INSIDE MACINTOSH

# Apple Events in Mac OS 8

**WWDC Release**

May 1996

Apple Computer, Inc.



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



Simultaneously published in the United States and Canada.

# Contents

# Introduction to the Mac OS 8 Event Model

## Contents

The Mac OS 8 event model is based on the Apple events mechanism introduced in System 7. It provides a unified interface for events throughout the system, avoids the problems created by polling, and enhances application responsiveness to user actions.

This chapter introduces the Mac OS 8 event model. Before you read this chapter, you should be familiar with the book *Inside Macintosh: Interapplication Communication*. An electronic copy is included with this developer release.

For descriptions of the Apple Event Manager functions you can use to take advantage of the event model, see "Apple Event Communications Model Reference" (page 2-3). Later developer releases will include sample code that demonstrates their use.

For information about the way the Toolbox routes standard Apple events, see the accompanying document *Human Interface Toolbox.*

Mac OS 8 supports the classic System 7 Event Manager as described in *Inside Macintosh: Macintosh Toolbox Essentials* for backward compatibility only. Supporting the Mac OS 8 event model allows you to take full advantage of the improved performance and flexibility that Mac OS 8 makes possible.

▲ **WARNING**
This document is preliminary and incomplete. All information presented here is subject to change. ▲

# Apple Events in Mac OS 8

An **Apple event** identifies itself by event class and event ID, names its own destination, and contains additional data structures that vary depending on the kind of event. First introduced in System 7 to support interapplication communication, Apple events provide the primary mechanism for communication throughout Mac OS 8.

In addition to this standard communications model, Apple events provide a hierarchical data format for organizing the information they convey. You use Apple Event Manager functions to support both the communication model (for sending and receiving Apple events) and the data model (for creating or interpreting the data structures used to construct Apple events).

The Apple event data model and Apple event communications model form the foundation for the Mac OS 8 event model, which defines programming concepts and protocols used throughout the system for communication among tasks, server programs, system services, device drivers, and other types of software as well as communication related to the user's interactions with your application.

## Apple Event Communications Model

The essence of the Mac OS 8 Apple event communication model is simple. When any program that supports this model runs, it expresses an interest in receiving certain Apple events. It then informs the Apple Event Manager that it is ready to receive, and the Apple Event Manager blocks the calling task until an event in which the program has expressed an interest arrives. This arrangement takes maximum advantage of priority-based preemptive scheduling, allowing other programs and tasks to receive processing time when your program doesn't need it.

The Mac OS 8 Apple Event Manager replaces the System 7 functions used to dispatch, receive, and send events with entirely new functions. To take full advantage of the Mac OS 8 communication model, you should use the new functions. Mac OS 8 supports the System 7 functions for backward compatibility only.

For descriptions of the new Mac OS 8 functions and a list of the System 7 functions they replace, see "Apple Event Communications Model Reference" (page 2-3).

## Apple Event Data Model

In general, System 7 functions used to implement the Apple event data model—that is, the functions used to add data to or get data from Apple events—have not changed. You can use most of these functions in the manner described in *Inside Macintosh: Interapplication Communication.*

One exception involves the way you obtain and clear the data in descriptor records. System 7 defines the data field of a descriptor record as a handle, requiring you to double dereference the handle and access the data directly. Although it defines accessors for related structures, System 7 doesn't provide accessors for descriptor records themselves.

As in other parts of the system, all data structures defined by the Mac OS 8 Apple Event Manager are opaque—that is, you can manipulate them only with the aid of accessor functions. Therefore, the Apple Event Manager provides new functions for obtaining and clearing the data in descriptor records. You should use these functions rather than accessing data in descriptor records directly.

The Mac OS 8 Apple Event Manager also provides functions that supplement the capabilities of the System 7 data model in the following areas:

- **Apple event streaming functions** allow you to write a series of descriptor records sequentially rather than explicitly inserting them one at a time using `AEPutNthPointer` or `AEPutKeyPtr`.
- **Apple event subdescriptor functions** allow you to examine the contents of recursively nested descriptor records without having to copy any data or create any new `AEDesc` structures.

Documentation for all these new functions will be available with later developer releases.

**Note**
Standard Apple events previously defined by Apple—for example, the Required suite of Apple events discussed in *Inside Macintosh: Interapplication Communication*—are still supported in Mac OS 8 and still play the same roles. ◆

# The Mac OS 8 Event Model

Mac OS 8 supports several forms of interprocess communication, including specialized low-level communication using shared data, shared memory areas, and the Microkernel Messaging Service as well as the higher-level Apple event communications model. Apple events provide the most pervasive form of interprocess communication in Mac OS 8. Applications, server programs, and system services can all use Apple events to communicate with each other.

This section introduces the concepts you need to understand to take advantage of the Mac OS 8 event model. In Mac OS 8, you typically implement an application (that is, a program with a human interface) as a cooperative program that may be supported by server programs. Every program has a main task that may spawn additional tasks. (For more information about tasks, cooperative programs, and server programs, see the accompanying document *Microkernel and Core System Services.)* The Mac OS 8 event model provides a unified mechanism for controlling programs and their associated tasks.

When launched, any program can call Apple Event Manager functions to install handlers for the events that it wants to handle. After installing its handlers, the program doesn't use `WaitNextEvent` to receive events and `AEProcessAppleEvent` to dispatch them. Instead, each task performs any initialization work the task requires, then calls the Mac OS 8 function `AEReceive`, which doesn't return unless a handler generates an error or intentionally terminates the call to `AEReceive`.

For the most part, both the main task and any additional tasks that it spawns execute from inside `AEReceive`, which takes care of many of the operations that are handled by a System 7 event loop. The `AEReceive` function blocks the calling task until an event the program can handle arrives, at which point the Apple Event Manager reawakens the task and dispatches the event to the appropriate handler. This cycle of blocking then waking the task continues until the task terminates or the application quits.

All Apple events are conveyed via the "bottleneck" of `AEReceive`. This avoids the multilevel dispatching required with the classic System 7 Event Manager, which recognizes three principal kinds of events (low-level events, operating system events, and high-level events such as Apple events), each requiring different treatment.

The Mac OS 8 event model treats all events the same way. At the same time, handlers installed for any event may translate low-level events into higher-level events that are more meaningful to the program. It may be helpful to think about this kind of translation or recycling of events in terms of three broad categories (whose boundaries may blur in specific cases):

- **Low-level events,** such as Mouse Down, are generated by the system in response to the user's manipulation of input devices. Low-level event handlers often translate events into higher-level synthetic or semantic events and resend the repackaged events. For example, if the user presses the mouse button while the pointer is over the File menu, the system sends corresponding Mouse Down event, which is resent by a Mouse Down handler as a Mouse Down in Content event.
- **Synthetic events,** such as Mouse Down in Content, mean more to most applications than low-level events. A single synthetic event may be produced by more than one low-level event; for example, a series of low-level Virtual Key events might generate a single synthetic text event that conveys a Kanji character. Synthetic event handlers can also translate events into higher-level semantic events; for example, the application's handler for a Mouse Down in Content event that is followed by a Mouse Up event while the pointer is over the Open command in the File menu translates those two events into an Open Document event.
- **Semantic events** are events such as Open Document or Quit Application with a specific meaning for an individual application. Several different kinds of synthetic events may be capable of generating the same semantic event. For example, the user can generate an Open Document event by releasing the mouse button while the Open command is selected in the File menu or by pressing the Command key and the O key at the same time. Semantic events may also be sent originally as semantic events, rather than cascading up through synthetic events from an original physical event. For example, a Get Data event is always generated by a script or another application; it is never generated directly by the user.

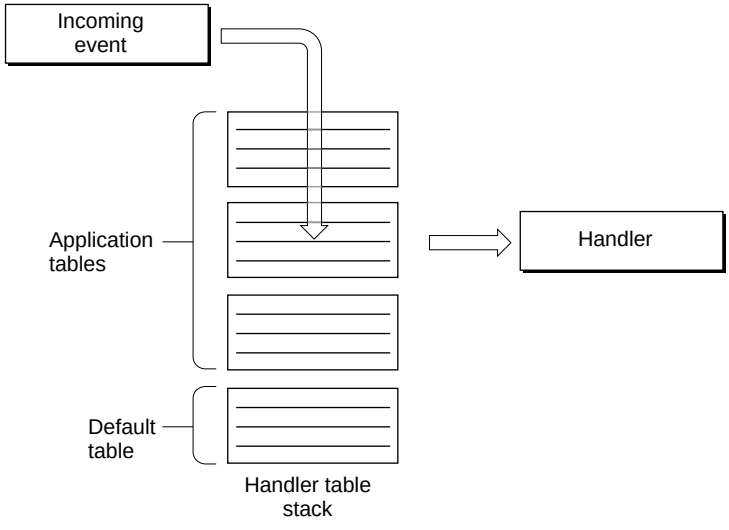
When you're creating the human interface for your application, you are primarily concerned with synthetic events. Most synthetic events are ultimately directed at a target within the application, such as a window or a menu. The Mac OS 8 event model allows the system to help your application arbitrate targets for some events while allowing you to override the default arbitration at any point. For some examples of default event routing provided by the system, see the accompanying document *Human Interface Toolbox.*

## Apple Event Dispatchers

To dispatch Apple events within a process, the Apple Event Manager uses one or more **Apple event dispatchers,** which combine an event queue and a stack of handler tables. Every process has a default dispatcher, and your application may create additional dispatchers as necessary.

Figure 1-1 illustrates the way the Apple Event Manager uses an Apple event dispatcher to dispatch an event. When an event arrives, the Apple Event Manager wakes up the task that called `AEReceive` and searches the stack of handler tables for a matching handler. An Apple event handler is identified by the combination of the event class and event ID of the event it handles—much the way Apple event handlers are stored in application and system handler tables in System 7.

**Figure 1-1** A handler table stack associated with an Apple event dispatcher



## Handler Tables

Each Apple event dispatcher maintains its own **handler table stack,** which consists of a default handler table and one or more application handler tables.

A **default handler table** contains the default handlers installed by the system. The default handlers for an application interpret standard events such as Mouse Down or Update and if necessary route them to the appropriate panels within a window.

Most applications need to add one or more of their own handler tables to the handler table stack to implement each application's unique behaviors. For example, when a user double-clicks one of your application's document icons in the Finder, the Finder sends the Open Document event to your application to request that it open the corresponding document. Therefore, every Mac OS 8 application must provide a handler for the Open Document event (in much the same way that System 7 applications do).

You use the Apple Event Manager to install Apple event handlers in one or more **application handler tables,** which you can add to or remove from the handler table stack at any time. When the Apple Event Manager searches for an event's handler, it starts from the top of the stack and looks down the chain of handler tables until it finds a match. By stacking one handler table on top of another, you can augment or override the behavior defined by the lower table with the behavior defined by the higher table.

If the Apple Event Manager can't find an entry for a particular event in an application handler table, it takes one of two actions, depending on the type of table. You can create two kinds of Apple event handler tables:

- **Nonfiltered handler table.** When a nonfiltered handler table contains no handler for a particular event, the Apple Event Manager passes the event on to the next handler table in the stack.
- **Filtered handler table.** When a filtered handler table contains no handler for a particular event, the Apple Event Manager immediately suspends the event, which remains in the event queue. After the filtered table has been removed from the handler table stack, the Apple Event Manager passes any suspended events on to the next handler table in the order in which they were originally received.

Nonfiltered tables are appropriate for most situations. Filtered handler tables are useful when your application is in a modal state and you want to suspend handling of certain events. For more information about filtered handler tables, see "Event Dispatching for Modal States" (page 1-17).

## Apple Event Handlers

If the event class and event ID for a particular event is sufficient for you to decide that your application doesn't need to handle that event, you don't need to install a handler for it. For example, you don't need to install handlers for text events if you intend to take advantage of the default text-handling provided by HI objects of class `HIEditText`.

If you need more information about an event before you can decide whether your application needs to handle it, you must install a handler for that event. For example, an application that supports the Get Data event for certain kinds of data must install a handler that receives all Get Data events, including some the application may not be able to handle.

When the Apple Event Manager finds an entry for an event in a handler table, it passes the event to that handler. The handler must make at least two decisions about the event:

1. **Handle the event or don't handle it.** If the handler can handle the event, it should do so, then proceed to step 2. If the handler can't handle the event for any reason, it should proceed directly to step 2.
2. **Consume the event or pass it on.** In most cases the handler simply handles the event, returns a result code such as `noErr` that indicates it did so successfully, and processing for that event stops. If the handler can't consume the event, it should return a result code that instructs the Apple Event Manager what to do with the event next. For example, if the handler returns the result code `errAEEventNotHandled`, the Apple Event Manager continues its search through the handler table stack for a handler for the event.

If the handler can't understand the event as specified—for example, a Get Data event requests the name of the current printer, which the application doesn't know anything about—returning the result code `errAEEventNotHandled` allows handlers for that event in lower tables, if any are installed, a chance to handle it. Passing on the event in this way can also be useful if the handler performs preliminary handling only and you want to take advantage of additional handling provided by handlers in lower tables.

If a handler understands the event but the event is impossible to handle—for example, a Get Data event specifies the fifth paragraph in a document that only has four paragraphs—the handler should return any appropriate nonzero

error. Returning any result code other than `errAEEventNotHandled` prevents the Apple Event Manager from continuing to search the handler table stack, and the event dies.

## Tasking Models

Before reading this section, you should be familiar with the accompanying document *Microkernel and Core System Services.* As that document explains, every program has a main task and may have additional tasks. Only the main task of a cooperative program can specify the default Apple event dispatcher when it calls `AEReceive`. You can associate additional tasks with other dispatchers to take advantage of preemptive scheduling for processing that doesn't involve your application's human interface.

When any task calls `AEReceive`, the task specifies the Apple event dispatcher (and thus the event queue) in which it's interested. You can associate tasks with Apple event dispatchers in three principal ways:

- one task and one dispatcher
- multiple tasks and multiple dispatchers
- multiple tasks and one dispatcher

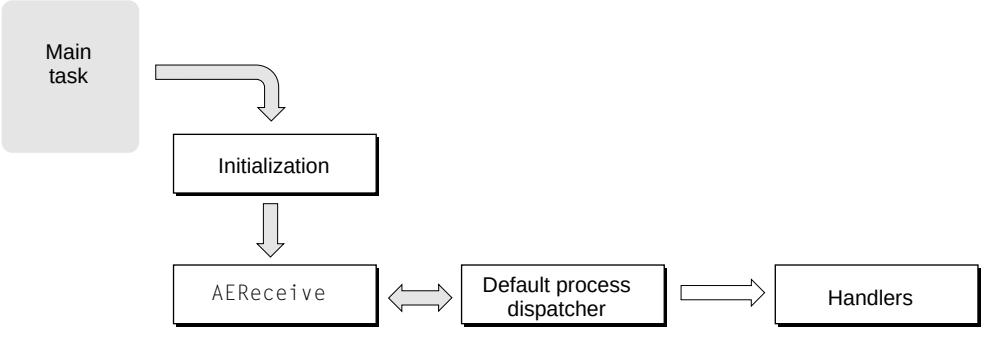
The sections that follow introduce these tasking models.

Identifying the particular arrangement of tasks and dispatchers appropriate for your program is a design decision. The first two models are appropriate for both cooperative programs and server programs. The third model, which associates multiple tasks with a single dispatcher, is intended for use by server programs only. All three models may be combined in various ways to support more complex relationships among tasks and dispatchers and to assign operations to server programs. The Mac OS 8 event model makes it easy to construct a multitasking "back end" for an application.

## One Task, One Dispatcher

Figure 1-2 shows the simplest case: a main task associated with the default Apple event dispatcher associated with a process.

**Figure 1-2** One task, one dispatcher



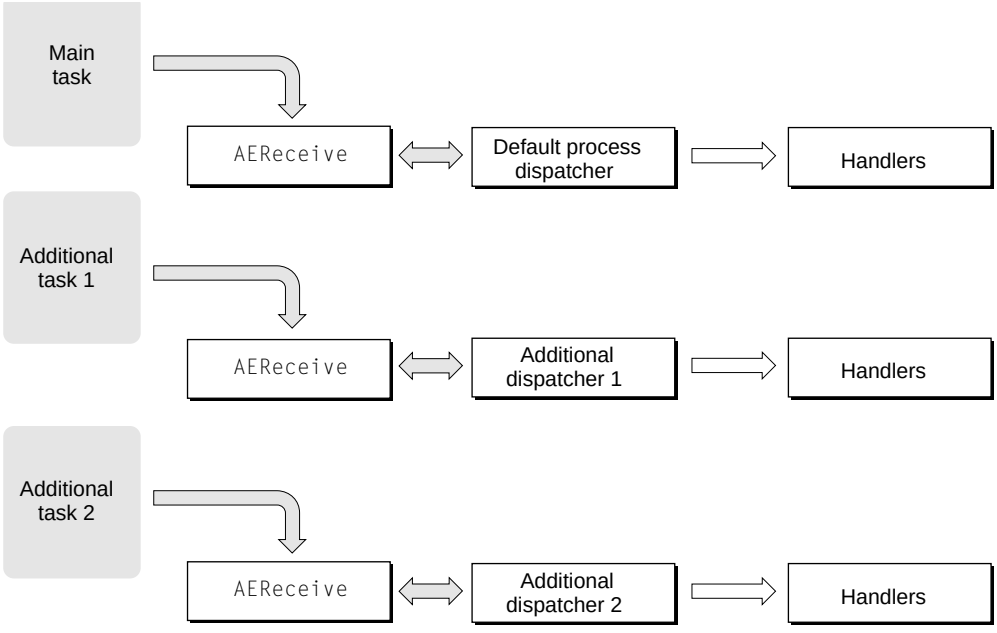

The arrangement shown in Figure 1-2 superficially resembles a System 7 event loop in the sense that a single task is responsible for all event handling. In Mac OS 8, the role played by the event loop in System 7 has been transferred to `AEReceive`, which blocks the main task until an event for which its dispatcher has a handler arrives.The Apple Event Manager the wakes the main task and runs the handler.

## Multiple Tasks, Multiple Dispatchers

A dispatcher and its handlers represent one kind of behavior or set of activities that your application can perform. For example, you must associate an application's main task with a single dispatcher for all handlers that use cooperative services, as in Figure 1-2.

It's also possible to create additional dispatchers for one or more additional tasks that use reentrant services only. Figure 1-3 illustrates this arrangement. Another alternative is to create additional tasks for a single dispatcher, as described in "Multiple Tasks, One Dispatcher" (page 1-14).

**Figure 1-3** Multiple tasks, multiple dispatchers




You can implement the tasking model shown in Figure 1-3 in several ways. It's possible, for example, to route events to a particular dispatcher. All human interface events must be routed through an application's default process dispatcher, but you can route other events to other dispatchers in any way you wish.

All events are sent to the default dispatcher initially. Its handlers forward certain events to one or more additional dispatchers. For example, a graphics program might have a menu command that transforms an image in some way by performing a series of calculations. The handler invoked by that command can in turn send an Apple event to a different dispatcher associated with a separate task that actually performs the calculations. The main task is then free to continue responding to the user's manipulation of the human interface while the second task, which doesn't involve the human interface, continues to execute in the background.

When the second task needs to inform the user of its progress, the handler that's performing the calculation can send an event back to the default process

dispatcher to update a progress indicator. Similarly, when the handler has completed its calculations, it can send an event back to the default process dispatcher to invoke the handler that actually draws the transformed image.

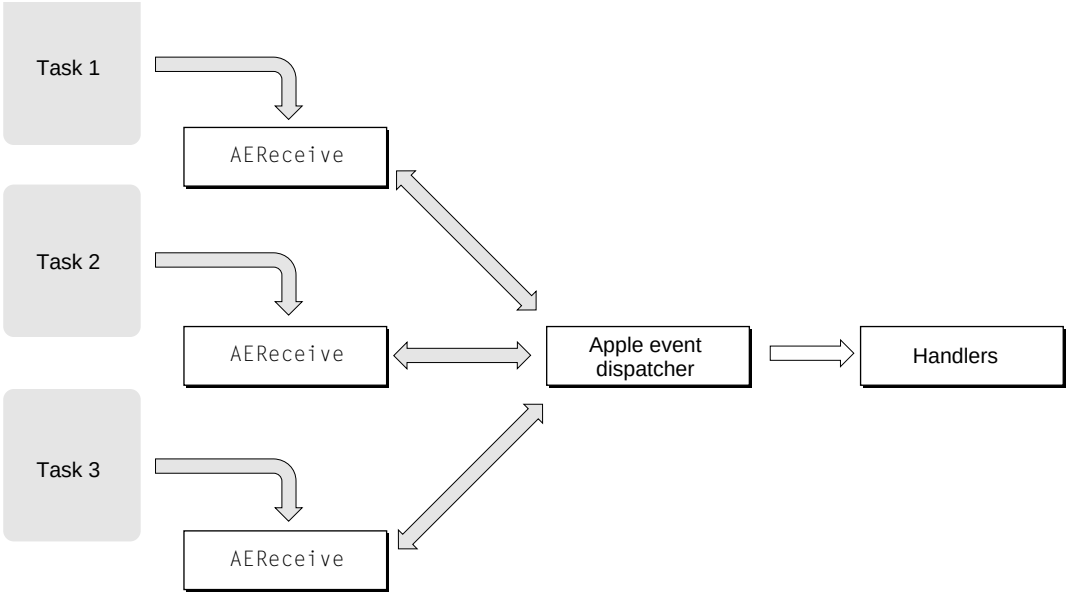
Because Mac OS 8 permits an application to use multiple tasks in addition to a single main task, the graphics application in this example could actually perform transformation calculations on several different images, starting each calculation at a different time and performing them all concurrently. Thus, the main task could be drawing the results of one calculation to the screen while another task is in the middle of calculating a transformation for a second image and still another task is just beginning to calculate a transformation for a third image. In effect this kind of arrangement allows you to create a "server within the application," even though the additional tasks don't necessarily have to be implemented as independent server programs in their own address spaces.

Because of the way the Mac OS 8 microkernel schedules tasks in this kind of situation, you can ensure that your application continues to be highly responsive to user actions even while additional tasks are executing—unlike System 7, in which background processing can seriously interfere with the application's responsiveness.

## Multiple Tasks, One Dispatcher

Figure 1-4 shows multiple tasks calling `AEReceive` with the same dispatcher. Each task has its own entry point and begins executing at a different time. The tasks don't necessarily have to be identical, but they must use the same set of handlers provided by the dispatcher and must all be equally qualified to deal with incoming events. All handlers in a dispatcher that is shared in this way must be fully reentrant.

**Figure 1-4** Multiple tasks, one dispatcher




This arrangement is most useful for server programs. For example, a database that receives requests continuously from several sources can spawn a series of identical tasks associated with the same Apple event dispatcher. All these tasks share the same stack of handler tables. The Apple event dispatcher pairs each task with each incoming request and looks up the corresponding handler in the stack of handler tables. As each task resumes execution, it can execute at the same time, if necessary, that previously woken tasks are executing. Thus, the database can handle a series of requests simultaneously.

The tasking model shown in Figure 1-4 is not appropriate for most cooperative programs.

## Benefits of the Mac OS 8 Event Model

Supporting the Mac OS 8 event model allows you to take advantage of all the human interface features provided by the Mac OS 8 Toolbox. The Mac OS 8 event model also provides these benefits:

- The use of blocking rather than polling improves performance for all applications running on the same machine and takes maximum advantage of priority-based preemptive scheduling.
- The use of Apple event handlers rather than event masks to distinguish events permits a much larger name space for events. This ensures that Apple can provide new default handlers and new behaviors with minimum impact on existing applications and also makes it easier to create specialized events for your own purposes.
- The use of Apple events throughout the system simplifies the programming you have to do to make your application scriptable and recordable.
- Events are always sent and dispatched the same way, which simplifies the overall Mac OS programming model.
- You can use the Mac OS 8 event model to construct a multitasking back end for your application.

Although Mac OS 8 supports the classic Event Manager for backward compatibility, many new Toolbox features require the new event model, and much of the information conveyed by Mac OS 8 Apple events is lost in the translation to classic events.

More information about the Mac OS 8 event model will be available with later developer releases. The best way to prepare your System 7 application for Mac OS 8 events is to support Apple events as described in *Inside Macintosh: Interapplication Communication*, including factoring your application and making it fully scriptable and recordable.

# Using the Mac OS 8 Event Model

This section introduces some of the functions provided by the Apple Event Manager that you can use to implement your application's event handling. For detailed descriptions of these functions, see "Apple Event Communications Model Reference" (page 2-3).

## Manipulating Dispatchers, Handler Tables, and Handlers

To get the default Apple event dispatcher for your application's process, use `AEGetDefaultDispatcher`. If you need to create your own dispatcher, use `AECreateEventDispatcher`. Both functions return a dispatcher reference that you can use to refer to the dispatcher when you call other functions. When you're finished with a dispatcher that you've created with `AECreateEventDispatcher`, use the `AEDisposeEventDispatcher` function to dispose of it.

To create a handler table, use `AENewHandlerTable` (for a nonfiltered table) or `AENewFilterHandlerTable` (for a filtered table). Both functions return a handler table reference that you can use to refer to the table when you call other functions. When you're finished with a handler table, use the `AEDisposeHandlerTable` function to dispose of it.

To add handlers to a handler table, use `AEInstallHandler`. To add a handler table to the top of a dispatcher's handler table stack, use `AEPushDispatcherHandlerTable`; to remove it from the stack, use `AEPopDispatcherHandlerTable`. To find out what handler table is currently on top of a handler table stack, use `AEGetDispatcherTopHandlerTable`.

## Event Dispatching for Modal States

Applications commonly modify the way they respond to events for short periods of time. For example, when a user chooses a pencil tool from a palette of tools provided by a painting program, then presses and holds the mouse button and moves the pencil tool around in a window, the application must track the pencil's movement until the mouse button is released. While it's tracking mouse movement, the application can't deal immediately with some of the events it may receive, such as an Open Document event. However, it may need to handle other events, such as Update events, right away. After the

user releases the mouse button, the application needs to resume handling of the events that arrived while the pencil tool was in use.

You can suspend events temporarily in this kind of modal situation by pushing and popping a filtered handler table. When the modal state begins—for example, in response to a Mouse Down in Content event—the handler for the event calls `AEPushDispatcherHandlerTable` to add a filtered handler table on top of the dispatcher's handler table stack. The handler then calls `AEReceive`, which blocks the calling task until an event with an entry in the filtered table arrives.

The call to `AEReceive` from the handler occurs within a previous call to `AEReceive` by the application's main task. Whenever the Apple Event Manager encounters a filtered handler table on the handler table stack, it suspends any incoming events that don't have entries in that table until a call to `AEPopDispatcherHandlerTable` removes it. Only the events that are relevant to the modal state get handled, and other events wait in the dispatcher's event queue until the modal state ends.

In the pencil example, you could create a filtered table with handlers for the Mouse Move, Update, and Mouse Up events. After you add this handler table to the handler table stack, the Apple Event Manager dispatches those events to their handlers, so the user can draw with the pencil and the screen will get updated as the mouse moves, but all other events wait in the event queue until the modal state ends.

The modal state ends when a handler installed in the filtered handler table—in this example, the Mouse Up handler—returns the result code `errAEReceiveEscapeCurrent`. When it receives this result code, the original Mouse Down in Content handler's call to `AEReceive` returns, the handler calls `AEPopDispatcherHandlerTable`, and the Apple Event Manager dispatches the suspended events in the order in which they were received. The calling task continues to block on its original call to `AEReceive`, which passes incoming events to the dispatcher just as it did before the modal state began.

You can also use the result code `errAEReceiveEscapeCurrent` to force the highest-level call to `AEReceive` to return. For example, a handler for a Quit Application event can return `errAEReceiveEscapeCurrent` to the main task's original call to `AEReceive`. This forces `AEReceive` to return, which allows the application's main function to return, which in turn allows the system to clean up. Alternatively, the Quit Application handler can call `ExitToShell` directly.

The HI object class `HIDialog` automatically performs most of the operations described here when you create a modal or movable modal dialog box. For more information, see the accompanying document *Human Interface Toolbox.*

# Apple Event Communications Model Reference

## Contents

# Apple Event Communications Constants and Data Types

This section describes constants and data types defined by the Mac OS 8 Apple Event Manager for manipulating Apple event dispatchers and handler tables and for receiving and sending Apple events.

For an introduction to the role of Apple events in Mac OS 8, see "Introduction to the Mac OS 8 Event Model" (page 1-3).

## Apple Event Dispatchers

Dispatcher references identify dispatchers within the scope of a program's process, and dispatcher IDs identify a program's dispatchers to sources of Apple events outside of its process. Dispatcher references provide maximum efficiency within a process.

For an introduction to the role of Apple event dispatchers, see "The Mac OS 8 Event Model" (page 1-6).

### Dispatcher References

Every task that calls the `AEReceive` function (page 2-34) must provide a reference to an Apple event dispatcher as an input parameter. A reference to an Apple event dispatcher, which must be created by `AECreateEventDispatcher` (page 2-11) or `AEGetDefaultDispatcher` (page 2-10), identifies a particular dispatcher within a single process.

```
typedef struct OpaqueAEDispatcher* AEDispatcher;
```

Several other Apple Event Manager functions, including `AEGetDispatcherTopHandlerTable` (page 2-33), `AEPushDispatcherHandlerTable` (page 2-30), and `AEPopDispatcherHandlerTable` (page 2-31), also take a dispatcher reference as an input parameter.

Dispatcher references provide maximum efficiency within the scope of a single process. To refer to a dispatcher in a manner that remains valid for any process, you must use a dispatcher ID.

## Dispatcher IDs

If you need to identify one of your program's dispatchers for use outside the scope of your program, for example by a server program from which you are requesting a service, you must use a dispatcher ID rather than a dispatcher reference.

```
typedef struct OpaqueAEDispatcherID* AEDispatcherID;
```

For example, when you specify the address of the reply event for `AESendEventQueueReply`, you must use a dispatcher ID if you want to specify a specific dispatcher as the target of the reply event, because the dispatcher can potentially be any dispatcher in any process.

To get a dispatcher ID for one of the dispatchers associated with your program, you pass a dispatcher reference to the `AEGetEventDispatcherID` function (page 2-13).

# Handler Table References

The stack of Apple event handler tables provided by each Apple event dispatcher consists of a default handler table plus additional handler tables, such as those installed by an application.

A reference to a handler table, which is first allocated by `AENewHandlerTable` (page 2-17) or `AENewFilterHandlerTable` (page 2-18), identifies a particular handler table.

```
typedef struct OpaqueAEHandlerTable* AEHandlerTableRef;
```

You can use this reference with the `AEInstallHandler` (page 2-24), `AEPushDispatcherHandlerTable` (page 2-30), and `AEPopDispatcherHandlerTable` (page 2-31) functions to add handlers to the table and to add the table to or remove it from a particular dispatcher's handler table stack.

To get a new reference to an existing handler table, use the `AEShareHandlerTable` function (page 2-21). `AEShareHandlerTable` increments the reference count for the handler table and returns a new reference. When you are finished with any reference to a handler table, call the `AEDisposeHandlerTable` function (page 2-23). `AEDisposeHandlerTable` decrements the reference count, disposing of the original object only when the reference count reaches 0.

## Receive Modes

When you call the `AEReceive` function (page 2-34), you specify in the `receiveMode` parameter how long you want the call to `AEReceive` to last.

The enumerators for receive modes are defined by the `AEReceiveMode` data type.

```
typedef UInt32 AEReceiveMode;
enum {
    kAEReceiveForever               = 0x00000000,
    kAEReceiveOneEvent              = 0x00000001,
};
```

#### Enumerator descriptions

`kAEReceiveForever` Receive events until the calling task terminates, a handler returns an error, or a handler intentionally terminates the call to `AEReceive` by returning `errAEReceiveEscapeCurrent`. This is the most commonly used receive mode.

`kAEReceiveOneEvent` Receive a single event, then return. This can be useful, for example, for an additional task spawned by a main task to handle a single event.

## Constants for Use With Send Functions

### Apple Event Send Options

The `AESendEvent` (page 2-37), `AESendEventQueueReply` (page 2-39), and `AESendEventToSelf` (page 2-41) functions allow you to specify several options of type `AESendOptions` when you send an event.

```
typedef OptionBits AESendOptions;
enum {
    kAENeverInteract    = 0x00000010,
    kAECanInteract      = 0x00000020,
    kAEAlwaysInteract   = 0x00000030,
    kAECanSwitchLayer   = 0x00000040,
    kAEDontRecord       = 0x00001000,
    kAEDontExecute      = 0x00002000
};
```

#### Enumerator descriptions

The enumerators `kAENeverInteract`, `kAECanInteract`, `kAEAlwaysInteract`, and `kAECanSwitchLayer` are supported as documented in *Inside Macintosh: Interapplication Communication* for backward compatibility only. Information about the way the Mac OS 8 Apple Event Manager handles user interaction will be available with later developer releases. Mac OS 8 fully supports the `kAEDontRecord` and `kAEDontExecute` options.

`kAEDontRecord` Your application is sending an event to itself but does not want the event recorded. When Apple event recording is on, the Apple Event Manager records a copy of every event your application sends to itself except for those events for which this option is set.

`kAEDontExecute` Your application is sending an Apple event to itself for recording purposes only—that is, you want the Apple Event Manager to send a copy of the event to the recording process but you do not want your application actually to receive the event.

## Apple Event Send Priorities

The `AESendEvent` (page 2-37) and `AESendEventQueueReply` (page 2-39) functions allow you to specify priorities of type `AESendPriority` when you send an event.

```
typedef SInt16 AESendPriority;
enum {
    kAENormalPriority   = 0x00000000,   /* post event at back of event queue */
    kAEHighPriority     = 0x00000001    /* post event at front of event queue */
};
```

#### Enumerator descriptions

`kAENormalPriority` Post event in the normal fashion at the back of the event queue, so it gets handled after all other pending events.

`kAEHighPriority` Post event at the front of the event queue, so it gets handled before any other pending events.

## Apple Event Handler

For each Apple event you want your program to handle, you must provide an Apple event handler. An Apple event handler performs any action requested by the Apple event, adds parameters to the reply Apple event if appropriate, and returns a result code.

```
typedef OSStatus (*AEEventHandlerProc)(
                    const AppleEvent *theAppleEvent,
                    AppleEvent *reply,
                    void *handlerRefcon,
                    AEHandlerTableRef handlerTable);
```

You use the `AEInstallHandler` function (page 2-24) to install an Apple event handler in a handler table, and the `AEPushDispatcherHandlerTable` function (page 2-30) to add a handler table to a particular dispatcher's stack of handler tables. For information about writing your own Apple event handlers, see "Application-Defined Function" (page 2-42).

# Apple Event Communications Functions

This section describes the Apple Event Manager functions you can use in Mac OS 8 to send, receive, dispatch, and handle Apple events.

When first launched, your program creates one or more Apple event handler tables and installs them in one or more Apple event dispatchers. The handler tables inform the Apple Event Manager which events your program is interested in receiving, and the handlers identified by the table entries define your program's responses to different kinds of events.

To manipulate Apple event dispatchers and install handler tables in them, use the functions described in "Creating and Manipulating Apple Event Dispatchers" (page 2-10) and "Creating and Manipulating Handler Tables" (page 2-16).

After your program has installed its handler tables, any tasks it creates call the `AEReceive` function (page 2-34), which blocks the calling task until an event your program can handle arrives.

Before you can send an Apple event, you must create it using functions described in *Inside Macintosh: Interapplication Communication*. After you have

created an Apple event, you can use one of the three functions described in "Sending Events" (page 2-35) to send it.

Table 2-1 summarizes the functions described in this chapter and the System 7 functions they replace. The new functions allow you to take full advantage of the Mac OS 8 event model.

**Table 2-1** Apple event communications functions: System 7 compared with Mac OS 8

| Programming domain | System 7 functions | Equivalent functions in Mac OS 8 |
|---|---|---|
| Manipulating system and application dispatch tables | System 7 uses system and application Apple event dispatch tables to dispatch events.<br><br>System 7 defines these functions for creating and maintaining system and application dispatch tables:<br><br>`AEInstallEventHandler`<br>`AEGetEventHandler`<br>`AERemoveEventHandler` | Mac OS 8 uses Apple event dispatchers and Apple event handler tables to dispatch events.<br><br>Mac OS 8 replaces the System 7 functions related to dispatching with these functions:<br><br>`AEGetDefaultDispatcher`<br>`AECreateEventDispatcher`<br>`AEGetEventDispatcherID`<br>`AEDisposeEventDispatcher`<br><br>`AENewHandlerTable`<br>`AENewFilterHandlerTable`<br>`AEDisposeHandlerTable`<br><br>`AEInstallHandler`<br>`AERemoveHandler`<br>`AEGetHandler`<br><br>`AEPushDispatcherHandlerTable`<br>`AEPopDispatcherHandlerTable`<br>`AEGetDispatcherTopHandlerTable`<br><br>These functions are described in "Creating and Manipulating Apple Event Dispatchers" (page 2-10) and "Creating and Manipulating Handler Tables" (page 2-16). |

**Table 2-1** Apple event communications functions: System 7 compared with Mac OS 8

| Programming domain | System 7 functions | Equivalent functions in Mac OS 8 |
|---|---|---|
| Manipulating special handler dispatch tables | System 7 defines these functions for creating and maintaining the special handler dispatch tables:<br>`AEInstallSpecialHandler`<br>`AEGetSpecialHandler`<br>`AERemoveSpecialHandler`<br>These functions are used for specialized handlers, including object callback functions. | Mac OS 8 defines these functions for creating and maintaining object callback functions:<br>`AEInstallSpecialCallback`<br>`AEGetSpecialCallback`<br>`AERemoveSpecialCallback`<br>Documentation for these functions will be provided with later developer releases. |
| Suspending and resuming event handling | System 7 defines these functions for suspending and resuming event handling:<br>`AESuspendTheCurrentEvent`<br>`AEResumeTheCurrentEvent`<br>`AESetTheCurrentEvent`<br>`AEGetTheCurrentEvent` | Mac OS 8 supports suspension and resumption of event handling for specific events by means of filtered handler tables.<br>Mac OS 8 replaces the System 7 functions for suspending and resuming event handling with `AENewFilterHandlerTable` and related functions listed above for manipulating handler tables. |
| Receiving and processing Apple events | In System 7, all events are received by the function `WaitNextEvent`, which continuously polls the system for new events. Apple events are then dispatched separately by `AEProcessAppleEvent`. | Mac OS 8 provides a single "bottleneck" function, `AEReceive`, that blocks the calling task until an event in which your program is interested arrives. |
| Sending Apple events | System 7 defines the function `AESend` for sending Apple events. | Mac OS 8 defines these functions for sending Apple events:<br>`AESendEvent`<br>`AESendEventQueueReply`<br>`AESendEventToSelf` |

## Creating and Manipulating Apple Event Dispatchers

Every process has a default Apple event dispatcher used by the program's main task, and you can install additional dispatchers as necessary. You must specify the dispatcher you want to use when you call the `AEReceive` function (page 2-34).

To get a reference to the default process dispatcher, use the `AEGetDefaultDispatcher` function (page 2-10). To create one or more additional dispatchers for use by your program, use the `AECreateEventDispatcher` function (page 2-11). To dispose of any dispatcher created by `AECreateEventDispatcher`, use the `AEDisposeEventDispatcher` function (page 2-14). You cannot dispose of a default dispatcher.

If you need to identify one of your program's dispatchers for use by a different program, such as a server program from which you are requesting a service, you must use a dispatcher ID rather than a dispatcher reference. Dispatcher references are valid only within a single process, whereas dispatcher IDs are valid across all processes. To get a dispatcher's ID, you pass a dispatcher reference to the `AEGetEventDispatcherID` function (page 2-13).

To get a reference to the dispatcher associated with a window, use the window's `GetEventDispatcher` method. Documentation for the HI object class `HIWindow` will be available with later developer releases.

Every Apple event dispatcher has a default handler table at the bottom of the handler table stack. You can use the functions described under "Creating and Manipulating Handler Tables" (page 2-16) to add your own handler tables to a handler table stack.

### AEGetDefaultDispatcher

Returns a reference to the default Apple event dispatcher associated with a process.

```
AEDispatcherRef AEGetDefaultDispatcher (void);
```

*function result* A reference to the default dispatcher created automatically when the process associated with your program was created. For more information, see "Dispatcher References" (page 2-3).

**DISCUSSION**

Every process has a default Apple event dispatcher. The Apple Event Manager disposes of the default dispatcher when your program terminates; you shouldn't attempt to dispose of it yourself.

**EXECUTION ENVIRONMENT**

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

**CALLING RESTRICTIONS**

`AEGetDefaultDispatcher` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

**SEE ALSO**

For an overview of related functions, see "Creating and Manipulating Apple Event Dispatchers" (page 2-10).

For an introduction to the role of dispatchers, see "Apple Event Dispatchers" (page 1-8).

## AECreateEventDispatcher

Creates a new Apple event dispatcher.

```
OSStatus AECreateEventDispatcher (
                    AEDispatcherRef *newDispatcher,
                    MemAllocatorRef allocator);
```

`newDispatcher` A pointer to a dispatcher reference. On output, the reference identifies the new dispatcher. For more information, see "Dispatcher References" (page 2-3).

| | |
|---|---|
| `allocator` | A reference to a memory allocator. A memory allocator specifies a memory allocation policy defined by your program. If you want the Apple Event Manager to use its default allocation policy, pass `NIL` in this parameter. If you want the Apple Event Manager to use some other allocation policy, pass a reference to the allocator for that policy in this parameter. |
| *function result* | A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes. |

**DISCUSSION**

A Mac OS 8 application always has a main task and may create additional tasks that perform specialized processing on behalf of the main task. It is usually desirable to associate these additional tasks with additional dispatchers. This arrangement allows an application to use additional tasks for specialized services, such as graphics calculations or database operations, without slowing down the application's responsiveness from a user's point of view.

A Mac OS 8 server program also has a main task and may create additional tasks that perform specialized processing on behalf of the main task. Server programs can use a variety of tasking models.

**EXECUTION ENVIRONMENT**

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

**CALLING RESTRICTIONS**

`AECreateEventDispatcher` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

**SEE ALSO**

For an overview of related functions, see "Creating and Manipulating Apple Event Dispatchers" (page 2-10).

For an introduction to the role of dispatchers, see "Apple Event Dispatchers" (page 1-8).

For an introduction to Mac OS 8 tasking models and the role of Apple event dispatchers in managing tasks, see "Tasking Models" (page 1-11).

## AEGetEventDispatcherID

Gets the dispatcher ID that corresponds to a given dispatcher reference.

```
OSStatus AEGetEventDispatcherID (AEDispatcherRef dispatcher,
                    AEDispatcherID *globalIdentity);
```

`dispatcher` A dispatcher reference. On input, you supply the dispatcher reference for which you want to obtain an equivalent dispatcher ID. For more information, see "Dispatcher References" (page 2-3).

`globalIdentity`
A pointer to a dispatcher ID. On output, this parameter contains the dispatcher ID that corresponds to the dispatcher reference passed in the `dispatcher` parameter. For more information, see "Dispatcher IDs" (page 2-4).

*function result* A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes.

### DISCUSSION

Dispatcher references are valid only within a single process, whereas dispatcher IDs are valid across all processes. You must use dispatcher references with Apple Event Manager functions to manipulate your program's dispatchers. Dispatcher references provide an optimized means of identification within a single process, but they aren't valid as targets for events sent from other processes. If you need to identify one of your program's dispatchers in events sent from other processes, you must use a dispatcher ID rather than a dispatcher reference.

For example, the main task for a database application might use the `AESendEventQueueReply` function (page 2-39) to send a connection request event to a database engine running on a different computer. The main task could also spawn an additional task, with its own separate dispatcher, that performs

a database search when the database engine returns the event that establishes the connection.

To specify the address of this additional task's dispatcher for use by the database engine when it returns the connection event, the main task of the database application must first pass the dispatcher reference to `AEGetEventDispatcherID` to get a dispatcher ID. Although the dispatcher reference is valid within the database application, only the equivalent dispatcher ID is valid for sending the event from the database engine back to one of the original application's dispatchers.

#### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

#### CALLING RESTRICTIONS

`AEGetEventDispatcherID` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

#### SEE ALSO

For an overview of related functions, see "Creating and Manipulating Apple Event Dispatchers" (page 2-10).

For an introduction to the role of dispatchers, see "Apple Event Dispatchers" (page 1-8).

## AEDisposeEventDispatcher

Disposes of an Apple event dispatcher previously created with `AECreateEventDispatcher`.

```
OSStatus AEDisposeEventDispatcher (AEDispatcher deadDispatcher);
```

`deadDispatcher`
A reference to the Apple event dispatcher you want to dispose of. For more information, see "Dispatcher References" (page 2-3).

*function result* A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes.

#### DISCUSSION

You must use the `AEDisposeEventDispatcher` function to dispose of any dispatcher that you create with the `AECreateEventDispatcher` function. Do not use `AEDisposeEventDispatcher` to dispose of the default dispatcher, which the Apple Event Manager disposes of when your program terminates.

You can only dispose of dispatchers associated with your program. Dispatcher references for dispatchers in other processes won't work with `AEDisposeEventDispatcher` or with any other function that takes a dispatcher reference.

#### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

#### CALLING RESTRICTIONS

`AEDisposeEventDispatcher` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

#### SEE ALSO

For an overview of related functions, see "Creating and Manipulating Apple Event Dispatchers" (page 2-10).

# Creating and Manipulating Handler Tables

## Creating, Getting, and Disposing of Handler Tables

You can create two kinds of Apple event handler tables:

- **Nonfiltered handler table.** When a nonfiltered handler table contains no handler for a particular event, the event is passed on to the next event handler table in the handler table stack. To create a new nonfiltered handler table, use the `AENewHandlerTable` function (page 2-17).
- **Filtered handler table.** When a filtered table contains no handler for a particular event, the event is immediately suspended but remains in the event queue to be handled at a later time. To create a new filtered table, use the `AENewFilterHandlerTable` function (page 2-18).

Nonfiltered tables are appropriate for most situations. Filtered handler tables are useful when your program is in a modal state and you want to suspend handling of certain events. For an introduction to the role of handler tables, see "Apple Event Dispatchers" (page 1-8).

When you create any new handler table, you can associate a reference constant with it. For example, if your program installs the same handler in more than one handler table, you can use the handler table reference constant to identify which handler table the handler has been invoked from. To get the reference constant associated with a table, use the `AEGetHandlerTableRefCon` function (page 2-20).

It is often convenient to share a single handler table among several dispatchers. The Apple Event Manager provides life cycle management for handler tables. To do so, it keeps track of references to every handler table your program creates, incrementing the table's reference count whenever a new reference gets created and decrementing the count whenever a reference gets released.

To get a new reference to an existing handler table, use the `AEShareHandlerTable` function (page 2-21). `AEShareHandlerTable` increments the reference count for the handler table and returns a new reference. When you are finished with any reference to a handler table, call the `AEDisposeHandlerTable` function (page 2-23). `AEDisposeHandlerTable` decrements the reference count, disposing of the original object only when the reference count reaches 0.

Every call to `AENewHandlerTable`, `AENewFilterHandlerTable`, or `AEShareHandlerTable` must be matched by an equivalent call to `AEDisposeHandlerTable`.

## AENewHandlerTable

Creates an empty nonfiltered Apple event handler table.

```
OSStatus AENewHandlerTable (
                    AEHandlerTableRef *newTable
                    void *refCon);
```

`newTable` A pointer to a handler table reference. On output, the reference identifies the new handler table. You can pass this reference to the `AEInstallHandler` function (page 2-24) to add handlers to the handler table.

`refCon` A reference constant. On input, you supply whatever reference constant you want to associate with the table. This may consist either of a pointer to another data structure or a simple value; interpretation of the reference constant is entirely up to your program.

*function result* A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes.

### DISCUSSION

The `AENewHandlerTable` function creates a new nonfiltered handler table. The table doesn't contain any handlers until you add them explicitly with the `AEInstallHandler` function.

When a nonfiltered table contains no handler for a particular event, or if it contains a handler that returns `errAEEventNotHandled`, the Apple Event Manager continues searching the remaining handler tables in the handler table stack. In general, you should use nonfiltered handler tables unless your program is in a modal state and you want to temporarily suspend events for which the table has no handlers.

The handler table reference identified by the `newTable` parameter is exactly the same as a shared reference obtained with the `AEShareHandlerTable` function (page 2-21). That is, it is a shared reference to a handler table with a reference count of 1, so that a subsequent call to `AEDisposeHandlerTable` reduces the reference count to 0 and thus causes the Apple Event Manager to deallocate the table.

#### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

#### CALLING RESTRICTIONS

`AENewHandlerTable` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

#### SEE ALSO

For an overview of related functions, see "Creating, Getting, and Disposing of Handler Tables" (page 2-16).

To add any handler table to the handler table stack associated with a particular dispatcher, use the `AEPushDispatcherHandlerTable` function (page 2-30).

For an introduction to the role of handler tables, see "Apple Event Dispatchers" (page 1-8).

## AENewFilterHandlerTable

Creates an empty filtered Apple event handler table.

```
OSStatus AENewFilterHandlerTable (
                    AEHandlerTableRef *newTable
                    void *refCon);
```

| | |
|---|---|
| `newTable` | A pointer to a filtered handler table reference. On output, the reference identifies the new filtered handler table. You can pass this reference to the `AEInstallHandler` function (page 2-24) to add handlers to the handler table. |
| `refCon` | A pointer to a reference constant. On input, you supply whatever reference constant you want to associate with the table. This may consist either of a pointer to another data structure or a simple value; interpretation of the reference constant is entirely up to your program. |
| *function result* | A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes. |

**DISCUSSION**

The `AENewFilterHandlerTable` function creates a new filtered handler table. The table doesn't contain any handlers until you add them explicitly with the `AEInstallHandler` function.

When a filtered table contains no handler for a particular event, the event is immediately suspended but remains in the event queue to be handled at a later time. Filtered handler tables are useful when your program is in a modal state and you want to suspend temporarily events for which the table has no handlers.

The handler table reference identified by the `newTable` parameter is exactly the same as a shared reference obtained with the `AEShareHandlerTable` function (page 2-21). That is, it is a shared reference to a filtered handler table with a reference count of 1, so that a subsequent call to `AEDisposeHandlerTable` reduces the reference count to 0 and thus causes the Apple Event Manager to deallocate the table.

**EXECUTION ENVIRONMENT**

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

**CALLING RESTRICTIONS**

`AENewFilterHandlerTable` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

**SEE ALSO**

For an overview of related functions, see "Creating, Getting, and Disposing of Handler Tables" (page 2-16).

To add handlers to any handler table, use the `AEInstallHandler` function (page 2-24).

To add any handler table to the handler table stack associated with a particular dispatcher, use the `AEPushDispatcherHandlerTable` function (page 2-30).

For an introduction to filtered handler tables and modal states, see "Event Dispatching for Modal States" (page 1-17).

## AEGetHandlerTableRefCon

Gets a handler table's reference constant.

```
OSStatus AEGetHandlerTableRefCon(
                    AEHandlerTableRef table,
                    void **refCon);
```

`table` A reference to the handler table whose reference constant you want to identify. For more information, see "Handler Table References" (page 2-4).

`refCon` A pointer to a reference constant. On output, the reference constant is the one that was associated with the handler table when it was created.

*function result* A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes.

#### DISCUSSION

When you create any new handler table, you can associate a reference constant with it for use by your program. For example, if your program installs the same handler in more than one handler table, you can use the handler table reference constant to identify which handler table the handler has been invoked from.

An Apple event handler always receives, as one of its parameters, a handler table reference for the table from which it was dispatched. To obtain the reference constant associated with that table, the handler should pass the handler table reference to `AEGetHandlerTableRefCon`.

#### CALLING RESTRICTIONS

`AEGetHandlerTableRefCon` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

#### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

#### SEE ALSO

For an overview of related functions, see "Creating, Getting, and Disposing of Handler Tables" (page 2-16).

## AEShareHandlerTable

Creates a handler table reference for a handler table that already exists.

```
OSStatus AEShareHandlerTable(
                    AEHandlerTableRef table,
                    void *newRefcon,
                    AEHandlerTableRef *newSharedReference);
```

| | |
|---|---|
| `table` | A reference to a handler table you want to share among several handler table stacks. |
| `newRefcon` | A reference constant. On input, you supply whatever reference constant you want to associate with the shared handler table reference. Each new reference to the same original table has a unique reference constant. This may consist either of a pointer to another data structure or a simple value; interpretation of the reference constant is entirely up to your program. |
| `newSharedReference` | A pointer to a handler table reference. On output, the reference is unique to the shared table identified by the `table` parameter. |
| *function result* | A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes. |

**DISCUSSION**

In some situations, it may be convenient to install a single handler table in multiple dispatchers. For example, this can facilitate dispatching within an application or, in a server program, the simultaneous execution of identical tasks, such as calculating image transformations. To share a handler table in this way among multiple dispatchers, you use a handler table reference created by `AEShareHandlerTable`.

`AEShareHandlerTable` creates a new handler table reference (with a unique reference constant) to an existing handler table. The Apple Event Manager increases its reference count for each new reference to the same table. When such a reference is passed to `AEDisposeHandlerTable`, the Apple Event Manager decreases the reference count. When this count reaches 0, the Apple Event Manager deallocates the original handler table.

**EXECUTION ENVIRONMENT**

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

#### CALLING RESTRICTIONS

`AEShareHandlerTable` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

#### SEE ALSO

For an overview of related functions, see "Creating, Getting, and Disposing of Handler Tables" (page 2-16).

## AEDisposeHandlerTable

Decrements the reference count for an Apple event handler table and, if the count reaches 0, disposes of the table.

```
OSStatus AEDisposeHandlerTable (AEHandlerTableRef table);
```

| | |
|---|---|
| `table` | A reference to the handler table you want to dispose of. For more information, see "Handler Table References" (page 2-4). |
| *function result* | A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes. |

#### DISCUSSION

You typically use `AEDisposeHandlerTable` to decrement a handler table's reference count right after using `AEPopDispatcherHandlerTable` (page 2-31) to remove it from a handler table stack. Do not pass a handler table that's still part of a handler table stack to `AEDisposeHandlerTable`.

#### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

**CALLING RESTRICTIONS**

`AEDisposeHandlerTable` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

**SEE ALSO**

For an overview of related functions, see "Creating, Getting, and Disposing of Handler Tables" (page 2-16).

## Installing, Getting, and Removing Handlers

Once you've created an Apple event handler table, you call the `AEInstallHandler` function (page 2-24) repeatedly to install entries in the table.

To get a pointer to the handler for a particular entry, use the `AEGetHandler` function (page 2-26). To remove a handler, use the `AERemoveHandler` function (page 2-28).

When you've finished installing handlers in a handler table, you can add it to a dispatcher's handler table stack. For details, see "Pushing and Popping Handler Tables" (page 2-29).

**IMPORTANT**

Modifying the entries in a handler table after the table has been added to a handler table stack degrades performance and should be avoided. Whenever possible, you should manipulate a handler table before you add it to or after you have removed it from a handler table stack. ▲

## AEInstallHandler

Installs an entry for an Apple event handler in a handler table.

```
OSStatus AEInstallHandler (
                    AEHandlerTableRef table,
                    AEEventClass handlerClass,
                    AEEventID handlerID,
                    AEEventHandlerProc handler,
                    void *handlerRefcon);
```

| | |
|---|---|
| `table` | A reference to the handler table in which you want to install an entry. For more information, see "Handler Table References" (page 2-4). |
| `handlerClass` | The event class for the Apple event or events to be dispatched for this entry. |
| `handlerID` | The event ID of events handled by the handler. |
| `handler` | A pointer to the handler for this table entry. |
| `handlerRefcon` | A reference constant. On input, you supply whatever reference constant you want to associate with the handler. This may consist either of a pointer to another data structure or a simple value; interpretation of the reference constant is entirely up to your program. |
| *function result* | A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes. |

**DISCUSSION**

Once you've created an Apple event handler table, you call the `AEInstallHandler` function repeatedly to install entries in the table.

The `AEInstallHandler` function creates an entry with the specified event class and event ID in the handler table specified by the `table` parameter for the handler specified by the `handler` parameter. If you use the same handler for several different kinds of events, you should install a separate entry for each event class and event ID.

If the handler table already contains an entry for the specified `handlerClass` and `handlerID`, `AEInstallHandler` overrides the existing handler with the one specified by the `handler` parameter. You can use `AERemoveHandler` to remove it explicitly before you install a new handler with `AEInstallHandler`.

**IMPORTANT**

Although Mac OS 8 supports the use of `typeWildCard` as discussed in *Inside Macintosh: Interapplication Communication* to install a single handler for multiple event classes or event IDs, this strategy degrades performance and should be avoided. ▲

#### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

#### CALLING RESTRICTIONS

`AEInstallHandler` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

#### SEE ALSO

To create a handler table, use the `AENewHandlerTable` function (page 2-17) or the `AENewFilterHandlerTable` function (page 2-18).

To get a reference to the handler table that is currently the topmost handler table for a particular dispatcher, use the `AEGetDispatcherTopHandlerTable` function (page 2-33).

## AEGetHandler

Gets the handler and reference constant from a handler table for a specified event class and event ID.

```
OSStatus AEGetHandler (
                    AEHandlerTableRef table,
                    AEEventClass handlerClass,
                    AEEventID handlerID,
                    AEEventHandlerProc *handler,
                    void **handlerRefcon);
```

`table` A reference to the handler table from which you want to retrieve a handler. For more information, see "Handler Table References" (page 2-4).

`handlerClass` The event class for the Apple event or events dispatched to the handler you want to retrieve.

| | |
|---|---|
| `handlerID` | The event ID of events handled by the handler you want to retrieve. |
| `handler` | A pointer to an Apple event handler. On output, the pointer identifies the handler for the specified `handlerClass` and `handlerId` parameters. |
| `handlerRefcon` | A pointer to a reference constant. On output, the reference constant is the same one the Apple Event Manager passes to the handler each time the handler is called. |
| *function result* | A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes. |

**DISCUSSION**

The `AEGetHandler` function allows your program to examine the handler currently installed in a specified handler table for any event class and event ID.

**EXECUTION ENVIRONMENT**

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

**CALLING RESTRICTIONS**

`AEGetHandler` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

**SEE ALSO**

To remove a handler, use the `AERemoveHandler` function (page 2-28).

To install a handler, use the `AEInstallHandler` function (page 2-24).

## AERemoveHandler

Removes the handler for a specified event class and event ID from a handler table.

```
OSStatus AERemoveHandler (
                    AEHandlerTableRef table,
                    AEEventClass handlerClass,
                    AEEventID handlerID,
                    AEEventHandlerProc handler);
```

`table` A reference to the handler table from which you want to remove a handler. For more information, see "Handler Table References" (page 2-4)

`handlerClass` The event class for the Apple event or events dispatched to the handler you want to remove.

`handlerID` The event ID of events handled by the handler you want to remove.

`handler` A pointer to an Apple event handler. On output, the pointer identifies the handler for the specified `handlerClass` and `handlerID` parameters.

*function result* A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes.

### DISCUSSION

You can use `AERemoveHandler` to remove an existing entry in a handler table before you use `AEInstallHandler` to install a new handler for the same event class and event ID.

### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

**CALLING RESTRICTIONS**

`AERemoveHandler` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

**SEE ALSO**

To install a handler, use the `AEInstallHandler` function (page 2-24).

## Pushing and Popping Handler Tables

Most programs need to add one or more of their own handler tables to the handler table stack to implement unique behaviors. After you've created a handler table and installed handler entries in it, you use the `AEPushDispatcherHandlerTable` function (page 2-30) to add the handler table to a dispatcher's stack of handler tables. By stacking one handler table on top of another, you can augment or override the behavior defined by the lower table with the behavior defined by the higher table.

When you no longer want the behavior defined by the table that is currently topmost in the handler stack, you use the `AEPopDispatcherHandlerTable` function (page 2-31) to remove it. This ability to push and pop handler tables allows you to tailor your program's behavior to different circumstances, including various kinds of modal states.

For an introduction to filtered handler tables and modal states, see "Event Dispatching for Modal States" (page 1-17).

To determine the topmost handler table in a dispatcher's handler table stack—that is, the table the dispatcher checks first for handlers—use the `AEGetDispatcherTopHandlerTable` function (page 2-33).

**IMPORTANT**

Modifying the entries in a handler table after the table has been added to a handler table stack degrades performance and should be avoided. Whenever possible, you should manipulate a handler table before you add it to or after you have removed it from a handler table stack. ▲

## AEPushDispatcherHandlerTable

Adds a handler table to the top of a dispatcher's handler table stack.

```
OSStatus AEPushDispatcherHandlerTable (
                    AEDispatcherRef dispatcher,
                    AEHandlerTableRef table);
```

`dispatcher` A reference to the dispatcher to which you want to add the handler table specified by the `table` parameter. For more information, see "Dispatcher References" (page 2-3).

`table` A reference to the handler table to be added to the dispatcher specified by the `dispatcher` parameter. For more information, see "Handler Table References" (page 2-4).

*function result* A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes.

**DISCUSSION**

You use the `AEPushDispatcherHandlerTable` function to add handler tables to a specified dispatcher's handler table stack. When the Apple Event Manager searches for an event handler, it starts from the top of the stack and looks down the chain of handler tables until it finds a match. Thus, as handler tables stack up, their handlers augment or override the behavior defined by handlers in the tables below them.

You can call `AEPushDispatcherHandlerTable` as many times as necessary, but each call must be matched eventually by a call to `AEPopDispatcherHandlerTable`.

You can suspend events temporarily to enforce a modal state by pushing a filtered handler table, and you can stop suspending events by popping the table. When the modal state begins, the handler for the event that triggers the modal state calls `AEPushDispatcherHandlerTable` to add a filtered handler table on top of the dispatcher's handler table stack. The handler then calls `AEReceive`, which blocks the calling task until an event with an entry in the filtered table arrives.

The call to `AEReceive` from the handler occurs within a previous call to `AEReceive` by the program's main task. When the Apple Event Manager encounters a filtered handler table, it suspends any incoming events that don't have entries in that table. When call to `AEPopDispatcherHandlerTable` removes

the table, the Apple Event Manager dispatches the suspended events in the order in which they were received.

#### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

#### CALLING RESTRICTIONS

`AEPushDispatcherHandlerTable` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

#### SEE ALSO

To pop a handler table that you have added to a handler stack with the `AEPushDispatcherHandlerTable` function, use `AEPopDispatcherHandlerTable` (page 2-31).

For more information about using `AEPushDispatcherHandlerTable` and `AEPopDispatcherHandlerTable`, see "Event Dispatching for Modal States" (page 1-17).

## AEPopDispatcherHandlerTable

Removes the topmost handler table from a dispatcher's handler table stack.

```
OSStatus AEPopDispatcherHandlerTable (
                    AEDispatcher dispatcher,
                    AEHandlerTableRef *table);
```

`dispatcher` A reference to the dispatcher whose topmost handler table you want to remove. For more information, see "Dispatcher References" (page 2-3)

| | |
|---|---|
| `table` | A pointer to a handler table reference. On output, the reference identifies the removed handler table. If you no longer need the removed handler table, pass this reference to the `AEDisposeHandlerTable` function (page 2-23) to dispose of the table. |
| *function result* | A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes. |

#### DISCUSSION

You use the `AEPopDispatcherHandlerTable` function to remove handler tables added to a dispatcher's stack with `AEPushDispatcherHandlerTable`. Every call to `AEPushDispatcherHandlerTable` (page 2-30) must eventually be matched by a separate call to `AEPopDispatcherHandlerTable`.

#### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

#### CALLING RESTRICTIONS

`AEPopDispatcherHandlerTable` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

#### SEE ALSO

To dispose of a handler table that you no longer need, use the `AEDisposeHandlerTable` function (page 2-23).

For more information about using `AEPushDispatcherHandlerTable` and `AEPopDispatcherHandlerTable`, see "Event Dispatching for Modal States" (page 1-17).

## AEGetDispatcherTopHandlerTable

Gets a reference to the topmost handler table in a dispatcher's handler table stack.

```
OSStatus AEGetDispatcherTopHandlerTable(
                    AEDispatcherRef dispatcher,
                    AEHandlerTableRef *table);
```

`dispatcher` A reference to the dispatcher whose topmost handler table you want to get a reference to. For more information, see "Dispatcher References" (page 2-3).

`table` A pointer to a handler table reference. On output, the reference identifies the topmost handler table in the dispatcher identified by the `dispatcher` parameter. You can pass this reference to the `AEInstallHandler` function (page 2-24) to install additional entries in the handler table.

*function result* A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes.

### DISCUSSION

The `AEGetDispatcherTopHandlerTable` function is useful whenever you need to determine the topmost handler table in a dispatcher's handler table stack.

### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

### CALLING RESTRICTIONS

`AEGetDispatcherTopHandlerTable` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

**SEE ALSO**

For an overview of related functions, see "Pushing and Popping Handler Tables" (page 2-29).

## Receiving Events

### AEReceive

Blocks the calling task and dispatches incoming events.

```
OSStatus AEReceive (
                    AEDispatcher waitDispatcher,
                    AEReceiveMode receiveMode);
```

`waitDispatcher` A reference to the dispatcher—and thus the event queue—in which the calling task is interested. For more information, see "Dispatcher References" (page 2-3).

`receiveMode` A receive mode indicating how long this call to `AEReceive` should continue receiving events. You identify the receive mode with one of the values defined in the `AEReceiveMode` enumeration (page 2-5).

*function result* A result code. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes.

**DISCUSSION**

The `AEReceive` function blocks the calling task until an event the program can handle arrives, then reawakens the task and uses the dispatcher specified in the `waitDispatcher` parameter to dispatch the event to the appropriate handler. This cycle of blocking and then waking the task continues until the task terminates a handler generates an error, or a handler returns `errAEReceiveEscapeCurrent`.

All events in Mac OS 8 are conveyed via the "bottleneck" of the `AEReceive` function, which replaces the `WaitNextEvent` function used in earlier versions of

the Mac OS. Except for any initialization work required and the execution of handlers, the calling task generally executes from within `AEReceive`.

#### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

#### CALLING RESTRICTIONS

`AEReceive` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

#### SEE ALSO

For an overview of the role of `AEReceive`, see "The Mac OS 8 Event Model" (page 1-6).

## Sending Events

Before you can send an Apple event, you must first create the event and add the appropriate parameters and attributes. To do so, you use both functions described in *Inside Macintosh: Interapplication Communication* and new functions provided by Mac OS 8.

**Note**
In System 7, the construction of large, hierarchical events requires users to construct every branch of the hierarchy explicitly, recopying each subbranch at each step. The Mac OS 8 Apple Event Manager includes additional streaming functions that allow you to write a series of descriptor records sequentially rather than explicitly inserting them one at a time using `AEPutNthPointer` or `AEPutKeyPtr`. Documentation for these functions will be available with later developer releases. ◆

Once you have created an Apple event, you must use one of the new functions defined by Mac OS 8 to send it. Each function is optimized for a different purpose:

- `AESendEvent` blocks the calling task until the function returns with the reply event (if you provide one). In Mac OS 8, you create the original reply event in the manner described in *Inside Macintosh: Interapplication Communication* or by using the Mac OS 8 function `AECreateReplyAppleEvent`. You can also specify `NIL` for the reply, in which case `AESendEvent` returns immediately after sending the event successfully.
- `AESendEventQueueReply` allows the calling task to keep running, receiving other events as usual from the event queue associated with its dispatcher. The Apple Event Manager creates the reply event, which may return via the original task's dispatcher or potentially via some other dispatcher.
- `AESendEventToSelf` sends an event directly to the handler table stack for one of your program's dispatchers, bypassing the event queue altogether. In Mac OS 8, you create the original reply event in the manner described in *Inside Macintosh: Interapplication Communication* or by using the Mac OS 8 function `AECreateReplyAppleEvent`.
- `AESendDelayed` sends the event at some later time. The calling task uses `AESendDelayed` to send periodic events, such as an event that blinks a cursor, or to delay the sending of an event until a single specified time. Information about this function will be provided with later developer releases.

**IMPORTANT**

Remote addresses, including the use of `typeSessionID` and `typeTargetID` to specify target addresses, will be supported in future developer releases. You can specify a local address with an address descriptor record of type `typeDispatcherID`, `typeProcessSerialNumber`, `typeKernelProcessID`, or `typeApplSignature`. ▲

## AESendEvent

Sends an Apple event and blocks the calling task until the function returns with the reply event or, if the reply is specified as NIL

```
OSStatus AESendEvent(
                    const AppleEvent *theAppleEvent,
                    AppleEvent *reply,
                    AESendOptions sendOpts,
                    AESendPriority sendPriority,
                    Duration timeOutDuration);
```

`theAppleEvent` A pointer to the Apple event to be sent.

`reply` A pointer to a reply Apple event. On input, you must provide the reply event or specify `NIL`. You can use the new Mac OS 8 function `AECreateReplyAppleEvent` to preallocate the reply event. If you don't want a reply, specify `NIL`. On output, the reply event, if any, contains information provided by the recipient. Your program is responsible for using `AEDisposeDesc` to dispose of the descriptor record returned in the reply parameter.

`sendOpts` The send options for the Apple event. You identify the send options you want with the values defined in the `AESendOptions` enumeration (page 2-5). The send options determine whether and how the recipient of the event can interact with the user, whether such interaction can involve a layer switch, and whether the event should be recorded.

`sendPriority` The send priority for the Apple event. You identify the event's send priority with one of the values defined in the `AESendPriority` enumeration (page 2-6). For events originally generated by the user, the send priority determines whether the Apple Event Manager places the Apple event at the back (indicated by `kAENormalPriority`) or the front (indicated by `kAEHighPriority`) of the dispatcher's event queue. Note, however, that events sent by the system may still have a higher priority than other events sent with a send priority of `kAEHighPriority`.

`timeOutDuration`
A value of type `Duration` specifying the length of time the sender is willing to wait for the reply from the recipient before timing out; for example, `kDurationForever` or `kDurationMinute`. (The microkernel defines these and other duration values for use throughout Mac OS 8.)

*function result* If the Apple Event Manager cannot find a handler for an Apple event with the aid of the dispatcher to which the event is addressed, `AESendEvent` returns the result code `errAEEventNotHandled`. If the Apple event is successfully handled, `AESendEvent` returns `noErr`.

#### DESCRIPTION

Use `AESendEvent` either to send events that don't require a reply or to send events whose reply must be received by the calling task before it can continue executing.

#### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

#### CALLING RESTRICTIONS

`AESendEvent` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

#### SEE ALSO

For an overview of related functions, see "Sending Events" (page 2-35).

## AESendEventQueueReply

Sends an Apple event and returns a reply event in the event queue associated with a specified dispatcher or with the default dispatcher for a specified process.

```
OSStatus AESendEventQueueReply(
                    const AppleEvent *theAppleEvent,
                    const AEAddressDesc *replyAddress,
                    AESendOptions sendOpts,
                    AESendPriority sendPriority);
```

`theAppleEvent` A pointer to the Apple event to be sent.

`replyAddress` A pointer to an address descriptor record. On input, you provide a descriptor record of type `typeDispatcherID`, `typeProcessSerialNumber`, `typeKernelProcessID`, or `typeApplSignature`. Use this descriptor record to identify the target of the reply event, which can be any local dispatcher. To get a dispatcher ID for one of your program's dispatchers, you pass a dispatcher reference to `AEGetEventDispatcherID` (page 2-13). If you use a process serial number, kernel process ID, or application signature, the event is routed to the default dispatcher for the specified program's process.

`sendOpts` The send options for the Apple event. You identify the options you want with the values defined in the `AESendOptions` enumeration (page 2-5). The send options determine whether and how the recipient of the event can interact with the user, whether such interaction can involve a layer switch, and whether the event should be recorded.

`sendPriority` The send priority for the Apple event. You identify the event's send priority with one of the values defined in the `AESendPriority` enumeration (page 2-6). For events originally generated by the user, the send priority determines whether the Apple Event Manager places the Apple event at the back (indicated by `kAENormalPriority`) or the front (indicated by `kAEHighPriority`) of the dispatcher's event queue. Note, however, that events sent by the system may still have a higher priority than other events sent with a send priority of `kAEHighPriority`.

*function result* A result code. If the Apple event is successfully sent to the target, `AESendEventQueueReply` returns `noErr`. See "Apple Event Manager Result Codes" (page 2-44) for a list of other possible result codes.

#### DESCRIPTION

`AESendEventQueueReply` allows the calling task to keep running, receiving other events as usual from the event queue associated with its dispatcher. The Apple Event Manager creates the reply event, which may return via the original tasks's dispatcher or potentially via some other dispatcher specified by the `replyAddress` parameter.

Remote addresses for Apple events, including the use of `typeSessionID` and `typeTargetID` to specify target addresses, will be supported in later developer releases.

#### EXECUTION ENVIRONMENT

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

#### CALLING RESTRICTIONS

`AESendEventQueueReply` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

#### SEE ALSO

For an overview of related functions, see "Sending Events" (page 2-35).

## AESendEventToSelf

Sends an event directly to the handler table stack for one of your program's dispatchers, bypassing the event queue altogether.

```
OSStatus AESendEventToSelf(
                    const AppleEvent *theAppleEvent,
                    const AppleEvent *reply,
                    AEDispatcherRef whichDispatcher,
                    AESendOptions sendOpts);
```

`theAppleEvent` A pointer to the Apple event to be sent. The address of the target is ignored.

`reply` A pointer to a reply Apple event. On input, you must provide the reply event. You can use the Mac OS 8 function `AECreateReplyAppleEvent` to preallocate the reply event. If you don't want a reply, specify `NIL` or a null Apple event. On output, the reply event, if any, contains information provided by the recipient. Your program is responsible for using `AEDisposeDesc` to dispose of the descriptor record returned in the reply parameter.

`whichDispatcher` A reference to the dispatcher you want to receive the event.

`sendOpts` The send options for the Apple event. You identify the send options you want with the values defined in the `AESendOptions` enumeration (page 2-5). The send options determine whether and how the recipient of the event can interact with the user, whether such interaction can involve a layer switch, and whether the event should be recorded.

*function result* A result code. If the Apple Event Manager cannot find a handler for an Apple event with the aid of the dispatcher to which the event is addressed, `AESendEventToSelf` returns the result code `errAEEventNotHandled`. If the Apple event is successfully handled, `AESendEventToSelf` returns `noErr`. See "Apple Event Manager Result Codes" (page 2-44) for a list of possible result codes.

**DESCRIPTION**

`AESendEventToSelf` provides the fastest way to send an event to one of your program's dispatchers.

**EXECUTION ENVIRONMENT**

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

**CALLING RESTRICTIONS**

`AESendEventToSelf` cannot be called by hardware interrupt handlers or secondary interrupt handlers.

**SEE ALSO**

For an overview of related functions, see "Sending Events" (page 2-35).

## Application-Defined Function

### MyEventHandler

Handles a specified Apple event.

```
OSStatus MyEventHandler (
                    const AppleEvent *theAppleEvent,
                    AppleEvent *reply,
                    void *handlerRefcon,
                    AEHandlerTableRef handlerTable);
```

`theAppleEvent` A pointer to an Apple event. On input, this is the event to be handled.

| | |
|---|---|
| `reply` | A pointer to a reply Apple event. On input, this is an empty reply event. On output, your program can add any parameters that might be useful to the Apple event's sender. |
| `handlerRefcon` | The reference constant stored in the dispatcher for the event's class and ID. |
| `handlerTable` | A reference to the handler table in which the handler is installed. |
| *function result* | A result code. Your handler should always set its function result to `noErr` if it successfully handles the Apple event. If an error occurs, your handlers should return either `errAEEventNotHandled` or some other nonzero result code. If the error occurs because your handler cannot understand the event, return `errAEEventNotHandled`, in case a handler in the next handler table in the dispatcher's handler table stack can handle it. 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. |

**DISCUSSION**

Your handler uses both functions described in *Inside Macintosh: Interapplication Communication* and new subdescriptor functions provided by Mac OS 8 to extract parameters and attributes from the Apple event and perform any processing required. 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. If any of the parameters include object specifier records, the handler should call `AEResolve` to resolve them—that is, to locate what they describe.

If your handler cannot understand the event, it should return `errAEEventNotHandled`. If the event is impossible to handle as specified, your handler should return the result code returned by whatever function caused the failure, or whatever other result code is appropriate.

For example, suppose your program receives a Get Data event that requests the name of the current printer, and your program cannot handle such an event. In this situation, you should return `errAEEventNotHandled` in case a handler in the next table in the dispatcher's handler table stack can handle it. This strategy

allows your program to take advantage of the default handlers provided by the system.

However, if your program 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 result code, because further attempts to handle the event are pointless.

**EXECUTION ENVIRONMENT**

| Reentrant? | Call at secondary interrupt level? | Call at hardware interrupt level? |
|---|---|---|
| Yes | No | No |

# Apple Event Manager Result Codes

| | | |
|---|---|---|
| `noErr` | 0 | No error |
| `paramErr` | –50 | Parameter error (for example, value of handler pointer is `NIL` or odd) |
| `MemFullErr` | –108 | Not enough room in heap zone |
| `userCanceledErr` | –128 | User canceled an operation |
| `procNotFound` | –600 | No eligible process with specified process serial number |
| `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 runtime 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 |
| `errAERecordingIsAlreadyOn` | –1732 | Attempt to turn recording on when it is already on |
| `errAEReceiveEscapeCurrent` | –1734 | Force only the current call to `AEReceive` to return |
| `errAEEventFiltered` | –1735 | Event has been filtered and should not be propagated |
| `errAEStreamBadNesting` | –1737 | Nesting violation while streaming |
| `errAEStreamAlreadyConverted` | –1738 | Attempt to convert a stream that has already been converted |
| `errAEDescIsNull` | –1739 | Attempt to perform an invalid operation on a null descriptor record |