEventHandler` function on page 4-62.

Dispatching Apple Events

After receiving a high-level event (and optionally determining whether it is a type of high-level event other than an Apple event that your application might support), your application typically calls the `AEProcessAppleEvent` function to determine the type of Apple event received and call the corresponding handler.

AEProcessAppleEvent

You can use the `AEProcessAppleEvent` function to call the appropriate handler for a specified Apple event.

```
FUNCTION AEProcessAppleEvent
                        (theEventRecord: EventRecord): OSErr;
```

`theEventRecord`
The event record for the Apple event.

DESCRIPTION

The `AEProcessAppleEvent` function looks first in the application's special handler dispatch table for an entry that was installed with the constant `keyPreDispatch`. If the application's special handler dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the function looks in the application's Apple event dispatch table for an entry that matches the event class and event ID of the specified Apple event.

If the application's Apple event dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the `AEProcessAppleEvent` function looks in the system special handler dispatch table for an entry that was installed with the constant `keyPreDispatch`. If the system special handler dispatch table does not include such a handler or if the handler returns `errAEEventNotHandled`, the function looks in the system Apple event dispatch table for an entry that matches the event class and event ID of the specified Apple event.

If the system Apple event dispatch table does not include such a handler, the Apple Event Manager returns the result code `errAEEventNotHandled` to the server application and, if the client application is waiting for a reply, to the client application.

If `AEProcessAppleEvent` finds an entry in one of the dispatch tables that matches the event class and event ID of the specified Apple event, it calls the corresponding handler.

SPECIAL CONSIDERATIONS

If an Apple event dispatch table contains one entry for an event class and a specific event ID, and also contains another entry that is identical except that it specifies a wildcard value for either the event class or the event ID, the Apple Event Manager dispatches the more specific entry. For example, if an Apple event dispatch table includes one entry that specifies the event class as `kAECoreSuite` and the event ID as `kAEDelete`, and another entry that specifies the event class as `kAECoreSuite` and the event ID as `typeWildCard`, the Apple Event Manager dispatches the Apple event handler associated with the entry that specifies the event ID as `kAEDelete`.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`bufferIsSmall`	–607	Buffer is too small
`noOutstandingHLE`	–608	No outstanding high-level event
`errAECorruptData`	–1702	Data in an Apple event could not be read
`errAENewerVersion`	–1706	Need a newer version of the Apple Event Manager
`errAEEventNotHandled`	–1708	Event wasn't handled by an Apple event handler

SEE ALSO

For an example of the use of `AEProcessAppleEvent`, see Listing 4-2 on page 4-6.

For a description of an Apple event handler, see page 4-105.

For more information about event processing, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

Getting Data or Descriptor Records Out of Apple Event Parameters and Attributes

The Apple Event Manager provides four functions that allow you to get data from Apple event parameters and attributes. The `AEGetParamPtr` and `AEGetParamDesc` functions get data from a specified Apple event parameter. The `AEGetAttributePtr` and `AEGetAttributeDesc` functions get data from a specified Apple event attribute.

AEGetParamPtr

You can use the `AEGetParamPtr` function to get a pointer to a buffer that contains the data from a specified Apple event parameter.

```
FUNCTION AEGetParamPtr (theAppleEvent: AppleEvent;
                        theAEKeyword: AEKeyword;
                        desiredType: DescType;
                        VAR typeCode: DescType; dataPtr: Ptr;
                        maximumSize: Size;
                        VAR actualSize: Size): OSErr;
```

`theAppleEvent`
The Apple event containing the desired parameter.

`theAEKeyword`
The keyword that specifies the desired parameter.

`desiredType`
The desired descriptor type for the data to be returned; if the requested Apple event parameter is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of `desiredType`

is `typeWildCard`, no coercion is performed, and the descriptor type of the returned data is the same as the descriptor type of the Apple event parameter.

`typeCode` The descriptor type of the returned data.

`dataPtr` A pointer to the buffer in which the returned data is stored.

`maximumSize`
The maximum length, in bytes, of the data to be returned. You must allocate at least this amount of storage for the buffer specified by the `dataPtr` parameter.

`actualSize`
The length, in bytes, of the data for the specified Apple event parameter. If this value is larger than the value of the `maximumSize` parameter, not all of the data for the parameter was returned.

DESCRIPTION

The `AEGetParamPtr` function uses a buffer to return the data from a specified Apple event parameter, which it attempts to coerce to the descriptor type specified by the `desiredType` parameter.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested descriptor type
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEReplyNotArrived`	–1718	Reply has not yet arrived

SEE ALSO

For examples of the use of `AEGetParamPtr`, see "Getting Data Out of an Apple Event," which begins on page 4-25.

AEGetParamDesc

You can use the `AEGetParamDesc` function to get the descriptor record for a specified Apple event parameter.

```
FUNCTION AEGetParamDesc (theAppleEvent: AppleEvent;
                         theAEKeyword: AEKeyword;
                         desiredType: DescType;
                         VAR result: AEDesc): OSErr;
```

`theAppleEvent`
The Apple event containing the desired parameter.

`theAEKeyword`
The keyword that specifies the desired parameter.

`desiredType`
The desired descriptor type for the descriptor record to be returned; if the requested Apple event parameter is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of `desiredType` is `typeWildCard`, no coercion is performed, and the descriptor type of the returned data is the same as the descriptor type of the Apple event parameter.

`result` The descriptor record from the desired Apple event parameter coerced to the descriptor type specified in `desiredType`.

DESCRIPTION

The `AEGetParamDesc` function returns, in the `result` parameter, the descriptor record for a specified Apple event parameter, which it attempts to coerce to the descriptor type specified by the `desiredType` parameter. Your application should call the `AEDisposeDesc` function to dispose of the resulting descriptor record after your application has finished using it.

If `AEGetParamDesc` returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested descriptor type
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEReplyNotArrived`	–1718	Reply has not yet arrived

SEE ALSO

For an example of the use of `AEGetParamDesc`, see "Getting Data Out of an Apple Event Parameter," which begins on page 4-26.

AEGetAttributePtr

You can use the `AEGetAttributePtr` function to get a pointer to a buffer that contains the data from a specified Apple event attribute.

```
FUNCTION AEGetAttributePtr (theAppleEvent: AppleEvent;
                            theAEKeyword: AEKeyword;
                            desiredType: DescType;
                            VAR typeCode: DescType; dataPtr: Ptr;
                            maximumSize: Size;
                            VAR actualSize: Size): OSErr;
```

`theAppleEvent`
The Apple event containing the desired attribute.

`theAEKeyword`
The keyword that specifies the desired attribute.

```
TYPE AEKeyword = PACKED ARRAY[1..4] OF Char;
```

The keyword can be any of the constants listed in the description that follows.

`desiredType`
The desired descriptor type for the data to be returned; if the requested Apple event attribute is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of `desiredType` is `typeWildCard`, no coercion is performed, and the descriptor type of the returned data is the same as the descriptor type of the Apple event attribute.

`typeCode` The descriptor type of the returned data.

`dataPtr` A pointer to the buffer in which the returned data is stored.

`maximumSize`
The maximum length, in bytes, of the data to be returned. You must allocate at least this amount of storage for the buffer specified by the `dataPtr` parameter.

`actualSize`
The length, in bytes, of the data for the specified Apple event attribute. If this value is larger than the value of the `maximumSize` parameter, not all of the data for the attribute was returned.

DESCRIPTION

The `AEGetAttributePtr` function uses a buffer to return the data from an Apple event attribute with the specified keyword, which it attempts to coerce to the descriptor type specified by the `desiredType` parameter. You can specify the parameter `theAEKeyWord` using any of these constants:

```
CONST
   keyAddressAttr          = 'addr'; {address of target or }
                                     { client application}
   keyEventClassAttr       = 'evcl'; {event class}
   keyEventIDAttr          = 'evid'; {event ID}
   keyEventSourceAttr      = 'esrc'; {nature of source }
                                     { application}
   keyInteractLevelAttr    = 'inte'; {settings to allow the }
                                     { Apple Event Manager to }
                                     { bring server application }
                                     { to the foreground}
   keyMissedKeywordAttr    = 'miss'; {first required parameter }
                                     { remaining in Apple event}
   keyOptionalKeywordAttr  = 'optk'; {list of optional }
                                     { parameters for Apple }
                                     { event}
   keyOriginalAddressAttr  = 'from'; {address of original source }
                                     { of Apple event; available }
                                     { beginning with version }
                                     { 1.01 of Apple Event }
                                     { Manager}
   keyReturnIDAttr         = 'rtid'; {return ID for reply Apple }
                                     { event}
   keyTimeoutAttr          = 'timo'; {length of time in ticks }
                                     { that client will wait }
                                     { for reply or result from }
                                     { the server}
   keyTransactionIDAttr    = 'tran'; {transaction ID identifying }
                                     { a series of Apple events}
```

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested descriptor type
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEReplyNotArrived`	–1718	Reply has not yet arrived

SEE ALSO

For an example of the use of the `AEGetAttributePtr` function, see "Getting Data Out of an Attribute" and "Writing Apple Event Handlers," which begin on page 4-28 and page 4-33, respectively.

AEGetAttributeDesc

You can use the `AEGetAttributeDesc` function to get the descriptor record for a specified Apple event attribute.

```
FUNCTION AEGetAttributeDesc (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword;
                             desiredType: DescType;
                             VAR result: AEDesc): OSErr;
```

`theAppleEvent`
The Apple event containing the desired attribute.

`theAEKeyword`
The keyword that specifies the desired attribute.

```
TYPE AEKeyword = PACKED ARRAY[1..4] OF Char;
```

The keyword can be any of the constants listed in the description of `AEGetAttributePtr` on page 4-71.

`desiredType`
The desired descriptor type for the descriptor record to be returned; if the requested Apple event attribute is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of `desiredType` is `typeWildCard`, no coercion is performed, and the descriptor type of the returned data is the same as the descriptor type of the Apple event attribute.

`result` A copy of the descriptor record from the desired attribute coerced to the descriptor type specified by the `desiredType` parameter.

DESCRIPTION

The `AEGetAttributeDesc` function returns, in the `result` parameter, the descriptor record for the Apple event attribute with the specified keyword. Your application should call the `AEDisposeDesc` function to dispose of the resulting descriptor record after your application has finished using it.

If `AEGetAttributeDesc` returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested descriptor type
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEReplyNotArrived`	–1718	Reply has not yet arrived

Counting the Items in Descriptor Lists

The `AECountItems` function counts the number of descriptor records in any descriptor list, including an Apple event record.

AECountItems

You can use the `AECountItems` function to count the number of descriptor records in any descriptor list.

```
FUNCTION AECountItems (theAEDescList: AEDescList;
                       VAR theCount: LongInt): OSErr;
```

`theAEDescList`
The descriptor list to be counted.

`theCount` The `AECountItems` function returns the number of descriptor records in the specified descriptor list in this parameter.

RESULT CODES

`noErr`	0	No error
`errAENotAEDesc`	–1704	Not a valid descriptor record

SEE ALSO

For an example of the use of `AECountItems`, see "Getting Data Out of a Descriptor List," which begins on page 4-31.

Getting Items From Descriptor Lists

The Apple Event Manager provides three functions that allow you to get items from any descriptor list, including an Apple event record. The `AEGetNthPtr` and `AEGetNthDesc` functions give you access to the data in a descriptor list. The `AEGetArray` function gets data from an array contained in a descriptor list.

AEGetNthPtr

You can use the `AEGetNthPtr` function to get a pointer to a buffer that contains a copy of a descriptor record from any descriptor list.

```
FUNCTION AEGetNthPtr (theAEDescList: AEDescList; index: LongInt;
                      desiredType: DescType;
                      VAR theAEKeyword: AEKeyword;
                      VAR typeCode: DescType; dataPtr: Ptr;
                      maximumSize: Size;
                      VAR actualSize: Size): OSErr;
```

`theAEDescList`
The descriptor list containing the desired descriptor record.

`index` The position of the desired descriptor record in the list (for example, 2 specifies the second descriptor record).

`desiredType`
The desired descriptor type for the copy of the descriptor record to be returned; if the desired descriptor record is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of `desiredType` is `typeWildCard`, no coercion is performed, and the descriptor type of the copied descriptor record is the same as the descriptor type of the original descriptor record.

`theAEKeyword`
The keyword of the specified descriptor record, if you are getting data from a list of keyword-specified descriptor records; otherwise, `AEGetNthPtr` returns the value `typeWildCard`.

`typeCode` The descriptor type of the returned descriptor record.

`dataPtr` A pointer to the buffer in which the returned descriptor record is stored.

`maximumSize`
The maximum length, in bytes, of the data to be returned. You must allocate at least this amount of storage for the buffer specified by the `dataPtr` parameter.

`actualSize`
The length, in bytes, of the data for the specified descriptor record. If this value is larger than the value of the `maximumSize` parameter, not all of the data for the descriptor record was returned.

DESCRIPTION

The `AEGetNthPtr` function uses a buffer to return a specified descriptor record from a specified descriptor list; the function attempts to coerce the descriptor record to the descriptor type specified by the `desiredType` parameter.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested descriptor type
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEReplyNotArrived`	–1718	Reply has not yet arrived

SEE ALSO

For an example of the use of `AEGetNthPtr`, see Listing 4-10 on page 4-33.

AEGetNthDesc

You can use the `AEGetNthDesc` function to get a copy of a descriptor record from any descriptor list.

```
FUNCTION AEGetNthDesc (theAEDescList: AEDescList; index: LongInt;
                       desiredType: DescType;
                       VAR theAEKeyword: AEKeyword;
                       VAR result: AEDesc): OSErr;
```

`theAEDescList`
The descriptor list containing the desired descriptor record.

`index` The position of the desired descriptor record in the list (for example, 2 specifies the second descriptor record).

`desiredType`
The desired descriptor type for the copy of the descriptor record to be returned; if the desired descriptor record is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of `desiredType` is `typeWildCard`, no coercion is performed, and the descriptor type of the copied descriptor record is the same as the descriptor type of the original descriptor record.

`theAEKeyword`
The keyword of the specified descriptor record, if you are getting data from a list of keyword-specified descriptor records; otherwise, `AEGetNthDesc` returns the value `typeWildCard`.

`result` A copy of the desired descriptor record coerced to the descriptor type specified by the `desiredType` parameter.

DESCRIPTION

The `AEGetNthDesc` function returns a specified descriptor record from a specified descriptor list.Your application should call the `AEDisposeDesc` function to dispose of the resulting descriptor record after your application has finished using it.

If `AEGetNthDesc` returns a nonzero result code, it returns a descriptor record of descriptor type `typeNull`. A descriptor record of this type does not contain any data.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested descriptor type
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEReplyNotArrived`	–1718	Reply has not yet arrived

AEGetArray

You can use the `AEGetArray` function to convert an Apple event array (an array created with the `AEPutArray` function and stored in a descriptor list) to the corresponding Pascal or C array and place the converted array in a buffer for which you have provided a pointer.

```
FUNCTION AEGetArray (theAEDescList: AEDescList;
                     arrayType: AEArrayType;
                     arrayPtr: AEArrayDataPointer;
                     maximumSize: Size;
                     VAR itemType: DescType; VAR itemSize: Size;
                     VAR itemCount: LongInt): OSErr;
```

`theAEDescList`
A descriptor list containing the desired array. If the array is of type `kAEDataArray`, `kAEPackedArray`, or `kAEHandleArray`, the descriptor list must be factored.

`arrayType` The Apple event array type to be converted. This is specified by one of the following constants: `kAEDataArray`, `kAEPackedArray`, `kAEHandleArray`, `kAEDescArray`, or `kAEKeyDescArray`.

`arrayPtr` A pointer to the buffer for storing the array.

`maximumSize`
The maximum length, in bytes, of the buffer for storing the array.

`itemType` For arrays of type `kAEDataArray`, `kAEPackedArray`, or `kAEHandleArray`, the `AEGetArray` function returns the descriptor type of the returned array items in this parameter.

`itemSize` For arrays of type `kAEDataArray` or `kAEPackedArray`, the `AEGetArray` function returns the size (in bytes) of the returned array items in this parameter.

`itemCount` The `AEGetArray` function returns the number of items in the resulting array in this parameter.

DESCRIPTION

The `AEGetArray` function uses a buffer identified by the pointer in the `arrayPtr` parameter to return the converted data for the Apple event array specified by the `theAEDescList` parameter. Even if the descriptor list that contains the array is factored, the converted data for each array item includes the data common to all the descriptor records in the list. The Apple Event Manager automatically reconstructs the common data for each item when you call `AEGetArray`.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEReplyNotArrived`	–1718	Reply has not yet arrived

SEE ALSO

For more information about data types and constants used with `AEGetArray`, see "Apple Event Array Data Types" on page 4-60.

For information about creating and factoring descriptor lists for Apple event arrays, see the description of `AECreateList` on page 5-29. For information about adding an Apple event array to a descriptor list, see the description of `AEPutArray` on page 5-32.

Getting Data and Keyword-Specified Descriptor Records Out of AE Records

The Apple Event Manager provides two functions, `AEGetKeyPtr` and `AEGetKeyDesc`, that allow you to get data and descriptor records out of an AE record or an Apple event record.

AEGetKeyPtr

You can use the `AEGetKeyPtr` function to get a pointer to a buffer that contains the data from a keyword-specified descriptor record. You can use this function to get data from an AE record or an Apple event record.

```
FUNCTION AEGetKeyPtr (theAERecord: AERecord;
                      theAEKeyword: AEKeyword;
                      desiredType: DescType;
                      VAR typeCode: DescType;
                      dataPtr: Ptr; maximumSize: Size;
                      VAR actualSize: Size): OSErr;
```

`theAERecord`
The AE record containing the desired data.

`theAEKeyword`
The keyword that specifies the desired descriptor record.

`desiredType`
The desired descriptor type for the data to be returned; if the requested data is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of `desiredType` is `typeWildCard`, no coercion is performed, and the descriptor type of returned data is the same as the descriptor type of the original data.

`typeCode` The descriptor type of the returned data.

`dataPtr` A pointer to the buffer for storing the data.

`maximumSize`
The maximum length, in bytes, of the data to be returned. You must allocate at least this amount of storage for the buffer specified by the `dataPtr` parameter.

`actualSize`
The length, in bytes, of the data for the keyword-specified descriptor record. If this value is larger than the value of the `maximumSize` parameter, not all of the data for the parameter was returned.

DESCRIPTION

The `AEGetKeyPtr` function uses a buffer to return the data from a keyword-specified Apple event parameter, which the function attempts to coerce to the descriptor type specified by the `desiredType` parameter.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested descriptor type
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEReplyNotArrived`	–1718	Reply has not yet arrived

AEGetKeyDesc

You can use the `AEGetKeyDesc` function to get the descriptor record for a keyword-specified descriptor record. You can use this function to get a descriptor record out of an AE record or an Apple event record.

```
FUNCTION AEGetKeyDesc (theAERecord: AERecord;
                       theAEKeyword: AEKeyword;
                       desiredType: DescType;
                       VAR result: AEDesc): OSErr;
```

`theAERecord`
The AE record containing the desired descriptor record.

`theAEKeyword`
The keyword that specifies the desired descriptor record.

`desiredType`
The desired descriptor type for the descriptor record to be returned; if the requested descriptor record is not of this type, the Apple Event Manager attempts to coerce it to this type. If the value of `desiredType` is `typeWildCard`, no coercion is performed, and the descriptor type of the returned descriptor record is the same as the descriptor type of the original descriptor record.

`result` A copy of the keyword-specified descriptor record, coerced to the descriptor type specified in the `desiredType` parameter.

DESCRIPTION

The `AEGetKeyDesc` function returns a copy of the descriptor record for a keyword-specified descriptor record. Your application should call the `AEDisposeDesc` function to dispose of the resulting descriptor record after your application has finished using it.

If `AEGetKeyDesc` returns a nonzero result code, it returns a descriptor record of descriptor type `typeNull`. A descriptor record of this type does not contain any data.

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	–1701	Descriptor record was not found
errAENotAEDesc	–1704	Not a valid descriptor record
errAEReplyNotArrived	–1718	Reply has not yet arrived

Requesting User Interaction

The Apple Event Manager provides three functions that allow you to set or request user interaction levels and to initiate user interaction when your application is the server application. The `AESetInteractionAllowed` and `AEGetInteractionAllowed` functions specify and return, respectively, the current user interaction preferences. Your application should call the `AEInteractWithUser` function before actually interacting with the user in response to an Apple event.

AESetInteractionAllowed

You can use the `AESetInteractionAllowed` function to specify your application's user interaction preferences for responding to an Apple event.

```
FUNCTION AESetInteractionAllowed
                                (level: AEInteractAllowed): OSErr;
```

`level` The user interaction level to be set.

DESCRIPTION

The `AESetInteractionAllowed` function sets the user interaction level for a server application's response to an Apple event. The `level` parameter must be one of three flags: `kAEInteractWithSelf`, `kAEInteractWithLocal`, or `kAEInteractWithAll`.

Specifying the `kAEInteractWithSelf` flag allows the server application to interact with the user in response to an Apple event only when the client application and server application are the same—that is, only when the application is sending the Apple event to itself.

Specifying the `kAEInteractWithLocal` flag allows the server application to interact with the user in response to an Apple event only if the client application is on the same computer as the server application; this is the default if the `AESetInteractionAllowed` function is not used.

Specifying the `kAEInteractWithAll` flag allows the server application to interact with the user in response to an Apple event sent from any client application on any computer.

RESULT CODE

`noErr`	0	No error

SEE ALSO

For more information about setting user preferences for a server application, see "Setting the Server Application's User Interaction Preferences" on page 4-48.

AEGetInteractionAllowed

You can use the `AEGetInteractionAllowed` function to get the current user interaction preferences for responding to an Apple event.

```
FUNCTION AEGetInteractionAllowed
                        (VAR level: AEInteractAllowed): OSErr;
```

`level` The current user interaction level, using the data type `AEInteractAllowed`.

```
TYPE AEInteractAllowed = (kAEInteractWithSelf,
                          kAEInteractWithLocal,
                          kAEInteractWithAll);
```

DESCRIPTION

The `AEGetInteractionAllowed` function returns, in the `level` parameter, a value that indicates the user interaction preferences for responding to an Apple event. The value, set by a previous call to `AESetInteractionAllowed`, is one of the following flags: `kAEInteractWithSelf`, `kAEInteractWithLocal`, or `kAEInteractWithAll`. The default value of `kAEInteractWithLocal` is returned if your application has not used `AESetInteractionAllowed` to set the interaction level explicitly.

The `kAEInteractWithSelf` flag indicates that the server application may interact with the user in response to an Apple event only when the client application and server application are the same—that is, only when the application is sending the Apple event to itself.

The `kAEInteractWithLocal` flag indicates that the server application may interact with the user in response to an Apple event only if the client application is on the same computer as the server application. This is the default if your application has not used the `AESetInteractionAllowed` function to set the interaction level explicitly.

The `kAEInteractWithAll` flag indicates that the server application may interact with the user in response to an Apple event sent from any client application on any computer.

RESULT CODE

`noErr`	0	No error

AEInteractWithUser

You can use the `AEInteractWithUser` function to initiate interaction with the user when your application is a server application responding to an Apple event.

```
FUNCTION AEInteractWithUser (timeOutInTicks: LongInt;
                             nmReqPtr: NMRecPtr;
                             idleProc: IdleProcPtr): OSErr;
```

`timeOutInTicks`
The amount of time (in ticks) that your handler is willing to wait for a response from the user. You can specify a number of ticks or use one of the following constants:

```
CONST kAEDefaultTimeout = -1; {value determined }
                              { by AEM}
      kNoTimeOut        = -2; {wait until reply }
                              { comes back}
```

`nmReqPtr` A pointer to a Notification Manager record provided by your application. You can specify `NIL` for this parameter to get the default notification handling provided by the Apple Event Manager.

`idleProc` A pointer to your application's idle function, which handles events while waiting for the Apple Event Manager to return control.

DESCRIPTION

Your application should call the `AEInteractWithUser` function before displaying a dialog box or alert box or otherwise interacting with the user in response to an Apple event. If the user interaction preference settings permit the application to come to the foreground, this function brings your application to the front, either directly or by posting a notification request.

Your application should normally pass a notification record in the `nmReqPtr` parameter rather than specifying `NIL` for default notification handling. If you specify `NIL`, the Apple Event Manager looks for an application icon with the ID specified by the application's bundle (`'BNDL'`) resource and the application's file reference (`'FREF'`) resource. The Apple Event Manager first looks for an `'SICN'` resource with the specified ID; if it can't find an `'SICN'` resource, it looks for the `'ICN#'` resource and compresses the icon to fit in the menu bar. The Apple Event Manager won't look for any members of an icon family other than the icon specified in the `'ICN#'` resource.

If the application doesn't have `'SICN'` or `'ICN#'` resources, or if it doesn't have a file reference resource, the Apple Event Manager passes `NIL` to the Notification Manager, and no icon appears in the upper-right corner of the screen. Therefore, if you want to display any icon other than those of type `'SICN'` or `'ICN#'`, you must specify a notification record as the second parameter to the `AEInteractWithUser` function.

Note
If you want the Notification Manager to use a color icon when it posts a notification request, you should provide a Notification Manager record that specifies a `'cicn'` resource. ◆

The `AEInteractWithUser` function checks whether the client application set the `kAENeverInteract` flag for the Apple event and, if so, returns an error. If not, then the `AEInteractWithUser` function checks the server application's preference set by the `AESetInteractionAllowed` function and compares it against the source of the Apple event—that is, whether it came from the same application, another process on the same computer, or a process running on another computer. The `AEInteractWithUser` function returns the `errAENoUserInteraction` result code if the user interaction preferences don't allow user interaction. If user interaction is allowed, the Apple Event Manager brings your application to the front, either directly or by posting a notification request. If `AEInteractWithUser` returns the `noErr` result code, then your application is in the foreground and is free to interact with the user.

RESULT CODES

`noErr`	0	No error
`errAETimeout`	–1712	Apple event timed out
`errAENoUserInteraction`	–1713	No user interaction allowed

SEE ALSO

For information about idle functions, see "Writing an Idle Function" on page 5-22.

For examples of the use of the `AEInteractWithUser` function, see "Interacting With the User," which begins on page 4-45.

Requesting More Time to Respond to Apple Events

The `AEResetTimer` function resets the timeout value for an Apple event to its starting value. A server application can call this function when it knows it cannot fulfill a client application's request (either by returning a result or by sending back a reply Apple event) before the client application is due to time out.

AEResetTimer

You can use the `AEResetTimer` function to reset the timeout value for an Apple event to its starting value.

```
FUNCTION AEResetTimer (reply: AppleEvent): OSErr;
```

`reply` The default reply for an Apple event, provided by the Apple Event Manager.

DESCRIPTION

When your application calls `AEResetTimer`, the Apple Event Manager for the server application uses the default reply to send a Reset Timer event to the client application; the Apple Event Manager for the client application's computer intercepts this Apple event and resets the client application's timer for the Apple event. (The Reset Timer event is never dispatched to a handler, so the client application does not need a handler for it.)

RESULT CODE

`noErr`	0	No error
`errAEReplyNotValid`	–1709	`AEResetTimer` was passed an invalid reply

Suspending and Resuming Apple Event Handling

When your application calls `AEProcessAppleEvent` and one of your event handlers is invoked, the Apple Event Manager normally assumes that your application has finished handling the event when the event handler returns. At this point, the Apple Event Manager disposes of the event. However, some applications, such as multi-session servers or any applications that implement their own internal event queueing, may need to defer handling of the event.

The `AESuspendTheCurrentEvent`, `AEResumeTheCurrentEvent`, `AESetTheCurrentEvent`, and `AEGetTheCurrentEvent` functions described in this section allow you to suspend and resume Apple event handling, specify the Apple event to be handled, and identify an Apple event that is currently being handled.

AESuspendTheCurrentEvent

You can use the `AESuspendTheCurrentEvent` function to suspend the processing of the Apple event that is currently being handled.

```
FUNCTION AESuspendTheCurrentEvent
                              (theAppleEvent: AppleEvent): OSErr;
```

`theAppleEvent`
The Apple event whose handling is to be suspended. Although the Apple Event Manager doesn't need this parameter to identify the Apple event currently being handled, providing it is a safeguard that you are suspending the correct Apple event.

DESCRIPTION

After a server application makes a successful call to the `AESuspendTheCurrentEvent` function, it is no longer required to return a result or a reply for the Apple event that was being handled. It can, however, return a result if it later calls the `AEResumeTheCurrentEvent` function to resume event processing.

The Apple Event Manager does not automatically dispose of Apple events that have been suspended or their default replies. (The Apple Event Manager does, however, automatically dispose of a previously suspended Apple event and its default reply if the server later resumes processing of the Apple event by calling the `AEResumeTheCurrentEvent` function.) If your server application does not resume processing of a suspended Apple event, it is responsible for using the `AEDisposeDesc` function to dispose of both the Apple event and its default reply when your application has finished using them.

SPECIAL CONSIDERATIONS

If your application suspends handling of an Apple event it sends to itself, the Apple Event Manager immediately returns from the `AESend` call with the error code `errAETimeout`, regardless of whether the `kAEQueueReply`, `kAEWaitReply`, or `kAENoReply` flags were set, even if the `timeout` parameter is set to `kNoTimeOut`. The routine calling `AESend` should take the timeout error as confirmation that the event was sent.

As with other calls to `AESend` that return a timeout error, the handler continues to process the event nevertheless. The handler's reply, if any, is provided in the reply event when the handling is completed. The Apple Event Manager provides no notification that the reply is ready. If no data has yet been placed in the reply event, the Apple Event Manager returns `errAEReplyNotArrived` when your application attempts to extract data from the reply.

RESULT CODE

`noErr`	0	No error

AEResumeTheCurrentEvent

You can use the `AEResumeTheCurrentEvent` function to inform the Apple Event Manager that your application wants to resume the handling of a previously suspended Apple event or that it has completed the handling of the Apple event.

```
FUNCTION AEResumeTheCurrentEvent
                            (theAppleEvent, reply: AppleEvent;
                             dispatcher: EventHandlerProcPtr;
                             handlerRefcon: LongInt): OSErr;
```

`theAppleEvent`
The Apple event to be resumed.

`reply` The default reply provided by the Apple Event Manager for the Apple event.

`dispatcher`
One of the following:

- a pointer to a routine for handling the event
- the `kAEUseStandardDispatch` constant, which tells the Apple Event Manager to dispatch the resumed event using the standard dispatching scheme it uses for other Apple events
- the `kAENoDispatch` constant, which tells the Apple Event Manager that the Apple event has been completely processed and need not be dispatched

`handlerRefcon`
If the value of the `dispatcher` parameter is not `kAEUseStandardDispatch`, this parameter is the reference constant passed to the handler when the handler is called. If the value of the `dispatcher` parameter is `kAEUseStandardDispatch`, the Apple Event Manager ignores the `handlerRefcon` parameter and instead passes the reference constant stored in the Apple event dispatch table entry for the Apple event. (You may wish to pass the same reference constant that is stored in the Apple event dispatch table. If so, call the `AEGetEventHandler` function.)

DESCRIPTION

When your application calls the `AEResumeTheCurrentEvent` function, the Apple Event Manager resumes handling the specified Apple event using the handler specified in the `dispatcher` parameter, if any. If `kAENoDispatch` is specified in the `dispatcher` parameter, `AEResumeTheCurrentEvent` simply informs the Apple Event Manager that the specified event has been handled.

SPECIAL CONSIDERATIONS

An Apple event handler that suspends an event should not immediately call `AEResumeTheCurrentEvent`, or else the handler will generate an error. Instead, the handler should return just after suspending the event.

When your application calls `AEResumeTheCurrentEvent` for an event that was not directly dispatched, the Apple Event Manager disposes of the event and the reply, just as it normally does, after the event handler returns to `AEProcessAppleEvent`. Make sure all processing involving the event or the reply has been completed before your application calls `AEResumeTheCurrentEvent`. Do not call `AEResumeTheCurrentEvent` for an event that was not suspended.

When your application calls `AEResumeTheCurrentEvent` for an event that was directly dispatched, your application is responsible for disposing of the original event and the reply, since it is acts as both the server and the client.

RESULT CODE

`noErr`	0	No error

AESetTheCurrentEvent

You can use the `AESetTheCurrentEvent` function to specify the Apple event to be handled.

```
FUNCTION AESetTheCurrentEvent (theAppleEvent: AppleEvent): OSErr;
```

`theAppleEvent`
The Apple event to be handled.

DESCRIPTION

There is usually no reason for your application to use the `AESetTheCurrentEvent` function. Instead of calling this function, your application should let the Apple Event Manager set the current Apple event through the dispatch tables.

If you need to avoid the dispatch tables, you must use the `AESetTheCurrentEvent` function only in the following way:

1. Your application suspends handling of an Apple event by calling the `AESuspendTheCurrentEvent` function.
2. Your application calls the `AESetTheCurrentEvent` function. This informs the Apple Event Manager that your application is handling the suspended Apple event. In this way, any routines that call the `AEGetTheCurrentEvent` function can ascertain which event is currently being handled.
3. When your application finishes handling the Apple event, it calls the `AEResumeTheCurrentEvent` function with the value `kAENoDispatch` to tell the Apple Event Manager that the event has been processed and need not be dispatched.

RESULT CODE

`noErr`	0	No error

AEGetTheCurrentEvent

You can use the `AEGetTheCurrentEvent` function to get the Apple event that is currently being handled.

```
FUNCTION AEGetTheCurrentEvent
                          (VAR theAppleEvent: AppleEvent): OSErr;
```

`theAppleEvent`
The Apple event that is currently being handled; if no Apple event is currently being handled, `AEGetTheCurrentEvent` returns a null descriptor record in this parameter.

DESCRIPTION

In many applications, the handling of an Apple event involves one or more long chains of calls to internal routines. The `AEGetTheCurrentEvent` function makes it unnecessary for these calls to include the current Apple event as a parameter; the routines can simply call `AEGetTheCurrentEvent` to get the current Apple event when it is needed.

You can also use the `AEGetTheCurrentEvent` function to make sure that no Apple event is currently being handled. For example, suppose your application always uses an application-defined routine to delete a file. That routine can first call `AEGetTheCurrentEvent` and delete the file only if `AEGetTheCurrentEvent` returns a null descriptor record (that is, only if no Apple event is currently being handled).

RESULT CODE

`noErr`	0	No error

Getting the Sizes and Descriptor Types of Descriptor Records

The Apple Event Manager provides four routines that allow you to get the sizes and descriptor types of descriptor records that are not part of an Apple event record. The `AESizeOfNthItem` function returns the size and descriptor type of a descriptor record in a descriptor list.The `AESizeOfKeyDesc` function returns the size and descriptor type of a keyword-specified descriptor record in an AE record. You can get the size and descriptor type of an Apple event parameter or Apple event attribute using the `AESizeOfParam` and `AESizeOfAttribute` functions.

AESizeOfNthItem

You can use the `AESizeOfNthItem` function to get the size and descriptor type of a descriptor record in a descriptor list.

```
FUNCTION AESizeOfNthItem (theAEDescList: AEDescList;
                          index: LongInt; VAR typeCode: DescType;
                          VAR dataSize: Size): OSErr;
```

`theAEDescList`
The descriptor list containing the descriptor record.

`index` The position of the descriptor record in the list (for example, 2 specifies the second descriptor record).

`typeCode` The descriptor type of the descriptor record.

`dataSize` The length (in bytes) of the data in the descriptor record.

RESULT CODES

`noErr`	0	No error
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAEReplyNotArrived`	–1718	Reply has not yet arrived

AESizeOfKeyDesc

You can use the `AESizeOfKeyDesc` function to get the size and descriptor type of a keyword-specified descriptor record in an AE record.

```
FUNCTION AESizeOfKeyDesc (theAERecord: AERecord;
                          theAEKeyword: AEKeyword;
                          VAR typeCode: DescType;
                          VAR dataSize: Size): OSErr;
```

`theAERecord`
The AE record containing the desired keyword-specified descriptor record.

`theAEKeyword`
The keyword that specifies the desired descriptor record.

`typeCode` The descriptor type of the keyword-specified descriptor record.

`dataSize` The length, in bytes of the data in the keyword-specified descriptor record.

RESULT CODES

noErr	0	No error
errAEDescNotFound	–1701	Descriptor record was not found
errAENotAEDesc	–1704	Not a valid descriptor record
errAEReplyNotArrived	–1718	Reply has not yet arrived

AESizeOfParam

You can use the `AESizeOfParam` function to get the size and descriptor type of an Apple event parameter.

```
FUNCTION AESizeOfParam (theAppleEvent: AppleEvent; theAEKeyword:
                        AEKeyword; VAR typeCode: DescType;
                        VAR dataSize: Size): OSErr;
```

`theAppleEvent`
The Apple event containing the parameter.

`theAEKeyword`
The keyword that specifies the desired parameter.

`typeCode` The descriptor type of the Apple event parameter.

`dataSize` The length, in bytes, of the data in the Apple event parameter.

RESULT CODES

noErr	0	No error
errAEDescNotFound	–1701	Descriptor record was not found
errAENotAEDesc	–1704	Not a valid descriptor record
errAEReplyNotArrived	–1718	Reply has not yet arrived

AESizeOfAttribute

You can use the `AESizeOfAttribute` function to get the size and descriptor type of an Apple event attribute.

```
FUNCTION AESizeOfAttribute (theAppleEvent: AppleEvent;
                            theAEKeyword: AEKeyword;
                            VAR typeCode: DescType;
                            VAR dataSize: Size): OSErr;
```

`theAppleEvent`
The Apple event containing the desired attribute.

`theAEKeyword`
The keyword that specifies the attribute.

`typeCode` The descriptor type of the attribute.

`dataSize` The length, in bytes, of the data in the attribute.

RESULT CODES

`noErr`	0	No error
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEReplyNotArrived`	–1718	Reply has not yet arrived

Deleting Descriptor Records

The Apple Event Manager provides three functions that allow you to delete descriptor records. The `AEDeleteItem`, `AEDeleteKeyDesc`, and `AEDeleteParam` functions allow you to delete descriptor records from a descriptor list, an AE record, and an Apple event parameter, respectively.

AEDeleteItem

You can use the `AEDeleteItem` function to delete a descriptor record from a descriptor list. All subsequent descriptor records will then move up one place.

```
FUNCTION AEDeleteItem (theAEDescList: AEDescList;
                       index: LongInt): OSErr;
```

`theAEDescList`
The descriptor list containing the descriptor record to be deleted.

`index` The position of the descriptor record to delete (for example, 2 specifies the second item).

RESULT CODES

`noErr`	0	No error
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

AEDeleteKeyDesc

You can use the `AEDeleteKeyDesc` function to delete a keyword-specified descriptor record from an AE record.

```
FUNCTION AEDeleteKeyDesc (theAERecord: AERecord;
                          theAEKeyword: AEKeyword): OSErr;
```

`theAERecord`
The AE record containing the keyword-specified descriptor record to be deleted.

`theAEKeyword`
The keyword that specifies the descriptor record to be deleted.

RESULT CODES

`noErr`	0	No error
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

AEDeleteParam

You can use the `AEDeleteParam` function to delete an Apple event parameter.

```
FUNCTION AEDeleteParam (theAppleEvent: AppleEvent;
                        theAEKeyword: AEKeyword): OSErr;
```

`theAppleEvent`
The Apple event containing the parameter to be deleted.

`theAEKeyword`
The keyword that specifies the parameter to be deleted.

RESULT CODES

`noErr`	0	No error
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

Deallocating Memory for Descriptor Records

The `AEDisposeDesc` function deallocates the memory used by a descriptor record. Because all Apple event structures (except for keyword-specified descriptor records) are descriptor records, you can use `AEDisposeDesc` for any of them.

AEDisposeDesc

You can use the `AEDisposeDesc` function to deallocate the memory used by a descriptor record.

```
FUNCTION AEDisposeDesc (VAR theAEDesc: AEDesc): OSErr;
```

`theAEDesc` The descriptor record to deallocate. The function returns a null descriptor record in this parameter. If you pass a null descriptor record in this parameter, `AEDisposeDesc` returns `noErr`.

RESULT CODE

`noErr`	0	No error

SEE ALSO

For more information about using `AEDisposeDesc`, see "Disposing of Apple Event Data Structures," which begins on page 4-39.

Coercing Descriptor Types

The Apple Event Manager provides two functions that allow you to coerce descriptor types. The `AECoercePtr` function takes a pointer to data and a desired descriptor type and attempts to coerce the data to a descriptor record of the desired descriptor type. The `AECoerceDesc` function attempts to coerce the data in an existing descriptor record to another descriptor type.

AECoercePtr

You can use the `AECoercePtr` function to coerce data to a desired descriptor type. If successful, it creates a descriptor record containing the newly coerced data.

```
FUNCTION AECoercePtr (typeCode: DescType; dataPtr: Ptr;
                      dataSize: Size; toType: DescType;
                      VAR result: AEDesc): OSErr;
```

`typeCode` The descriptor type of the source data.

`dataPtr` A pointer to the data to be coerced.

`dataSize` The length, in bytes, of the data to be coerced.

`toType` The desired descriptor type of the resulting descriptor record.

`result` The resulting descriptor record.

DESCRIPTION

The `AECoercePtr` function creates a new descriptor record by coercing the specified data to a descriptor record of the specified descriptor type. You should use the `AEDisposeDesc` function to dispose of the resulting descriptor record once you are finished using it.

If `AECoercePtr` returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested descriptor type

SEE ALSO

For a description of the `AEDisposeDesc` function, see page 4-94.

AECoerceDesc

You can use the `AECoerceDesc` function to coerce the data in a descriptor record to another descriptor type.

```
FUNCTION AECoerceDesc (theAEDesc: AEDesc; toType: DescType;
                       VAR result: AEDesc): OSErr;
```

`theAEDesc`	The descriptor record whose data is to be coerced.
`toType`	The desired descriptor type of the resulting descriptor record.
`result`	The resulting descriptor record.

DESCRIPTION

The `AECoerceDesc` function attempts to create a new descriptor record by coercing the specified descriptor record. Your application is responsible for using the `AEDisposeDesc` function to dispose of the resulting descriptor record once you are finished using it.

If `AECoerceDesc` returns a nonzero result code, it returns a null descriptor record (a descriptor record of type `typeNull`, which does not contain any data) unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to requested descriptor type

SEE ALSO

For a list of the descriptor types for which the Apple Event Manager provides coercions, see Table 4-1, which begins on page 4-43.

Creating and Managing the Coercion Handler Dispatch Tables

The Apple Event Manager provides three functions that allow you to create and manage the coercion handler dispatch tables. The `AEInstallCoercionHandler` function installs a coercion handler routine in either the application or system coercion dispatch table. The `AEGetCoercionHandler` function returns the handler for a specified descriptor type coercion. The `AERemoveCoercionHandler` function removes a coercion handler from either the application or system coercion table.

AEInstallCoercionHandler

You can use the `AEInstallCoercionHandler` function to install a coercion handler routine in either the application or system coercion handler dispatch table.

```
FUNCTION AEInstallCoercionHandler (fromType: DescType;
                                   toType: DescType;
                                   handler: ProcPtr;
                                   handlerRefcon: LongInt;
                                   fromTypeIsDesc: Boolean;
                                   isSysHandler: Boolean): OSErr;
```

`fromType` The descriptor type of the data coerced by the handler.

`toType` The descriptor type of the resulting data. If there was already an entry in the specified coercion handler table for the same source descriptor type and result descriptor type, the existing entry is replaced.

`handler` A pointer to the coercion handler. Note that a handler in the system coercion table must reside in the system heap; thus, if the value of the `isSysHandler` parameter is `TRUE`, the `handler` parameter should point to a location in the system heap. Otherwise, if you put your system handler code in your application heap, you should use `AERemoveCoercionHandler` to remove the handler when your application quits.

`handlerRefcon`
A reference constant passed by the Apple Event Manager to the handler each time the handler is called. If your handler doesn't expect a reference constant, use 0 as the value of this parameter.

`fromTypeIsDesc`
Specifies the form of the data to be coerced. If the value of this parameter is `TRUE`, the coercion handler expects the data to be passed as a descriptor record. If the value is `FALSE`, the coercion handler expects a pointer to the data. Because it is more efficient for the Apple Event Manager to provide a pointer to data than to a descriptor record, all coercion routines should accept a pointer to data if possible.

`isSysHandler`
Specifies the coercion table to which the handler is added. If the value of this parameter is `TRUE`, the handler is added to the system coercion table and made available to all applications. If the value is `FALSE`, the handler is added to the application coercion table. Note that a handler in the system coercion table must reside in the system heap; thus, if the value of the `isSysHandler` parameter is `TRUE`, the handler parameter must point to a location in the system heap.

DESCRIPTION

Before using `AEInstallCoercionHandler` to install a handler for a particular descriptor type into the system coercion handler dispatch table, use the `AEGetCoercionHandler` function to determine whether the table already contains a coercion handler for that descriptor type. If an entry exists, `AEGetCoercionHandler` returns a reference constant and a pointer to that handler. Chain these to your coercion handler by providing, in the `handlerRefcon` parameter of `AEInstallCoercionHandler`, pointers to the previous handler and its reference constant. If your coercion handler returns the error `errAECoercionFail`, use these pointers to call the previous handler. If you remove your system coercion handler, be sure to reinstall the chained handlers.

SPECIAL CONSIDERATIONS

Before an application calls a system coercion handler, system software has set up the A5 register for the calling application. For this reason, if you provide a system coercion handler, it should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system handler may crash.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone

AEGetCoercionHandler

You can use the `AEGetCoercionHandler` function to get the handler for a specified descriptor type coercion.

```
FUNCTION AEGetCoercionHandler (fromType: DescType;
                               toType: DescType;
                               VAR handler: ProcPtr;
                               VAR handlerRefcon: LongInt;
                               VAR fromTypeIsDesc: Boolean;
                               isSysHandler: Boolean): OSErr;
```

`fromType` The descriptor type of the data coerced by the handler.

`toType` The descriptor type of the resulting data.

`handler` A pointer to the desired coercion handler.

`handlerRefcon`
The reference constant for the desired handler. The Apple Event Manager passes this reference constant to the handler each time the handler is called.

`fromTypeIsDesc`
If the `AEGetCoercionHandler` function returns `TRUE` in this parameter, the coercion handler expects the data to be passed as a descriptor record. If the function returns `FALSE`, the coercion handler expects a pointer to the data.

`isSysHandler`
Specifies the coercion table from which to get the handler. If the value of this parameter is `TRUE`, the handler is taken from the system coercion table. If the value is `FALSE`, the handler is taken from the application coercion table.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAEHandlerNotFound`	–1717	No coercion handler found

AERemoveCoercionHandler

You can use the `AERemoveCoercionHandler` function to remove a coercion handler from either the application or system coercion handler dispatch table.

```
FUNCTION AERemoveCoercionHandler (fromType: DescType;
                                  toType: DescType;
                                  handler: ProcPtr;
                                  isSysHandler: Boolean): OSErr;
```

`fromType` The descriptor type of the data coerced by the handler.

`toType` The descriptor type of the resulting data.

`handler` A pointer to the coercion handler. Although the `fromType` and `toType` parameters would be sufficient to identify the handler to be removed, providing the `handler` parameter is a safeguard to ensure that you remove the correct handler.

`isSysHandler`
The coercion table from which to remove the handler. If the value of this parameter is `TRUE`, the handler is removed from the system coercion table. If the value is `FALSE`, the handler is removed from the application coercion dispatch table.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAEHandlerNotFound`	–1717	No coercion handler found

Creating and Managing the Special Handler Dispatch Tables

The Apple Event Manager provides three functions that allow you to create and manage the special handler dispatch tables. The `AEInstallSpecialHandler` function installs an entry for a special handler in either the application or system special handler dispatch table. The `AEGetSpecialHandler` function returns the handler for a specified special handler. The `AERemoveSpecialHandler` function removes a special handler from either the application or system special handler dispatch table.

You can also use the `AEInstallSpecialHandler`, `AEGetSpecialHandler`, and `AERemoveSpecialHandler` functions to install, get, and remove object callback functions—including system object callback functions, which cannot be installed with the `AESetObjectCallbacks` function. When calling any of these three functions, use one of the following constants as the value of the `functionClass` parameter to specify the object callback function:

Object callback function	Constant
Object-counting function	`keyAECountProc`
Object-comparison function	`keyAECompareProc`
Token disposal function	`keyDiposeTokenProc`
Error callback function	`keyAEGetErrDescProc`
Mark token function	`keyAEMarkTokenProc`
Object-marking function	`keyAEMarkProc`
Mark-adjusting function	`keyAEAdjustMarksProc`

You can also use the `AERemoveSpecialHandler` function to disable all the Apple Event Manager routines that support object specifier records. To do this, specify the constant `keySelectProc` in the `functionClass` parameter as described on page 4-102.

AEInstallSpecialHandler

You can use the `AEInstallSpecialHandler` function to install a special handler in either the application or system special handler dispatch table.

```
FUNCTION AEInstallSpecialHandler (functionClass: AEKeyword;
                                  handler: ProcPtr;
                                  isSysHandler: Boolean): OSErr;
```

`functionClass`
The keyword for the special handler that is installed. The `keyPreDispatch` constant identifies a handler with the same parameters as an Apple event handler called immediately before the Apple Event Manager dispatches an Apple event. Any of the constants for object callback functions listed above can also be specified in this parameter. If there was already an entry in the specified special handler dispatch table for the same value of `functionClass`, it is replaced.

`handler` A pointer to the special handler. Note that a handler in the system special handler dispatch table must reside in the system heap; thus, if the value of the `isSysHandler` parameter is `TRUE`, the `handler` parameter should point to a location in the system heap. Otherwise, if you put your system handler code in your application heap, use `AERemoveSpecialHandler` to remove the handler when your application quits.

`isSysHandler`
The special handler dispatch table to which to add the handler. If the value of this parameter is `TRUE`, the handler is added to the system handler dispatch table and made available to all applications. If the value is `FALSE`, the handler is added to the application handler table.

DESCRIPTION

The `AEInstallSpecialHandler` function creates an entry in either your application's special handler dispatch table or the system special handler dispatch table. You must supply parameters that specify the keyword for the special handler that is installed, the handler routine, and whether the handler is to be added to the system special handler dispatch table or your application's special handler dispatch table.

SPECIAL CONSIDERATIONS

Before an application calls a system special handler, system software has set up the A5 register for the calling application. For this reason, a system special handler should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system handler may crash.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Parameter error (handler pointer is `NIL` or odd)
`memFullErr`	–108	Not enough room in heap zone
`errAENotASpecialFunction`	–1714	Wrong keyword for a special function

AEGetSpecialHandler

You can use the `AEGetSpecialHandler` function to get a specified special handler.

```
FUNCTION AEGetSpecialHandler (functionClass: AEKeyword;
                              VAR handler: ProcPtr;
                              isSysHandler: Boolean): OSErr;
```

`functionClass`
The keyword for the special handler that is installed. The `keyPreDispatch` constant identifies a handler with the same parameters as an Apple event handler that is called immediately before the Apple Event Manager dispatches an Apple event. Any of the constants for object callback functions listed on page 4-100 can also be specified in this parameter.

`handler` A pointer to the special handler.

`isSysHandler`
Specifies the special handler dispatch table from which to get the handler. If the value of this parameter is `TRUE`, the handler is taken from the system special handler dispatch table. If the value is `FALSE`, the handler is taken from the application's special handler dispatch table.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAENotASpecialFunction`	–1714	Wrong keyword for a special handler

AERemoveSpecialHandler

You can use the `AERemoveSpecialHandler` function to remove a handler from a special handler table.

```
FUNCTION AERemoveSpecialHandler (functionClass: AEKeyword;
                                 handler: ProcPtr;
                                 isSysHandler: Boolean): OSErr;
```

`functionClass`
The keyword for the special handler to be removed. In addition to the constants for object callback functions listed on page 4-100, two other values are allowed for the `functionClass` parameter: `keyPreDispatch` and `keySelectProc`. The `keyPreDispatch` constant identifies a handler with the same parameters as an Apple event handler that is called immediately before the Apple Event Manager dispatches an Apple event. The `keySelectProc` constant indicates that you want to disable the Object Support Library—that is, all the routines described in the chapter "Resolving and Creating Object Specifier Records" in this book (see the description that follows for more information).

`handler` A pointer to the special handler to be removed. Although the `functionClass` parameter would be sufficient to identify the handler to be removed, providing the `handler` parameter is a safeguard that you remove the correct handler.

`isSysHandler`
Specifies the special handler dispatch table from which to remove the handler. If the value of this parameter is `TRUE`, the handler is taken from the system special handler dispatch table. If the value is `FALSE`, the handler is removed from the application special handler dispatch table.

DESCRIPTION

In addition to using the `AERemoveSpecialHandler` function to remove specific special handlers, you can use the function to disable, within your application only, all Apple Event Manager routines that support Apple event objects—that is, all the routines available to your application as a result of linking the Object Support Library (OSL) and calling the `AEObjectInit` function.

An application that expects its copy of the OSL to move after it is installed—for example, an application that keeps it in a stand-alone code resource—would need to do this. When an application calls `AEObjectInit` to initialize the OSL, the OSL installs the addresses of its routines as extensions to the pack. If those routines move, the addresses become invalid.

To disable the OSL, you should pass the keyword `keySelectProc` in the `functionClass` parameter, `NIL` in the `handler` parameter, and `FALSE` in the `isSysHandler` parameter. Once you have called the `AERemoveSpecialHandler` function with these parameters, subsequent calls by your application to any of the Apple Event Manager routines that support Apple event objects will return errors. To initialize the OSL after disabling it with the `AERemoveSpecialHandler` function, your application must call `AEObjectInit` again.

If you expect to initialize the OSL and disable it several times, you should call `AERemoveObjectAccessor` to remove your application's object accessor functions from your application's object accessor dispatch table before you call `AERemoveSpecialHandler`.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAENotASpecialFunction`	–1714	Wrong keyword for a special function

Getting Information About the Apple Event Manager

The `AEManagerInfo` routine allows you to get two kinds of information related to Apple events on the current computer: the number of processes currently recording Apple events and the version of the Apple Event Manager. If you decide to make your application recordable, this information may be useful when your application is responding to Apple events that it sends to itself.

You can find out whether the Apple Event Manager is available in system software by using the `Gestalt` function. See page 4-4 for details.

AEManagerInfo

You can use the `AEManagerInfo` function to obtain information about the version of the Apple Event Manager currently available or the number of processes that are currently recording Apple events. This function is available only in version 1.01 and later versions of the Apple Event Manager.

```
FUNCTION AEManagerInfo (keyword: AEKeyword;
                        VAR result: LongInt): OSErr;
```

`keyword` A value that determines what kind of information `AEManagerInfo` returns. The value can be represented by one of these constants:

```
CONST keyAERecorderCount         = 'recr';
      keyAEVersion               = 'vers';
```

`result` If the value of the `keyword` parameter is `keyAERecorderCount`, this parameter is an integer that indicates the number of processes that are currently recording Apple events. If the value of the `keyword` parameter is `keyAEVersion`, this parameter is an integer that provides information about the version of the Apple Event Manager available on the current computer, using the same format as a `'vers'` resource.

RESULT CODE

`noErr`	0	No error

SEE ALSO

For information about using the `AEManagerInfo` function to check whether Apple event recording is on or not, see the chapter "Recording Apple Events" in this book.

For information about using `Gestalt` to determine whether the Apple Event Manager is available, see "Handling Apple Events" on page 4-4.

For information about the `'vers'` resource, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Application-Defined Routines

For each Apple event your application supports, you must provide an Apple event handler. The `AEProcessAppleEvent` function calls one of your Apple event handlers when it processes an Apple event. An Apple event handler (`MyEventHandler`) should perform any action described by the Apple event, add parameters to the reply Apple event if appropriate, and return a result code.

You can also provide your own coercion handlers to coerce data to descriptor types other than those for which the Apple Event Manager provides coercion handling. The `MyCoercePtr` function accepts a pointer to data and returns a descriptor record, and the `MyCoerceDesc` function accepts a descriptor record and returns a descriptor record.

MyEventHandler

An Apple event handler has the following syntax:

```
FUNCTION MyEventHandler (theAppleEvent: AppleEvent;
                         reply: AppleEvent;
                         handlerRefcon: LongInt): OSErr;
```

`theAppleEvent`
The Apple event to handle.

`reply` The default reply Apple event provided by the Apple Event Manager.

`handlerRefcon`
The reference constant stored in the Apple event dispatch table for the Apple event.

DESCRIPTION

An Apple event handler should extract any parameters and attributes from the Apple event, perform the requested action, and add parameters to the reply Apple event if appropriate.

Your handler should always set its function result to `noErr` if it successfully handles the Apple event. If an error occurs, your handler should return either `errAEEventNotHandled` or some other nonzero result code. If the error occurs because your application cannot understand the event, return `errAEEventNotHandled`, in case a handler in the system special handler or system Apple event dispatch tables might be able to handle the event. If the error occurs because the event is impossible to handle as specified, return the result code returned by whatever function caused the failure, or whatever other result code is appropriate.

For example, suppose your application receives a Get Data event that requests the name of the current printer, and your application cannot handle such an event. In this situation, you should return `errAEEventNotHandled` in case another handler available to the Apple Event Manager can handle the event. This strategy allows users to take advantage of system capabilities from within your application via system handlers.

However, if your application cannot handle a Get Data event that requests the fifth paragraph in a document because the document contains only four paragraphs, you should return some other nonzero error, because further attempts to handle the event are pointless.

If your Apple event handler calls the `AEResolve` function and `AEResolve` calls an object accessor function in the system object accessor dispatch table, your Apple event handler may not recognize the descriptor type of the token returned by the function. In this case, your handler should return the result code `errAEUnknownObjectType`. When your handler returns this result code, the Apple Event Manager attempts to locate a system Apple event handler that can recognize the token.

SEE ALSO

For more information about Apple event handlers, see "Writing Apple Event Handlers" on page 4-33.

For a discussion of the dispatching of object accessor functions and the use of the result code `errAEUnknownObjectType`, see "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

MyCoercePtr

A coercion handler that accepts a pointer to data has the following syntax:

```
FUNCTION MyCoercePtr (typeCode: DescType; dataPtr: Ptr;
                      dataSize: Size; toType: DescType;
                      handlerRefcon: LongInt;
                      VAR result: AEDesc): OSErr;
```

`typeCode` The descriptor type of the original data.

`dataPtr` A pointer to the data to coerce.

`dataSize` The length, in bytes, of the data to coerce.

`toType` The desired descriptor type for the resulting descriptor record.

`handlerRefcon`
A reference constant that is stored in the coercion dispatch table entry for the handler and passed to the handler by the Apple Event Manager whenever the handler is called.

`result` The resulting descriptor record.

DESCRIPTION

Your coercion handler should coerce the data to the desired descriptor type and return the resulting data in the descriptor record specified by the `result` parameter. Your handler should return the `noErr` result code if your handler successfully performs the coercion, and a nonzero result code otherwise.

SEE ALSO

For more information, see "Writing and Installing Coercion Handlers" on page 4-41.

MyCoerceDesc

A coercion handler that accepts a descriptor record has the following syntax:

```
FUNCTION MyCoerceDesc (theAEDesc: AEDesc; toType: DescType;
                       handlerRefcon: LongInt;
                       VAR result: AEDesc): OSErr;
```

`theAEDesc` The descriptor record that contains the data to be coerced.

`toType` The desired descriptor type for the resulting descriptor record.

`handlerRefcon`
A reference constant that is stored in the coercion dispatch table entry for the handler and passed to the handler by the Apple Event Manager whenever the handler is called.

`result` The resulting descriptor record.

DESCRIPTION

Your coercion handler should coerce the data in the descriptor record to the desired descriptor type and return the resulting data in the descriptor record specified by the `result` parameter. Your handler should return an appropriate result code.

SEE ALSO

For more information, see “Writing and Installing Coercion Handlers” on page 4-41.

Summary of Responding to Apple Events

Pascal Summary

Constants

```
CONST
   gestaltAppleEventsAttr     = 'evnt';   {selector for Apple events}
   gestaltAppleEventsPresent  = 0;        {if this bit is set, then Apple }
                                          { Event Manager is available}

   {Apple event descriptor types}
   typeBoolean             = 'bool';      {1-byte Boolean value}
   typeChar                = 'TEXT';      {unterminated string}
   typeSMInt               = 'shor';      {16-bit integer}
   typeInteger             = 'long';      {32-bit integer}
   typeSMFloat             = 'sing';      {SANE single}
   typeFloat               = 'doub';      {SANE double}
   typeLongInteger         = 'long';      {32-bit integer}
   typeShortInteger        = 'shor';      {16-bit integer}
   typeLongFloat           = 'doub';      {SANE double}
   typeShortFloat          = 'sing';      {SANE single}
   typeExtended            = 'exte';      {SANE extended}
   typeComp                = 'comp';      {SANE comp}
   typeMagnitude           = 'magn';      {unsigned 32-bit integer}
   typeAEList              = 'list';      {list of descriptor records}
   typeAERecord            = 'reco';      {list of keyword-specified }
                                          { descriptor records}
   typeAppleEvent          = 'aevt';      {Apple event record}
   typeTrue                = 'true';      {TRUE Boolean value}
   typeFalse               = 'fals';      {FALSE Boolean value}
   typeAlias               = 'alis';      {alias record}
   typeEnumerated          = 'enum';      {enumerated data}
   typeType                = 'type';      {four-character code for }
                                          { event class or event ID}
   typeAppParameters       = 'appa';      {Process Manager launch parameters}
   typeProperty            = 'prop';      {Apple event property}
   typeFSS                 = 'fss ';      {file system specification}
```

```
typeKeyword             = 'keyw';      {Apple event keyword}
typeSectionH            = 'sect';      {handle to a section record}
typeWildCard            = '****';      {matches any type}
typeApplSignature       = 'sign';      {application signature}
typeSessionID           = 'ssid';      {session reference number}
typeTargetID            = 'targ';      {target ID record}
typeProcessSerialNumber = 'psn ';      {process serial number}
typeNull                = 'null';      {NULL or nonexistent data}

{keywords for Apple event parameters}
keyDirectObject         = '----';      {direct parameter}
keyErrorNumber          = 'errn';      {error number parameter}
keyErrorString          = 'errs';      {error string parameter}
keyProcessSerialNumber  = 'psn ';      {process serial number param}

{keywords for Apple event attributes}
keyTransactionIDAttr    = 'tran';      {transaction ID}
keyReturnIDAttr         = 'rtid';      {return ID}
keyEventClassAttr       = 'evcl';      {event class}
keyEventIDAttr          = 'evid';      {event ID}
keyAddressAttr          = 'addr';      {address of target or }
                                       { client application}
keyOptionalKeywordAttr  = 'optk';      {list of optional parameters }
                                       { for the Apple event}
keyTimeoutAttr          = 'timo';      {number of ticks the client }
                                       { will wait}
keyInteractLevelAttr    = 'inte';      {settings to allow Apple Event }
                                       { Manager to bring server }
                                       { to foreground}
keyEventSourceAttr      = 'esrc';      {nature of source }
                                       { application}
keyMissedKeywordAttr    = 'miss';      {first required parameter }
                                       { remaining in an Apple event}
keyOriginalAddressAttr  = 'from';      {address of original source; }
                                       { available only in version }
                                       { 1.01 and later versions of }
                                       { the Apple Event Manager}

{keywords for special handlers}
keyPreDispatch          = 'phac';      {identifies a handler routine }
                                       { called immediately before the }
                                       { Apple Event Manager dispatches }
                                       { an Apple event}
```

```
keySelectProc           = 'selh';      {selector used with }
                                       { AERemoveSpecialHandler to }
                                       { disable the OSL}

{keywords for use with AEManagerInfo; available only in version }
{ 1.0.1 and later versions of the Apple Event Manager}
keyAERecorderCount      = 'recr';      {keyword for recording info}
keyAEVersion            = 'vers';      {keyword for version info}

{event class}
kCoreEventClass         = 'aevt';      {event class for required Apple }
                                       { events}

{event IDs for required Apple events}
kAEOpenApplication      = 'oapp';      {event ID for Open }
                                       { Application event}
kAEOpenDocuments        = 'odoc';      {event ID for Open Documents event}
kAEPrintDocuments       = 'pdoc';      {event ID for Print Documents }
                                       { event}
kAEQuitApplication      = 'quit';      {event ID for Quit Application }
                                       { event}
kAEAnswer               = 'ansr';      {event ID for Apple event replies}
kAEApplicationDied      = 'obit';      {event ID for Application Died }
                                       { event}

{constants for setting the sendMode parameter of AESend}
kAENoReply              = $00000001;   {client doesn't want reply}
kAEQueueReply           = $00000002;   {client wants server to }
                                       { reply in event queue}
kAEWaitReply            = $00000003;   {client wants a reply and }
                                       { will give up processor}
kAENeverInteract        = $00000010;   {server application should }
                                       { not interact with user }
                                       { for this Apple event}
kAECanInteract          = $00000020;   {server may interact with }
                                       { user for this Apple event }
                                       { to supply information}

kAEAlwaysInteract       = $00000030;   {server may interact with user }
                                       { for this Apple event even if }
                                       { no information is required}
kAECanSwitchLayer       = $00000040;   {server should come directly }
                                       { to foreground when appropriate}
```

```
kAEDontReconnect        = $00000080;   {don't reconnect if there }
                                       { is a PPC session closed error}
kAEWantReceipt          = nReturnReceipt; {client wants return }
                                       { receipt}
kAEDontRecord           = $00001000;   {don't record this event}
kAEDontExecute          = $00002000;   {don't excecute this event}

{constants for setting the sendPriority parameter of AESend}
kAENormalPriority       = $00000000;   {put event at the back of }
                                       { event queue}
kAEHighPriority         = nAttnMsg;    {put event at the front of }
                                       { the event queue}

{event IDs for recording events; available only in version 1.01 and }
{ later versions of the Apple Event Manager}
kAEStartRecording       = 'reca';      {event ID for Start Recording }
                                       { event}
kAEStopRecording        = 'recc';      {event ID for Stop Recording }
                                       { event}
kAENotifyStartRecording = 'rec1';      {event ID for Recording On event}
kAENotifyStopRecording  = 'rec0';      {event ID for Recording Off event}
kAENotifyRecording      = 'recr';      {event ID for Receive Recordable }
                                       { Event event}

{constant for the returnID parameter of AECreateAppleEvent}
kAutoGenerateReturnID   = -1;          {tells Apple Event Manager to }
                                       { generate a unique return ID}

{constant for transaction IDs}
kAnyTransactionID       = 0;           {the Apple event is not }
                                       { part of a transaction}

{constants for timeout durations}
kAEDefaultTimeout       = -1;          {use default timeout value}
kNoTimeOut              = -2;          {never time out}

{constants for the dispatcher parameter of AEResumeTheCurrentEvent}
kAENoDispatch           = 0;           {don't redispatch the Apple event}
kAEUseStandardDispatch  = -1;          {redispatch the Apple event }
                                       { by using its entry in the }
                                       { Apple event dispatch table}
```

Data Types

```
TYPE
   AEEventClass =
      PACKED ARRAY[1..4] OF Char;            {event class for a high-level }
                                             { event}
   AEEventID =
      PACKED ARRAY[1..4] OF Char;            {event ID for a high-level }
                                             { event}

   AEKeyword =
      PACKED ARRAY[1..4] OF Char;            {keyword for a descriptor }
                                             { record}

   DescType                   = ResType;     {descriptor type}

   AEDesc =                                  {descriptor record}
   RECORD
      descriptorType:         DescType;      {type of data being passed}
      dataHandle:             Handle;        {handle to data being passed}
   END;

   AEKeyDesc =                               {keyword-specified }
   RECORD                                    { descriptor record}
      descKey:                AEKeyword;     {keyword}
      descContent:            AEDesc;        {descriptor record}
   END;

   AEAddressDesc              = AEDesc;      {address descriptor record}

   AEDescList                 = AEDesc;      {list of descriptor records}

   AERecord                   = AEDescList;  {list of keyword-specified }
                                             { descriptor records}

   AppleEvent                 = AERecord;    {list of attributes and }
                                             { parameters necessary for }
                                             { an Apple event}

   AESendMode                 = LongInt;     {flags that determine how }
                                             { an Apple event is sent}

   AESendPriority             = Integer;     {send priority of an Apple }
                                             { event}
```

```
AEInteractAllowed = (kAEInteractWithSelf, kAEInteractWithLocal,
                     kAEInteractWithAll); {what processes may }
                                          { interact with the user}

AEEventSource = (kAEUnknownSource, kAEDirectCall, kAESameProcess,
                  kAELocalProcess, kAERemoteProcess);
                                          {the source of an Apple }
                                          { event}

AEArrayType = (kAEDataArray, kAEPackedArray, kAEHandleArray,
               kAEDescArray, kAEKeyDescArray);
                                          {type of an Apple event array}
AEArrayData =
RECORD                                    {data for an Apple event array}
   CASE AEArrayType OF
   kAEDataArray:
      (AEDataArray: ARRAY[0..0] OF Integer);
   kAEPackedArray:
      (AEPackedArray: PACKED ARRAY[0..0] OF Char);
   kAEHandleArray:
      (AEHandleArray: ARRAY[0..0] OF Handle);
   kAEDescArray:
      (AEDescArray: ARRAY[0..0] OF AEDesc);
   kAEKeyDescArray:
      (AEKeyDescArray: ARRAY[0..0] OF AEKeyDesc);
END;

AEArrayDataPointer = ^AEArrayData;

EventHandlerProcPtr = ProcPtr;            {pointer to an Apple event }
                                          { handler}
IdleProcPtr = ProcPtr;                    {pointer to an application's }
                                          { idle function}
EventFilterProcPtr = ProcPtr;             {pointer to an application's }
                                          { filter function}
```

Routines for Responding to Apple Events

Creating and Managing the Apple Event Dispatch Tables

```
FUNCTION AEInstallEventHandler
                            (theAEEventClass: AEEventClass;
                             theAEEventID: AEEventID;
                             handler: EventHandlerProcPtr;
                             handlerRefcon: LongInt;
                             isSysHandler: Boolean): OSErr;
FUNCTION AEGetEventHandler  (theAEEventClass: AEEventClass;
                             theAEEventID: AEEventID;
                             VAR handler: EventHandlerProcPtr;
                             VAR handlerRefcon: LongInt;
                             isSysHandler: Boolean): OSErr;
FUNCTION AERemoveEventHandler
                            (theAEEventClass: AEEventClass; theAEEventID:
                             AEEventID; handler: EventHandlerProcPtr;
                             isSysHandler: Boolean): OSErr;
```

Dispatching Apple Events

```
FUNCTION AEProcessAppleEvent
                            (theEventRecord: EventRecord): OSErr;
```

Getting Data or Descriptor Records Out of Apple Event Parameters and Attributes

```
FUNCTION AEGetParamPtr      (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword;
                             desiredType: DescType;
                             VAR typeCode: DescType;
                             dataPtr: Ptr; maximumSize: Size;
                             VAR actualSize: Size): OSErr;
FUNCTION AEGetParamDesc     (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword; desiredType: DescType;
                             VAR result: AEDesc): OSErr;
FUNCTION AEGetAttributePtr  (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword; desiredType: DescType;
                             VAR typeCode: DescType;
                             dataPtr: Ptr; maximumSize: Size;
                             VAR actualSize: Size): OSErr;
FUNCTION AEGetAttributeDesc (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword; desiredType: DescType;
                             VAR result: AEDesc): OSErr;
```

Counting the Items in Descriptor Lists

```
FUNCTION AECountItems       (theAEDescList: AEDescList;
                             VAR theCount: LongInt): OSErr;
```

Getting Items From Descriptor Lists

```
FUNCTION AEGetNthPtr        (theAEDescList: AEDescList; index: LongInt;
                             desiredType: DescType;
                             VAR theAEKeyword: AEKeyword;
                             VAR typeCode: DescType; dataPtr: Ptr;
                             maximumSize: Size;
                             VAR actualSize: Size): OSErr;
FUNCTION AEGetNthDesc       (theAEDescList: AEDescList; index: LongInt;
                             desiredType: DescType;
                             VAR theAEKeyword: AEKeyword;
                             VAR result: AEDesc): OSErr;
FUNCTION AEGetArray         (theAEDescList: AEDescList;
                             arrayType: AEArrayType;
                             arrayPtr: AEArrayDataPointer;
                             maximumSize: Size;
                             VAR itemType: DescType; VAR itemSize: Size;
                             VAR itemCount: LongInt): OSErr;
```

Getting Data and Keyword-Specified Descriptor Records Out of AE Records

```
FUNCTION AEGetKeyPtr        (theAERecord: AERecord;
                             theAEKeyword: AEKeyword;
                             desiredType: DescType; VAR typeCode: DescType;
                             dataPtr: Ptr; maximumSize: Size;
                             VAR actualSize: Size): OSErr;
FUNCTION AEGetKeyDesc       (theAERecord: AERecord;
                             theAEKeyword: AEKeyword;
                             desiredType: DescType;
                             VAR result: AEDesc): OSErr;
```

Requesting User Interaction

```
FUNCTION AESetInteractionAllowed
                            (level: AEInteractAllowed): OSErr;
FUNCTION AEGetInteractionAllowed
                            (VAR level: AEInteractAllowed): OSErr;
FUNCTION AEInteractWithUser (timeOutInTicks: LongInt; nmReqPtr: NMRecPtr;
                             idleProc: IdleProcPtr): OSErr;
```

Requesting More Time to Respond to Apple Events

```
FUNCTION AEResetTimer       (reply: AppleEvent): OSErr;
```

Suspending and Resuming Apple Event Handling

```
FUNCTION AESuspendTheCurrentEvent
                            (theAppleEvent: AppleEvent): OSErr;
FUNCTION AEResumeTheCurrentEvent
                            (theAppleEvent, reply: AppleEvent;
                             dispatcher: EventHandlerProcPtr;
                             handlerRefcon: LongInt): OSErr;
FUNCTION AESetTheCurrentEvent
                            (theAppleEvent: AppleEvent): OSErr;
FUNCTION AEGetTheCurrentEvent
                            (VAR theAppleEvent: AppleEvent): OSErr;
```

Getting the Sizes and Descriptor Types of Descriptor Records

```
FUNCTION AESizeOfNthItem    (theAEDescList: AEDescList; index: LongInt;
                             VAR typeCode: DescType;
                             VAR dataSize: Size): OSErr;
FUNCTION AESizeOfKeyDesc    (theAERecord: AERecord;
                             theAEKeyword: AEKeyword;
                             VAR typeCode: DescType;
                             VAR dataSize: Size): OSErr;
FUNCTION AESizeOfParam      (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword;
                             VAR typeCode: DescType;
                             VAR dataSize: Size): OSErr;
FUNCTION AESizeOfAttribute  (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword;
                             VAR typeCode: DescType;
                             VAR dataSize: Size): OSErr;
```

Deleting Descriptor Records

```
FUNCTION AEDeleteItem       (theAEDescList: AEDescList;
                             index: LongInt): OSErr;
FUNCTION AEDeleteKeyDesc    (theAERecord: AERecord;
                             theAEKeyword: AEKeyword): OSErr;
FUNCTION AEDeleteParam      (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword): OSErr;
```

Deallocating Memory for Descriptor Records

```
FUNCTION AEDisposeDesc      (VAR theAEDesc: AEDesc): OSErr;
```

Coercing Descriptor Types

```
FUNCTION AECoercePtr        (typeCode: DescType; dataPtr: Ptr;
                             dataSize: Size; toType: DescType;
                             VAR result: AEDesc): OSErr;
FUNCTION AECoerceDesc       (theAEDesc: AEDesc; toType: DescType;
                             VAR result: AEDesc): OSErr;
```

Creating and Managing the Coercion Handler Dispatch Tables

```
FUNCTION AEInstallCoercionHandler
                            (fromType: DescType; toType: DescType;
                             handler: ProcPtr; handlerRefcon: LongInt;
                             fromTypeIsDesc: Boolean;
                             isSysHandler: Boolean): OSErr;
FUNCTION AEGetCoercionHandler
                            (fromType: DescType; toType: DescType;
                             VAR handler: ProcPtr;
                             VAR handlerRefcon: LongInt;
                             VAR fromTypeIsDesc: Boolean;
                             isSysHandler: Boolean): OSErr;
FUNCTION AERemoveCoercionHandler
                            (fromType: DescType; toType: DescType;
                             handler: ProcPtr;
                             isSysHandler: Boolean): OSErr;
```

Creating and Managing the Special Handler Dispatch Tables

```
FUNCTION AEInstallSpecialHandler
                            (functionClass: AEKeyword; handler: ProcPtr;
                             isSysHandler: Boolean): OSErr;
FUNCTION AEGetSpecialHandler
                            (functionClass: AEKeyword;
                             VAR handler: ProcPtr;
                             isSysHandler: Boolean): OSErr;
FUNCTION AERemoveSpecialHandler
                            (functionClass: AEKeyword; handler: ProcPtr;
                             isSysHandler: Boolean): OSErr;
```

Getting Information About the Apple Event Manager

```
{available only in version 1.01 and later versions of Apple Event Manager}
FUNCTION AEManagerInfo      (keyword: AEKeyword;
                             VAR result: LongInt): OSErr;
```

Application-Defined Routines

```
FUNCTION MyEventHandler     (theAppleEvent: AppleEvent; reply: AppleEvent;
                             handlerRefcon: LongInt): OSErr;
FUNCTION MyCoercePtr        (typeCode: DescType; dataPtr: Ptr;
                             dataSize: Size; toType: DescType;
                             handlerRefcon: LongInt;
                             VAR result: AEDesc): OSErr;
FUNCTION MyCoerceDesc       (theAEDesc: AEDesc; toType: DescType;
                             handlerRefcon: LongInt;
                             VAR result: AEDesc): OSErr;
```

C Summary

Constants

```
enum {
      #define gestaltAppleEventsAttr  'evnt' /*selector for Apple events*/
      gestaltAppleEventsPresent        = 0   /*if this bit is set, then */
                                             /* Apple Event Manager is */
};                                           /* available*/

      /*Apple event descriptor types*/
enum  {
      typeBoolean             = 'bool',      /*1-byte Boolean value*/
      typeChar                = 'TEXT',      /*unterminated string*/
      typeSMInt               = 'shor',      /*16-bit integer*/
      typeInteger             = 'long',      /*32-bit integer*/
      typeSMFloat             = 'sing',      /*SANE single*/
      typeFloat               = 'doub',      /*SANE double*/
      typeLongInteger         = 'long',      /*32-bit integer*/
      typeShortInteger        = 'shor',      /*16-bit integer*/
      typeLongFloat           = 'doub',      /*SANE double*/
      typeShortFloat          = 'sing',      /*SANE single*/
      typeExtended            = 'exte',      /*SANE extended*/
```

```
      typeComp                = 'comp',      /*SANE comp*/
      typeMagnitude           = 'magn',      /*unsigned 32-bit integer*/
      typeAEList              = 'list',      /*list of descriptor records*/
      typeAERecord            = 'reco',      /*list of keyword-specified */
                                             /* descriptor records*/
      typeAppleEvent          = 'aevt',      /*Apple event record*/
      typeTrue                = 'true',      /*TRUE Boolean value*/
      typeFalse               = 'fals',      /*FALSE Boolean value*/
      typeAlias               = 'alis',      /*alias record*/
      typeEnumerated          = 'enum'       /*enumerated data*/
};

enum {
      typeType                = 'type',      /*four-character code for */
                                             /* event class or event ID*/
      typeAppParameters       = 'appa',      /*Process Manager launch */
                                             /* parameters*/
      typeProperty            = 'prop',      /*Apple event property*/
      typeFSS                 = 'fss ',      /*file system specification*/
      typeKeyword             = 'keyw',      /*Apple event keyword*/

      typeSectionH            = 'sect',      /*handle to a section record*/
      typeWildCard            = '****',      /*matches any type*/
      typeApplSignature       = 'sign',      /*application signature*/
      typeSessionID           = 'ssid',      /*session ID*/
      typeTargetID            = 'targ',      /*target ID record*/
      typeProcessSerialNumber = 'psn ',      /*process serial number*/
      typeNull                = 'null'       /*NULL or nonexistent data*/
};

      /*keywords for Apple event parameters*/
enum {
      keyDirectObject         = '----',      /*direct parameter*/
      keyErrorNumber          = 'errn',      /*error number parameter*/
      keyErrorString          = 'errs',      /*error string parameter*/
      keyProcessSerialNumber  = 'psn '       /*process serial number param*/
};

      /*keywords for Apple event attributes*/
enum {
      keyTransactionIDAttr    = 'tran',      /*transaction ID*/
      keyReturnIDAttr         = 'rtid',      /*return ID*/
      keyEventClassAttr       = 'evcl',      /*event class*/
```

```
      keyEventIDAttr          = 'evid',      /*event ID*/
      keyAddressAttr          = 'addr',      /*address of target or */
                                             /* client application*/
      keyOptionalKeywordAttr  = 'optk',      /*list of optional parameters */
                                             /* for the Apple event*/
      keyTimeoutAttr          = 'timo',      /*number of ticks the client */
                                             /* will wait*/
      keyInteractLevelAttr    = 'inte',      /*settings to allow Apple */
                                             /* Event Mgr to bring */
                                             /* server to foreground*/
      keyEventSourceAttr      = 'esrc',      /*nature of source */
                                             /* application*/
      keyMissedKeywordAttr    = 'miss',      /*first required parameter */
                                             /* remaining in an Apple */
                                             /* event*/
      keyOriginalAddressAttr  = 'from'       /*address of original source; */
                                             /* available only in version */
                                             /* 1.01 and later versions of */
                                             /* the Apple Event Manager*/
};

      /*keywords for special handlers*/
enum {
      keyPreDispatch          = 'phac',      /*identifies a handler */
                                             /* routine that is called */
                                             /* immediately before the */
                                             /* Apple Event Manager */
                                             /* dispatches an Apple event*/
      keySelectProc           = 'selh',      /*selector used with */
                                             /* AERemoveSpecialHandler to */
                                             /* disable the OSL*/

      /*keywords for use with AEManagerInfo, available only in version */
      /* 1.0.1 and later versions of the Apple Event Manager*/
      keyAERecorderCount      = 'recr',      /*keyword for recording info*/
      keyAEVersion            = 'vers',      /*keyword for version info*/

      /*event class*/
      kCoreEventClass         = 'aevt'       /*event class for required */
                                             /* Apple events*/
};
```

```
      /*event IDs for required Apple events*/
enum {
      kAEOpenApplication      = 'oapp',      /*event ID for Open */
                                             /* Application event*/
      kAEOpenDocuments        = 'odoc',      /*event ID for Open */
                                             /* Documents event*/

      kAEPrintDocuments       = 'pdoc',      /*event ID for Print */
                                             /* Documents event*/
      kAEQuitApplication      = 'quit',      /*event ID for Quit */
                                             /* Application event*/
      kAEAnswer               = 'ansr',      /*event ID for Apple event */
                                             /* replies*/
      kAEApplicationDied      = 'obit'       /*event ID for Application */
                                             /* Died event*/
};
      /*constants for setting the sendMode parameter of AESend*/
enum {
      kAENoReply              = 0x00000001,  /*client doesn't want reply*/
      kAEQueueReply           = 0x00000002,  /*client wants server to */
                                             /* reply in event queue*/
      kAEWaitReply            = 0x00000003,  /*client wants a reply and */
                                             /* will give up processor*/
      kAENeverInteract        = 0x00000010,  /*server application should */
                                             /* not interact with user */
                                             /* for this Apple event*/
      kAECanInteract          = 0x00000020,  /*server may interact with */
                                             /* user for this Apple event */
                                             /* to supply information*/
      kAEAlwaysInteract       = 0x00000030,  /*server may interact with */
                                             /* user for this Apple event */
                                             /* even if no information */
                                             /* is required*/
      kAECanSwitchLayer       = 0x00000040,  /*server should come */
                                             /* directly to foreground */
                                             /* when appropriate*/
      kAEDontReconnect        = 0x00000080,  /*don't reconnect if there */
                                             /* is a PPC session closed */
                                             /* error*/
      kAEWantReceipt          = nReturnReceipt, /*client wants return */
                                             /* receipt*/
      kAEDontRecord           = 0x00001000,  /*don't record this event*/
      kAEDontExecute          = 0x00002000,  /*don't excecute this event*/
```

```
      /*constants for setting the sendPriority parameter of AESend*/
      kAENormalPriority       = 0x00000000,  /*post message at end of */
                                             /* event queue*/
      kAEHighPriority         = nAttnMsg     /*post message at front of */
                                             /* event queue*/
};

      /*event IDs for recording events; available only in version 1.01 and */
      /* later versions of the Apple Event Manager*/
enum {
      kAEStartRecording       = 'reca',      /*event ID for Start */
                                             /* Recording event*/
      kAEStopRecording        = 'recc',      /*event ID for Stop */
                                             /* Recording event*/
      kAENotifyStartRecording = 'rec1',      /*event ID for Recording On*/
                                             /* event*/
      kAENotifyStopRecording  = 'rec0',      /*event ID for Recording Off */
                                             /* event*/
      kAENotifyRecording      = 'recr'       /*event ID for Receive */
                                             /* Recordable Event event*/
};
enum {
      /*constant for the returnID parameter of AECreateAppleEvent*/
      kAutoGenerateReturnID   = -1,          /*tells Apple Event Manager */
                                             /* to generate a unique */
                                             /* return ID*/

      /*constant for transaction IDs*/
      kAnyTransactionID       = 0,           /*the Apple event is not */
                                             /* part of a transaction*/

      /*constants for timeout durations*/
      kAEDefaultTimeout       = -1,          /*use default timeout value*/
      kNoTimeOut              = -2,          /*never time out*/

      /*constants for the dispatcher parameter of AEResumeTheCurrentEvent*/
      kAENoDispatch           = 0,           /*don't redispatch the */
                                             /* Apple event*/
      kAEUseStandardDispatch  = -1        /*redispatch the Apple event */
                                             /* by using its entry in the */
                                             /* Apple event dispatch table*/
};
```

Data Types

```
typedef unsigned long AEEventClass;          /*event class for a */
                                             /* high-level event*/
typedef unsigned long AEEventID;             /*event ID for a high-level */
                                             /* event*/

typedef unsigned long AEKeyword;             /*keyword for a descriptor */
                                             /* record*/
typedef ResType DescType;                    /*descriptor type*/

struct AEDesc {                              /*descriptor record*/
   DescType descriptorType;                  /*type of data being passed*/
   Handle dataHandle;                        /*handle to data being passed*/
};
typedef struct AEDesc AEDesc;

struct AEKeyDesc {                           /*keyword-specified */
                                             /* descriptor record*/
   AEKeyword descKey;                        /*keyword*/
   AEDesc descContent;                       /*descriptor record*/
};
typedef struct AEKeyDesc AEKeyDesc;

typedef AEDesc AEAddressDesc;                /*address descriptor record*/
typedef AEDesc AEDescList;                   /*list of descriptor records*/
typedef AEDescList AERecord;                 /*list of keyword-specified */
                                             /* descriptor records*/
typedef AERecord AppleEvent;                 /*list of attributes and */
                                             /* parameters necessary for */
                                             /* an Apple event*/

typedef long AESendMode;                     /*flags that determine how */
                                             /* an Apple event is sent*/

typedef short AESendPriority;                /*send priority of an Apple */
                                             /* event*/

enum { kAEInteractWithSelf, kAEInteractWithLocal,
       kAEInteractWithAll };                 /*what processes may */
typedef unsigned char AEInteractAllowed;     /* interact with the user*/
```

```
enum { kAEUnknownSource, kAEDirectCall, kAESameProcess, kAELocalProcess,
       kAERemoteProcess };                  /*the source of an Apple */
typedef unsigned char AEEventSource;        /* event*/
enum { kAEDataArray, kAEPackedArray, kAEHandleArray,
       kAEDescArray, kAEKeyDescArray };     /*type of an Apple event */
typedef unsigned char AEArrayType;          /* array*/

union AEArrayData {                            /*data for an Apple event */
   short       kAEDataArray[1];                /* array*/
   char        kAEPackedArray[1];
   Handle      kAEHandleArray[1];
   AEDesc      kAEDescArray[1];
   AEKeyDesc   kAEKeyDescArray[1];
};
typedef union AEArrayData AEArrayData;

typedef AEArrayData *AEArrayDataPointer;

typedef ProcPtr EventHandlerProcPtr;        /*pointer to an Apple event */
                                            /* handler*/
typedef ProcPtr IdleProcPtr;                /*pointer to an application's */
                                            /* idle function*/
typedef ProcPtr EventFilterProcPtr;         /*pointer to an application's */
                                            /* filter function*/
```

Routines for Responding to Apple Events

Creating and Managing the Apple Event Dispatch Tables

```
pascal OSErr AEInstallEventHandler
                            (AEEventClass theAEEventClass,
                             AEEventID theAEEventID,
                             EventHandlerProcPtr handler,
                             long handlerRefcon, Boolean isSysHandler);
pascal OSErr AEGetEventHandler
                            (AEEventClass theAEEventClass,
                             AEEventID theAEEventID,
                             EventHandlerProcPtr *handler,
                             long *handlerRefcon, Boolean isSysHandler);
pascal OSErr AERemoveEventHandler
                            (AEEventClass theAEEventClass,
                             AEEventID theAEEventID,
                             EventHandlerProcPtr handler,
                             Boolean isSysHandler);
```

Dispatching Apple Events

```
pascal  OSErr  AEProcessAppleEvent
                            (const EventRecord *theEventRecord);
```

Getting Data or Descriptor Records Out of Apple Event Parameters and Attributes

```
pascal OSErr AEGetParamPtr  (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType desiredType,
                             DescType *typeCode, void* dataPtr,
                             Size maximumSize, Size *actualSize);
pascal OSErr AEGetParamDesc (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType desiredType,
                             AEDesc *result);
pascal OSErr AEGetAttributePtr
                            (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType desiredType,
                             DescType *typeCode, void* dataPtr,
                             Size maximumSize, Size *actualSize);
pascal OSErr AEGetAttributeDesc
                            (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType desiredType,
                             AEDesc *result);
```

Counting the Items in Descriptor Lists

```
pascal OSErr AECountItems   (const AEDescList *theAEDescList,
                             long *theCount);
```

Getting Items From Descriptor Lists

```
pascal OSErr AEGetNthPtr    (const AEDescList *theAEDescList, long index,
                             DescType desiredType, AEKeyword *theAEKeyword,
                             DescType *typeCode, void* dataPtr,
                             Size maximumSize, Size *actualSize);
pascal OSErr AEGetNthDesc   (const AEDescList *theAEDescList, long index,
                             DescType desiredType, AEKeyword *theAEKeyword,
                             AEDesc *result);
pascal OSErr AEGetArray     (const AEDescList *theAEDescList,
                             AEArrayType arrayType,
                             AEArrayDataPointer arrayPtr, Size maximumSize,
                             DescType *itemType, Size *itemSize,
                             long *itemCount);
```

Getting Data and Keyword-Specified Descriptor Records Out of AE Records

```
pascal OSErr AEGetKeyPtr    (const AERecord *theAERecord,
                             AEKeyword theAEKeyword, DescType desiredType,
                             DescType *typeCode, void* dataPtr,
                             Size maximumSize, Size *actualSize);
pascal OSErr AEGetKeyDesc   (const AERecord *theAERecord,
                             AEKeyword theAEKeyword, DescType desiredType,
                             AEDesc *result);
```

Requesting User Interaction

```
pascal OSErr AESetInteractionAllowed
                            (AEInteractAllowed level);
pascal OSErr AEGetInteractionAllowed
                            (AEInteractAllowed *level);
pascal OSErr AEInteractWithUser
                            (long timeOutInTicks, NMRecPtr nmReqPtr,
                             IdleProcPtr idleProc);
```

Requesting More Time to Respond to Apple Events

```
pascal OSErr AEResetTimer   (const AppleEvent *reply);
```

Suspending and Resuming Apple Event Handling

```
pascal OSErr AESuspendTheCurrentEvent
                            (const AppleEvent *theAppleEvent);
pascal OSErr AEResumeTheCurrentEvent
                            (const AppleEvent *theAppleEvent,
                             const AppleEvent *reply,
                             EventHandlerProcPtr dispatcher,
                             long handlerRefcon);
pascal OSErr AESetTheCurrentEvent
                            (const AppleEvent *theAppleEvent);
pascal OSErr AEGetTheCurrentEvent
                            (AppleEvent *theAppleEvent);
```

Getting the Sizes and Descriptor Types of Descriptor Records

```
pascal OSErr AESizeOfNthItem
                            (const AEDescList *theAEDescList, long index,
                             DescType *typeCode, Size *dataSize);
pascal OSErr AESizeOfKeyDesc
                            (const AERecord *theAERecord,
                             AEKeyword theAEKeyword, DescType *typeCode,
                             Size *dataSize);
```

```
pascal OSErr AESizeOfParam  (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType *typeCode,
                             Size *dataSize);
pascal OSErr AESizeOfAttribute
                            (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType *typeCode,
                             Size *dataSize);
```

Deleting Descriptor Records

```
pascal OSErr AEDeleteItem   (const AEDescList *theAEDescList, long index);
pascal OSErr AEDeleteKeyDesc
                            (const AERecord *theAERecord,
                             AEKeyword theAEKeyword);
pascal OSErr AEDeleteParam  (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword);
```

Deallocating Memory for Descriptor Records

```
pascal OSErr AEDisposeDesc  (AEDesc *theAEDesc);
```

Coercing Descriptor Types

```
pascal OSErr AECoercePtr    (DescType typeCode, const void* dataPtr,
                             Size dataSize, DescType toType,
                             AEDesc *result);
pascal OSErr AECoerceDesc   (const AEDesc *theAEDesc, DescType toType,
                             AEDesc *result);
```

Creating and Managing the Coercion Handler Dispatch Tables

```
pascal OSErr AEInstallCoercionHandler
                            (DescType fromType, DescType toType,
                             ProcPtr handler, long handlerRefcon,
                             Boolean fromTypeIsDesc, Boolean isSysHandler);
pascal OSErr AEGetCoercionHandler
                            (DescType fromType, DescType toType,
                             ProcPtr *handler, long *handlerRefcon,
                             Boolean *fromTypeIsDesc,
                             Boolean isSysHandler);
pascal OSErr AERemoveCoercionHandler
                            (DescType fromType, DescType toType,
                             ProcPtr handler, Boolean isSysHandler);
```

Creating and Managing the Special Handler Dispatch Tables

```
pascal OSErr AEInstallSpecialHandler
                            (AEKeyword functionClass, ProcPtr handler,
                             Boolean isSysHandler);
pascal OSErr AEGetSpecialHandler
                            (AEKeyword functionClass, ProcPtr *handler,
                             Boolean isSysHandler);
pascal OSErr AERemoveSpecialHandler
                            (AEKeyword functionClass, ProcPtr handler,
                             Boolean isSysHandler);
```

Getting Information About the Apple Event Manager

```
/*available only in version 1.01 and later versions of Apple Event Manager*/
pascal OSErr AEManagerInfo  (AEKeyword keyword, long *result);
```

Application-Defined Routines

```
pascal OSErr MyEventHandler (const AppleEvent *theAppleEvent,
                             const AppleEvent *reply, long handlerRefcon);
pascal OSErr MyCoercePtr    (DescType typeCode, const void* dataPtr,
                             Size dataSize, DescType toType,
                             long handlerRefcon, AEDesc *result);
pascal OSErr MyCoerceDesc   (const AEDesc *theAEDesc, DescType toType, long
                             handlerRefcon, AEDesc *result);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

```
_Pack8
```

Selector	Routine
$011E	AESetInteractionAllowed
$0204	AEDisposeDesc
$0219	AEResetTimer
$021A	AEGetTheCurrentEvent
$021B	AEProcessAppleEvent

Selector	Routine
$021D	AEGetInteractionAllowed
$022B	AESuspendTheCurrentEvent
$022C	AESetTheCurrentEvent
$0407	AECountItems
$040E	AEDeleteItem
$0413	AEDeleteKeyDesc
$0413	AEDeleteParam
$0441	AEManagerInfo
$0500	AEInstallSpecialHandler
$0501	AERemoveSpecialHandler
$052D	AEGetSpecialHandler
$0603	AECoerceDesc
$061C	AEInteractWithUser
$0720	AERemoveEventHandler
$0723	AERemoveCoercionHandler
$0812	AEGetKeyDesc
$0812	AEGetParamDesc
$0818	AEResumeTheCurrentEvent
$0826	AEGetAttributeDesc
$0828	AESizeOfAttribute
$0829	AESizeOfKeyDesc
$0829	AESizeOfParam
$082A	AESizeOfNthItem
$091F	AEInstallEventHandler
$0921	AEGetEventHandler
$0A02	AECoercePtr
$0A22	AEInstallCoercionHandler
$0A0B	AEGetNthDesc
$0B24	AEGetCoercionHandler
$0D0C	AEGetArray
$0E11	AEGetKeyPtr
$0E11	AEGetParamPtr
$0E15	AEGetAttributePtr
$100A	AEGetNthPtr

Result Codes

noErr	0	No error
paramErr	–50	Parameter error (for example, value of handler pointer is NIL or odd)
eLenErr	–92	Buffer too big to send
memFullErr	–108	Not enough room in heap zone
userCanceledErr	–128	User canceled an operation
procNotFound	–600	No eligible process with specified process serial number
bufferIsSmall	–607	Buffer is too small
noOutstandingHLE	–608	No outstanding high-level event
connectionInvalid	–609	Nonexistent signature or session ID
noUserInteractionAllowed	–610	Background application sends event requiring authentication
noPortErr	–903	Client hasn't set 'SIZE' resource to indicate awareness of high-level events
destPortErr	–906	Server hasn't set 'SIZE' resource to indicate awareness of high-level events, or else is not present
sessClosedErr	–917	The kAEDontReconnect flag in the sendMode parameter was set, and the server quit and then restarted
errAECoercionFail	–1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	–1701	Descriptor record was not found
errAECorruptData	–1702	Data in an Apple event could not be read
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed
errAENewerVersion	–1706	Need a newer version of the Apple Event Manager
errAENotAppleEvent	–1707	Event is not an Apple event
errAEEventNotHandled	–1708	Event wasn't handled by an Apple event handler
errAEReplyNotValid	–1709	AEResetTimer was passed an invalid reply
errAEUnknownSendMode	–1710	Invalid sending mode was passed
errAEWaitCanceled	–1711	User canceled out of wait loop for reply or receipt
errAETimeout	–1712	Apple event timed out
errAENoUserInteraction	–1713	No user interaction allowed
errAENotASpecialFunction	–1714	The keyword is not valid for a special function
errAEParamMissed	–1715	Handler cannot understand a parameter the client considers required
errAEUnknownAddressType	–1716	Unknown Apple event address type
errAEHandlerNotFound	–1717	No handler found for an Apple event or a coercion, or no object callback function found
errAEReplyNotArrived	–1718	Reply has not yet arrived
errAEIllegalIndex	–1719	Not a valid list index
errAEImpossibleRange	–1720	The range is not valid because it is impossible for a range to include the first and last objects that were specified; an example is a range in which the offset of the first object is greater than the offset of the last object
errAEWrongNumberArgs	–1721	The number of operands provided for the kAENot logical operator is not 1
errAEAccessorNotFound	–1723	There is no object accessor function for the specified object class and token descriptor type

errAENoSuchLogical	–1725	The logical operator in a logical descriptor record is not kAEAnd, kAEOr, or kAENot
errAEBadTestKey	–1726	The descriptor record in a test key is neither a comparison descriptor record nor a logical descriptor record
errAENotAnObjectSpec	–1727	The objSpecifier parameter of AEResolve is not an object specifier record
errAENoSuchObject	–1728	A run-time resolution error, for example: object specifier record asked for the third element, but there are only two
errAENegativeCount	–1729	Object-counting function returned negative value
errAEEmptyListContainer	–1730	The container for an Apple event object is specified by an empty list
errAEUnknownObjectType	–1731	Descriptor type of token returned by AEResolve is not known to server application
errAERecordingIsAlreadyOn	–1732	Attempt to turn recording on when it is already on