Event Manager

Contents

Introduction to Events 2-4
Low-Level Events 2-8
Operating-System Events 2-10
High-Level Events 2-13
Priority of Events 2-15
Switching Contexts 2-15
About the Event Manager 2-16
Using the Event Manager 2-17
Obtaining Information About Events 2-18
Processing Events 2-21
Using the WaitNextEvent Function 2-22
Writing an Event Loop 2-24
Setting the Event Mask 2-26
Handling Events in a Dialog Box 2-29
Creating a Size Resource 2-30
Handling Low-Level Events 2-32
Responding to Mouse Events 2-33
Responding to Keyboard Events 2-38
Scanning for a Cancel Event 2-46
Responding to Update Events 2-47
Responding to Activate Events 2-50
Responding to Disk-Inserted Events 2-55
Responding to Null Events 2-57
Handling Operating-System Events 2-58
Responding to Suspend and Resume Events 2-60
Responding to Mouse-Moved Events 2-62
Handling High-Level Events 2-67
Responding to Events From Other Applications 2-69
Searching for a Specific High-Level Event 2-71
Determining the Sender of a High-Level Event 2-72

Sending High-Level Events 2-73
Requesting Return Receipts 2-77
Handling Apple Events 2-78
Event Manager Reference 2-78
Data Structures 2-79
The Event Record 2-79
The Target ID Record 2-81
The High-Level Event Message Record 2-82
The Event Queue 2-83
Event Manager Routines 2-84
Receiving Events 2-84
Sending Events 2-100
Converting Process Serial Numbers and Port Names 2-105
Reading the Mouse 2-108
Reading the Keyboard 2-110
Getting Timing Information 2-112
Application-Defined Routine 2-114
Filter Function for Searching the High-Level Event Queue 2-114
Resource 2-115
The Size Resource 2-115
Summary of the Event Manager 2-120
Pascal Summary 2-120
Constants 2-120
Data Types 2-122
Event Manager Routines 2-123
Application-Defined Routine 2-124
C Summary 2-125
Constants 2-125
Data Types 2-127
Event Manager Routines 2-128
Application-Defined Routine 2-129
Assembly-Language Summary 2-130
Data Structures 2-130
Trap Macros 2-130
Global Variables 2-131
Result Codes 2-132

This chapter describes how your application can use the Toolbox Event Manager to receive information about actions performed by the user, to receive notice of changes in the processing status of your application, and to communicate with other applications.

For example, you can retrieve information from the Toolbox Event Manager that gives your application details about whether the user has pressed a key or the mouse button, whether one of your application's windows needs updating, or whether some other hardware-related or software-related action requires a response from your application.

Your application also uses the Event Manager to support the cooperative, multitasking environment available on Macintosh computers. This environment allows users to switch between many open applications and allows other applications to receive background processing time. By using Event Manager routines, you allow the system software to coordinate the scheduling of processing time between your application and other applications.

The Event Manager and Process Manager maintain the cooperative, multitasking environment. The Process Manager coordinates the scheduling of applications, and the Event Manager communicates information about changes in your application's processing status to your application.

See the chapter "Process Manager" in *Inside Macintosh: Processes* for complete information on how the Process Manager schedules applications for execution.

You can use the Event Manager to communicate with other applications. Your application can also communicate with other applications using the services of the Apple Event Manager.

The Event Manager and Apple Event Manager routines that let your application communicate with other applications depend on the services of the Program-to-Program Communications (PPC) Toolbox. The services performed by the Event Manager and Apple Event Manager meet the needs of most applications for interapplication communication. However, to get additional control or capabilities not provided by the Event Manager or Apple Event Manager, you can choose to access the PPC Toolbox directly. The chapter "Program-to-Program Communications Toolbox" in *Inside Macintosh: Interapplication Communication* describes the PPC Toolbox routines that are available to your application.

For a comparison of the services provided by the Event Manager, Apple Event Manager, and PPC Toolbox, see *Inside Macintosh: Interapplication Communication*. For additional information about Apple events, including descriptions of how to process the required Apple events, see *Inside Macintosh: Interapplication Communication*.

This chapter describes both the Toolbox Event Manager and the Operating System Event Manager. The Operating System Event Manager maintains a queue in which it stores hardware-related occurrences that you may want your application to respond to. The Toolbox Event Manager communicates the information maintained by the Operating System Event Manager to your application. In most cases, your application needs to interact only with the Toolbox Event Manager. In this chapter, the name *Event Manager* refers to the Toolbox Event Manager.

This chapter provides a general introduction to events and then explains how you can use the Event Manager to

- receive keypresses and mouse clicks as input for your application
- receive indication that your application's windows need to be activated or updated
- allow other applications to use the available system resources when your application isn't using them
- communicate with other applications

Introduction to Events

Most Macintosh applications receive information about hardware and software occurrences that require a response from the application, through events. An **event** is the means by which the Event Manager communicates information about user actions, changes in the processing status of the application, and other occurrences that require a response from the application.

The Event Manager communicates information about events that occur through the event record. The `EventRecord` data type defines the event record. The **event record** contains information about what type of event occurred (a mouse click or keypress, for example) and contains additional information associated with the event (for example, for a keypress the Event Manager also reports which key was pressed).

Most Macintosh applications are event-driven—that is, they respond to various changes or occurrences and take action based on the nature of the event. Typically, a Macintosh application repeatedly checks to see if an event has occurred and, if so, responds to the event. If no events are pending, the application can choose to relinquish the processor for a specified amount of time or can perform other tasks before checking again to see whether an event has occurred.

Your application typically retrieves events from the Event Manager and also relinquishes processor time by using the `WaitNextEvent` function. If any events are pending for your application, the `WaitNextEvent` function returns the event to your application. If no events are pending for your application, the `WaitNextEvent` function may allocate processing time to other applications.

When multiple applications are open, the user chooses one to interact with at any given time. The active application (or **foreground process**) is the one currently interacting with the user. The foreground process displays its menu bar, and its windows are in front of the windows of all other applications. (The term **process** refers to an open application or, in some cases, an open desk accessory.)

There can be only one foreground process at any one time; however, multiple processes can exist in the background. A **background process** is a process that is not currently interacting with the user. The foreground process has first priority for accessing the CPU. Other processes can access the CPU only when the foreground process yields time to them.

By using `WaitNextEvent` to retrieve events, you allow other applications to make use of processing time that your application would otherwise not use. When your application is in the background, it in turn can receive processing time when other applications relinquish the CPU. Using `WaitNextEvent` also allows users to switch between multiple open applications.

An application that is in the background can get CPU time but can't interact with the user while it is in the background. (However, the user can choose to bring the application to the foreground—for example, by clicking in one of the application's windows.) An application can also post a notification request using the Notification Manager if the application is in the background and requires the user's attention. Any application that has the `canBackground` flag set in its size (`'SIZE'`) resource is eligible to obtain access to the CPU when it is in the background.

At any given time a process is either in the foreground or the background; a process can switch between the two states at well-defined times.

The Event Manager ensures that switching between applications occurs in a smooth fashion—by sending your application an event when it is about to be suspended and sending it an event when it has processing time again and can resume executing. The Event Manager and Process Manager coordinate this switching and scheduling of processor time among many applications.

Your application can receive various types of events: low-level events, operating-system events, and high-level events.

The Event Manager returns low-level events to your application for occurrences such as the user pressing the mouse button, releasing the mouse button, pressing a key on the keyboard, or inserting a disk. The Event Manager also returns low-level events to your application if your application needs to activate (make changes to a window based on whether it is in front or not) or update (redraw the contents of) one of its windows. When your application requests an event and there are no other events to report, the Event Manager returns a null event.

The Event Manager returns operating-system events to your application when the processing status of your application is about to change or has changed. For example, if a user brings your application to the foreground, the Process Manager sends an event through the Event Manager to your application. Some of the work of reactivating your application is done automatically, both by the Process Manager and by the Window Manager; your application must take care of any further processing needed as a result of your application being reactivated.

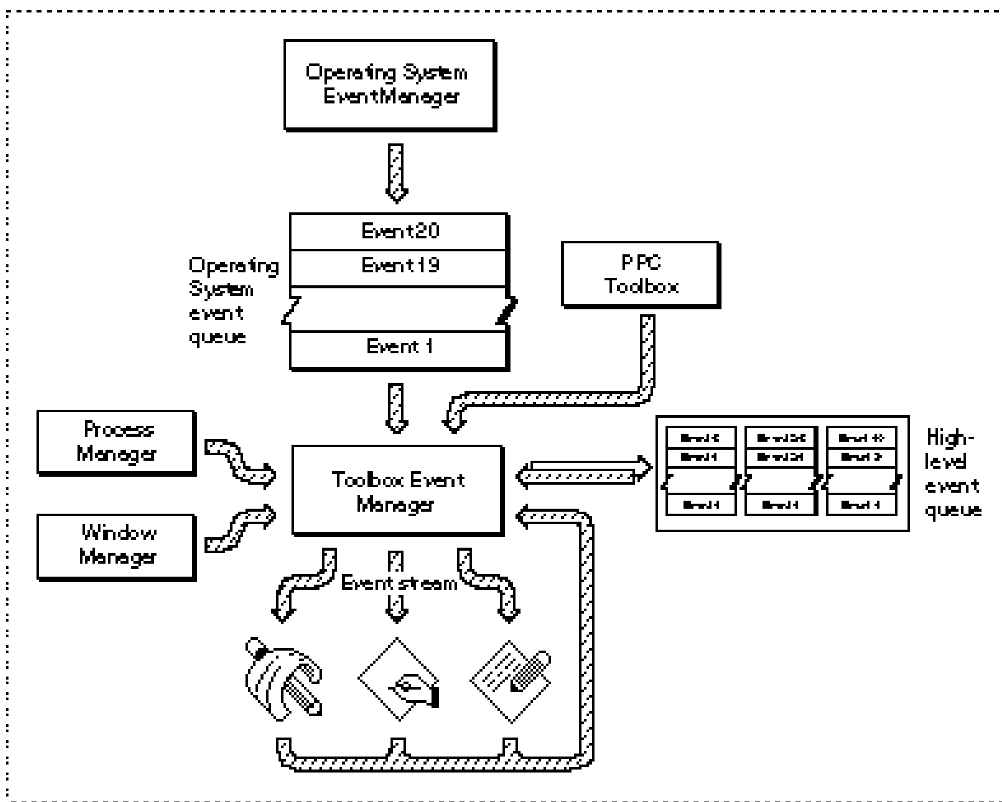
The Event Manager returns high-level events to your application as a result of communication directed to your application from another application or process.

Low-level events, except for update events and null events, are always directed to the foreground process. Operating-system events are also always directed to the foreground process. High-level events, update events, and null events can be directed to the foreground process or background processes.

You can specify which types of events you want your application to receive. You do this by specifying an event mask as a parameter to various Event Manager routines. An **event mask** allows you to mask out the events you are not interested in receiving. For example, you can accept all events except high-level events.

Events can originate from a number of different sources: the Operating System Event Manager, Window Manager, Process Manager, and PPC Toolbox. Figure 2-1 shows the relationships between the Toolbox Event Manager, other parts of the system software, and your application.

Figure 2-1 Sources of events sent to your application



The Operating System Event Manager creates and maintains a queue referred to as the **Operating System event queue.** The Operating System Event Manager detects and reports low-level hardware-related events such as mouse clicks, keypresses, and disk insertions. The Operating System Event Manager places these events in the Operating System event queue. The Toolbox Event Manager retrieves events from this event queue and returns events, one at a time at your application's request, to your application.

A maximum of 20 events can be pending in the Operating System event queue. If the Operating System event queue becomes full, the Operating System Event Manager begins to discard old events to make room for new ones as events are posted. The Operating System Event Manager always discards the oldest event in the queue when it must discard an event. However, this is not a common occurrence; your application typically processes events much faster than the user can generate them. The actual capacity of the event queue is determined by system startup information stored on the startup volume; see the chapter "File Manager" in *Inside Macintosh: Files* for system startup information.

The Event Manager can also report events from the Window Manager and Process Manager. If a window needs to be updated, activated, or deactivated, the Window Manager directs an event to the Toolbox Event Manager. Similarly, the Process Manager directs an event to the Toolbox Event Manager if the processing status of your application changes. The Toolbox Event Manager reports these events to your application.

Note

On computers running System 6, MultiFinder provides some of the capabilities supplied by the Process Manager in System 7. On computers running System 6 without MultiFinder, only a single-application environment is supported. ◆

Your application can use the Event Manager to send and receive high-level events. To transmit high-level events between applications, the Event Manager uses the PPC Toolbox on behalf of your application. For each open application capable of receiving high-level events, the Event Manager maintains a separate queue, referred to as the application's **high-level event queue,** to store high-level events. The size of an application's high-level event queue is limited only by the amount of available memory.

Your application's event stream consists of those events that are available to your application for retrieval when it makes a request for an event. For example, when your application is in the background, its event stream can contain only update events, null events, and high-level events.

When your application asks the Event Manager for the next event, the Event Manager returns the next available event according to its priority. In general, the Event Manager returns events in this order of priority:

1. low-level events
2. operating-system events
3. high-level events

The next sections describe low-level events, operating-system events, and high-level events in greater detail.

Low-Level Events

The Event Manager uses **low-level events** to report very low-level hardware and software occurrences. Low-level events report

- actions by the user (such as pressing the mouse button, typing on the keyboard, or inserting a disk)
- changes in windows on the screen
- that the Event Manager has no other events to report

Low-level events that report actions by the user include mouse-down, mouse-up, key-down, key-up, auto-key, and disk-inserted events. The Event Manager reports any of these events when the user performs the action associated with each event.

Mouse-down and **mouse-up events** report that the user pressed or released the mouse button. For these events the Event Manager returns the location of the cursor at the time the mouse button was pressed or released. **Key-down** and **key-up events** report that the user pressed or released a key. **Auto-key events** report that the user has held a key down for a certain amount of time. For keyboard-related events, the Event Manager reports which key was pressed. For mouse-related and keyboard-related events, the Event Manager also reports the state of the **modifier keys** (the Option, Command, Caps Lock, Control, and Shift keys) at the time of the event.

When the user inserts a disk, the Operating System attempts to mount the volume on the disk by calling the File Manager function `PBMountVol`. The Operating System Event Manager then generates a **disk-inserted event.** If the user is interacting with a standard file dialog box, the Standard File Package intercepts the disk-inserted event and handles it. Otherwise, the event is left in the event queue for your application to retrieve. In most cases your application can handle unexpected disk-inserted events by simply checking to see if the volume was successfully mounted.

Update events and activate events are two types of low-level events that the Event Manager can report as a result of changes in the appearance of windows on the screen. For example, if a user is working with several open documents belonging to your application, you can allow the user to switch from one document to another when the user clicks in the appropriate window. You can determine whether the user clicked in another window by using the Window Manager function `FindWindow`; if the user clicked in another window, you can then use the Window Manager procedure `SelectWindow` to generate the necessary activate events. Before the Event Manager sends your application any activate events relating to this occurrence, the Window Manager does some work for you, such as unhighlighting the deactivated window and highlighting the newly activated window. At your application's next request for an event, the Event Manager returns an activate event.

An **activate event** indicates the window involved and whether the window is becoming activated or deactivated. Your application should perform any other necessary actions to complete the transformation of the window from active to inactive or vice versa. For example, when a window becomes active, your application should show any scroll bars and restore any selections.

Your application typically receives an activate event for the window being deactivated, followed by an activate event for the window becoming active at your application's next request for an event.

Note

If the user switches between your application and another application (by clicking in the window of another application, for example), your application is responsible for activating or deactivating any windows as appropriate. Your application is notified of this occurrence through operating-system events. If your application has the `acceptSuspendResumeEvents` and `doesActivateOnFGSwitch` flags set in its `'SIZE'` resource, your application is notified of the switch through an operating-system event and does not receive a separate activate event when the user switches between applications. ◆

The Window Manager generates **update events** to control the appearance of windows on the screen. The Window Manager keeps track of the front-to-back ordering of windows and allows windows to overlap other windows. The Window Manager coordinates the display of windows. When one window covers another window and then the user moves the first window, the Window Manager generates an update event so that the contents of the newly exposed area can be updated. The Event Manager reports update events as needed to the applications whose windows need updating. Unlike other low-level events, update events can be directed to the foreground process or background processes.

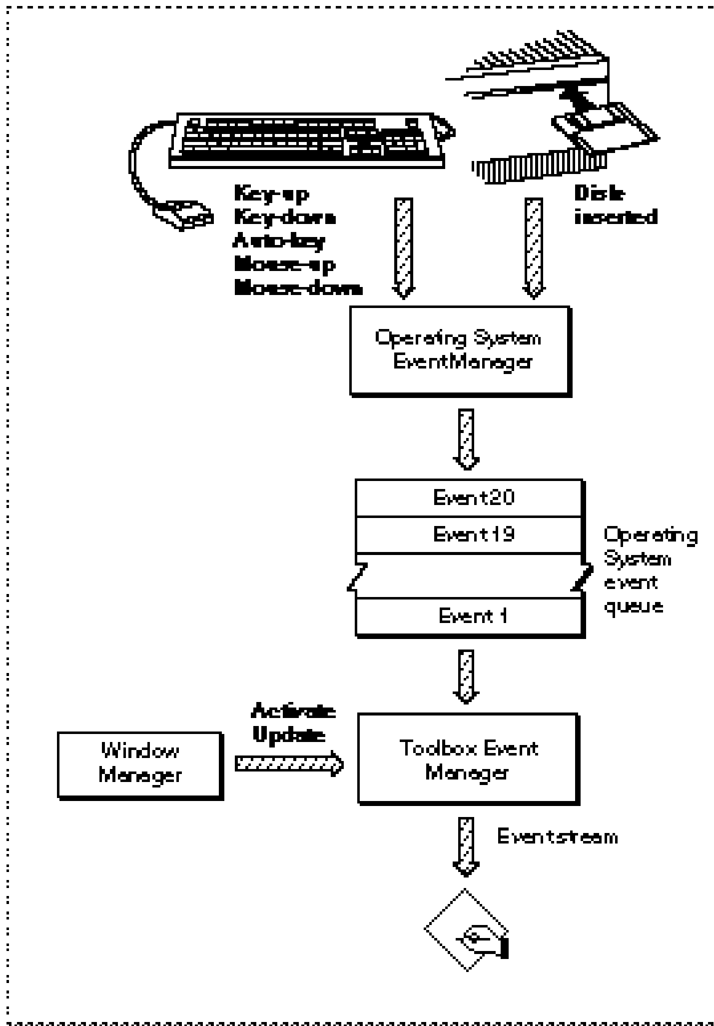
Activate and update events generated by the Window Manager are not placed into the Operating System event queue but are sent directly to the Event Manager.

The Event Manager reports a **null event** when your application requests an event and your application's event stream does not contain any of the requested event types. By using the `WaitNextEvent` function, you can yield time to other processes when null events are the only pending events for your application.

When your application receives a null event, your application can do idle processing (such as blinking the caret) if it is in the foreground or do other tasks if it is in the background. If you want your application to receive null events when it is in the background, you must set the `canBackground` flag in your application's `'SIZE'` resource. If your application does not perform any processing in response to null events when it is in the background, then set the `cannotBackground` flag. If you set the `cannotBackground` flag, the Event Manager does not report null events to your application when it is in the background. However, the Event Manager still reports update events (and high-level events if the `isHighLevelEventAware` flag is set in the `'SIZE'` resource) to your application when it is in the background regardless of how the background flag is set.

Figure 2-2 shows the various kinds of low-level events your application can receive. See "Handling Low-Level Events" beginning on page 2-32 for complete details of how your application should respond to low-level events.

Figure 2-2 Low-level events



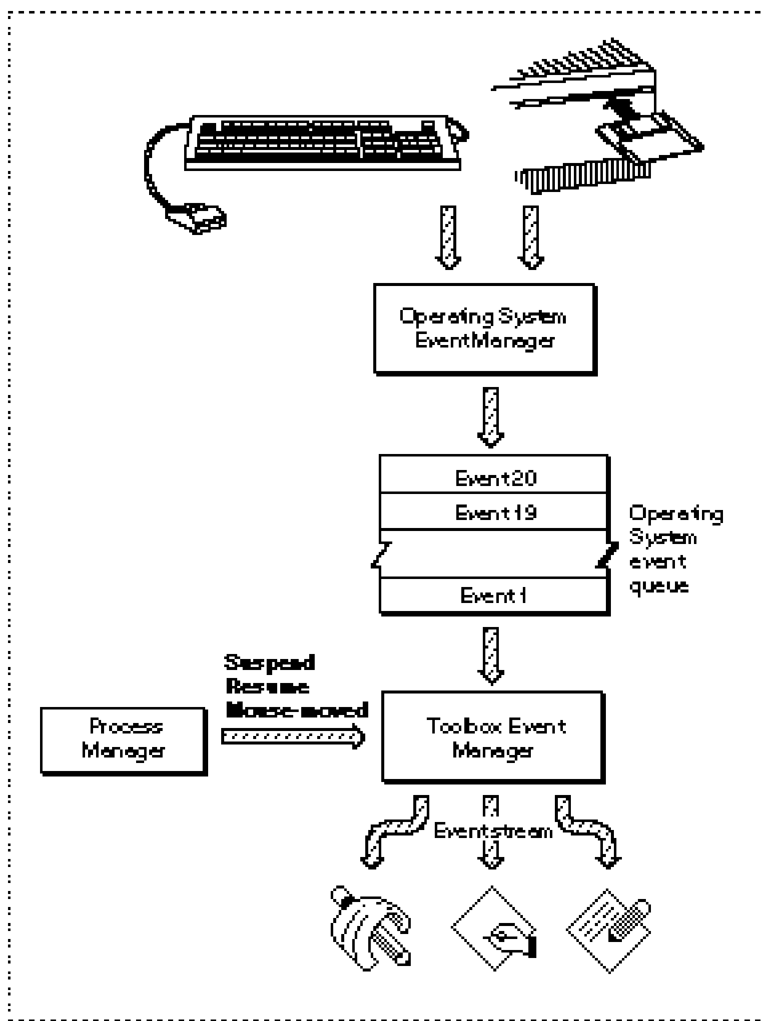
Operating-System Events

The cooperative, multitasking environment allows the user to interact with your application and with other applications. The Process Manager coordinates the scheduling of applications, and the Event Manager communicates information about changes in the operating status of applications to the applications involved.

For example, when your application is about to be switched into the background, the Event Manager sends it a **suspend event.** Then, when your application is switched back into the foreground, it receives a **resume event.** These types of events, as well as a special type of mouse event, the **mouse-moved event,** are known as **operating-system events.**

Figure 2-3 illustrates how the Event Manager helps provide this cooperative, multitasking environment. The Process Manager generates suspend, resume, and mouse-moved events, and the Event Manager reports these events to applications.

Figure 2-3 Operating-system events



Note

If your application sets the `acceptSuspendResumeEvents` and `doesActivateOnFGSwitch` flags in its `'SIZE'` resource, your application is also responsible for activating or deactivating any windows as appropriate in response to operating-system events. For maximum compatibility, your application should set these flags and handle suspend and resume events. See "The Size Resource" beginning on page 2-115 for more information on these and other flags in the `'SIZE'` resource. ◆

When your application receives a suspend event, it does not actually switch to the background until it makes its next request to receive events from the Event Manager. At the time that it receives the suspend event, your application should convert any private scrap into the global scrap if necessary. Your application should hide scroll bars, remove the highlighting from any selections, and hide any floating windows. If your application

shows a window that displays the Clipboard contents, you should hide this window also. Then you should call `WaitNextEvent` to relinquish the CPU and allow the Operating System to schedule other processes for execution. It is important to minimize the processing you do in response to a suspend event so that the computer appears responsive to the user.

When control returns to your application, the first event it receives is a resume event. Your application should convert the global scrap back to its private scrap, if necessary. Your application should also restore any windows to the state the user left them in at the time of the previous suspend event. For example, your application should show any scroll bars, highlight any selections, and show any floating windows. See "Responding to Suspend and Resume Events" beginning on page 2-60 for complete details of how your application should respond to these events.

The events that your application can receive in the background are update, null, and high-level events. When your application is in the background, it should not perform any processing that would make the foreground process appear unresponsive to the user. When receiving events in the background, your application should perform any needed action in response to an event and then quickly return.

Your application should never interact with the user when it is in the background. If you need to notify the user of some special occurrence while your application is executing in the background, you should use the Notification Manager to queue a notification request. You should not attempt to display an alert box while your application is in the background. Instead, your application can specify that the Notification Manager play a sound, display an alert box, cause a small icon representing your application to blink in alternation with the Application menu icon, display a diamond next to your application's name in the Application menu, or put a combination of these actions into effect.

These actions are designed to alert the user that another application needs the user's attention. By using the Notification Manager you help maintain the user interface principle of giving the user control, as the user can choose to bring the application requesting attention to the foreground at the user's convenience. See the chapter "Notification Manager" in *Inside Macintosh: Processes* for examples of how to post notification requests.

Another kind of operating-system event is the mouse-moved event. You can request that the Event Manager send your application a mouse-moved event whenever the cursor is outside of a region that you specify to the `WaitNextEvent` function. For example, you can use mouse-moved events as a convenient way for your application to change the appearance of the cursor as the user moves the cursor from the text area of a document to the scroll bar. See "Responding to Mouse-Moved Events" beginning on page 2-62 for detailed examples.

High-Level Events

The Event Manager provides routines that let applications communicate with each other by exchanging high-level events. A **high-level event** is an event that your application can send to another application to give it some information, to receive some information from it, or to have it perform some action.

For example, your application can send a high-level event to another application instructing that application to perform a specific action, such as adding a row to a spreadsheet or changing the font size of a paragraph. Your application can also send a high-level event to another application requesting information from that application—for example, requesting a dictionary application to return the definition of a particular word. When you send a high-level event to another application, you can also include additional information or commands in an optional data buffer. For example, your application can use a high-level event to send a list of new words and definitions to a dictionary application.

Note
High-level events are available only in system software version 7.0 or later. ◆

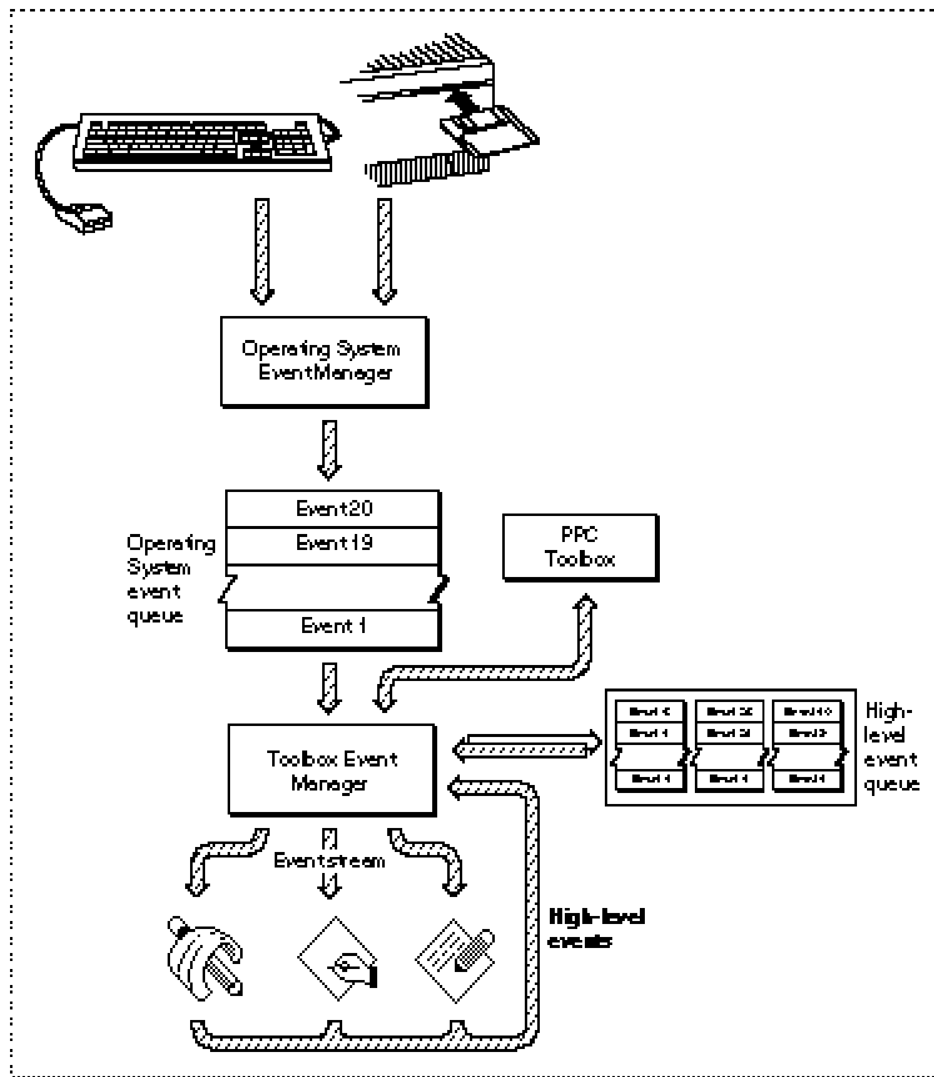
Figure 2-4 on the next page shows three different applications communicating with each other by sending and receiving high-level events. The Event Manager uses the PPC Toolbox to transmit high-level events. The Event Manager maintains a high-level event queue for each application that has identified itself as capable of receiving high-level events. The high-level event queues are limited in size only by available memory.

For effective communication between applications, your application must define the set of high-level events it responds to and let other applications know the events it accepts. By implementing the capabilities to send events to and receive events from other applications, you allow other applications to interact with your application and provide enhanced capabilities to your users.

Generally, there is no restriction on the type of processing that one application can request from another by sending it a high-level event. For a high-level event sent by one application to be understood by another application, however, the sender and receiver must agree on a protocol, that is, on the way the event is to be interpreted. **Apple events** are high-level events whose structure and interpretation are determined by the Apple Event Interprocess Messaging Protocol (AEIMP).

Your application should support the required Apple events, as described in *Inside Macintosh: Interapplication Communication*. The Finder uses the required Apple events to provide your application with information when it is opened and to give it the names of documents to open or print when the user opens or prints documents from the Finder.

Figure 2-4 High-level events



In addition, you may want your application to support other common Apple events. For example, the Edition Manager uses Apple events to communicate information about document sections among the various applications that may publish sections or subscribe to them. The Edition Manager sends the appropriate Apple events to applications that want to maintain up-to-date subscriber sections within their documents. If a user alters a section of a document that has previously been published and updates the edition, the Edition Manager might post an Apple event to the application indicating that a new edition is available. The application receiving the Apple event can then update the subscriber or ignore the information, as the user dictates. For complete information on responding to Apple events sent by the Edition Manager, see the chapter "Edition Manager" in *Inside Macintosh: Interapplication Communication*.

To ensure compatibility and smooth interaction with other Macintosh applications, you should use the Apple event protocol for high-level events whenever possible. You should define new protocols only if your application must communicate with applications on other computers that use different protocols or if your application has other special needs. For complete information about Apple events and about implementing the required set of Apple events, see *Inside Macintosh: Interapplication Communication*.

Note
All Macintosh system software that sends or receives high-level events uses the Apple events protocol. ◆

Priority of Events

Each type of event has a certain priority. The Event Manager returns events in this order of priority:

1. activate events
2. mouse-down, mouse-up, key-down, key-up, and disk-inserted events in FIFO (first-in, first-out) order
3. auto-key events
4. update events (in front-to-back order of windows)
5. operating-system events (suspend, resume, mouse-moved)
6. high-level events
7. null events

Several of the Event Manager routines can be restricted to operate on one or more specific types of events. You do this by disabling (or "masking out") the events you are not interested in receiving. See "Setting the Event Mask" beginning on page 2-26 for details about specifying the types of events you wish to receive.

Switching Contexts

Processes running in the background receive processing time when the foreground process makes an event call (that is, calls `WaitNextEvent` or `EventAvail`) and there are no events pending for that foreground process. A process running in the background should relinquish the CPU regularly to ensure a timely return to the foreground process when necessary.

In System 7 (or with MultiFinder in earlier versions), the available processing time is distributed among multiple processes through a procedure known as *context switching* (or just *switching*). All switching occurs at a well-defined time, namely, when an application calls `WaitNextEvent`. When a context switch occurs, the Process Manager allocates processing time to a process other than the one that had been receiving processing time. Two types of context switching may occur: major and minor.

A **major switch** is a complete context switch: an application's windows are moved from the back to the front, or vice versa. In a major switch, two applications are involved, the one being switched to the foreground and the one being switched to the background. The Process Manager switches the A5 worlds of both applications, as well as the relevant low-memory environments. If those applications receive suspend and resume events, they are so notified at the time that a major switch occurs.

A **minor switch** occurs when the Process Manager gives time to a background process without bringing the background process to the front. The two processes involved in a minor switch can be two background processes or a foreground process and a background process. As in a major switch, the Process Manager switches the A5 worlds and the low-memory environments of the two processes. However, the order of windows is not switched, and neither process receives either suspend or resume events.

When the frontmost window is an alert box or a modal dialog box, major switching does not occur, although minor switching can. To determine whether major switching can occur, the Operating System checks (among other things) to see if the window definition procedure of the frontmost window is `dBoxProc`, because the type `dBoxProc` is specifically reserved for alert boxes and modal dialog boxes. (If the frontmost window is a movable modal dialog box, major switching can still occur.)

Note
Your application can also get switched out if it calls a system software routine that makes an event call. For example, when your application calls `ModalDialog`, a minor switch can occur. ◆

Your application can receive processing time and perform tasks in the background, but your application should not interact with the user or perform tasks that would slow down the responsiveness of the foreground process.

Your application indicates scheduling options to the Operating System, such as whether the application can use null-event processing time when in the background, whether it can accept suspend and resume events, and so forth, by setting flags in its **size** (`'SIZE'`) **resource.** Every application executing in System 7, as well as every application executing in System 6 with MultiFinder, should contain a `'SIZE'` resource. See "Creating a Size Resource" beginning on page 2-30 for details on how to specify this information.

About the Event Manager

The Toolbox Event Manager provides routines that communicate information about actions performed by the user and give notice of changes in the processing status of your application. The Event Manager also provides routines that your application can use to communicate with other applications. You can control the scheduling of your application for execution by using the Event Manager.

The rest of this chapter explains

- how to structure your main event loop to receive and process events
- how to create a `'SIZE'` resource to specify your application's memory requirements and scheduling options
- how to respond to most types of events
- how to receive and process high-level events
- how to send high-level events to other applications

Using the Event Manager

You can use the Event Manager to receive information about hardware-related events, about changes in the appearance of your application's windows, or about changes in the operating status of your application. You can also use the Event Manager to communicate directly with other applications. This communication can include sending events to other applications, receiving events from other applications, and searching for specific events from other applications.

Your application can both send and receive high-level events, but it generally only receives low-level events and should not send them. Your application receives low-level events, operating-system events, and high-level events in the same way, which is by asking the Event Manager for the next available event. If the event your application receives is a high-level event, your application might need to use another Event Manager or Apple Event Manager routine to retrieve an optional data buffer and additional information accompanying that event.

Before using the Event Manager, you can use the `Gestalt` function to determine if certain features of the Event Manager are available. See the chapter "Gestalt Manager" in ***Inside Macintosh: Operating System Utilities*** for information on the `Gestalt` function.

If your application sends or receives high-level events, you should use the `Gestalt` function with the `gestaltPPCToolboxAttr` selector to determine whether the PPC Toolbox is present. Use the `Gestalt` function with the `gestaltOSAttr` selector to see if the Process Manager is available. If the PPC Toolbox and the Process Manager are present, then the system software provides support for high-level events.

If your application sends or receives Apple events, use the `Gestalt` function with the `gestaltAppleEventsAttr` selector to determine whether the Apple Event Manager is available.

Your application needs to initialize QuickDraw, the Font Manager, and the Window Manager before using the Event Manager. Your application can accomplish this initialization by using the `InitGraf`, `InitFonts`, and `InitWindows` procedures.

When your application starts, you can call the `FlushEvents` procedure to empty the Operating System event queue of any low-level events left unprocessed by another application. For example, you might want to remove any mouse-down events or keyboard events that the user might have entered while the Finder launched your application.

This section shows how to retrieve events from the Event Manager, how to mask out unwanted events, how to specify memory and scheduling options for your application, and how to handle each type of event received from the Event Manager.

Obtaining Information About Events

You get information about events through the event record. The `EventRecord` data type defines the event record and has this structure:

```
TYPE  EventRecord =
      RECORD
         what:       Integer;       {event code}
         message:    LongInt;       {event message}
         when:       LongInt;       {ticks since startup}
         where:      Point;         {mouse location}
         modifiers:  Integer;       {modifier flags}
      END;
```

Field descriptions

`what` The `what` field indicates the type of event received. The type of event can be identified by these constants:

```
CONST
nullEvent          = 0;  {no other pending events}
mouseDown          = 1;  {mouse button pressed}
mouseUp            = 2;  {mouse button released}
keyDown            = 3;  {key pressed}
keyUp              = 4;  {key released}
autoKey            = 5;  {key repeatedly held down}
updateEvt          = 6;  {window needs updating}
diskEvt            = 7;  {disk inserted}
activateEvt        = 8;  {activate/deactivate window}
osEvt              = 15;{operating-system event-- }
                         { resume, suspend, or }
                         { mouse-moved}
kHighLevelEvent    = 23;{high-level event}
```

`message` The `message` field contains additional information associated with the event. The interpretation of this information depends on the event type. The contents of the `message` field for each event type are summarized here:

Event type	Event message
null, mouse-up, mouse-down	Undefined.
key-up, key-down, auto-key	Character code and virtual key code in low-order word. For Apple Desktop Bus (ADB) keyboards, the low byte of the high-order word contains the ADB address of the keyboard where the keyboard event occurred. The high byte of the high-order word is reserved.
update, activate	Pointer to the window to update, activate, or deactivate.
disk-inserted	Drive number in low-order word, File Manager result code in high-order word.
resume	The `suspendResumeMessage` constant in bits 24–31 and a 1 in bit 0 to indicate the event is a resume event. Bit 1 contains either a 1 or a 0 to indicate if Clipboard conversion is required, and bits 2–23 are reserved.
suspend	The `suspendResumeMessage` constant in bits 24–31 and a 0 in bit 0 to indicate the event is a suspend event. Bit 1 is undefined, and bits 2–23 are reserved.
mouse-moved	The `mouseMovedMessage` constant in bits 24–31. Bits 2–23 are reserved, and bit 0 and bit 1 are undefined.
high-level	Class of events to which the high-level event belongs. The `message` and `where` fields of a high-level event define the specific type of high-level event received.

`when` The `when` field indicates the time when the event was posted (in ticks since system startup). When needed, you can use the `when` field to compare how much time has elapsed between successive mouse events.

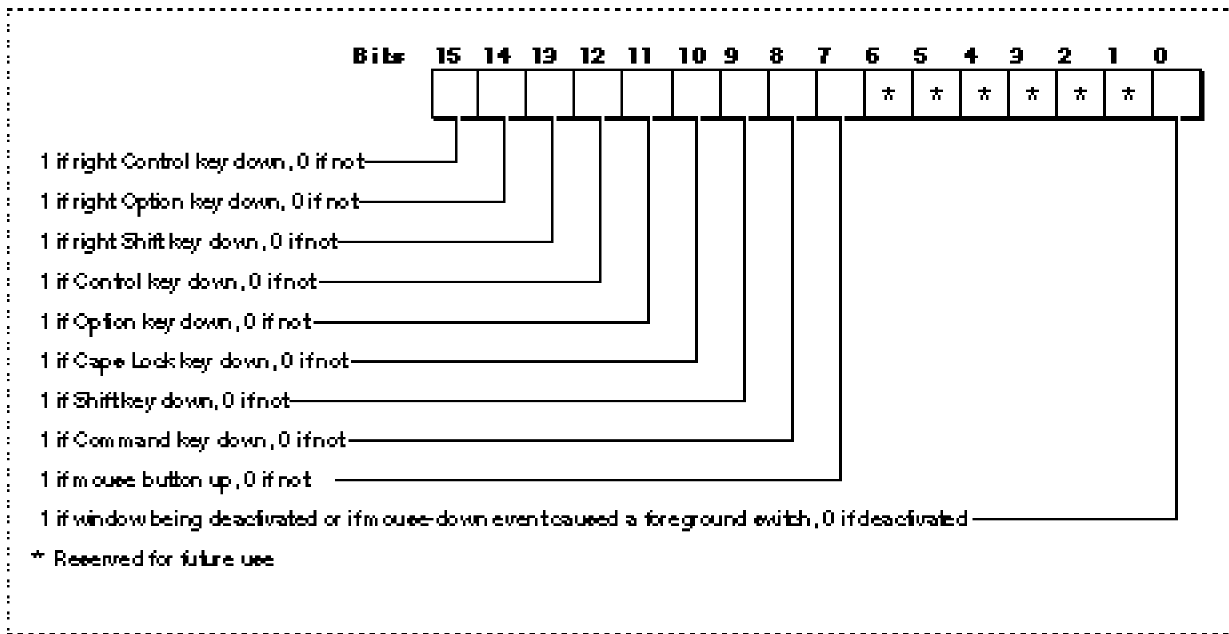
`where` For low-level events and operating-system events, the `where` field contains the location of the cursor at the time the event was posted (in global coordinates).

For high-level events, the `where` field contains a second event specifier, the event ID. The event ID defines the particular type of event within the class of events defined by the `message` field of the high-level event. For high-level events, you should interpret the `where` field as having the data type `OSType`, not `Point`.

`modifiers` The `modifiers` field contains information about the state of the modifier keys and the mouse button at the time the event was posted. For activate events, this field also indicates whether the window should be activated or deactivated. In System 7 it also indicates whether a mouse-down event caused your application to switch to the foreground.

Each of the modifier keys is represented by a specific bit in the `modifiers` field of the event record. Figure 2-5 shows how to interpret the `modifiers` field. You can examine the `modifiers` field of the event record to determine which, if any, of the modifier keys were pressed at the time of the event. The modifier keys include the Option, Command, Caps Lock, Control, and Shift keys. If your application attaches special meaning to any of these keys in combination with other keys or when the mouse button is down, you can test the state of the `modifiers` field to determine the action your application should take. For example, you can use this information to determine whether the user pressed the Command key and another key at the same time to make a menu selection.

Figure 2-5 The `modifiers` field of the event record



Bit 0 in the `modifiers` field gives additional information that is valid only if the event is an activate event or a mouse-down event.

For activate events, the value of bit 0 is 1 if the window pointed to by the event message should be activated, and the value is 0 if the window should be deactivated.

For mouse-down events in System 7, bit 0 indicates whether a mouse-down event caused your application to switch to the foreground. If so, bit 0 contains 1; otherwise, it contains 0.

You can also use these constants as masks to test the setting of various bits in the modifiers field:

```
CONST activeFlag  = 1;     {set if window being activated or if }
                           { mouse-down event caused fgnd switch}
      btnState    = 128;   {set if mouse button up}
      cmdKey      = 256;   {set if Command key down}
      shiftKey    = 512;   {set if Shift key down}
      alphaLock   = 1024;  {set if Caps Lock key down}
      optionKey   = 2048;  {set if Option key down}
      controlKey  = 4096;  {set if Control key down}
```

Note that the bit giving information about the mouse button is set if the mouse button is up. The bits for the modifier keys are set if the corresponding key is down.

Some keyboards do not distinguish between the right or left Control, Shift, and Option keys; for example, the virtual key code for the right Shift key and left Shift key might be the same. For these keyboards, if the user presses the Control, Shift, or Option key, the Event Manager sets only the bits corresponding to the shiftKey, optionKey, and controlKey constants. For keyboards that do distinguish between these keys, the Event Manager sets the bits in the modifiers field to indicate whether the right or left Control, Shift, or Option keys were pressed. For example, the Event Manager sets bit 13 in the modifiers field if the user presses the right Shift key and sets bit 9 if the user presses the left Shift key. In most cases your application should not need to distinguish between the left and right Control, Shift, and Option keys.

Processing Events

Applications receive events one at a time by asking the Event Manager for the next available event. You use Event Manager routines to receive (or in the case of EventAvail, simply to look at) the next available event that is pending for your application. You supply an event record as a parameter to the Event Manager routines that retrieve events. The Event Manager fills out the event record with the relevant information about that event and returns it to your application.

Your application can use the WaitNextEvent function to retrieve events from the Event Manager. If no events are pending for your application, the WaitNextEvent function may allocate processing time to other applications. If an event is pending for your application, the WaitNextEvent function returns the next available event of a specified type and removes the returned event from your application's event stream.

The EventAvail function gets the next available event of a specified type and returns it to your application, but does not remove the event from your application's event stream. EventAvail thus allows your application to look at an event in the event stream without actually processing the event.

Note
You can also use the `GetNextEvent` function to retrieve and remove an event; however, you should use `WaitNextEvent` to provide greater support for multitasking. ◆

Using the WaitNextEvent Function

Your application typically calls `WaitNextEvent` repeatedly. The next section, "Writing an Event Loop," shows how to use `WaitNextEvent` with other routines to process events. This discussion focuses on the `WaitNextEvent` function itself.

The `WaitNextEvent` function requires four parameters:

- an event mask (`eventMask`)
- an event record (`theEvent`)
- a sleep value (`sleep`)
- a mouse region (`mouseRgn`)

When `WaitNextEvent` returns, the event record contains information about the retrieved event, if any.

The `eventMask` parameter specifies the events you are interested in receiving. `WaitNextEvent` returns events one at a time, in order of priority and at your application's request, according to the value you specify in the `eventMask` parameter. If your application specifies that it doesn't want to receive particular types of events, those events are not returned to your application when it makes a request for an event. However, those events are not removed from the event stream. (To remove events from the Operating System event queue, you can use the `FlushEvents` procedure with a mask specifying only those events you wish to remove from the queue.) See "Setting the Event Mask" beginning on page 2-26 for examples of how to use constants to set the value of the `eventMask` parameter.

The `sleep` parameter specifies the amount of time (in ticks) for which your application agrees to relinquish the processor if no events are pending for it. When that time expires or when an event becomes available for your application, the Process Manager schedules your application for execution. In general, you should specify a value greater than 0 in the `sleep` parameter so that other applications can receive processing time if they need it. If the user is editing text and your application needs to blink the caret at periodic intervals or uses TextEdit to blink the caret, your application should not specify a value greater than the value returned by the `GetCaretTime` function.

In the `mouseRgn` parameter you specify a screen region inside of which the Event Manager does *not* generate mouse-moved events. You should specify the region in global coordinates. If the user moves the cursor outside of this region and your application is the foreground process, the Event Manager reports mouse-moved events. Your application should recalculate the `mouseRgn` parameter when it receives a mouse-moved event; otherwise it will continue to receive mouse-moved events as long as the cursor is outside of the original region. If you pass an empty region or a `NIL` region handle, the Event Manager does not return mouse-moved events. You can use the

mouseRgn parameter as a convenient way to change the shape of the cursor—for example, when the user moves the cursor from the content area of a window to the scroll bar. See "Responding to Mouse-Moved Events" beginning on page 2-62 for information on how to set and change the mouseRgn parameter.

Listing 2-1 shows an example of using the WaitNextEvent function.

Listing 2-1 Using the WaitNextEvent function

```
VAR
   eventMask:      Integer;
   event:          EventRecord;
   cursorRgn:      RgnHandle;
   mySleep:        LongInt;
   gotEvent:       Boolean;

   eventMask := everyEvent;   {accept all events}
   mySleep := MyGetSleep;     {set an appropriate sleep value}
   cursorRgn := MyGetRgn;     {set the region as appropriate}
   gotEvent := WaitNextEvent(eventMask,event,mySleep,cursorRgn);
```

The code in Listing 2-1 specifies that WaitNextEvent should return the next pending event of any kind, give up the processor if no events are pending, and return a mouse-moved event if the user moves the cursor out of the specified region.

The WaitNextEvent function returns after retrieving an event or after the time specified in the sleep parameter has expired. If there are no events of the types specified by the eventMask parameter (other than null events) pending for your application, and the time specified in the sleep parameter has not expired, WaitNextEvent may allocate processing time to background processes. Once an event for your application occurs or the time specified in the sleep parameter expires, your application receives processing time again.

WaitNextEvent returns a function result of TRUE if it has retrieved any event other than a null event. If there are no events of the types specified by the eventMask parameter (other than null events) pending for the application, WaitNextEvent returns FALSE.

Before returning an event to your application, WaitNextEvent performs other processing and may intercept the event. The WaitNextEvent function:

- Calls the Operating System Event Manager function SystemEvent to determine whether the event should be handled by your application or the Operating System. For example, if the event is a Command–Shift–number key sequence, the Event Manager intercepts the event and calls the corresponding 'FKEY' resource to perform the associated action.
- Makes the alarm go off if the alarm is set and the current time is the alarm time. The user sets the alarm using the Alarm Clock desk accessory.

- Calls the `SystemTask` procedure, which gives time to each open desk accessory or device driver to perform any periodic action defined for it. A desk accessory or device driver specifies how often the periodic action should occur, and `SystemTask` gives time to the desk accessory or device driver at the appropriate interval.

In System 7, the `WaitNextEvent` function reports a suspend event to your application when

- your application is in the foreground and the user opens a desk accessory or other item from the Apple menu,
- the user clicks in the window belonging to a desk accessory or another application, or
- the user chooses another process from the Application menu.

After your application is switched out, the Event Manager directs events (other than events your application can receive in the background) to the newly activated process until the user switches back to your application or another application.

Writing an Event Loop

In applications that are event-driven (that is, applications that decide what to do at any time by receiving and responding to events), you can obtain information about pending events by calling Event Manager routines. Since you call these routines repeatedly, the section of code in which you request events from the Event Manager usually takes the form of a loop; this section of code is called the ***event loop***.

Listing 2-2 shows a simple event loop (an application-defined procedure called `MyEventLoop`) for an application running in System 7.

Listing 2-2 An event loop

```
PROCEDURE MyEventLoop;
VAR
   cursorRgn:     RgnHandle;
   gotEvent:      Boolean;
   event:         EventRecord;
BEGIN
   cursorRgn := NewRgn; {pass an empty region the first time thru}
   REPEAT
      gotEvent := WaitNextEvent(everyEvent, event, MyGetSleep,
                                cursorRgn);
      IF (event.what <> kHighLevelEvent) AND (NOT gInBackground)
         THEN MyAdjustCursor(event.where, cursorRgn);
      IF gotEvent THEN  {the event isn't a null event, }
         DoEvent(event) { so handle it}
      ELSE              {no event (other than null) to handle }
         DoIdle(event); { right now, so do idle processing}
   UNTIL gDone;         {loop until user quits}
END;
```

The MyEventLoop procedure repeatedly uses WaitNextEvent to retrieve events. The WaitNextEvent function returns a Boolean value of FALSE if there are no events of the specified types other than null events pending for the application. WaitNextEvent returns TRUE if it has retrieved any event other than a null event.

After WaitNextEvent returns, the MyEventLoop procedure first calls an application-defined routine, MyAdjustCursor, to adjust the cursor as necessary. You usually adjust the cursor in response to mouse-moved events, and often in response to other events as well. This code adjusts the cursor once every time through the event loop, when the application receives any event other than a high-level event. The code does not adjust the cursor if the event is a high-level event, because the where field of a high-level event contains the event ID, not the location of the cursor. The code also does not adjust the cursor if this application is in the background, as the foreground process is responsible for setting the appearance of the cursor.

If WaitNextEvent retrieved any event other than a null event, the event loop calls DoEvent, an application-defined procedure, to process the event. Otherwise, the procedure calls an application-defined idling procedure, DoIdle.

Note

If your application uses modeless dialog boxes, you need to appropriately handle events in them. You can choose to handle events for modeless dialog boxes using the same routines that you use to handle events in other windows; this is the approach used throughout this chapter. Alternatively, you can choose to call the IsDialogEvent function in your event loop. See “Handling Events in a Dialog Box” on page 2-29 for information on handling events in alert boxes, modal dialog boxes, movable modal dialog boxes, and modeless dialog boxes. For additional information on dialog boxes, see the chapter “Dialog Manager” in this book. ◆

If you intend to design your application to run in either a single-application environment (such as System 6 without MultiFinder) or a multiple-application environment, the very beginning of your event loop should test to make sure the WaitNextEvent function is available. If WaitNextEvent is not available, your code should use GetNextEvent to retrieve events. If your code uses GetNextEvent, it should also call SystemTask to allow desk accessories to perform periodic actions. However, your code should always use WaitNextEvent if it is available, rather than GetNextEvent. If your application calls WaitNextEvent, it should not call the SystemTask procedure.

The event loop shown in Listing 2-2 calls an application-defined procedure, DoEvent, to determine what kind of event the call to WaitNextEvent retrieved. Listing 2-3 defines a simple DoEvent procedure. The DoEvent procedure examines the value of the what field of the event record to determine the type of event received and then calls an appropriate application-defined routine to further process the event.

Listing 2-3 Processing events

```
PROCEDURE DoEvent (event: EventRecord);
VAR
   window:     WindowPtr;
   activate:   Boolean;
BEGIN
   CASE event.what OF
      mouseDown:
         DoMouseDown(event);
      mouseUp:
         DoMouseUp(event);
      keyDown, autoKey:
         DoKeyDown(event);
      activateEvt:
         BEGIN
            window := WindowPtr(event.message);
            activate := BAnd(event.modifiers, activeFlag) <> 0;
            DoActivate(window, activate, event);
         END;
      updateEvt:
         DoUpdate(WindowPtr(event.message));
      diskEvt:
         DoDiskEvent(event);
      osEvt:
         DoOSEvent(event);
      kHighLevelEvent:
         DoHighLevelEvent(event);
   END; {of case}
END;
```

The next sections describe how to set the event mask, handle events in dialog boxes, and create your application's `'SIZE'` resource. Following sections show code that can handle each kind of event.

Setting the Event Mask

Several of the Event Manager routines can be restricted to operate on a specific event type or group of types. You do this by specifying the event types you want your application to receive, thereby disabling (or "masking out") the events you are not interested in receiving. To specify which event types an Event Manager routine governs, you supply a parameter known as an event mask.

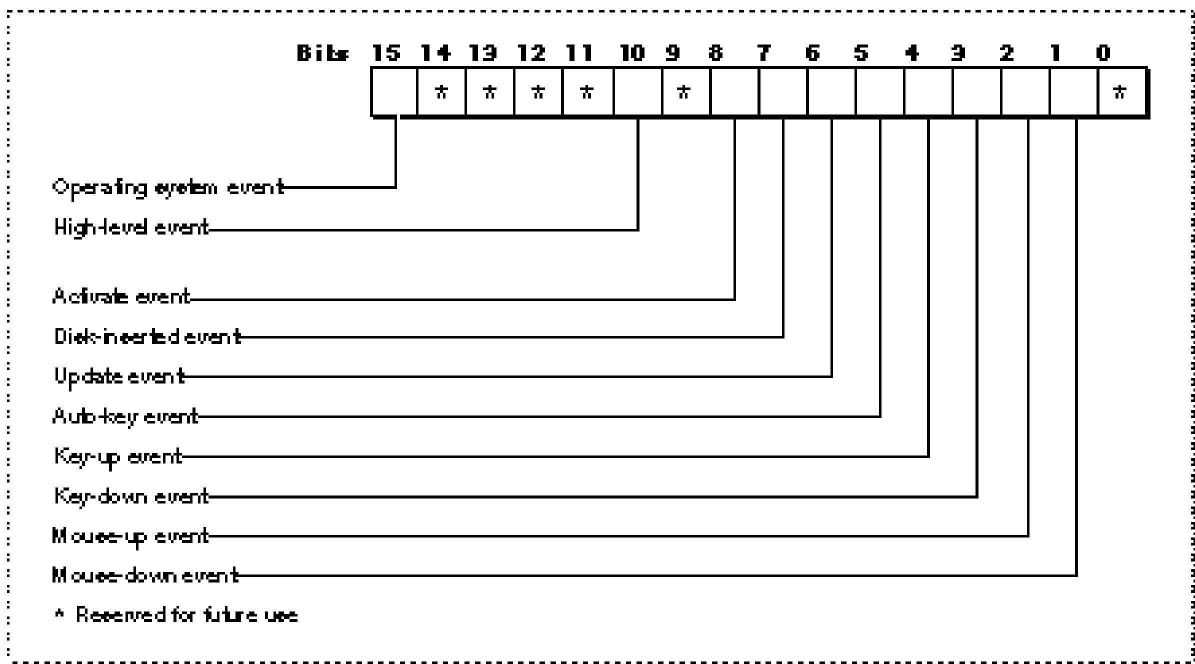
The **event mask** is an integer with one bit position for each event type. If the bit representing a particular event type is set, then the Event Manager returns events of

that type. If the bit is set to 0, the Event Manager does not return events of that type. To accept all types of events, set every bit of the event mask to 1. You can do this using the constant `everyEvent`.

```
CONST everyEvent           =  -1;       {every event}
```

Figure 2-6 shows the bits corresponding to each event type in the event mask.

Figure 2-6 The event mask



You can use these constants when referring to the bits in the event mask that correspond to each individual event type:

```
CONST mDownMask          =  2;       {mouse-down event       (bit 1)}
      mUpMask            =  4;       {mouse-up event         (bit 2)}
      keyDownMask        =  8;       {key-down event         (bit 3)}
      keyUpMask          =  16;      {key-up event           (bit 4)}
      autoKeyMask        =  32;      {auto-key event         (bit 5)}
      updateMask         =  64;      {update event           (bit 6)}
      diskMask           =  128;     {disk-inserted event    (bit 7)}
      activMask          =  256;     {activate event         (bit 8)}
      highLevelEventMask =  1024;    {high-level event       (bit 10)}
      osMask             =  -32768;  {operating-system event (bit 15)}
```

You can select any subset of events by adding or subtracting these constants. For example, you can use this code to accept only high-level events and mouse-down events and mask out all other events:

```
myErr := WaitNextEvent(highLevelEventMask + mDownMask, myEvent,
                         mySleep, myMRgnHnd);
```

The `everyEvent` constant indicates that you wish to receive every type of event. To accept all events except mouse-up events, you can use the code:

```
myErr := WaitNextEvent(everyEvent - mUpMask, myEvent, mySleep,
                         myMRgnHnd);
```

Masking out specific types of events does not remove those events from the event stream. If a type of event is masked out, the Event Manager simply ignores it when reporting events from the event stream. Note that you cannot mask out null events by setting the event mask. The Event Manager always returns a null event if no other events are pending. However, if you do not want the Event Manager to report null events to your application when it is in the background, you can set the `cannotBackground` flag in your application's `'SIZE'` resource.

In most cases you should always use `everyEvent` as your event mask. The user expects most applications to respond to keyboard, mouse, update, and other events.

The types of events returned to your application are also affected by the system event mask. The Event Manager maintains a system event mask for each application. The system event mask controls which low-level event types get posted in the Operating System event queue. The Event Manager uses the system event mask of the current process (the process that is currently executing and the process associated with the `CurrentA5` global variable) when determining which low-level events to post in the Operating System event queue. The system event mask is an integer with 1 bit for each corresponding low-level event type. These constants refer to the bits that represent the corresponding low-level event types in the system event mask:

```
CONST mDownMask           =  2;       {mouse-down    (bit 1)}
      mUpMask             =  4;       {mouse-up      (bit 2)}
      keyDownMask         =  8;       {key-down      (bit 3)}
      keyUpMask           =  16;      {key-up        (bit 4)}
      autoKeyMask         =  32;      {auto-key      (bit 5)}
      diskMask            =  128;     {disk-inserted (bit 7)}
```

When a low-level event (other than an update or activate event) occurs, the Operating System Event Manager posts the low-level event in the Operating System event queue only if the bit corresponding to the low-level event type is set in the system event mask of the current process. When your application starts, the Operating System initializes the system event mask of your application to post mouse-up, mouse-down, key-down, auto-key, and disk-inserted events in the Operating System event queue. Thus, the system event mask has this initial setting:

```
systemEventMask := everyEvent - keyUpMask;
```

Your application should not change the system event mask except to enable key-up events if your application needs to respond to key-up events. (Most applications ignore key-up events.) If your application needs to receive key-up events, you can change the system event mask using the Operating System Event Manager procedure `SetEventMask`. Note that your application cannot rely on receiving key-up events when it is not the current process. For example, if your application is the foreground (and current) process and a minor switch occurs, the Event Manager uses the system event mask of the background process (now the current process) when posting low-level event types. When your application becomes the current process again, the Event Manager uses the system event mask of your application when posting low-level events.

Handling Events in a Dialog Box

If your application uses alert boxes, modal dialog boxes, movable modal dialog boxes, or modeless dialog boxes, you need to make sure your application handles events for them appropriately.

To display and handle events in alert boxes, you use the Dialog Manager functions `Alert`, `NoteAlert`, `CautionAlert`, and `StopAlert`. The Dialog Manager handles all of the events generated by the user until the user clicks a button (typically the OK or Cancel button). When the user clicks the OK or Cancel button, the alert box functions highlight the button that was clicked, close the alert box, and report the user's selection to your application. Your application is responsible for performing the appropriate action associated with that button.

For modal dialog boxes, you can use the Dialog Manager procedure `ModalDialog`. The Dialog Manager handles most of the user interaction until the user selects an item. The `ModalDialog` procedure then reports that the user selected an enabled item, and your application is responsible for performing the action associated with that item. Your application typically calls `ModalDialog` repeatedly, responding to clicks on enabled items as reported by `ModalDialog`, until the user selects OK or Cancel.

For alert boxes and modal dialog boxes, you should also supply an event filter function as one of the parameters to the alert box functions or `ModalDialog` procedure. As the user interacts with the alert or modal dialog box, these functions pass events to your event filter function before handling each event. Your event filter function can handle any events not handled by the Dialog Manager or, if necessary, can choose to handle events normally handled by the Dialog Manager. For more information on filter functions for alert and dialog boxes, see the chapter "Dialog Manager" in this book.

To handle events in movable modal dialog boxes, you can use the Dialog Manager functions `IsDialogEvent` and `DialogSelect` or you can use other Toolbox routines to handle events without using the Dialog Manager.

For modeless dialog boxes, you can choose to handle events in them using an approach similar to the one you use to handle events in other windows; that is, when you receive an event, you first determine the type of event that occurred and then take the appropriate action based on the type of window that is in front. If a modeless dialog box is in front, you can provide code that takes any actions specific to that modeless dialog box and call the `DialogSelect` function to handle any events that your code doesn't

handle. This is the approach used throughout this chapter. Alternatively, you can choose to call the `IsDialogEvent` function in your event loop. If you do this, you can use the `IsDialogEvent` function to determine whether the event involves a modeless dialog box that belongs to your application. If the event involves a modeless dialog box (including null events) and a modeless dialog box is active, `IsDialogEvent` returns `TRUE`. Otherwise, `IsDialogEvent` returns `FALSE`.

If `IsDialogEvent` returns `TRUE`, your application can check to see what type of event occurred and, depending on the type of event, it can choose to handle the event itself.

Regardless of the approach you use, if your application chooses not to handle the event, it should call `DialogSelect`. The `DialogSelect` function handles events for modeless dialog boxes (including null events). It also blinks the caret in editable text items when necessary.

For more information on the `DialogSelect` function and events in dialog boxes, see the chapter "Dialog Manager" in this book.

Creating a Size Resource

Your application should include a size (`'SIZE'`) resource. You use a `'SIZE'` resource to inform the Operating System about the memory size requirements for your application so that the Operating System can set up a partition of the appropriate size for your application. You also use the `'SIZE'` resource to indicate certain scheduling options to the Operating System, such as whether your application can accept suspend and resume events.

You can also specify additional information in the `'SIZE'` resource in System 7, indicating whether your application is 32-bit clean, whether your application supports stationery documents, whether your application uses TextEdit's inline input services, whether your application wishes to receive notification of the termination of any applications it has launched, and whether your application wishes to receive high-level events.

A `'SIZE'` resource consists of a 16-bit flags field, followed by two 32-bit size fields. The flags field specifies operating characteristics of your application, and the size fields indicate the minimum and preferred partition sizes for your application. The **minimum partition size** is the actual limit below which your application will not run. The **preferred partition size** is the memory size at which your application can run most effectively and that the Operating System attempts to secure upon launch of your application. If that amount of memory is unavailable, your application is placed into the largest contiguous block available, provided that it is larger than the specified minimum size.

Note

If the amount of available memory is between the minimum and the preferred sizes, the Finder displays a dialog box asking if the user wants to run the application using the amount of memory available. If your application does not have a `'SIZE'` resource, it is assigned a default partition size of 512 KB and the Process Manager uses a default value of `FALSE` for all specifications normally defined by constants in the flags field. ◆

When you define a 'SIZE' resource, you should give it a resource ID of –1. A user can modify the preferred size in the Finder's information window for your application. If the user does alter the partition size, the Operating System creates a new 'SIZE' resource having a resource ID of 0. At application launch time, the Process Manager looks for a 'SIZE' resource with ID 0; if this resource is not found, it uses your original 'SIZE' resource with ID –1. This new 'SIZE' resource is also created when the user modifies any of the other settings in the resource.

When creating a 'SIZE' resource, you first need to determine the various operating characteristics of your application. For example, if your application has nothing useful to do when it is in the background, then you should not set the canBackground flag. Similarly, if you have not tested your application in an environment that uses all 32 bits of a handle or pointer for memory addresses, then you should not set the is32BitCompatible flag.

Listing 2-4 shows the Rez input for a sample 'SIZE' resource. (Rez is a resource compiler available with the MPW environment.)

Listing 2-4 The Rez input for a sample 'SIZE' resource

```
resource 'SIZE' (-1) {
   reserved,                      /*reserved*/
   acceptSuspendResumeEvents,     /*accepts suspend&resume events*/
   reserved,                      /*reserved*/
   canBackground,                 /*can use background null */
                                  /* events*/
   doesActivateOnFGSwitch,        /*activates own windows in */
                                  /* response to OS events*/
   backgroundAndForeground,       /*application has a user */
                                  /* interface*/
   dontGetFrontClicks,            /*don't return mouse events */
                                  /* in front window on resume*/
   ignoreAppDiedEvents,           /*doesn't want app-died events*/
   is32BitCompatible,             /*works with 24- or 32-bit addr*/
   isHighLevelEventAware,         /*supports high-level events*/
   localAndRemoteHLEvents,        /*also remote high-level events*/
   isStationeryAware,             /*can use stationery documents*/
   dontUseTextEditServices,       /*can't use inline input */
                                  /* services*/
   reserved,                      /*reserved*/
   reserved,                      /*reserved*/
   reserved,                      /*reserved*/
   kPrefSize * 1024,              /*preferred memory size*/
   kMinSize * 1024                /*minimum memory size*/
};
```

The `'SIZE'` resource specification in Listing 2-4 indicates, among other things, that the application accepts suspend and resume events, does processing in the background using null events, activates or deactivates any windows as necessary in response to operating-system events, can execute in both the foreground and background, and doesn't want to receive any mouse event associated with a resume event that was caused by the user clicking in the application's front window. It also indicates that the application doesn't want to receive Application Died events, can work in 24-bit or 32-bit addressing mode, does accept high-level events, including both local and network high-level events, does handle stationery documents, and doesn't use TextEdit's inline input services. In this example, the Rez-input file must define values for the constants `kPrefSize` and `kMinSize`; for example, if `kPrefSize` is set to 50, the preferred partition size is 50 KB.

The numbers you specify as your application's preferred and minimum memory sizes depend on the particular memory requirements of your application. Your application's memory requirements depend on the size of your application's static heap, dynamic heap, A5 world, and stack. (See "Introduction to Memory Management" in *Inside Macintosh: Memory* for complete details about these areas of your application's partition.)

The static heap size includes objects that are always present during the execution of your application—for example, code segments, Toolbox data structures for window records, and so on.

Dynamic heap requirements come from various objects created on a per-document basis (which may vary in size proportionally with the document itself) and objects that are required for specific commands or functions.

The size of the A5 world depends on the amount of global data and the number of intersegment jumps your application contains.

The stack contains variables, return addresses, and temporary information. The size of the application stack varies among computers, so you should base your values for the stack size according to the stack size required on a Macintosh Plus computer (8 KB). The Process Manager automatically adjusts your requested amount of memory to compensate for the different stack sizes on different machines. For example, if you request 512 KB, more stack space (approximately 16 KB) will be allocated on machines with larger default stack sizes.

Unfortunately, it is difficult to forecast all of these conditions with any great degree of reliability. You should be able to determine reasonably accurate estimates for the stack size, static heap size, A5 world, and jump table. In addition, you can use tools such as MacsBug's heap-exploring commands to help you empirically determine your application's dynamic memory requirements.

See "The Size Resource" beginning on page 2-115 for additional information on the meaning of each of the fields and flags of a `'SIZE'` resource.

Handling Low-Level Events

Low-level events include hardware-related occurrences stored in the Operating System event queue and activate and update events generated by the Window Manager. When your application receives a low-level event, your application needs to determine the type

of event and respond appropriately. The following sections discuss how to respond to mouse events, keyboard events (including certain specific keyboard events, such as when the user presses the Command key and period key at the same time), update events, activate events, disk-inserted events, and null events.

Responding to Mouse Events

Whenever the user presses or releases the mouse button, the Operating System Event Manager records the action in the Operating System event queue. These actions are stored in the event queue as mouse-down and mouse-up events. Your application can retrieve these events using the `WaitNextEvent` function.

Events related to movements of the mouse are not stored in the event queue. The mouse driver automatically tracks the mouse and displays the cursor as the user moves the mouse. Therefore, the Operating System Event Manager does not report an event if the user simply moves the mouse.

However, you can request that the Event Manager report mouse-moved events if the user moves the cursor out of a region that you specify to the `WaitNextEvent` function. For example, your application can use mouse-moved events in this way to change the shape of the cursor from an I-beam to an arrow when the user moves the cursor from a text area to the scroll bar of a window.

The rest of this section describes how your application responds to mouse-down or mouse-up events. See "Responding to Mouse-Moved Events" beginning on page 2-62 for specific details on mouse-moved events.

The user expects that pressing the mouse button correlates to particular actions in an application. Your application is responsible for providing feedback or performing any actions in response to the user. For example, if the user presses the mouse button while the cursor is in the menu bar, your application should use the Menu Manager function `MenuSelect` to allow the user to choose a menu command.

Your application can receive and respond to mouse-down and mouse-up events. Most applications respond to mouse-down events and use the routines of various managers (such as `MenuSelect`, `DragWindow`, `TEClick`, `TrackBox`, `TrackGoAway`, and `TrackControl`) to handle the corresponding mouse-up events. You can also provide code to respond to mouse-up events if it's appropriate for your application. For example, if your application implements its own text-editing capabilities, you might let the user select lines of text by dragging the mouse and use mouse-up events to signal the end of the selection.

In System 7, your application receives mouse-down events only when it is the foreground process and the user clicks in the menu bar, in a window belonging to your application, or in a window belonging to a desk accessory that was launched in your application's partition. If the user clicks in a window belonging to another application, the Event Manager sends your application a suspend event and performs a major switch to the other application.

When your application receives a mouse-down event, you need to first determine the location of the cursor at the time the mouse button was pressed (the **mouse location**) and respond appropriately. You can use the Window Manager function `FindWindow` to

find which of your application's windows, if any, the mouse button was pressed in and, if applicable, to find which part of the window it was pressed in. The `FindWindow` function also reports whether the given mouse location is in the menu bar or, in some cases, in a window belonging to a desk accessory (if the desk accessory was launched in your application's partition).

The `what` field of the event record for a mouse event contains the `mouseDown` or `mouseUp` constant to report that the mouse button was pressed or released. The `message` field is undefined. The `when` field contains the number of ticks since the system last started up. You can use the `when` field to compare how much time has elapsed between successive mouse events; for example, you might use this information to help detect mouse double clicks.

The `where` field of the event record contains the location of the cursor at the time the mouse button was pressed or released. You can pass this location to the `FindWindow` function; the `FindWindow` function maps the given mouse location to particular areas of the screen.

The `modifiers` field contains information about the state of the modifier keys at the time the mouse button was pressed or released. Your application can perform different actions based on the state of the modifier keys. For example, your application might let the user extend a selection or select multiple objects at a time if the Shift key was down at the time of the mouse-down event.

Listing 2-5 shows code that handles mouse-down events. The `DoMouseDown` procedure is an application-defined procedure that is called from the `DoEvent` procedure. (Listing 2-3 on page 2-26 shows the `DoEvent` procedure.)

Listing 2-5 Handling mouse-down events

```
PROCEDURE DoMouseDown (event: EventRecord);
VAR
   part:             Integer;
   thisWindow:       WindowPtr;
BEGIN
   {map location of the cursor (at the time of mouse-down event) }
   { to general areas of the screen}
   part := FindWindow(event.where, thisWindow);

   CASE part OF {take action based on the mouse location}
   inMenuBar: {mouse down in menu bar, respond appropriately}
      BEGIN
         {first adjust marks and enabled state of menu items}
         MyAdjustMenus;
         {let user choose a menu command}
         DoMenuCommand(MenuSelect(event.where));
      END;
  inSysWindow: {cursor in a window belonging to a desk accessory}
      SystemClick(event, thisWindow);
```

```
  inContent: {mouse down occurred in the content area of }
             { one of your application's windows}
   IF thisWindow <> FrontWindow THEN
   BEGIN {mouse down occurred in a window other than the front }
         { window--make the window clicked in the front window, }
         { unless the front window is movable modal}
      IF MyIsMovableModal(FrontWindow) THEN
         SysBeep(30)
      ELSE
         SelectWindow(thisWindow);
   END
   ELSE  {mouse down was in the content area of front window}
      DoContentClick(thisWindow, event);

  inDrag:               {handle mouse down in drag area}
      IF (thisWindow <> FrontWindow) AND
         (MyIsMovableModal(FrontWindow))
      THEN
         SysBeep(30)
      ELSE
        DragWindow(thisWindow, event.where, GetGrayRgn^^.rgnBBox);

   inGrow:              {handle mouse down in grow region}
      DoGrowWindow(thisWindow, event);
   inGoAway:            {handle mouse down in go-away region}
      IF TrackGoAway(thisWindow, event.where) THEN
         DoCloseCmd;
   inZoomIn, inZoomOut: {handle mouse down in zoom box region}
      IF TrackBox(thisWindow, event.where, part) THEN
         DoZoomWindow(thisWindow, part);
   END; {end of CASE}
END;{of DoMouseDown}
```

When your application retrieves a mouse-down event, call the Window Manager function `FindWindow` to map the location of the cursor to particular areas of the screen. Given a mouse location, the `FindWindow` function returns as its function result a value that indicates whether the mouse location is in the menu bar, in one of your application's windows, or, in some cases, in a desk accessory window. If the mouse location is in an application window, the function result indicates which part of the window the mouse location is in. You can test the function result of `FindWindow` against these constants to determine the mouse location at the time of the mouse-down event:

```
CONST inDesk      = 0;{none of the following}
      inMenuBar   = 1;{in the menu bar}
      inSysWindow = 2;{in a desk accessory window}
```

```
inContent   = 3;{anywhere in content region except the }
                { grow region if the window is active, }
                { anywhere in content region including the }
                { grow region if the window is inactive}
inDrag      = 4;{in drag (title bar) region}
inGrow      = 5;{in grow region (active window only)}
inGoAway    = 6;{in go-away region (active window only)}
inZoomIn    = 7;{in zoom-in region (active window only)}
inZoomOut   = 8;{in zoom-out region (active window only)}
```

The `FindWindow` function reports the `inDesk` constant if the mouse location is not in the menu bar, desk accessory window, or any window of your application. For example, the `FindWindow` function may report this constant if the location of the cursor is inside a window frame but not in the drag region or go-away region of the window; your application seldom receives the `inDesk` constant.

If `FindWindow` returns the `inMenuBar` constant, the mouse location is in the menu bar. In this case your application should first adjust its menus. The application-defined `MyAdjustMenus` procedure adjusts its menus—enabling and disabling items and setting marks—based on the context of the active window. For example, if the active window is a document window that contains a selection, your application should enable the Cut and Copy commands in the Edit menu, add marks to the appropriate items in the Font, Size, and Style menus, and adjust any other menu items accordingly. After adjusting your application's menus, call the Menu Manager function `MenuSelect`, passing it the location of the mouse, to allow the user to choose a menu command. The `MenuSelect` function handles all user interaction until the user releases the mouse button. The `MenuSelect` function returns as its function result a long integer indicating the menu selection made by the user. As shown in Listing 2-5 on page 2-34, the `DoMouseDown` routine calls an application-defined routine, `DoMenuCommand`, to perform the menu command selected by the user. See the chapter "Menu Manager" in this book for a listing that gives the code for the `MyAdjustMenus` and `DoMenuCommand` routines and for more information about responding to specific menu commands.

In System 7, the `FindWindow` function seldom returns the `inSysWindow` constant. The `FindWindow` function returns this constant only when a mouse-down event occurred in a desk accessory that was launched in the application's partition. Normally, if the user clicks in a desk accessory's window, the Event Manager sends your application a suspend event and brings the desk accessory to the foreground. From that point on, mouse-down events and other events are handled by the desk accessory until the user again clicks in one of your application's windows.

If `FindWindow` does return the `inSysWindow` constant, the mouse location is in a window belonging to a desk accessory that was launched in your application's partition. In this case, your application should call the `SystemClick` procedure. The `SystemClick` procedure routes the event to the desk accessory as appropriate. If the mouse button was pressed while the cursor was in the content region of the desk accessory's window and the window is inactive, `SystemClick` makes it the active window. It does this by sending your application an activate event to deactivate its front window and directing an event to the desk accessory to activate its window.

FindWindow can return any of the constants inContent, inDrag, inGrow, inGoAway, inZoomIn, or inZoomOut if the given mouse location is in your application's active window. If the cursor is in the content area, your application should perform any actions appropriate to your application. Note that scroll bars are part of the content region. In most cases, if the cursor is in the content area, your application first needs to determine whether the mouse location is in the scroll bar or any other controls and then respond appropriately. The DoMouseDown procedure calls the application-defined procedure DoContentClick to handle mouse-down events in the content area of the active window. If your application needs to determine whether the mouse-down event caused a foreground switch (and you set the getFrontClicks flag in your application's 'SIZE' resource), your DoContentClick procedure can test bit 0 in the modifiers field of the event record (normally your application does not test for this condition). See the chapter "Control Manager" in this book for an example DoContentClick procedure and for detailed information on implementing controls in your application's windows.

If the mouse location is in any of the other specified regions of an active application window, your application should perform the action corresponding to that region. For example, if the cursor is in the drag region, your application should call the Window Manager procedure DragWindow to allow the user to drag the window to a new location.

If the mouse location is in an inactive application window, FindWindow can return the inContent or inDrag constant, but does not distinguish between any other areas of the window. In this case, if FindWindow reports the inContent constant, your application should bring the inactive window to the front using the SelectWindow procedure (unless the active window is a movable modal dialog box). If the active window is a movable modal dialog box, then your application should use the SysBeep procedure to play the system alert sound rather than activating the selected window. Also, if your application interprets the first mouse click in an inactive window as a request to activate the window *and* perform an action, you can process the event again. However, note that most users expect the first click in an inactive window to activate the window without performing any additional action. If FindWindow reports inDrag for an inactive application window, your application should call the DragWindow procedure to allow the user to drag the window to a new location (unless the active window is a movable modal dialog box, in which case your application should simply play the system alert sound).

If you're using TextEdit to handle text editing and call TEClick, TEClick automatically interprets mouse double clicks appropriately, including allowing the user to select a word by double-clicking it. Your application must provide the means to allow double-clicking in this manner in all other contexts.

You can detect mouse double clicks by comparing the time and location of a mouse-up event with that of the immediately following mouse-down event. The GetDblTime function returns the recommended difference in ticks that should exist between the occurrence of a mouse-up and mouse-down event for those two mouse events to be considered a double click.

You should interpret mouse events as a double click if both of these conditions are true:

- The times of the mouse-up event and mouse-down event differ by a number of ticks less than or equal to the value returned by the `GetDblTime` function.
- The locations of the two mouse-down events separated by the mouse-up event are sufficiently close to each other. How you determine this value depends on your application and the context in which the mouse-down events occurred. For example, in a word-processing application, you might consider two mouse-down events a double click if the mouse locations both mapped to the same character, whereas in a graphics application you might consider it a double click if the sum of the horizontal and vertical difference between the two mouse locations is no more than five pixels.

The Event Manager also provides other routines that give information about the mouse. You can find the current mouse location using the `GetMouse` procedure. You can determine the current state of the mouse button using the `Button`, `StillDown`, and `WaitMouseUp` functions. See "Reading the Mouse" beginning on page 2-108 for detailed information on these routines.

Responding to Keyboard Events

Your application can receive keyboard events to notify you when the user has pressed or released a key or continued to hold down a key. When the user presses a key, the Operating System Event Manager stores a key-down event in the Operating System event queue. Your application can retrieve the event from the queue; determine which key was pressed; determine which modifier keys, if any, were pressed at the time of the event; and respond appropriately. Typically, your application provides feedback by echoing (displaying) the glyph representing the character generated by the pressed key on the screen.

When the user holds down a key for a certain amount of time, the Event Manager generates auto-key events. The Event Manager generates an auto-key event after a certain initial delay (the auto-key threshold) has elapsed since the original key-down event. The Event Manager generates subsequent auto-key events whenever a certain repeat interval (the auto-key rate) has elapsed since the last auto-key event and while the original key is still held down. The user can set the initial delay and rate of repetition using the Keyboard control panel. The default value for the auto-key threshold is 16 ticks, and the default value for the auto-key rate is 4 ticks. Current values of the auto-key threshold and auto-key rate are stored in the system global variables `KeyThresh` and `KeyRepThresh`.

In addition to getting keyboard events when the user presses or releases a key, you can directly read the keyboard (and keypad) using the `GetKeys` procedure.

When the user presses a key or a combination of keys, your application should respond appropriately. Your application should follow the guidelines in *Macintosh Human Interface Guidelines* for consistent use of and response to keyboard events. For example, your application should allow the user to choose a frequently used menu command by using a keyboard equivalent for that menu command—usually a combination of the Command key and another key. Your application should also respond to the user pressing the arrow keys, Shift key, or other keys according to the guidelines provided in *Macintosh Human Interface Guidelines*.

Also note that certain keyboards have different physical layouts or contain additional keys, such as function keys. If your application supports function keys or other special keys, you should follow the guidelines in *Macintosh Human Interface Guidelines* when determining what action to take when the user presses one of these keys.

Certain keystroke combinations are handled by the Event Manager and not returned to your application. If the user holds down the Command and Shift keys while pressing a numeric key to produce a special effect, that special effect occurs. Apple provides three standard Command–Shift–number key sequences. The standard Command–Shift–number key sequences are 1 for ejecting the disk in the internal drive, 2 for ejecting the disk in a second internal drive or for ejecting the disk in an external drive if the computer has only one internal drive, and 3 for taking a snapshot of the screen and storing it as a TeachText document on the startup volume.

The action corresponding to a Command–Shift–number key sequence is implemented as a routine that takes no parameters and is stored in an 'FKEY' resource with a resource ID that corresponds to the number that activates it. Apple reserves 'FKEY' resources with resource IDs 1 through 4 for its own use; if you provide an 'FKEY' resource, use a resource ID between 5 and 9.

You can disable the Event Manager's processing of Command–Shift–number key sequences for numbers 3 through 9 by setting the system global variable ScrDmpEnb (a byte) to 0. However, in most cases you should not disable the Event Manager's processing of these events.

The what field of the event record for a keyboard-related event contains either the keyDown or keyUp constant to indicate that the key was pressed or released, or the autokey constant to indicate that the key is being held down.

The Event Manager sets the system event mask of your application to accept all events except key-up events. Most applications ignore key-up events. If your application needs to receive key-up events, you can change the system event mask of your application using the Operating System Event Manager procedure SetEventMask.

In the low-order word the message field contains the character code and virtual key code that corresponds to the key pressed by the user.

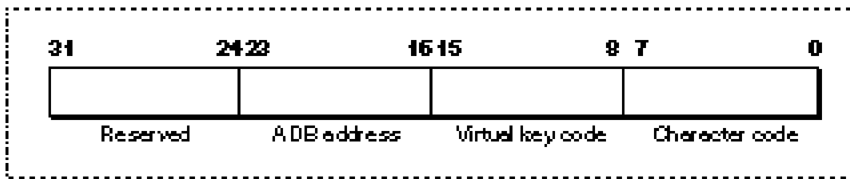
The **virtual key code** represents the key pressed or released by the user; this value is always the same for a specific physical key on a particular keyboard. For example, on the Apple Keyboard II, ISO layout, the virtual key code for the fifth key to the right of the Tab key (the key labeled "T") is always $11, regardless of which modifier keys are also pressed.

To determine the virtual key code that corresponds to a specific physical key, system software uses a hardware-specific key-map ('KMAP') resource that specifies the virtual key codes for a particular keyboard. After determining the virtual key code of the key pressed by the user, system software uses a script-specific keyboard-layout ('KCHR') resource to map a virtual key code to a specific character code. Any given script system has one or more 'KCHR' resources. For example, a particular computer might contain the French 'KCHR' resource in addition to the standard U.S. 'KCHR' resource. In this situation, the current 'KCHR' resource determines whether virtual key codes are mapped to the French or U.S. character set.

The **character code** represents a particular character. The character code that is generated depends on the virtual key code, the state of the modifier keys, and the current 'KCHR' resource. For example, the U.S. 'KCHR' resource specifies that for the virtual key code $2D (the fifth key to the left of the Shift key and labeled "N" on an Apple Keyboard II, Domestic layout), the character code is $6E when no modifier keys are pressed; the character code is $4E when this key is pressed in combination with the Shift key. Character codes for the Roman script system are specified in the extended version of ASCII (the American Standard Code for Information Interchange).

The `message` field contains additional information for ADB keyboards. The low-order byte of the high-order word contains the ADB address of the keyboard where the keyboard event occurred. Figure 2-7 shows the structure of the `message` field of the event record for keyboard events.

Figure 2-7 The `message` field of the event record for keyboard events



Usually your application uses the character code, rather than the virtual key code, when responding to keyboard events. You can use these two constants to access the virtual key code and character code in the `message` field:

```
CONST charCodeMask      = $000000FF;{mask for character code}
      keyCodeMask       = $0000FF00;{mask for virtual key code}
```

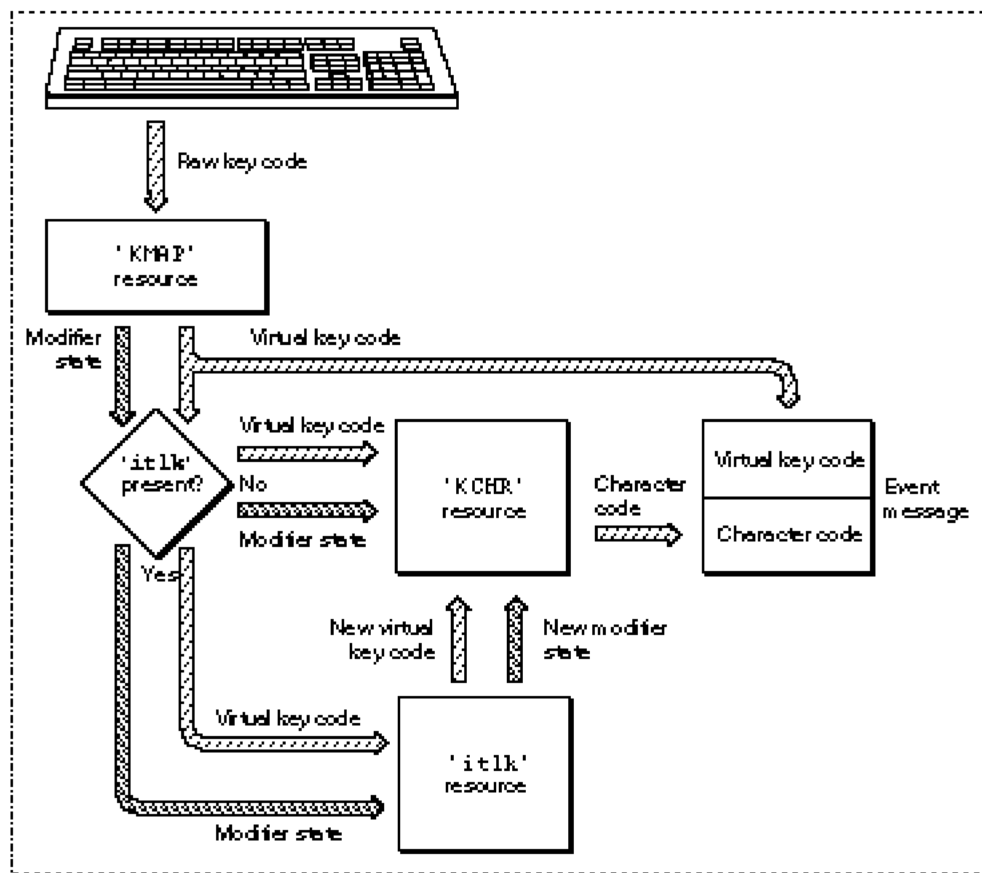
The `when` field contains the number of ticks since the system last started up. You can use the `when` field to compare how much time has expired between successive keyboard events.

The `where` field of the event record contains the location of the cursor at the time the key was pressed or released. You typically disregard the mouse location when processing keyboard events.

The `modifiers` field contains information about the state of the modifier keys at the time the key was pressed or released. Your application can perform different actions based on the state of the modifier keys. For example, your application might perform an action associated with a corresponding menu command if the Command key was down at the time of the key-down event.

System software can support a number of different types of keyboards, for example, the Apple Keyboard II, the Apple Extended keyboards, or other keyboards. The system software uses various keyboard resources and international resources to manage different types of keyboards. Figure 2-8 illustrates how system software maps keys to character codes.

Figure 2-8 Keyboard translation



When a user presses or releases a key on the keyboard, the keyboard generates a raw key code. The system software uses a 'KMAP' resource to map the raw key code to a hardware-independent virtual key code and to set bits indicating the state of the modifier keys. A 'KMAP' resource specifies the physical arrangement of a particular keyboard and indicates the virtual key codes that correspond to each physical key.

If the optional key-remap ('itlk') resource is present, the system software remaps the virtual key codes and modifier state for some key combinations on certain keyboards before using the 'KCHR' resource. The 'itlk' resource can reintroduce hardware dependence because certain scripts, languages, and regions need subtle differences in layout for specific keyboards. If present, the 'itlk' resource affects only a few keys.

After mapping the virtual key code and the state of the modifier keys through an optional 'itlk' resource, the system software uses a 'KCHR' resource to produce the character code representing the key that was pressed or released. The 'KCHR' resource specifies how to map the setting of the modifier keys and a virtual key code to a character code.

After mapping the key, the Event Manager returns the virtual key code and the character code in the `message` field of the event record.

Figure 2-9 shows the virtual key codes as specified by the `'KMAP'` resource for the Apple Keyboard II, ISO layout. The labels for the keys on the keyboard are shown using the U.S. keyboard layout. The virtual key codes are shown in hexadecimal.

Figure 2-9 Virtual key codes for the Apple Keyboard II, ISO layout

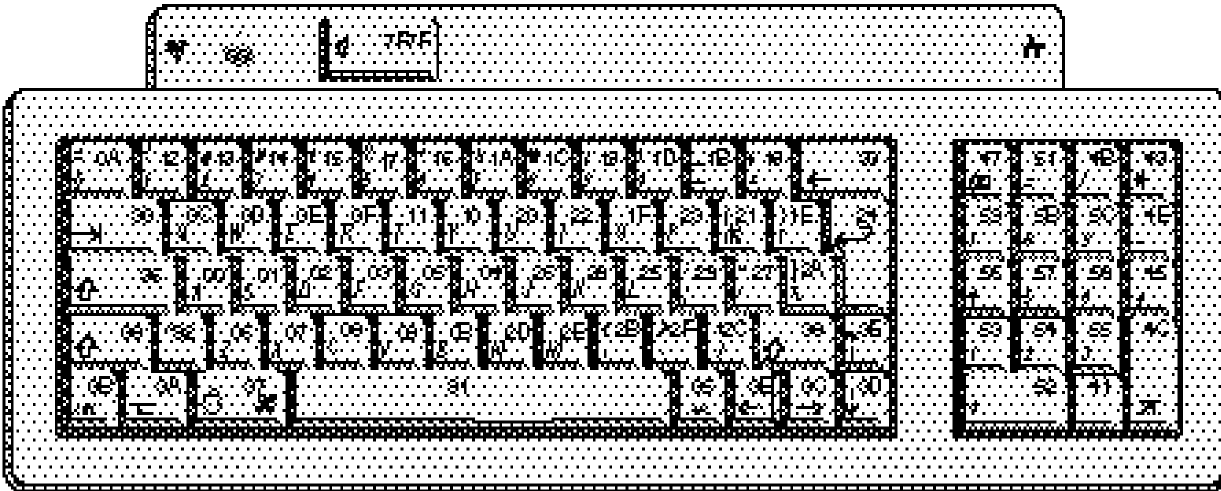


Figure 2-10 shows the virtual key codes as specified by the `'KMAP'` resource for the Apple Extended Keyboard II, one that uses the Domestic (ANSI) layout, and one that uses the ISO layout. The labels for the keys on the ISO keyboard are shown using the French keyboard layout. The virtual key codes are shown in hexadecimal.

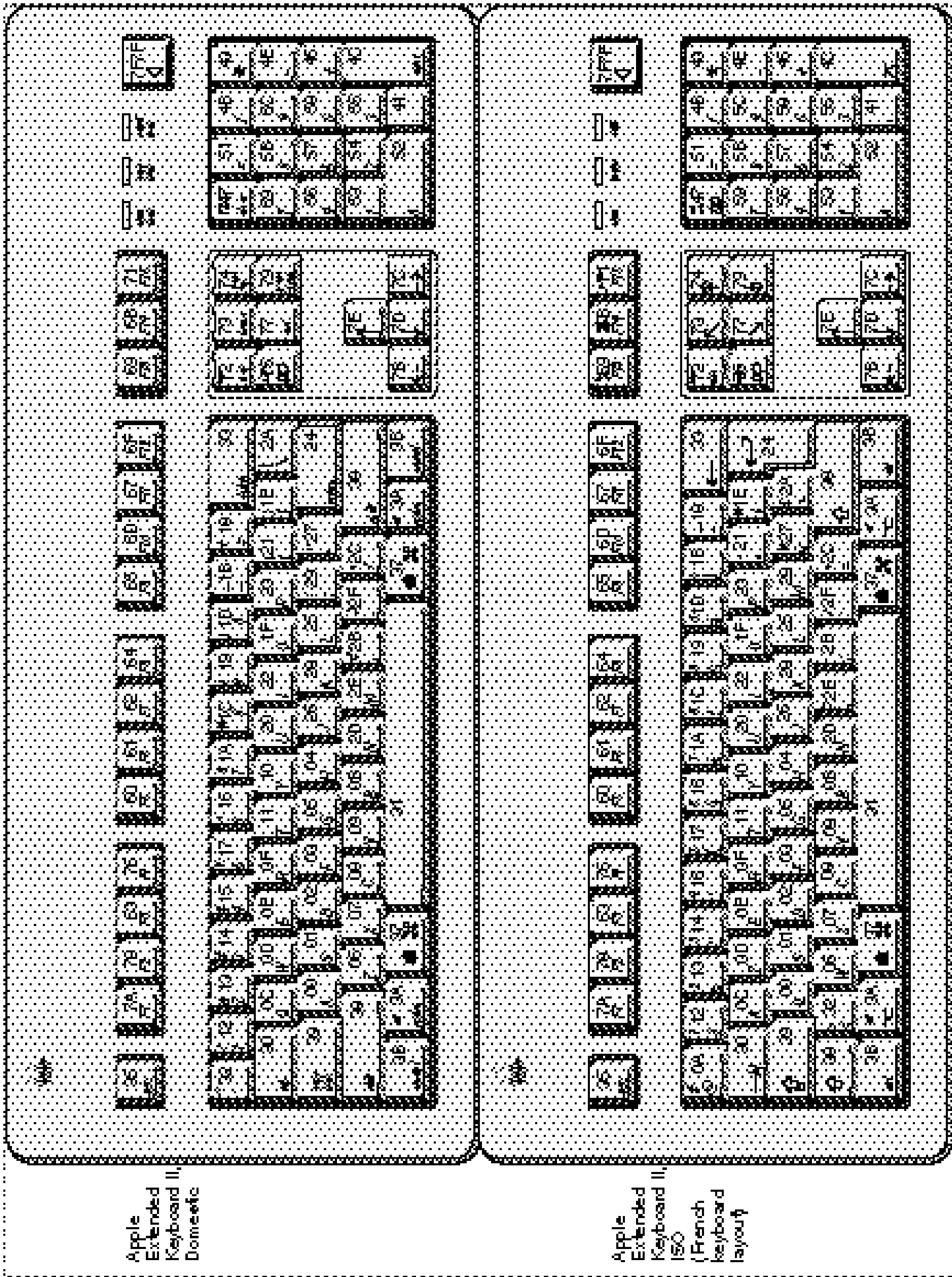
If a user of an Apple Extended Keyboard II (using the U.S. `'KCHR'` resource) presses the key labeled “C” and no modifier keys, the system software maps this through the `'KMAP'` and `'KCHR'` resources to produce a virtual key code of $08 and the character code $63 (the character “c”) in the `message` field of the event record. If the user presses the key labeled “C” and the Option key, then the system software maps this to virtual key code $08 and the character code $8D (the character “ç”) in the `message` field.

As another example, if a user of an Apple Extended Keyboard II, Domestic layout, is using the U.S. `'KCHR'` resource and presses the key labeled “M” the system software maps this through the `'KMAP'` and `'KCHR'` resources to produce a virtual key code of $2E and the character code $6D (the character “m”) in the `message` field of the event record.

If a user of an Apple Extended Keyboard II, ISO layout, is using the French `'KCHR'` resource and presses the key labeled “M” the system software maps this through the `'KMAP'` and `'KCHR'` resources to produce a virtual key code of $29 and the character code $6D (the character “m”) in the `message` field of the event record.

See *Inside Macintosh: Text* for additional information about the keyboard resources and how the Script Manager manages various scripts.

Figure 2-10 Virtual key codes for the Apple Extended Keyboard II



Listing 2-6 shows code that handles key-down and auto-key events. The `DoKeyDown` procedure is an application-defined procedure that is called from the `DoEvent` procedure. (Listing 2-3 on page 2-26 shows the `DoEvent` procedure.)

Listing 2-6 Handling key-down and auto-key events

```
PROCEDURE DoKeyDown (event: EventRecord);
VAR
   key:  Char;
BEGIN
   key := CHR(BAnd(event.message, charCodeMask));
   IF BAnd(event.modifiers, cmdKey) <> 0 THEN
   BEGIN                      {Command key down}
      IF event.what = keyDown THEN
      BEGIN {first enable/disable/check menu items as needed-- }
            { the MyAdjustMenus procedure adjusts the menus }
            { as appropriate for the current window}
         MyAdjustMenus;
         DoMenuCommand(MenuKey(key)); {handle the menu command}
      END;
   END
   ELSE
      MyHandleKeyDown(event);
END;
```

The `DoKeyDown` procedure in Listing 2-6 first extracts the character code of the key pressed from the `message` field of the event record. It then checks the `modifiers` field of the event record to determine if the Command key was pressed at the time of the event. If so, and if the event is a key-down event, the code calls the application-defined procedure `MyAdjustMenus`, and then calls another application-defined routine, `DoMenuCommand`, to perform the menu command associated with that key. (The `MyAdjustMenus` procedure adjusts the menus appropriately, and according to whether the current window is a document window or modeless dialog box. See the chapter "Menu Manager" in this book for code that defines the `MyAdjustMenus` procedure.) Otherwise, the code calls the application-defined procedure `MyHandleKeyDown` to handle the event.

Listing 2-7 shows the application-defined routine `MyHandleKeyDown`.

Listing 2-7 Handling key-down events

```
PROCEDURE MyHandleKeyDown (event: EventRecord);
VAR
   key:           Char;
   window:        WindowPtr;
```

```
   myData:         MyDocRecHnd;
   te:             TEHandle;
   windowType:     Integer;
BEGIN
   window := FrontWindow;
   {determine the type of window--document, modeless, etc.}
   windowType := MyGetWindowType(window);
   IF windowType = kMyDocWindow THEN
   BEGIN
      key := CHR(BAnd(event.message, charCodeMask));
      IF window <> NIL THEN
      BEGIN
         IF key = char(kTab) THEN {handle special characters}
            MyDoTab(event)
         ELSE
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(window));
            te := myData^^.editRec;
            IF
            (te^^.teLength - (te^^.selEnd - te^^.selStart) + 1
               < kMaxTELength) THEN
            BEGIN
               TEKey(key, te);   {insert character in document}
               MyAdjustScrollBars(window, FALSE);
               MyAdjustTE(window);
               myData^^.windowDirty := TRUE;
            END;
         END;
      END;
   END
   ELSE
      MyHandleKeyDownInModeless(event, windowType);
END;
```

The `MyHandleKeyDown` procedure in Listing 2-7 handles key-down events in any window of the application. For document windows, the code inserts the character represented by the key pressed by the user into the active document. It first finds the active document using the `FrontWindow` function, then handles the event as appropriate for the document window. For example, it treats the Tab key as a special character and calls an application-defined routine, `MyDoTab`, to handle this character appropriately for the document. For all other keys directed to the document window, the code gets the edit record associated with the document, and then it simply inserts the character into the document, using the TextEdit `TEKey` procedure. It also calls two other application-defined routines, `MyAdjustScrollBars` and `MyAdjustTE`, to update the document and edit record.

The MyHandleKeyDown procedure calls an application-defined routine, MyHandleKeyDownInModeless, to handle key-down events in modeless dialog boxes. See the chapter "Dialog Manager" in this book for more information on handling events in dialog boxes.

Scanning for a Cancel Event

Your application should allow the user to cancel a lengthy operation by using the Command-period combination. Your application can implement this cancel operation by periodically examining the state of the keyboard using the GetKeys procedure, or your application can scan the event queue for a keyboard event.

Listing 2-8 shows an application-defined function that scans the event queue for any occurrence of a Command-period event.

The UserDidCancel function in Listing 2-8 first checks to see if the user changed the script. The application maintains a global variable, gCurrentKeyScript, that keeps track of this information. The application also uses a global variable, gPeriodKeyCode, to hold the key code that maps to the period key according to the current script. If the current script has changed, the UserDidCancel function calls an application-defined routine, MySetPeriodKeyCode, to change the value of the gPeriodKeyCode global variable as necessary.

The UserDidCancel function then determines whether A/UX is running. You must use a different method to scan the event queue if A/UX is running. This code uses an application-defined function called MyCheckAUXEventQueue to search for a Command-period event if A/UX is running. Otherwise, the code checks the what field for a key-down event. If it finds a key-down event, it then checks the message field to determine whether the user pressed the period key and checks the modifiers field to determine whether the user also pressed the Command key. If it finds the Command-period combination, it sets the foundEvent variable to TRUE and returns this value. Otherwise, it looks at the next entry in the queue and continues to search the queue until it either finds a Command-period event or reaches the end of the queue.

Listing 2-8 Scanning for a Command-period event

```
FUNCTION UserDidCancel: Boolean;
VAR
   foundEvent:     Boolean;
   eventQPtr:      EvQElPtr;
   eventQHdr:      QHdrPtr;
   keyCode:        LongInt;
   isCmdKey:       LongInt;
BEGIN
   foundEvent := FALSE;             {assume the event is not there}
   {Check to see if the script has changed}
   IF (gCurrentKeyScript <> GetEnvirons(smKeyScript)) THEN
      MySetPeriodKeyCode;           {set gPeriodKeyCode to match new script}
```

```
   IF (GetAUXVersion > 0) THEN      {if A/UX is running use this method}
      foundEvent := MyCheckAUXEventQueue(gPeriodKeyCode, cmdKey)
   ELSE
   BEGIN                                          {scan event queue}
      eventQHdr := GetEvQHdr;                     {get the event queue header}
      eventQPtr := EvQElPtr(eventQHdr^.qHead);    {get first entry}
      WHILE (eventQPtr <> NIL) AND (NOT(foundEvent)) DO
      BEGIN                                       {look for key-down event}
         IF (eventQPtr^.evtQWhat = keyDown) THEN {found key-down event, }
         BEGIN                                    { look for Command-period}
            keyCode := BAND(eventQPtr^.evtQMessage, keyCodeMask);
            keyCode := BSR(keyCode, 8);
            isCmdKey := BAND(eventQPtr^.evtQModifiers, cmdKey);
            IF isCmdKey <> 0 THEN                 {Command key was pressed}
               IF keyCode = gPeriodKeyCode THEN
                  foundEvent := TRUE;             {key pressed was '.'}
         END;     {of found key-down}
         IF (NOT foundEvent) THEN                 {go to next entry}
            eventQPtr := EvQElPtr(eventQPtr^.qLink);
      END;     {of while}
   END;     {of scan event queue}
   UserDidCancel := foundEvent;                   {return result of search}
END;
```

Responding to Update Events

The Event Manager reports update events to your application whenever one of your application's windows needs updating. Upon receiving an update event, your application should update the contents of the specified window. Your application can call the Window Manager procedure `BeginUpdate`, draw the window's contents, and then call `EndUpdate` when your application has finished updating the window's contents.

Your application can also let the Window Manager automatically update the contents of a window by supplying in the window record a handle to a picture that contains the contents of the window. This technique is generally useful only for windows that contain static information that doesn't change or can't be edited. For example, if your application provides a window that always displays a picture of the earth, you can supply the handle to the picture, and the Window Manager automatically updates the window as needed, without sending your application an update event. In most cases, your application needs to perform the update itself.

The Window Manager maintains an update region for each window. The Window Manager keeps track of all areas in a window's content region that need to be redrawn and accumulates them in the window's update region. When an application calls `WaitNextEvent` or `EventAvail` (or `GetNextEvent`), the Event Manager checks to see if any windows have an update region that is not empty. If so, the Event Manager

reports update events to the appropriate applications; any applications with windows that require updating receive the necessary update events according to the normal processing of events.

If more than one window needs updating, the Event Manager issues update events for the frontmost window first. This means that updating of windows occurs in front-to-back order, which is what the user expects.

When one of your application's windows needs to be updated, the Window Manager calls the window definition function of that window, requesting that it draw the window frame. The Window Manager then generates an update event for that window. The Event Manager reports any update events for your application's windows to your application, and your application should update the window contents as necessary.

In response to an update event, your application should first call the `BeginUpdate` procedure. The `BeginUpdate` procedure temporarily replaces the visible region of the window's graphics port (that part of the window that is visible on the screen) with the intersection of the visible region and update region of the window. The `BeginUpdate` procedure then clears the update region of the window—preventing the update event for this occurrence from being reported again.

After calling `BeginUpdate`, your application should draw the window's contents, either entirely or in part. You can draw either the entire content region or only the area in the visible region. In either case, the Window Manager allows only what falls within the visible region to be drawn on the screen. (Because the `BeginUpdate` procedure intersects the visible region with the update region, the visible region at this point corresponds to any visible parts of the old update region.)

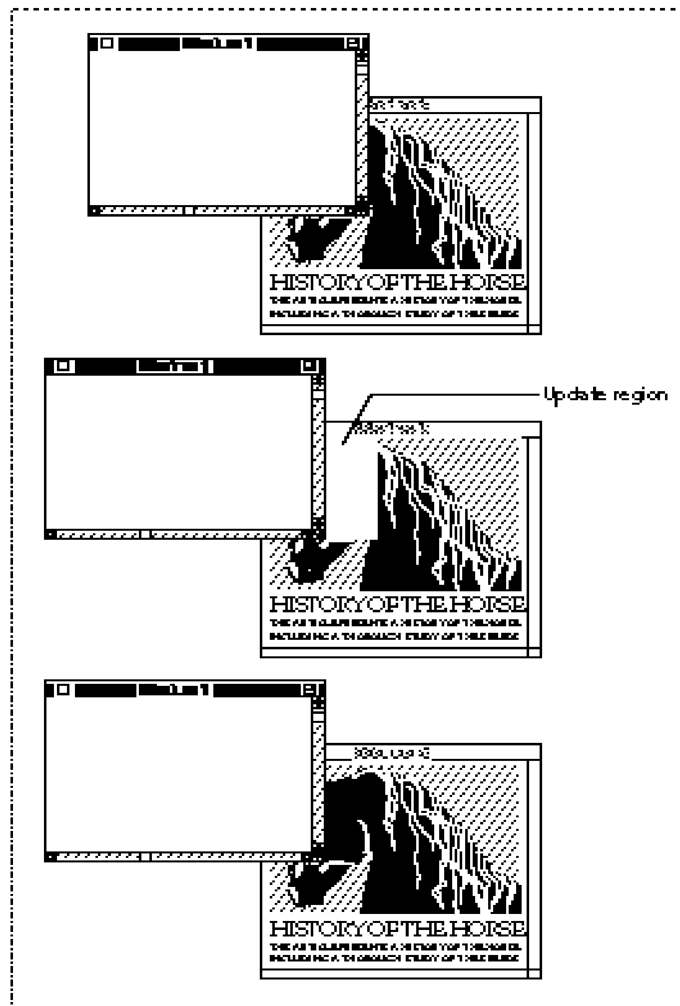
The `EndUpdate` procedure restores the normal visible region of the window's graphics port.

Figure 2-11 shows how an application updates its windows. In this example, Window 1 partially covers Window 2. When the user moves Window 1 so that more of Window 2 is exposed, the Window Manager requests the window definition function of the window to update the window frame, and accumulates the area requiring updating in the update region of the window.

When the application receives an update event for this window, the `message` field of the event record contains a pointer to the window that needs updating. Your application can call `BeginUpdate`, draw the window's contents, and then call `EndUpdate`. This completes the handling of the update event.

Your application can receive update events when it is in the foreground or in the background. In the example shown in Figure 2-11, Window 1 and Window 2 could belong to the same application or different applications. In either case, the Event Manager reports an update event to the application whose window contents need updating.

Figure 2-11 Responding to an update event for a window



Your application should respond to update events or at least call the `BeginUpdate` procedure in response to an update event. If you do not call the `BeginUpdate` procedure, your application continues to receive update events for the window (until the update region is empty). You should always make sure that you match a call to `BeginUpdate` with a call to `EndUpdate`. By calling the `BeginUpdate` and `EndUpdate` procedures, you indicate to the Window Manager that you have updated the window and handled the update event.

Listing 2-9 shows an example of an application-defined routine that responds to update events.

Listing 2-9 Responding to update events

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
   windowType: Integer;
BEGIN
   {determine the type of window--document, modeless, etc.}
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyDocWindow:
         BEGIN
            BeginUpdate(window);
            MyDrawWindow(window);
            EndUpdate(window);
         END;
      OTHERWISE
         DoUpdateMyDialog(window);
   END; {of CASE}
END;
```

The `DoUpdate` procedure in Listing 2-9 first determines if the window is a document window or a modeless dialog box. The `MyGetWindowType` function is an application-defined routine that returns the `kMyDocWindow` constant if the window is a document window and returns other application-defined constants if the window is a modeless dialog box.

If the window is a document window, the procedure does all its drawing of the window within calls to the `BeginUpdate` and `EndUpdate` procedures. The application-defined routine `MyDrawWindow` performs the actual updating of the document window contents. See the chapter "Window Manager" in this book for code that shows the `MyGetWindowType` and `MyDrawWindow` routines.

If the window is a modeless dialog box, the code calls the application-defined `DoUpdateMyDialog` procedure to update the contents of the dialog box. See the chapter "Dialog Manager" in this book for details on handling update events in dialog boxes.

Responding to Activate Events

When several windows belonging to your application are open, you should allow the user to switch from one window to another by clicking in the appropriate window. To implement this, whenever your application receives a mouse-down event, you should

first determine whether the user clicked in another window by using the Window Manager function FindWindow; if so, you can use the Window Manager procedure SelectWindow to generate the necessary activate events.

Before returning to your application and before your application receives any events relating to this occurrence, the SelectWindow procedure does some work for you, such as removing the highlighting from the window to be deactivated and highlighting the newly activated window. At your application's next request for an event, the Event Manager returns an activate event.

An activate event indicates the window involved and whether the window is being activated or deactivated. Your application should perform any other actions needed to complete the action of the window becoming active or inactive. For example, when a window becomes active, your application should show any scroll bars and restore selections as necessary.

Your application typically receives an activate event (with a flag that indicates the window should be deactivated) for the window being deactivated, followed by an activate event for the window becoming active.

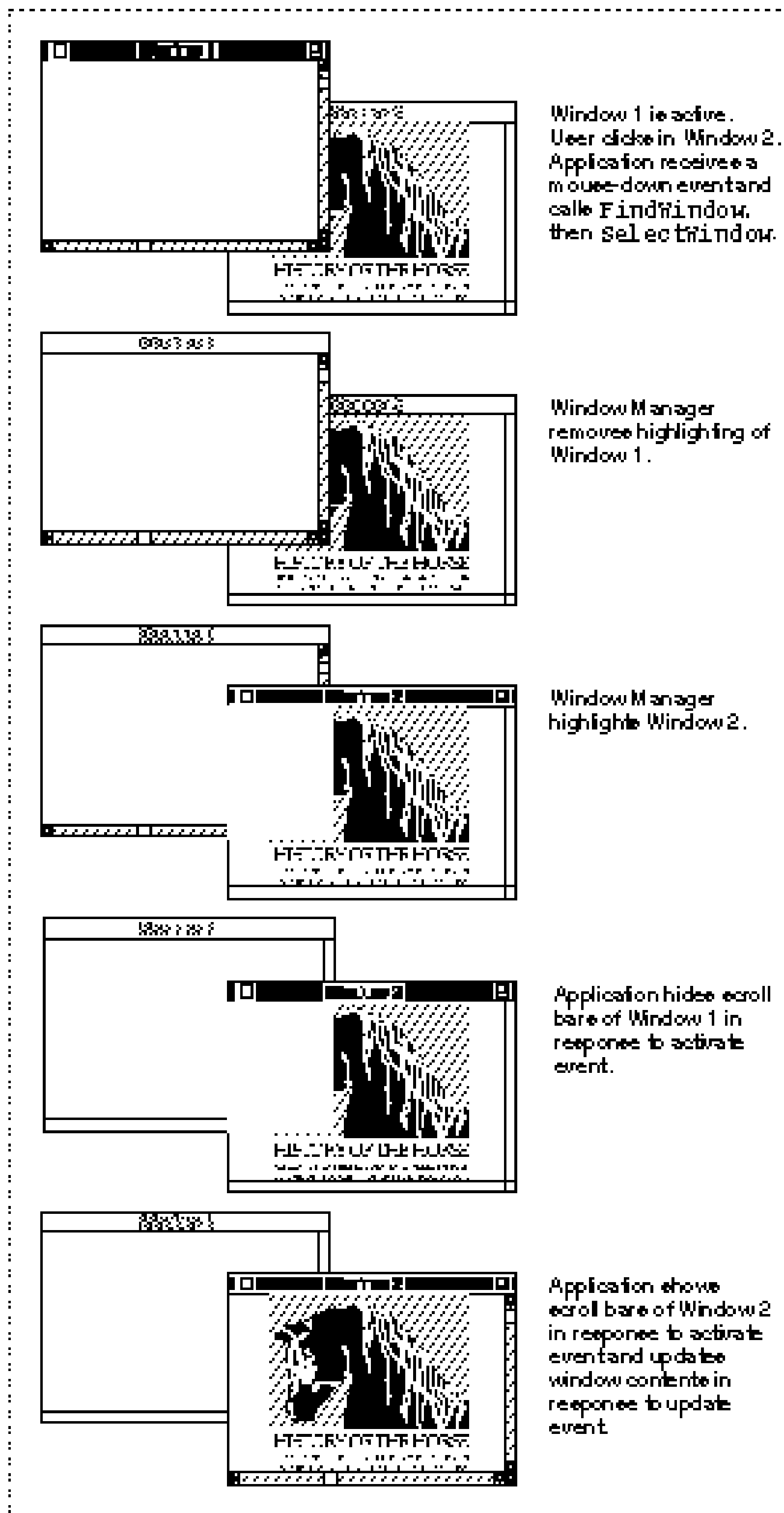
Activate events are not placed into the Operating System event queue but are sent directly to the Event Manager.

Figure 2-12 on the next page shows two documents belonging to the same application, with Window 1 the active window. When the user clicks in Window 2, your application receives a mouse-down event and can use the FindWindow function to determine whether the mouse location is in an inactive window. If so, your application should call the SelectWindow procedure. The SelectWindow procedure removes highlighting of Window 1, highlights Window 2, and generates activate events for both of these occurrences. The Event Manager reports the activate events one at a time to your application; in this example, the first activate event indicates that Window 1 should be deactivated. Your application should hide the scroll bars and remove the highlighting from any selections as necessary.

The next activate event indicates that Window 2 should be activated. Your application should show the scroll bars and restore any selections as necessary. If the window needs updating as a result of being activated, the Event Manager sends your application an update event so that your application can update the window contents.

Your application also needs to activate or deactivate windows in response to suspend and resume events. If you set the acceptSuspendResumeEvents flag and the doesActivateOnFGSwitch flag in your application's 'SIZE' resource, your application is responsible for activating or deactivating your application's windows in response to handling suspend and resume events. If you set the acceptSuspendResumeEvents flag and do not set the doesActivateOnFGSwitch flag, your application receives an activate event immediately following a suspend or resume event. In most cases, you should set both the acceptSuspendResumeEvents and doesActivateOnFGSwitch flags in your application's 'SIZE' resource.

Figure 2-12 Responding to activate events for a window



The what field of an event record for an activate event contains the activateEvt constant. The message field contains a pointer to the window being activated or deactivated. The modifiers field contains additional information about the activate event, along with information about the state of the modifier keys at the time the event was posted. Your application can examine bit 0 of the modifiers field of the event record to determine if the window should be activated or deactivated. Bit 0 of the modifiers field is 1 if the window should be activated and 0 if the window should be deactivated. You can use the activeFlag constant to test the state of this bit in the modifiers field.

The when field of the event record contains the number of ticks since the system last started up. The where field of the event record contains the location of the cursor at the time the activate event occurred.

Upon receiving an activate event that indicates the window is being deactivated, your application should hide any scroll bars and remove the highlighting from any selections as necessary.

Upon receiving an activate event that indicates the window is becoming active, your application should show any scroll bars, highlight any selections, and otherwise restore the window to the state it was in when it was last active. For example, your application should restore the insertion point to its previous position, and the document should be scrolled to the position in which the user last left it. Your application should also adjust its menus appropriately for the newly active window—adjusting the marks and enabled state of menu items based on the state of the active window.

Listing 2-10 shows an application-defined procedure that responds to activate events.

Listing 2-10 Responding to activate events

```
PROCEDURE DoActivate (window: windowPtr; activate: Boolean;
                      event: EventRecord);
VAR
   growRect:   Rect;          {window's grow rectangle}
   myData:     MyDocRecHnd;   {window's document record}
   windowType: Integer;
BEGIN
   {determine the type of window--document, modeless, etc.}
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kMyDocWindow:
   BEGIN
      myData := MyDocRecHnd(GetWRefCon(window));
      HLock(Handle(myData));
      WITH myData^^ DO
      IF activate THEN     {window is being activated}
```

```
      BEGIN
         {restore any selections or display caret}
         MyRestoreSelection(window);
         {adjust menus as appropriate for this document window}
         MyAdjustMenus;
         {activate any scroll bars}
         vScrollBar^^.contrlVis := kControlVisible;
         hScrollBar^^.contrlVis := kControlVisible;
         {invalidate area of scroll bars to force update}
         InvalRect(vScrollBar^^.contrlRect);
         InvalRect(hScrollBar^^.contrlRect);
         {invalidate area of size box, if any}
         growRect := window^.portRect;
         WITH growRect DO
         BEGIN
            top := bottom - kScrollbarAdjust;
            left := right - kScrollbarAdjust;
         END; {end of WITH growRect statement}
         InvalRect(growRect);
      END
      ELSE                 {window is being deactivated}
      BEGIN
         {unhighlight selection (if any) or hide the caret}
         MyHideSelection;
         HideControl(vScrollBar);   {hide any scroll bars}
         HideControl(hScrollBar);
         DrawGrowIcon(window);      {change size box immediately}
      END;
      HUnLock(Handle(myData));
   END; {end of kMyDocWindow}

   kMyGlobalChangesID:  {this window is a modeless dialog box }
                        { for this app's Global Changes command}
      MyDoActivateGlobalChangesDialog(window, event);
   {handle other modeless dialog boxes as appropriate}
   END; {of CASE}
END;
```

Listing 2-10 uses the application-defined function `MyGetWindowType` to determine what type of window is involved with the activate event. If the window is a document window, the `DoActivate` procedure uses the `GetWRefCon` function to get a handle to the window's document record. (The `DoActivate` procedure, and other application-defined routines, maintain information about the document associated with a window in a document record; the application stores a handle to the document record as the window's reference constant value when it creates a new window. See the chapter "Window Manager" in this book for information on defining a document record.)

If the document window should be activated, the code calls an application-defined routine, MyRestoreSelection. Your application should restore any selection or display the caret as appropriate. For example, if your application uses TextEdit to display text in the content area of windows, you can call the TextEdit procedure TEActivate to restore any selection or display a caret at the insertion point. The DoActivate procedure then calls another application-defined procedure, MyAdjustMenus, to adjust the menus as appropriate for the document window. (See the chapter "Menu Manager" for a listing of the MyAdjustMenus procedure.) After restoring any selections and adjusting its menus, the code shows the scroll bars and size box of the window being activated. It does this by invalidating the area of the scroll bars and size box, accumulating these areas into the update region. This causes an update event to be generated. The application redraws its controls as appropriate in response to update events.

If the document window should be deactivated, the code in Listing 2-10 unhighlights the selection and hides the caret by calling the application-defined procedure MyHideSelection. The code then hides the scroll bars and size box of the deactivated window.

If the window associated with the activate event is a modeless dialog box, for example, a Global Changes modeless dialog box, the DoActivate procedure calls an application-defined procedure to activate or deactivate the dialog box as needed. See the "Dialog Manager" chapter in this book for information on handling activate events in modeless dialog boxes.

Responding to Disk-Inserted Events

When your application uses the Standard File Package to allow the user to choose a file to open or choose a location for storing a file, the Standard File Package responds to disk-inserted events for your application while interacting with the user. In most cases, if your application receives an unexpected disk-inserted event, it can simply check to see if the disk was successfully mounted and use the Disk Initialization Manager function DIBadMount to notify the user if the disk was not successfully mounted.

When the user inserts a disk, the Operating System attempts to mount the volume on the disk by calling the File Manager function PBMountVol. If the volume is successfully mounted, an icon representing the disk appears on the desktop. The Operating System Event Manager then generates a disk-inserted event. If the user is interacting with a standard file dialog box, the Standard File Package intercepts the disk-inserted event and handles it. Otherwise, the event is left in the event queue for your application to retrieve. The Desk Manager also intercepts and handles disk-inserted events if a desk accessory is in front.

Usually your application should handle and not mask out disk-inserted events. The user might insert a disk at any time and expects to be warned if the disk is uninitialized or damaged. If your application receives a disk-inserted event and the volume was successfully mounted, your application usually does not need to take any further action. However, if the volume was not successfully mounted, then your application should give the user a chance to initialize or eject the uninitialized or damaged disk.

If you do mask out disk-inserted events, the event stays in the Operating System event queue until your application calls the Standard File Package or until an application that does handle disk-inserted events becomes the foreground process. This situation can be confusing to the user, so your application should handle disk-inserted events at the time that they occur.

If the volume was successfully mounted and your application either does not use the Standard File Package or prompts the user to insert a disk, then you can choose to respond to disk-inserted events in whatever way is appropriate for your application.

The Dialog Manager procedure `ModalDialog` masks out disk-inserted events. (The Standard File Package changes the mask in order to receive disk-inserted events.) If one of your application's modal dialog boxes needs to respond to disk-inserted events, then you can change the event mask from within the event filter function that you supply as one of the parameters to `ModalDialog`. Otherwise, your application can respond to the disk-inserted event after the user dismisses the modal dialog box.

The `what` field of the event record contains the `diskEvt` constant to indicate a disk-inserted event. The `message` field contains the drive number in the low-order word and the result code from the `PBMountVol` function in the high-order word. Your application can examine the high-order word to determine if the attempt to mount the volume was successful. If the volume was not successfully mounted, your application can notify the user using the Disk Initialization Manager function `DIBadMount`. If the volume was successfully mounted, your application can use the drive number returned in the low-order word for accessing the disk.

Listing 2-11 shows a procedure that handles disk-inserted events. If the disk was not successfully mounted, the procedure notifies the user using the `DIBadMount` function. Otherwise, it does not take any action. See the chapter "Disk Initialization Manager" in *Inside Macintosh: Files* for information on the routines provided by the Disk Initialization Manager.

Listing 2-11 Responding to disk-inserted events

```
PROCEDURE DoDiskEvent (event: EventRecord);
VAR
   thisPoint:  Point;
   myErr:      OSErr;
BEGIN
   IF HiWord(event.message) <> noErr THEN
   BEGIN                         {attempt to mount was unsuccessful}
      DILoad;                    {load Disk Initialization Manager}
      SetPt(thisPoint, 120, 120);
                                 {notify the user}
      myErr := DIBadMount(thisPoint, event.message);
      DIUnload;                  {unload Disk Initialization Manager}
   END
```

```
   ELSE                       {attempt to mount was successful}
      ;        {record the drive number or do other processing}
END;
```

Responding to Null Events

When the Event Manager has no other events to report, it returns a null event. The `WaitNextEvent` function reports a null event by returning a function result of `FALSE` and setting the `what` field of the returned event record to `nullEvt`. (The `EventAvail` and `GetNextEvent` functions also return null events in this manner.)

When your application receives a null event, it can perform idle processing. Your application should do minimum processing in response to a null event, so that other processes can use the CPU and so that the foreground process (or your application, if it is in the foreground) can respond promptly to the user.

For example, if your application receives a null event and it is in the foreground, it can make the caret blink in the active window.

If your application receives a null event in the background, it can perform tasks or do other processing while in the background. However, your application should not perform any tasks that would slow down the responsiveness of the foreground process. Your application also should not interact with the user if it is in the background.

If you don't want your application to receive null events when it is in the background, set the `cannotBackground` flag in your application's `'SIZE'` resource.

Listing 2-12 shows a procedure that performs idle processing in response to a null event. If the application is not in the background and the active window is a document window, this code calls the TextEdit procedure `TEIdle`. The `TEIdle` procedure makes a blinking caret appear at the insertion point in the text referred to by the edit record. (This application uses TextEdit to display text in its document windows; if you don't use TextEdit for your document windows, provide your own routine to blink the caret.) If the active window is a modeless dialog box, the `DoIdle` procedure calls the Dialog Manager function `DialogSelect` to blink the caret in any editable text item of the dialog box.

Listing 2-12 Handling null events

```
PROCEDURE DoIdle (event: EventRecord);
VAR
   window:     WindowPtr;
   myData:     MyDocRecHnd;
   windowType: Integer;
   itemHit:    Integer;
   result:     Boolean;
```

```
BEGIN
   window := FrontWindow;
   {determine the type of window--document, modeless, etc.}
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyDocWindow:
         IF (NOT gInBackground) THEN
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(window));
            TEIdle(myData^^.editRec);
         END;
      kMyGlobalChangesID:
         result := DialogSelect(event, window, itemHit);
   END; {of CASE}
END;
```

Handling Operating-System Events

Operating-system events include suspend, resume, and mouse-moved events. Your application receives suspend and resume events as a result of changes in its processing status. Your application can request that the Event Manager return mouse-moved events whenever the cursor is outside a specified region by specifying a nonempty region in the `mouseRgn` parameter to `WaitNextEvent`. If you specify an empty region or a `NIL` region handle in the `mouseRgn` parameter, the Event Manager does not report mouse-moved events.

Your application examines the event record to determine which event it received and to obtain additional information associated with the event.

The `what` field in the event record of an operating-system event contains the `osEvt` constant.

The `message` field in the event record of an operating-system event contains information indicating whether the event is a suspend, resume, or mouse-moved event. The `message` field also indicates whether Clipboard conversion is required when the application resumes execution. The bits in the `message` field give this information:

Bit	Contents
0	0 if a suspend event
	1 if a resume event
1	0 if Clipboard conversion not required
	1 if Clipboard conversion required
2–23	Reserved
24–31	`suspendResumeMessage` if a suspend or resume event `mouseMovedMessage` if a mouse-moved event

Note that you need to examine bits 24–31 of the message field to determine what kind of operating-system event you have received. Bits 24–31 in the message field contain one of these two constants:

```
CONST suspendResumeMessage = $01;      {suspend or resume event}
      mouseMovedMessage    = $FA;      {mouse-moved event}
```

If the event is a suspend or resume event, you need to examine bit 0 to determine whether that event is a suspend or resume event. Bits 0 and 1 are meaningful only if bits 24–31 indicate that the event is a suspend or resume event. You can use the resumeFlag constant to determine whether the event is a suspend or resume event. If the event is a resume event, you can use the convertClipboardFlag constant to determine whether Clipboard conversion from the Clipboard to your application's scrap is required:

```
CONST resumeFlag           = 1;  {resume event}
      convertClipboardFlag = 2;  {Clipboard conversion required}
```

Whenever the user performs a copy or cut operation, your application should copy the selected data either to its private scrap or, if your application doesn't have a private scrap, to the Clipboard. If your application uses a private scrap, you need to convert the data from your private scrap to the Clipboard whenever your application receives a suspend event. Likewise, you need to convert any data from the Clipboard (if it has changed) when your application receives a resume event. For resume events, the value of bit 1 of the message field is 1 if your application needs to read in the new contents of the Clipboard.

Listing 2-13 shows a procedure that responds to operating-system events.

Listing 2-13 Responding to operating-system events

```
PROCEDURE DoOSEvent (event: EventRecord);
BEGIN
   CASE BAnd(BRotL(event.message, 8), $FF) OF {get high byte}
      mouseMovedMessage:
         DoIdle(event); {mouse-moved same as idle for this app}
      suspendResumeMessage:
         DoSuspendResumeEvent(event);{handle supend/resume event}
   END;
END;
```

The DoOSEvent procedure in Listing 2-13 is called from the DoEvent procedure (shown in Listing 2-3 on page 2-26) whenever the application receives an operating-system event. The DoOSEvent procedure examines the high byte of the message field to determine whether the event is a mouse-moved, suspend, or resume event, and it then calls an application-defined procedure to handle the event. Note that most applications either adjust the cursor in response to mouse-moved events or adjust the cursor in their event loop whenever any type of event is received. The code in this chapter uses the

latter approach, and thus the `DoOSEvent` procedure simply calls its `DoIdle` procedure in response to mouse-moved events. The next two sections show the code that handles suspend, resume, and mouse-moved events.

Responding to Suspend and Resume Events

The `WaitNextEvent` function returns a suspend event when your application is about to be switched to the background. `WaitNextEvent` returns a resume event when your application becomes the foreground process again.

Upon receiving a suspend event, your application should deactivate the front window, remove the highlighting from any selections, and hide any floating windows. Your application should also convert any private scrap into the global scrap, if necessary. If your application shows a window that displays the Clipboard contents, you should hide this window also, as the user might change the contents of the Clipboard before returning to your application. Your application can also do anything else necessary to get ready for a major switch. Then your application should call `WaitNextEvent` to relinquish the processor and allow the Operating System to schedule other processes for execution.

Upon receiving a resume event, your application should activate the front window and restore any windows to the state the user left them in at the time of the previous suspend event. For example, your application should show scroll bars, restore any selections that were previously in effect, and show any floating windows. Your application should copy the contents of the Clipboard and convert the data back to its private scrap, if necessary. If your application shows a window that displays the Clipboard contents, you can update the contents of the window after reading in the scrap. Your application can then resume interacting with the user.

Responding to a suspend or resume event usually involves activating or deactivating windows. If you set the `acceptSuspendResumeEvents` flag and the `doesActivateOnFGSwitch` flag in your application's `'SIZE'` resource, your application is responsible for activating or deactivating your application's windows in response to handling suspend and resume events.

Note

If you set the `acceptSuspendResumeEvents` flag and do not set the `doesActivateOnFGSwitch` flag in your application's `'SIZE'` resource, your application receives an activate event immediately following a suspend or resume event. In most cases, you should set both the `acceptSuspendResumeEvents` and `doesActivateOnFGSwitch` flags in your application's `'SIZE'` resource. ◆

Your application can use the Scrap Manager functions `InfoScrap`, `ZeroScrap`, `PutScrap`, and `GetScrap` to read data from and write data to the Clipboard. See the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox* for additional details.

Note
If your application does not handle suspend and resume events (as indicated by a flag in its 'SIZE' resource), then the Operating System has to trick your application into performing scrap coercion to ensure that the contents of the Clipboard can be transferred from one application to another. This process adds to the time it takes to move the foreground application to the background and vice versa. ◆

Listing 2-14 shows a procedure that responds to suspend and resume events. The `DoSuspendResumeEvent` procedure first gets a pointer to the front window using the Window Manager function `FrontWindow`. It then examines bit 0 of the `message` field of the event record to determine whether the event is a suspend or resume event. If the event is a resume event, the code examines bit 1 of the `message` field of the event record to determine whether it needs to read in the contents of the scrap. If so, the code calls an application-defined routine, `MyConvertScrap`, that reads in the scrap and converts the contents to its private scrap. It then sets a private global flag, `gInBackground`, to `FALSE`, to indicate that the application is not in the background. It then calls another application-defined routine, `DoActivate` (shown in Listing 2-10), to activate the application's front window.

For suspend events, the `DoSuspendResumeEvent` procedure calls the application-defined `MyConvertScrap` procedure to copy the contents of its private scrap to the global scrap. It then sets a private global flag, `gInBackground`, to `TRUE`, to indicate that the application is in the background. Finally, it calls another application-defined routine to deactivate the application's front window.

Listing 2-14 Responding to suspend and resume events

```
PROCEDURE DoSuspendResumeEvent (event: EventRecord);
VAR
   currentFrontWindow: WindowPtr;
BEGIN                         {handle suspend/resume event}
   currentFrontWindow := FrontWindow;
   IF (BAnd(event.message, resumeFlag) <> 0) THEN
   BEGIN                {it's a resume event}
      IF (BAnd(event.message, convertClipboardFlag) <> 0) THEN
         MyConvertScrap(kClipboardToPrivate);
      gInBackground := FALSE;
                        {activate front window}
      DoActivate(currentFrontWindow, NOT gInBackground, event);
      MyShowClipboardWindow;  {show Clipboard window if it was }
                              { showing at last suspend event}
      MyShowFloatingWindows;  {show any floating windows}
   END
   ELSE
```

```
   BEGIN                {it's a suspend event}
      MyConvertScrap(kPrivateToClipboard);
      gInBackground := TRUE;
                        {deactivate front window}
      DoActivate(currentFrontWindow, NOT gInBackground, event);
      MyHideClipboardWindow; {hide Clipboard window if showing}
      MyHideFloatingWindows; {hide any floating windows}
   END;
END;
```

Your application can receive processing time while in the background and perform tasks in the background, but your application should not interact with the user or perform tasks that would slow down the responsiveness of the foreground process.

If you need to notify the user of some special occurrence while your application is executing in the background, you should use the Notification Manager to queue a notification request. See the chapter "Notification Manager" in *Inside Macintosh: Processes* for examples of how to post notification requests.

Responding to Mouse-Moved Events

Whenever the user moves the mouse, the mouse driver, the Event Manager, and your application are responsible for providing feedback to the user. The mouse driver performs low-level functions, such as continually polling the mouse for its location and status and maintaining the current location of the mouse in a global variable.

As the user moves the mouse, the user expects the cursor to move to a corresponding relative location on the screen. The low-level interrupt routines of the mouse driver map the movement of the mouse to relative locations on the screen. Whenever the user moves the mouse, a low-level interrupt routine of the mouse driver moves the cursor displayed on the screen and aligns the hot spot of the cursor with the new mouse location. A **hot spot** is a point that the mouse driver uses to align the cursor with the mouse location.

Your application is responsible for setting the initial appearance of the cursor, for restoring the cursor after `WaitNextEvent` returns, and for changing the appearance of the cursor as appropriate for your application. For example, most applications set the cursor to the I-beam when the cursor is inside a text-editing area of a document, and change the cursor to an arrow when the cursor is inside the scroll bar of a document. Your application can achieve this effect by requesting that the Event Manager report mouse-moved events if the user moves the cursor out of a region you specify in the `mouseRgn` parameter to the `WaitNextEvent` function.

The mouse driver and your application control the shape and appearance of the cursor. A cursor can be any 256-bit image, defined by a 16-by-16 bit square. The mouse driver displays the current cursor, which your application can change by using various cursor-handling routines (for example, the `SetCursor` procedure).

Figure 2-13 shows the standard arrow cursor. You can initialize the cursor to the standard arrow cursor using the `InitCursor` procedure. In Figure 2-13, the hot spot for the arrow cursor is at location (1,1). See *Inside Macintosh: Imaging* for information on

the cursor-handling routines and for specific details of how your application can define its own cursors.

Figure 2-13 The standard arrow cursor

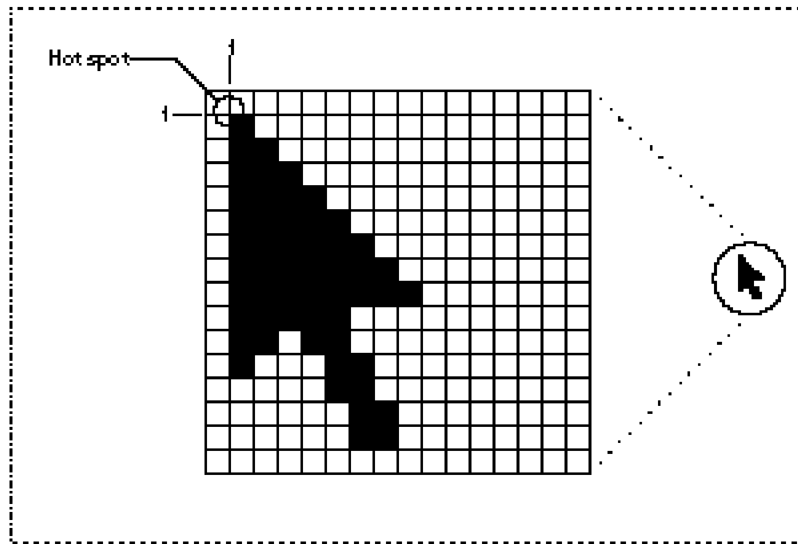
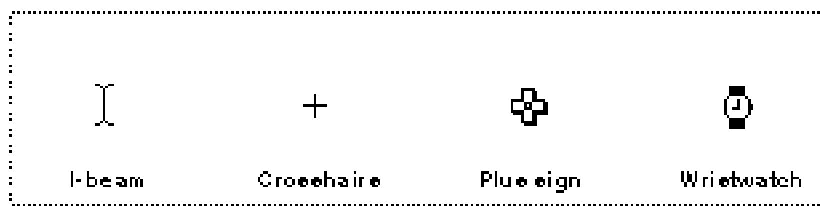


Figure 2-14 shows four other common cursors that are available to your application: the I-beam, crosshairs, plus sign, and wristwatch cursors.

Figure 2-14 The I-beam, crosshairs, plus sign, and wristwatch cursors



The I-beam, crosshairs, plus sign, and wristwatch cursors are defined as resources, and your application can get a handle to any of these cursors by specifying their corresponding resource IDs to the `GetCursor` function. These constants specify the resource IDs for the I-beam, crosshairs, plus sign, and wristwatch cursors:

```
CONST iBeamCursor = 1;{used in text editing}
      crossCursor = 2;{often used for manipulating graphics}
      plusCursor  = 3;{often used for selecting fields in }
                      { an array}
      watchCursor = 4;{used to mean a lengthy operation }
                      { is in progress}
```

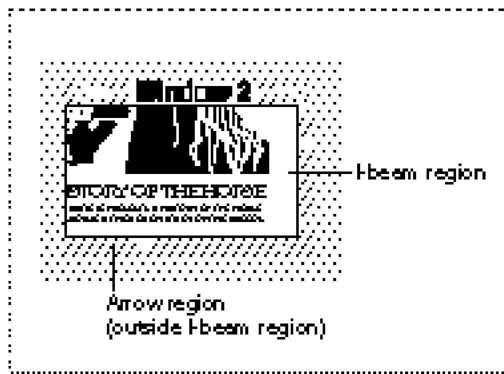
You can change the appearance of the cursor using the SetCursor procedure or other cursor-handling routines. You can also define your own cursors, store them in resources, and use them as needed in your application.

Your application usually needs to change the shape of the cursor as the user moves the cursor to different areas within a document. Your application can use mouse-moved events to accomplish this. Your application also needs to adjust the cursor in response to resume events. Most applications adjust the cursor once through the event loop in response to almost all events.

You can request that the Event Manager report mouse-moved events whenever the cursor is outside of a specified region that you pass as a parameter to the WaitNextEvent function. If you specify an empty region or a NIL handle to the WaitNextEvent function, WaitNextEvent does not report mouse-moved events.

If you specify a nonempty region in the mouseRgn parameter to the WaitNextEvent function, WaitNextEvent returns a mouse-moved event whenever the cursor is out of this region. For example, Figure 2-15 shows a document window. An application might define two regions: a region that encloses the text area of a window (the ***I-beam region***), and a region that defines the scroll bars and all other areas outside the text area (the ***arrow region***). By specifying the I-beam region to WaitNextEvent, the mouse driver continues to display the I-beam cursor until the user moves the cursor out of this region.

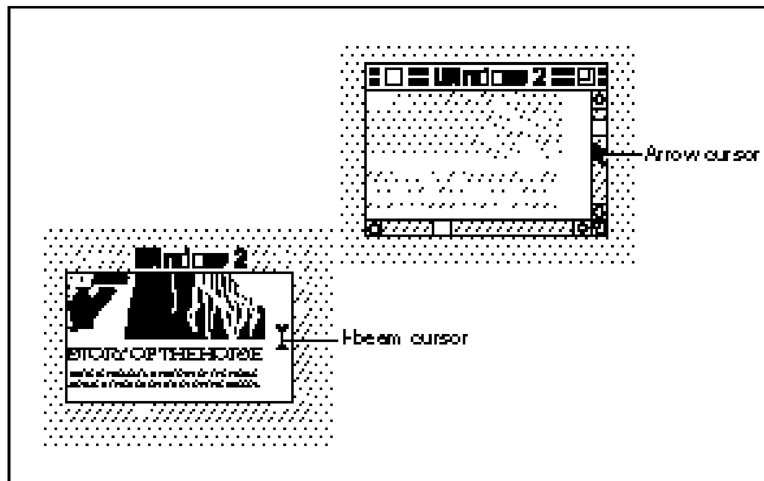
Figure 2-15 The arrow region and the I-beam region



When the user moves the cursor out of the I-beam region, WaitNextEvent reports a mouse-moved event. Your application can then change the I-beam cursor to the arrow cursor and change the mouseRgn parameter to the area defined by the scroll bars and all other areas outside of the I-beam region. The cursor now remains an arrow until the user moves the cursor out of this region, at which point your application receives a mouse-moved event.

Figure 2-16 shows how an application might change the cursor from the I-beam cursor to the arrow cursor after receiving a mouse-moved event.

Figure 2-16 Changing the cursor from the I-beam cursor to the arrow cursor



Note that your application should recalculate the `mouseRgn` parameter when it receives a mouse-moved event; otherwise, it will continue to receive mouse-moved events as long as the cursor position is outside the original region.

After receiving any event other than a high-level event, the `MyEventLoop` procedure (shown in Listing 2-2 on page 2-24) calls the application-defined procedure `MyAdjustCursor` to adjust the cursor. After adjusting the cursor, if the event is an operating-system event, the `DoEvent` procedure calls the `DoOSEvent` procedure. The `DoOSEvent` procedure calls the `DoIdle` procedure for mouse-moved events. The `DoIdle` procedure simply calls `TEIdle` to blink the caret in the text-editing window.

Listing 2-15 shows the application-defined routine `MyAdjustCursor`.

Listing 2-15 Changing the cursor

```
PROCEDURE MyAdjustCursor (mouse: Point; VAR region: RgnHandle);
VAR
   window:         WindowPtr;
   arrowRgn:       RgnHandle;
   iBeamRgn:       RgnHandle;
   iBeamRect:      Rect;
   myData:         MyDocRecHnd;
   windowType:     Integer;
BEGIN
window := FrontWindow;
{determine the type of window--document, modeless, etc.}
windowType := MyGetWindowType(window);
CASE windowType OF
   kMyDocWindow:
```

```
BEGIN
   {initialize regions for arrow and I-beam}
   arrowRgn := NewRgn;
   ibeamRgn := NewRgn;

   {set arrow region to large region at first}
   SetRectRgn(arrowRgn, -32768, -32768, 32766, 32766);

   {calculate I-beam region}
   {first get the document's TextEdit view rectangle}
   myData := MyDocRecHnd(GetWRefCon(window));
   iBeamRect := myData^^.editRec^^.viewRect;
   SetPort(window);
   WITH iBeamRect DO
   BEGIN
      LocalToGlobal(topLeft);
      LocalToGlobal(botRight);
   END;
   RectRgn(iBeamRgn, iBeamRect);
   WITH window^.portBits.bounds DO
      SetOrigin(-left, -top);
   {intersect I-beam region with window's visible region}
   SectRgn(iBeamRgn, window^.visRgn, iBeamRgn);
   SetOrigin(0,0);

   {calculate arrow region by subtracting I-beam region}
   DiffRgn(arrowRgn, iBeamRgn, arrowRgn);

   {change the cursor and region parameter as necessary}
   IF PtInRgn(mouse, iBeamRgn) THEN {cursor is in I-beam rgn}
   BEGIN
      SetCursor(GetCursor(iBeamCursor)^^);      {set to I-beam}
      CopyRgn(iBeamRgn, region);     {update the region param}
   END;

   {update cursor if in arrow region}
   IF PtInRgn(mouse, arrowRgn) THEN {cursor is in arrow rgn}
   BEGIN
      SetCursor(arrow);             {set cursor to the arrow}
      CopyRgn(arrowRgn, region);    {update the region param}
   END;
   DisposeRgn(iBeamRgn);
   DisposeRgn(arrowRgn);
END; {of kMyDocWindow}
```

```
   kMyGlobalChangesID:
      MyCalcCursorRgnForModelessDialogBox(window, region);

   kNil:
   BEGIN
      MySetRegionNoWindows(kNil, region);
      SetCursor(arrow);
   END;
 END; {of CASE}
END;
```


The MyAdjustCursor procedure sets the cursor appropriately, according to whether a document window or modeless dialog box is active.

For a document window, the code in Listing 2-15 defines two regions, specified by the arrowRgn and iBeamRgn variables. If the cursor is inside the region described by the arrowRgn variable, the code sets the cursor to the arrow cursor and returns the region described by arrowRgn. Similarly, if the cursor is inside the region described by the iBeamRgn variable, the code sets the cursor to the I-beam cursor and returns the region described by iBeamRgn.

The MyAdjustCursor procedure calculates the two regions by first setting the arrow region to the largest possible region. It then sets the I-beam region to the region described by the document's TextEdit view rectangle. This region typically corresponds to the content area of the window minus the scroll bars. (If your application doesn't use TextEdit for its document window, then set this region as appropriate to your application.) The code then adjusts the I-beam region so that it includes only the part of the content area that is in the window's visible region (for example, to take into account any floating windows that might be over the window). The code then sets the arrow region to include the entire screen except for the region occupied by the I-beam region.

The procedure then determines which region the cursor is in and sets the cursor and region parameter appropriately.

For modeless dialog boxes (for example, the Global Changes modeless dialog box), the MyAdjustCursor procedure calls an application-defined routine to appropriately adjust the cursor for the modeless dialog box. The MyAdjustCursor procedure also appropriately adjusts the cursor if no windows are currently open.

Handling High-Level Events

High-level events provide a means of communication between applications. Apple events are high-level events that follow the Apple Event Interprocess Messaging Protocol (AEIMP). In most cases, you should use Apple events rather than define your own high-level events if you wish to communicate with other applications. If you plan to use Apple events, see *Inside Macintosh: Interapplication Communication* for specific information on Apple events, and refer to this section for specific details about how the Event Manager reports high-level events.

To receive high-level events, you must set the appropriate flags in your application's 'SIZE' resource. You must set the isHighLevelEventAware flag if your application is to receive any high-level events. You must set the localAndRemoteHLEvents flag for your application to receive high-level events sent from another computer on the network. In addition, to receive high-level events from another computer, your application must be shared and Program Linking must be enabled. The user shares your application by selecting your application in the Finder and choosing Sharing from the File menu and enables Program Linking from the Sharing Setup control panel.

If you set the isHighLevelEventAware flag in your application's 'SIZE' resource, your application receives the Finder information in the form of Apple events. The Finder information is the information your application can use to determine which files to open or print. Your application must respond to the required Apple events (Open Application, Open Documents, Print Documents, and Quit Application) that are sent by the Finder if your application sends or receives high-level events.

The what field in the event record of a high-level event contains the kHighLevelEvent constant.

To determine the type of high-level event received, your application needs to examine the message and where fields of the event record. For high-level events, these two fields of the event record have special meanings.

The message field and the where field of the event record together define the specific type of high-level event received. Your application should interpret these fields as having the data type OSType, not LongInt or Point.

The message field contains the event class of the high-level event. For example, Apple events sent by the Edition Manager have the event class 'sect'. You can define your own group of events that are specific to your application. If you have registered your application signature with Apple Computer, Inc., then you can use your signature to define the class of events that belong to your application. Note, however, that Apple reserves the use of all event classes whose names contain only lowercase letters and nonalphabetic characters.

For high-level events, the where field in the event record contains a second message specifier, called the *event ID*. The event ID defines the particular type of event (or message) within the class of events defined by the event class. For example, the Section Read event sent by the Edition Manager has event class 'sect' and event ID 'read'. The Open Documents event sent by the Finder has event class 'aevt' and event ID 'odoc'. You can define your own set of event IDs corresponding to your own event class. For example, if the message field contains 'biff' and the where field contains 'cmd1', then the high-level event indicates the type of event defined by 'cmd1' within the class of events defined by the application with the signature 'biff'.

Note

If your application supports Apple events, you can call the AEProcessAppleEvent function to determine the type of Apple event received, rather than examining the message and where fields. ◆

Note that because the `where` field of an event record for a high-level event is used to select a specific kind of event (within the class determined by the `message` field), high-level event records do not contain the mouse location at the time of the event. You should not interpret the `where` field before interpreting the `what` field because different event classes can contain overlapping sets of event IDs.

Unlike low-level events and operating-system events, high-level events may not be completely determined by the event record returned to your application when it calls `WaitNextEvent`. For example, you might still need to know which other application sent you the high-level event or what additional data that application wants to send you. Your application can obtain this further information about the high-level event by calling the `AcceptHighLevelEvent` function. The additional information associated with a high-level event includes

- the identity of the sender of the event
- a unique number that identifies the request associated with the event or associates the particular event with a request from a previous event
- the address and length of a data buffer that can contain optional data

To obtain this additional information, your application must call `AcceptHighLevelEvent` before calling `WaitNextEvent` again. By convention, calling `AcceptHighLevelEvent` indicates that your application intends to process the high-level event.

To accept an Apple event, call the `AEProcessAppleEvent` function instead of the `AcceptHighLevelEvent` function. The Apple Event Manager also extracts any additional information associated with the Apple event at your application's request. This chapter discusses how to accept high-level events using the `AcceptHighLevelEvent` function; for information on the `AEProcessAppleEvent` function, see *Inside Macintosh: Interapplication Communication*.

Responding to Events From Other Applications

You can identify high-level events by the value in the `what` field of the event record. The `message` and `where` fields further classify the type of high-level event. Your application can choose to recognize as many events as are appropriate. Some high-level events may be fully specified by their event record only, while others may include additional information in an optional buffer. To get that additional information or to find the sender of the event, use the `AcceptHighLevelEvent` function.

Note

To respond to an Apple event, use the Apple Event Manager, as described in *Inside Macintosh: Interapplication Communication*. ◆

Listing 2-16 on the next page illustrates how to respond to a high-level event.

The `DoHighLevelEvent` procedure in Listing 2-16 first determines the type of high-level event received by checking the `message` and `where` fields of the event record. It then uses `AcceptHighLevelEvent` to get any additional data associated with the event. This particular application recognizes only one type of high-level event. If the event is not of this type, the code assumes that the event is an Apple event and calls `AEProcessAppleEvent` to handle the event.

In general, you cannot know in advance how big the optional data buffer is, so you can allocate a zero-length buffer and then resize it if the call to `AcceptHighLevelEvent` returns the `bufferIsSmall` result code.

Listing 2-16 Accepting a high-level event

```
PROCEDURE DoHighLevelEvent (event: EventRecord);
VAR
   myTarg:     TargetID;      {target ID record}
   myRefCon:   LongInt;
   myBuff:     Ptr;
   myLen:      LongInt;
   myErr:      OSErr;
BEGIN
   IF (event.message = LongInt(kMySpecialHLEventClass)) AND
      (LongInt(event.where) = LongInt(kMySpecialHLEventID)) THEN
   BEGIN
      {it's a high-level event that doesn't use AEIMP}
      myLen := 0;                 {start with a 0-byte buffer}
      myBuff := NIL;
      myErr:=AcceptHighLevelEvent(myTarg,myRefCon, myBuff, myLen);
      IF myErr = bufferIsSmall THEN
      BEGIN
         myBuff := NewPtr(myLen);{allocate needed storage}
         myErr := AcceptHighLevelEvent(myTarg, myRefCon, myBuff,
                                       myLen);
         IF myErr = noErr THEN
            ;  {perform any action requested by the event}
      END;
      IF myErr <> noErr THEN
         DoError(myErr);{perform the necessary error handling}
   END
   ELSE
   BEGIN {otherwise, assume that the event is an Apple event}
      myErr := AEProcessAppleEvent(event);
      IF myErr <> noErr THEN
         DoError(myErr);{perform the necessary error handling}
   END;
END;
```

The `AcceptHighLevelEvent` function returns additional information and data associated with the event. The ID of the sender of the event is returned in the first parameter, which is a target ID record. You can inspect the fields of that record to determine which application sent the event. The target ID record contains the session

reference number that identifies the connection with the other application as well as the port name and location name of the sender. If the high-level event requires that you return information, you can use the information returned in the target ID record to send an event back to the requesting application. See "Determining the Sender of a High-Level Event" on page 2-72 and "Sending High-Level Events" on page 2-73 for specific information on the target ID record.

The second parameter to `AcceptHighLevelEvent`, the reference constant parameter, is a unique number that identifies the request associated with the event or identifies that the particular event is related to a request from a previous event. If you send a response to this event, you should use the same value for the reference constant so that the sender of the event can associate the reply with the original request.

The third parameter points to any additional data associated with the event. Any data in this additional buffer is defined by the particular high-level event. On input, the fourth parameter to `AcceptHighLevelEvent`, the length parameter, contains the size of the buffer. If no error occurs, on output the length parameter contains the size of the message accepted. If the `AcceptHighLevelEvent` function returns the result code `bufferIsSmall`, the length parameter contains the size of the message yet to be received.

Searching for a Specific High-Level Event

Sometimes you do not want to accept the next available high-level event pending for your application. Instead, you might want to select one event from among all the high-level events in your application's high-level event queue. For example, you might want to look for a return receipt for a high-level event you previously posted before processing other high-level events.

You can select a specific high-level event by calling the `GetSpecificHighLevelEvent` function. One of the parameters you pass to this function is a filter function that you provide. Your filter function should examine an event in your application's high-level event queue and determine whether it is the kind of event you wish to receive. If it is, your filter function returns `TRUE`. This indicates that your filter function does not want to inspect any more events. If the filter function finds an event of the desired type, it should call `AcceptHighLevelEvent` to retrieve the event. When your function returns `TRUE`, the `GetSpecificHighLevelEvent` function itself returns `TRUE`.

If your filter function returns `FALSE` for an event in the high-level event queue, then `GetSpecificHighLevelEvent` looks at the next event in the high-level event queue and executes your filter function. If the filter function returns `FALSE` for all the high-level events in the queue, then `GetSpecificHighLevelEvent` itself returns `FALSE` to your application.

Here's how you declare the filter function whose address you pass to the `GetSpecificHighLevelEvent` function:

```
FUNCTION MyFilter (yourDataPtr: Ptr;
                   msgBuff: HighLevelEventMsgPtr;
                   sender: TargetID): Boolean;
```

When your application calls `GetSpecificHighLevelEvent`, you pass it a parameter that indicates the criteria your filter function should use to search for a specific event. The `GetSpecificHighLevelEvent` function passes this information to your filter function in the `yourDataPtr` parameter. The `GetSpecificHighLevelEvent` function also provides your filter function with information about the event record of the high-level event in the `msgBuff` parameter as well as information about the sender of the high-level event in the `sender` parameter.

The `msgBuff` parameter contains a pointer to a high-level event message record that has this structure:

```
TYPE  HighLevelEventMsg =
      RECORD
         HighLevelEventMsgHeaderLength:      Integer;
         version:                            Integer;
         reserved1:                          LongInt;
         theMsgEvent:                        EventRecord;
         userRefCon:                         LongInt;
         postingOptions:                     LongInt;
         msgLength:                          LongInt;
      END;

      HighLevelEventMsgPtr= ^HighLevelEventMsg;
```

When you call `GetSpecificHighLevelEvent` and it executes your filter function for a high-level event waiting in the high-level event queue, the fields of the high-level event message record are filled in by the Event Manager. You can then compare the fields of this record to the information in the `yourDataPtr` parameter to determine whether that event suits your needs. For example, the `yourDataPtr` parameter might contain the signature of a return receipt. You can test its value against the event class of the event record contained in the `theMsgEvent` field of the high-level event message record.

Determining the Sender of a High-Level Event

When you receive a high-level event, part of the information returned by `AcceptHighLevelEvent` is the identity of the sender of the event. You can use that information to respond selectively to requests made by other applications or to find which application to send any replies to. The information about the sender is provided in the form of a target ID record, defined as follows:

```
TYPE  TargetID =
      RECORD
         sessionID:  LongInt;          {session reference number}
         name:       PPCPortRec;       {sender's port name}
         location:   LocationNameRec;  {sender's location name}
         recvrName:  PPCPortRec;       {reserved}
      END;
```

The sessionID field corresponds to the session reference number created by the PPC Toolbox. This is a 32-bit number that uniquely identifies a PPC Toolbox session (or connection) with another application. The name and location fields contain the sender's port name and location name. If the sending application is on the same computer as the receiving application, you can determine the sending application's process serial number by calling the GetProcessSerialNumberFromPortName function.

Sending High-Level Events

You use the PostHighLevelEvent function to send a high-level event to another application. When doing so, you need to provide six pieces of information:

- an event record with the event class and event ID assigned appropriately
- the identity of the recipient of the event
- a unique number that identifies the communication associated with this particular event
- a data buffer that can contain optional data
- the length of the data buffer
- options determining how the event is posted

Note

To send an Apple event, use the Apple Event Manager function AESend. The Apple Event Manager uses the Event Manager to post Apple events. For information on posting Apple events, see *Inside Macintosh: Interapplication Communication*. ◆

When you post a high-level event to an application on the same computer, you can specify its recipient in one of four ways:

- by port name and location name (specified in a target ID record)
- by a session reference number
- by the application's creator signature
- by a process serial number

To specify the recipient of a high-level event sent across a network, you can use only the receiving application's port name and location name or its session reference number. You can use any of the four ways when sending high-level events to applications on the local computer.

You specify the recipient of a high-level event in the receiverID parameter when you use the PostHighLevelEvent function. To specify a port name and location name, provide the address of a target ID record in the receiverID parameter. To specify a process serial number, provide its address in the receiverID parameter. To specify a session reference number, or signature, provide the data in the receiverID parameter.

When you are replying to a high-level event, it is easy to identify the recipient because you can use the target ID record that you receive from AcceptHighLevelEvent, the

session reference number contained in that target ID record, or the process serial number (if the receiving process is local). Note that replying by session reference number is always the fastest way to respond to a high-level event.

When you are not replying to a previous event, you need to determine the identity of the target application yourself. You can use one of several methods to do this. If the target application is on the local computer, you can search for that application's creator signature or its process serial number by calling the `GetProcessInformation` function. See the chapter "Process Manager" in *Inside Macintosh: Processes* for a detailed explanation of the `GetProcessInformation` function and for examples of how to use it to generate a list of process serial numbers of all open processes on the local computer.

If the application to which you want to send a high-level event is located on a remote computer, you need to identify it either by its session reference number or by its port name and location name. You can call the `PPCBrowser` function to let the user browse for a specific port. You can call the `IPCListPorts` function to obtain a list of all ports registered with the target PPC Toolbox. See the chapter "Program-to-Program Communications Toolbox" in *Inside Macintosh: Interapplication Communication* for an explanation of both of these functions.

As just described, you can identify the recipient of the high-level event in one of four ways. Listing 2-17 illustrates how to send a high-level event to an application on the local computer using the application's creator signature. In this example, an application is sending a high-level event to the application with the creator signature of `'boff'`. The specific high-level event being sent is identified by the event class `'boff'` and the event ID `'cmd1'`.

Listing 2-17 Posting a high-level event by application signature

```
PROCEDURE MyPostTest;
VAR
   myEvent:    EventRecord;   {an event record}
   myRecvID:   OSType;        {receiver ID}
   myOpts:     LongInt;       {posting options}
   myErr:      OSErr;
BEGIN
   myEvent.what := kHighLevelEvent;
   myEvent.message := LongInt('boff');       {event class}
   myEvent.where := Point(LongInt('cmd1'));  {event ID}
   {the receiver is identified by its signature and }
   { a return receipt is requested}
   myOpts := receiverIDisSignature + nReturnReceipt;
   myRecvID := 'boff';                     {receiver's signature}
   myErr := PostHighLevelEvent(myEvent, Ptr(myRecvID), 0, NIL, 0,
                               myOpts);
   IF myErr <> noErr THEN
      DoError(myErr);
END;
```

In this example of using the PostHighLevelEvent function, there is no additional data to transmit, so the sending application provides NIL as the pointer to the data buffer and sets the buffer length to 0. The myOpts variable specifies posting options.

Posting options are of two types: delivery options and options associated with the receiverID parameter. You can specify one or more delivery options to indicate if you want the other application to receive the event at the next opportunity and to indicate if you want acknowledgment that the other application received the event. You use the options associated with the receiverID parameter to indicate how you are specifying the recipient of the event. To set the various posting options, use these constants:

```
CONST nAttnMsg             = $00000001;{give this message priority}
      nReturnReceipt       = $00000200;{return receipt requested}
      receiverIDisTargetID = $00005000;{ID is port name and location name}
      receiverIDisSessionID = $00006000;{ID is PPC session ref number}
      receiverIDisSignature = $00007000;{ID is creator signature}
      receiverIDisPSN      = $00008000;{ID is process serial number}
```

When you specify the receiving application in the receiverID parameter, you can use these constants to specify the receiver of the event by port name and location name, session reference number, process serial number, or signature. Any of these specifications allows you to send an event to another application on the local computer. For example, in Listing 2-17 the myOpts variable indicates that the receiver is identified by its creator signature, and the myRecvID variable contains the receiver's creator signature. To send events to an application on a remote computer, you can specify the recipient only by the session reference number or by the port name and location name.

When you specify the receiver of the event by port name and location name, use the receiverIDisTargetID constant in the posting options parameter and specify the address of a target ID record in the receiverID parameter.

```
TYPE  TargetID =
      RECORD
         sessionID:  LongInt;         {unused for posting}
         name:       PPCPortRec;      {recipient's port name}
         location:   LocationNameRec; {recipient's port loc}
         recvrName:  PPCPortRec;      {unused for posting}
      END;
```

When you pass a target ID record, you need to specify only the name and location fields. You can use the IPCListPorts function to list all of the existing port names along with information on whether the port will accept authenticated service on the computer specified by the location name. For information on how to use the IPCListPorts function, see the chapter "Program-to-Program Communications Toolbox" in *Inside Macintosh: Interapplication Communication*.

You can also use the PPCBrowser function to fill in a target ID record. Listing 2-18 on the next page illustrates how to use the PPCBrowser function to post a high-level event. In this example, the sending application wants to locate a dictionary application and have the dictionary return the definition of a word to it.

Listing 2-18 Using the `PPCBrowser` function to post a high-level event

```
FUNCTION MyPostWithPPCBrowser (aTextPtr: Ptr; textlength: LongInt): OSErr;
VAR
   myHLEvent:     EventRecord;
   myErr:         OSErr;
   myNumTries:    Integer;
   myPortInfo:    PortInfoRec;
   myTarget:      TargetID;
BEGIN
   {use PPCBrowser to get the target}
   myErr := PPCBrowser('Select an Application', 'Application', FALSE,
                       myTarget.location, myPortInfo, NIL, '');
   IF myErr = NoErr THEN
   BEGIN
      {copy port name into myTarget.name}
      myTarget.name := myPortInfo.name;

      myHLEvent.what := kHighLevelEvent;
      myHLEvent.message := LongInt('Dict');
      myHLEvent.where := Point(LongInt('Defn'));

      {if a connection is broken, then sessClosedErr is returned to }
      { PostHighLevelEvent; to reestablish the connection, just post }
      { the event one more time}
      myNumTries := 0;
      REPEAT
         myErr := PostHighLevelEvent(myHLEvent, @myTarget, 0, aTextPtr,
                                     textlength, receiverIDisTargetID);
         myNumTries := myNumTries + 1;
      UNTIL (myErr <> sessClosedErr) OR (myNumTries > 1);
   END;
   MyPostWithPPCBrowser := myErr;   {return any error}
END;
```

The application-defined function in Listing 2-18 uses the `PPCBrowser` function to display a dialog box asking the user to select a dictionary. (For additional information on the `PPCBrowser` function, see *Inside Macintosh: Interapplication Communication*.) If the user selects a dictionary, this code posts a high-level event to that dictionary application asking for the definition of the selected text. Note that the sending application and the receiving application must both agree that definition queries are to be of event class '`Dict`' and event ID '`Defn`'. It is necessary to define a private protocol only in cases in which no suitable Apple event exists.

Note
You should avoid passing handles to the receiving application in an attempt to share a block of data. It is better to put the relevant data into a buffer (as illustrated in Listing 2-18) and pass the address of the buffer. If you absolutely must share data by passing a handle, make sure that the block of data is located in the system heap. ◆

If a high-level event is posted successfully, `PostHighLevelEvent` returns the result code `noErr`, which indicates only that the event was successfully passed to the PPC Toolbox. Your application needs to call another Event Manager routine (`EventAvail`, `GetNextEvent`, or `WaitNextEvent`) to give the other application an opportunity to receive the event.

The event you send might require the other application to return some information to your application by sending a high-level event back to your application. You can scan for the response by using `GetSpecificHighLevelEvent`. If your application must wait for this event, you might want to display a wristwatch cursor or take other action as appropriate to your application. You also might want to implement a timeout mechanism in case your application never receives a response to the event.

Requesting Return Receipts

When you post a high-level event, you can request a return receipt by including the `nReturnReceipt` constant as one of the posting options. This requests that the Event Manager send your application a high-level event that tells you whether the other application accepted your event. Note that this does not necessarily mean that the other application performed any action you might have requested from it.

A **return receipt** is a high-level event having an event class and an event ID indicated by these two constants:

```
CONST HighLevelEventMsgClass  = 'jaym';
      rtrnReceiptMsgID        = 'rtrn';
```

Return receipts are posted by the Event Manager on the computer of the receiving application (and not by the receiving application itself). No data buffer is associated with a return receipt. However, the posting Event Manager sets the `modifiers` field of the high-level event record to one of the following values:

```
CONST msgWasNotAccepted       = 0;
      msgWasFullyAccepted     = 1;
      msgWasPartiallyAccepted = 2;
```

The `msgWasNotAccepted` constant indicates that your event was not accepted by the receiving application. This means that the receiving application was notified of the arrival of your event (through `WaitNextEvent`) but did not call `AcceptHighLevelEvent` to accept the event. The `msgWasFullyAccepted` constant indicates that the receiving application did call `AcceptHighLevelEvent` and retrieved all the data in the optional data buffer. The `msgWasPartiallyAccepted` constant

indicates that the receiving application called `AcceptHighLevelEvent`, but the application's data buffer was too small to hold the data sent with your application, and the receiving application called `WaitNextEvent` before retrieving the rest of the buffer.

Note that a return receipt does not indicate the identity of the receiving application. To determine on whose behalf the Event Manager has sent you a particular return receipt, you need to call `AcceptHighLevelEvent`. When `AcceptHighLevelEvent` returns successfully, the `sender` parameter contains a target ID record with the fields filled in for the receiving application. With return receipts, the `msgLen` parameter is 0, the `msgBuff` parameter is `NIL`, and the `msgRefCon` parameter contains the unique number of the `refCon` parameter of the original high-level event sender (that is, your application).

Handling Apple Events

If your application uses high-level events, your application must respond to the required Apple events sent by the Finder. The four required Apple events are Open Application, Open Documents, Print Documents, and Quit Application. See *Inside Macintosh: Interapplication Communication* for information on how to handle the required Apple events.

When your application receives a high-level event (as indicated by the `kHighLevelEvent` constant in the `what` field of the event record), and if your application supports Apple events, call the `AEProcessAppleEvent` function. The `AEProcessAppleEvent` function provides an easy way for your application to identify the event class and event ID of the Apple event and to direct the Apple Event Manager to call the code in your program that handles the Apple event.

To send Apple events to other applications, use the `AESend` function.

To ensure compatibility and smooth interaction with other Macintosh applications, you should use the Apple event protocol for high-level events whenever possible. By implementing the capabilities to send Apple events to and receive Apple events from other applications, you allow other applications to interact with your application and provide enhanced capabilities to your users.

See *Inside Macintosh: Interapplication Communication* for complete information on how to send and receive Apple events.

Event Manager Reference

This section describes the data structures and routines for the Event Manager and Operating System Event Manager. It also describes the `'SIZE'` resource.

Data Structures

This section describes the event record, target ID record, high-level event message record, and structure of the Operating System event queue. The Event Manager uses event records to return information about events. You can use a target ID record to specify or identify the address of another application or process with which your application is communicating. If your application supplies a filter function as a parameter to the `GetSpecificHighLevelEvent` function, your filter function receives information about high-level events in a high-level event message record.

The Event Record

When your application uses an Event Manager routine to retrieve an event, the Event Manager returns information about the retrieved event in an event record. The `EventRecord` data type defines the event record.

```
TYPE  EventRecord =
      RECORD
         what:        Integer;       {event code}
         message:     LongInt;       {event message}
         when:        LongInt;       {ticks since startup}
         where:       Point;         {mouse location}
         modifiers:   Integer;       {modifier flags}
      END;
```

Field descriptions

`what` The `what` field indicates the type of event received. The type of event can be identified by these constants:

```
CONST
nullEvent         = 0; {no other pending events}
mouseDown         = 1; {mouse button pressed}
mouseUp           = 2; {mouse button released}
keyDown           = 3; {key pressed}
keyUp             = 4; {key released}
autoKey           = 5; {key repeatedly held down}
updateEvt         = 6; {window needs updating}
diskEvt           = 7; {disk inserted}
activateEvt       = 8; {activate/deactivate window}
osEvt             = 15;{operating-system event-- }
                       { resume, suspend, or }
                       { mouse-moved}
kHighLevelEvent   = 23;{high-level event}
```

Note that in System 7, event types with the values 9 through 14 are undefined and reserved for future use by Apple. All other values for the `what` field are also reserved for use by Apple.

`message` Additional information associated with the event. The interpretation of this information depends on the event type. The contents of the `message` field for each event type are summarized here:

Event type	Event message
null, mouse-up, mouse-down	Undefined.
key-up, key-down, auto-key	Character code and virtual key code in low-order word. For Apple Desktop Bus (ADB) keyboards, the low byte of the high-order word contains the ADB address of the keyboard where the keyboard event occurred. The high byte of the high-order word is reserved.
update, activate	Pointer to the window to update, activate, or deactivate.
disk-inserted	Drive number in low-order word, File Manager result code in high-order word.
resume	The `suspendResumeMessage` constant in bits 24–31 and a 1 in bit 0 to indicate the event is a resume event. Bit 1 contains either a 1 or a 0 to indicate if Clipboard conversion is required, and bits 2–23 are reserved.
suspend	The `suspendResumeMessage` constant in bits 24–31 and a 0 in bit 0 to indicate the event is a suspend event. Bit 1 is undefined, and bits 2–23 are reserved.
mouse-moved	The `mouseMovedMessage` constant in bits 24–31. Bits 2–23 are reserved, and bit 0 and bit 1 are undefined.
high-level	Class of events to which the high-level event belongs. The `message` and `where` fields of a high-level event define the specific type of high-level event received.

`when` The `when` field indicates the time when the event was posted (in ticks since system startup).

`where` For low-level events and operating-system events, the `where` field contains the location of the cursor at the time the event was posted (in global coordinates).

For high-level events, the `where` field contains a second event specifier, the event ID. The event ID defines the particular type of event within the class of events defined by the `message` field of the high-level event. For high-level events, you should interpret the `where` field as having the data type `OSType`, not `Point`.

`modifiers` The `modifiers` field contains information about the state of the modifier keys and the mouse button at the time the event was posted. For activate events, this field also indicates whether the

window should be activated or deactivated. In System 7 it also indicates whether the mouse-down event caused your application to switch to the foreground.

Each of the modifier keys is represented by a specific bit in the `modifiers` field of the event record. Figure 2-5, on page 2-20, shows how to interpret the `modifiers` field. The modifier keys include the Option, Command, Caps Lock, Control, and Shift keys. If your application attaches special meaning to any of these keys in combination with other keys or when the mouse button is down, you can test the state of the `modifiers` field to determine the action your application should take. For example, you can use this information to determine whether the user pressed the Command key and another key to make a menu choice.

The Target ID Record

When you send a high-level event to another application, you can use the target ID record to specify the recipient of the event. When you receive a high-level event, the `AcceptHighLevelEvent` function uses a target ID record to return information about the sender of the event.

The `TargetID` data type defines the target ID record.

```
TYPE  TargetID =
      RECORD
         sessionID:  LongInt;          {session reference number}
         name:       PPCPortRec;       {port name}
         location:   LocationNameRec;  {location name}
         recvrName:  PPCPortRec;       {reserved}
      END;
```

Field descriptions

`sessionID` — For high-level events that your application receives, this field contains the session reference number created by the PPC Toolbox. This is a 32-bit number that uniquely identifies a PPC Toolbox session (or connection) with another application. This field is not used by your application when sending a high-level event to another process. (To send a high-level event that specifies the recipient by session reference number, provide a pointer to a session reference number in the `receiverID` parameter and use the `receiverIDisSessionID` constant in the `postingOptions` parameter to `PostHighLevelEvent`.)

`name` — For high-level events that your application receives, this field contains a PPC port record that specifies the port name of the process from which the high-level event originated. When sending a high-level event to a process on a local or remote computer, you can specify the port name of the recipient process in a PPC port record that you provide in this field.

If the sending application is on the same computer as the receiving application, you can determine the sending application's process serial number by calling the `GetProcessSerialNumberFromPortName` function.

`location` — For high-level events that your application receives, this field contains a location name record that identifies the location name of the process from which the high-level event originated. When sending a high-level event to a process on a local or remote computer, you can specify the location name of the recipient process in a location name record that you provide in this field.

`recvrName` — This field is reserved.

The High-Level Event Message Record

You can search your application's high-level event queue for a specific high-level event by using the `GetSpecificHighLevelEvent` function and providing a filter function. Your filter function receives a pointer to a high-level event message record that contains information about a high-level event. (See "Filter Function for Searching the High-Level Event Queue" on page 2-114 for information on how to define a filter function.)

The `HighLevelEventMsg` data type defines the structure of a high-level event message record.

```
TYPE  HighLevelEventMsg =
      RECORD
         HighLevelEventMsgHeaderLength:      Integer;
         version:                            Integer;
         reserved1:                          LongInt;
         theMsgEvent:                        EventRecord;
         userRefCon:                         LongInt;
         postingOptions:                     LongInt;
         msgLength:                          LongInt;
      END;
```

Field descriptions

`HighLevelEventMsgHeaderLength` — Reserved for use by the Event Manager.

`version` — Reserved for use by the Event Manager.

`reserved1` — Reserved for use by the Event Manager.

`theMsgEvent` — The event record of a high-level event. Your filter function can compare the fields of this event record to determine whether the high-level event is the desired event. If your filter function finds the desired event, it should call `AcceptHighLevelEvent` to accept the event and remove the event from the high-level event queue, and return `TRUE` as its function result.

userRefCon	A unique number that identifies the communication associated with this event.
postingOptions	Reserved for use by the Event Manager.
msgLength	Reserved for use by the Event Manager.

The Event Queue

The event queue is a standard Macintosh Operating System queue that the Operating System Event Manager maintains. Only mouse-up, mouse-down, key-up, key-down, auto-key, and disk-inserted events are stored in the Operating System event queue. In most cases, your application should not access the event queue directly. Instead you usually use the `WaitNextEvent` function, which can retrieve events from this queue as well as from other sources.

The event queue consists of a header followed by the actual entries in the queue. The event queue has the same header as all standard Macintosh Operating System queues. The `Qhdr` data type defines the queue header.

```
TYPE  QHdr =
      RECORD
         qFlags:  Integer;    {queue flags}
         qHead:   QElemPtr;   {first queue entry}
         qTail:   QElemPtr;   {last queue entry}
      END;
```

The `EvQEl` data type defines an entry in the Operating System event queue.

```
TYPE  EvQEl =
      RECORD
         qLink:          QElemPtr;   {next queue entry}
         qType:          Integer;    {queue type (ORD(evType))}
         evtQWhat:       Integer;    {event code}
         evtQMessage:    LongInt;    {event message}
         evtQWhen:       LongInt;    {ticks since startup}
         evtQWhere:      Point;      {mouse location}
         evtQModifiers:  Integer;    {modifier flags}
      END;
```

Each entry in the event queue begins with 4 bytes of flags followed by a pointer to the next queue entry. The flags are maintained by and internal to the Operating System Event Manager. The queue entries are linked by pointers, and the first field of the `EvQEl` data type, which represents the structure of a queue entry, begins with a pointer to the next queue entry. Thus you cannot directly access the flags using the `EvQEl` data type.

Event Manager Routines

The Event Manager includes routines for receiving events, receiving and sending high-level events, and searching for specific high-level events. The Event Manager also provides routines for converting between process serial numbers and port names, getting information about the state of the mouse button, reading the keyboard, and getting timing information.

Receiving Events

You can use the `WaitNextEvent` or `GetNextEvent` function to retrieve an event from the Event Manager and remove the event from the event stream. To provide greater support for multitasking, however, you should use the `WaitNextEvent` function instead of `GetNextEvent` whenever possible. You can use the `EventAvail` function to look at an event without removing it from the event stream. You can use the `AcceptHighLevelEvent` function to get additional information associated with a high-level event and `GetSpecificHighLevelEvent` to search for a specific high-level event.

The `FlushEvents` procedure removes all low-level events from the Operating System event queue. In general, your application should not empty the event queue.

You can use the `SystemClick` procedure to route events to desk accessories when necessary. The `SystemTask` and `SystemEvent` routines are used by the Event Manager, and your application usually does not need to call these two routines.

You usually use the functions provided by the Toolbox Event Manager to retrieve events from the event stream. Even if you are interested only in the events stored in the Operating System event queue, you can retrieve these events using the Toolbox Event Manager by setting the event mask to mask out all events except keyboard, mouse, and disk-inserted events. However, you can choose to use Operating System Event Manager routines to perform this task.

The Operating System Event Manager provides two functions, `GetOSEvent` and `OSEventAvail`, to retrieve events from the Operating System event queue. In most cases, your application will not need to use these two functions.

If your application needs to receive key-up events, you can change the system event mask of your application using the `SetEventMask` procedure. The `GetEvQHdr` function returns a pointer to the header of the Operating System event queue.

WaitNextEvent

You can use the `WaitNextEvent` function to retrieve events one at a time from the Event Manager.

```
FUNCTION WaitNextEvent (eventMask: Integer;
                        VAR theEvent: EventRecord; sleep: LongInt;
                        mouseRgn: RgnHandle): Boolean;
```

`eventMask` A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. You can use these constants to specify the event mask:

```
CONST
mDownMask     =  2;      {mouse-down event    (bit 1)}
mUpMask       =  4;      {mouse-up event      (bit 2)}
keyDownMask   =  8;      {key-down event      (bit 3)}
keyUpMask     =  16;     {key-up event        (bit 4)}
autoKeyMask   =  32;     {auto-key event      (bit 5)}
updateMask    =  64;     {update event        (bit 6)}
diskMask      =  128;    {disk-inserted event (bit 7)}
activMask     =  256;    {activate event      (bit 8)}
highLevelEventMask
              =  1024;   {high-level event    (bit 10)}
osMask        =  -32768; {operating-system    (bit 15)}
```

To accept all events, you can specify the `everyEvent` constant as the event mask:

```
CONST
everyEvent    =  -1;     {every event}
```

If no event of any of the designated types is available, `WaitNextEvent` returns a null event. `WaitNextEvent` determines the next available event to return based on the `eventMask` parameter and the priority of the event.

Events not designated by the event mask remain in the event stream until retrieved by an application. Low-level events in the Operating System event queue are kept in the queue until they are retrieved by your application or another application or until the queue becomes full. Once the queue becomes full, the Operating System Event Manager begins discarding the oldest events in the queue.

`theEvent` The next available event of the specified type or types. The `WaitNextEvent` function removes the returned event from the event stream and returns the information about the event in an event record. The event record includes the type of event received and other information. See "The Event Record," beginning on page 2-79, for a description of the fields in the event record.

In addition to the event record, high-level events can contain additional data; you use the AcceptHighLevelEvent or AEProcessAppleEvent functions to get additional data associated with these events.

sleep — The number of ticks (a tick is approximately 1/60 of a second) indicating the amount of time your application is willing to relinquish the processor if no events (other than null events) are pending for your application. If you specify a value greater than 0 for the sleep parameter, your application relinquishes the processor for the specified time or until an event occurs.

You should usually specify a value greater than 0 for the sleep parameter to allow background processes to receive processing time. You should not set the sleep parameter to a value greater than the number of ticks returned by GetCaretTime if your application provides text-editing capabilities. When the specified time expires, and if there are no pending events for your application, WaitNextEvent returns a null event in the parameter theEvent.

mouseRgn — A handle to a region that specifies a region inside of which mouse movement does not cause mouse-moved events. In other words, your application receives mouse-moved events only when the cursor is outside the specified region. You should specify the region in global coordinates. If you pass an empty region or a NIL region handle, the Event Manager does not report mouse-moved events to your application. Note that your application should recalculate the mouseRgn parameter when it receives a mouse-moved event, or it will continue to receive mouse-moved events as long as the cursor position is outside the original mouseRgn.

DESCRIPTION

The WaitNextEvent function returns FALSE as its function result if the event being returned is a null event or if WaitNextEvent has intercepted the event; otherwise, WaitNextEvent returns TRUE. The WaitNextEvent function calls the Operating System Event Manager function SystemEvent to determine whether the event should be handled by the application or the Operating System.

If no events are pending for your application, WaitNextEvent waits for a specified amount of time for an event. (During this time, processing time may be allocated to background processes.) If an event occurs, it is returned as the value of the parameter theEvent, and WaitNextEvent returns a function result of TRUE. If the specified time expires and there are no pending events for your application, WaitNextEvent returns a null event in theEvent and a function result of FALSE.

Before returning an event to your application, WaitNextEvent performs other processing and may intercept the event.

The WaitNextEvent function intercepts Command–Shift–number key sequences and calls the corresponding 'FKEY' resource to perform the associated action. The Event Manager's processing of Command–Shift–number key sequences with numbers 3 through 9 can be disabled by setting the ScrDmpEnable global variable (a byte) to 0.

The WaitNextEvent function also makes the alarm go off if the alarm is set and the current time is the alarm time. The user sets the alarm using the Alarm Clock desk accessory.

The `WaitNextEvent` function also calls the `SystemTask` procedure, which gives time to each open desk accessory or device driver to perform any periodic action defined for it. A desk accessory or device driver specifies how often the periodic action should occur, and `SystemTask` gives time to the desk accessory or device driver at the appropriate interval.

Some high-level events may be fully specified by their event records only, while others may include additional information in an optional buffer. To get any additional information and to find the sender of the event, use the `AcceptHighLevelEvent` function.

If the returned event is a high-level event and your application supports Apple events, use the Apple Event Manager function `AEProcessAppleEvent` to respond to the Apple event and to get additional information associated with the Apple event.

SPECIAL CONSIDERATIONS

In System 7, if your application is in the foreground and the user opens a desk accessory or other item from the Apple menu, clicks in the window belonging to another application or desk accessory, or chooses another process from the Application menu, the next event reported to your application by the `WaitNextEvent` function is a suspend event. After your application is switched out, the Event Manager directs events (other than events your application can receive in the background) to the newly activated process until the user switches back to your application or another application.

Note

In a single-application environment in System 6, and in a multiple-application environment in which the desk accessory is launched in the application's partition (for example, a desk accessory opened by the user from the Apple menu while holding down the Option key), the Event Manager handles events for desk accessories in a slightly different manner.

In these environments, when mouse-up, activate, update, and keyboard events (including keyboard equivalents of menu commands) occur, the Event Manager checks to see whether the active window belongs to a desk accessory and whether the desk accessory can handle the event. If so, it sends the event to the desk accessory and `WaitNextEvent` returns `FALSE` to your application. Also note that in these environments, the Event Manager returns `TRUE` for mouse-down events, regardless of whether the mouse-down event is for a desk accessory or not. For mouse-down events in these situations, if the mouse button was pressed while the cursor was in a desk accessory window (as indicated by the `inSystem` constant returned by the `FindWindow` function), your application should call the `SystemClick` procedure. The `SystemClick` procedure handles mouse-down events as appropriate for desk accessories, including sending your application an activate event to deactivate its front window if the desk accessory window needs to be activated. ◆

SEE ALSO

For examples that use the `WaitNextEvent` function, see Listing 2-1 on page 2-23 and Listing 2-2 on page 2-24.

To get information about the sender of a high-level event and to retrieve any additional data associated with the high-level event, see the description of the `AcceptHighLevelEvent` function on page 2-90. For details on how to process an Apple event, see the description of the `AEProcessAppleEvent` function in *Inside Macintosh: Interapplication Communication*.

For information on how to retrieve an event without removing it from the event stream, see the description of the `EventAvail` function, immediately following.

EventAvail

You can use the `EventAvail` function to retrieve the next available event from the Event Manager without removing the returned event from your application's event stream.

```
FUNCTION EventAvail (eventMask: Integer;
                     VAR theEvent: EventRecord): Boolean;
```

`eventMask` A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. You can use these constants to specify the event mask:

```
CONST
mDownMask     = 2;        {mouse-down event    (bit 1)}
mUpMask       = 4;        {mouse-up event      (bit 2)}
keyDownMask   = 8;        {key-down event      (bit 3)}
keyUpMask     = 16;       {key-up event        (bit 4)}
autoKeyMask   = 32;       {auto-key event      (bit 5)}
updateMask    = 64;       {update event        (bit 6)}
diskMask      = 128;      {disk-inserted event (bit 7)}
activMask     = 256;      {activate event      (bit 8)}
highLevelEventMask
              = 1024;     {high-level event    (bit 10)}
osMask        = -32768;   {operating-system    (bit 15)}
```

To accept all events, you can specify the `everyEvent` constant as the event mask:

```
CONST
everyEvent    =  -1;      {every event}
```

If no event of any of the designated types is available, `EventAvail` returns a null event.

`theEvent` The next available event of the specified type or types. The `EventAvail` function does not remove the returned event from the event stream, but does return the information about the event in an event record. The event record includes the type of event received and other information.

DESCRIPTION

`EventAvail` returns `FALSE` as its function result if the event being returned is a null event; otherwise, `EventAvail` returns `TRUE`.

Like `WaitNextEvent`, the `EventAvail` function calls the `SystemTask` procedure to give time to each open desk accessory or device driver to perform any periodic action defined for it. The `EventAvail` function also makes the alarm go off if the alarm is set and the current time is the alarm time. The user sets the alarm using the Alarm Clock desk accessory.

SPECIAL CONSIDERATIONS

If `EventAvail` returns a low-level event from the Operating System event queue, the event will not be accessible later if, in the meantime, the event queue becomes full and the event is discarded from it; however, this is not a common occurrence.

SEE ALSO

See "The Event Record," beginning on page 2-79, for a description of the fields in the event record.

GetNextEvent

Although you should normally use `WaitNextEvent`, you can also use the `GetNextEvent` function to retrieve events one at a time from the Event Manager.

```
FUNCTION GetNextEvent (eventMask: Integer;
                       VAR theEvent: EventRecord): Boolean;
```

`eventMask` A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants (listed in "Setting the Event Mask" beginning on page 2-26). If no event of any of the designated types is available, `GetNextEvent` returns a null event.

`theEvent` The next available event of the specified type or types. The `GetNextEvent` function removes the returned event from the event stream and returns the information about the event in an event record. The event record includes the type of event received and other information.

DESCRIPTION

GetNextEvent returns FALSE as its function result if the event being returned is a null event or if GetNextEvent has intercepted the event; otherwise, GetNextEvent returns TRUE. The GetNextEvent function calls the Operating System Manager function SystemEvent to determine whether the event should be handled by the application or the Operating System.

Like WaitNextEvent, the GetNextEvent function calls the SystemTask procedure to give time to each open desk accessory or device driver to perform any periodic action defined for it. The GetNextEvent function also makes the alarm go off if the alarm is set and the current time is the alarm time. (The user sets the alarm using the Alarm Clock desk accessory.)

The GetNextEvent function also intercepts Command–Shift–number key sequences and calls the corresponding 'FKEY' resource to perform the associated action. The Event Manager's processing of Command–Shift–number key sequences with numbers 3 through 9 can be disabled by setting the ScrDmpEnable global variable (a byte) to 0.

SPECIAL CONSIDERATIONS

For greater support of the multitasking environment, your application should use WaitNextEvent instead of GetNextEvent whenever possible. If your application does call GetNextEvent, it should also call the SystemTask procedure.

SEE ALSO

See "The Event Record," beginning on page 2-79, for a description of the fields in the event record. For information on the SystemTask procedure, see page 2-95.

AcceptHighLevelEvent

After receiving a high-level event (other than an Apple event), use the AcceptHighLevelEvent function to get any additional information associated with the event.

```
FUNCTION AcceptHighLevelEvent (VAR sender: TargetID;
                               VAR msgRefcon: LongInt;
                               msgBuff: Ptr;
                               VAR msgLen: LongInt): OSErr;
```

sender — Identifies the sender of the event; this information is returned in a target ID record. The sender parameter contains the session reference number that identifies the connection with the other application and the port name and location name of the sender.

msgRefcon	Uniquely identifies the communication associated with this event. If you send a response to this event, you should specify the same value for the msgRefcon parameter so that the sender of the event can associate the reply with the original request.
msgBuff	Specifies where the AcceptHighLevelEvent function should return any additional data associated with the event. Your application is responsible for allocating the memory for the additional data pointed to by the msgBuff parameter and for setting the msgLen parameter to the number of bytes that you have allocated for the data. If the msgBuff parameter points to an area in memory that is not large enough to hold all the data associated with the event, AcceptHighLevelEvent returns as much data as the specified memory area can hold, returns the amount of data remaining in the msgLen parameter, and returns the result code bufferIsSmall.
msgLen	Contains the size of the data (in bytes) pointed to by the msgBuff parameter. If AcceptHighLevelEvent returns the result code bufferIsSmall, the msgLen parameter contains the number of bytes remaining. You can call AcceptHighLevelEvent again to receive the rest of the data.

DESCRIPTION

When your application receives a high-level event, you can use the AcceptHighLevelEvent function to get additional data associated with the event. The AcceptHighLevelEvent function returns information that identifies the sender of the event and the unique message reference constant of the event.

Your application should allocate memory for any additional data associated with the event, then supply a pointer to the data area and also provide the length in bytes of the data area.

SPECIAL CONSIDERATIONS

The AcceptHighLevelEvent function may move or purge memory. You should not call this function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the AcceptHighLevelEvent function are

Trap macro	Selector
_OSDispatch	$0033

RESULT CODES

noErr	0	No error
bufferIsSmall	–607	Buffer is too small
noOutstandingHLE	–608	No outstanding high-level event

SEE ALSO

For details on how to process an Apple event using the `AEProcessAppleEvent` function, see *Inside Macintosh: Interapplication Communication*.

GetSpecificHighLevelEvent

You can use the `GetSpecificHighLevelEvent` function to select and optionally retrieve a specific high-level event from your application's high-level event queue.

```
FUNCTION GetSpecificHighLevelEvent
            (aFilter: GetSpecificFilterProcPtr;
             yourDataPtr: UNIV Ptr; VAR err: OSErr): Boolean;
```

`aFilter` Specifies the filter function that `GetSpecificHighLevelEvent` should use to search for a specific event. `GetSpecificHighLevelEvent` calls your filter function once for each event in your application's high-level event queue until your filter function returns `TRUE` or the end of the queue is reached.

`yourDataPtr`
Specifies the criteria your filter function should use to select a specific event. For example, in the `yourDataPtr` parameter you can specify a reference constant to search for a particular event, a pointer to a target ID record to search for a specific sender of an event, or an event class to search for a specific class of event.

`err` `GetSpecificHighLevelEvent` returns in this parameter a value indicating if any errors occurred. The `err` parameter contains the `noErr` constant if no errors occurred or `noOutstandingHLE` if no high-level events are pending in your application's high-level event queue.

DESCRIPTION

You can use the `GetSpecificHighLevelEvent` function to search for a specific high-level event in your application's high-level event queue. You provide a pointer to a filter function as one of the parameters to `GetSpecificHighLevelEvent`. The `GetSpecificHighLevelEvent` function calls your filter function once for every event in your application's high-level event queue, until your filter function returns `TRUE` or the end of the queue is reached.

The `GetSpecificHighLevelEvent` function passes the value you specify in the `yourDataPtr` parameter to your filter function. Your filter function also receives as parameters the event record associated with the high-level event and the target ID record that identifies the sender of the event. Your filter function can compare the contents of the `yourDataPtr` parameter with any of the other information it receives.

If your filter function finds a match, it can call `AcceptHighLevelEvent` if necessary, and then return `TRUE`. If your filter function does not find a match, then it should return `FALSE`.

If your filter function returns TRUE, the GetSpecificHighLevelEvent function returns TRUE. If your filter function returns FALSE for all high-level events in your application's event queue, or if there are no high-level events in the queue, GetSpecificHighLevelEvent returns FALSE.

SPECIAL CONSIDERATIONS

The GetSpecificHighLevelEvent function may move or purge memory. You should not call this function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the GetSpecificHighLevelEvent function are

Trap macro	Selector
_OSDispatch	$0045

SEE ALSO

See "Filter Function for Searching the High-Level Event Queue" on page 2-114 for more information about how to define a filter function and the parameters that GetSpecificHighLevelEvent passes to your filter function.

FlushEvents

The FlushEvents procedure removes low-level events from the Operating System event queue. Note that FlushEvents does not remove any types of events not stored in the Operating System event queue.

You can choose to use the FlushEvents procedure when your application first starts to empty the Operating System event queue of any keystrokes or mouse events generated by the user while the Finder loaded your application. In general, however, your application should not empty the queue at any other time as this loses user actions and makes your application and the computer appear unresponsive to the user.

```
PROCEDURE FlushEvents (whichMask: Integer; stopMask: Integer);
```

whichMask — A value that indicates which kinds of low-level events are to be removed from the Operating System event queue; this parameter is interpreted as a sum of event mask constants. The whichMask and stopMask parameters together specify which events to remove.

`stopMask` A value that limits which low-level events are to be removed from the Operating System event queue; this parameter is interpreted as a sum of event mask constants. `FlushEvents` does not remove any low-level events that are specified by the `stopMask` parameter. To remove all events specified by the `whichMask` parameter, specify 0 as the `stopMask` parameter.

DESCRIPTION

`FlushEvents` removes only low-level events stored in the Operating System event queue; it does not remove activate, update, operating-system, or high-level events.

You specify which low-level events to remove using the `whichMask` and `stopMask` parameters. `FlushEvents` removes the low-level events specified by the `whichMask` parameter, up to but not including the first event of any type specified by the `stopMask` parameter.

If the event queue doesn't contain any of the events specified by the `whichMask` parameter, `FlushEvents` does not remove any events from the queue.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register D0 with the event mask (`whichMask`) and stop mask before calling `FlushEvents`. When `FlushEvents` returns, register D0 contains 0 if all events were removed from the queue or, if all events were not removed from the queue, an event code that specifies the type of event that caused the removal process to stop.

Registers on entry

D0 Event mask (low-order word)
Stop mask (high-order word)

Registers on exit

D0 0 if all events were removed from the queue, or the event code of the event that stopped the search (low-order word)

SEE ALSO

See "Setting the Event Mask" beginning on page 2-26 for information on how to specify an event mask.

SystemClick

After receiving a mouse-down event, your application should call the Window Manager function `FindWindow` to determine where the cursor was when the mouse button was pressed. If `FindWindow` returns the `inSysWindow` constant, call the `SystemClick` procedure to handle the event.

```
PROCEDURE SystemClick (theEvent: EventRecord;
                        theWindow: WindowPtr);
```

`theEvent` The event record for the event.

`theWindow` The window in which the mouse-down event occurred. Pass the window pointer returned by `FindWindow` in this parameter.

DESCRIPTION

If a mouse-down event occurred in a desk accessory's window, the `SystemClick` procedure determines which part of the desk accessory's window the cursor was in when the mouse button was pressed and routes the event to the appropriate desk accessory as necessary.

If the mouse button was pressed while the cursor was in the content region of the desk accessory's window and the window is active, `SystemClick` sends the mouse-down event to the desk accessory to process. If the mouse-down event occurred in the content region of the window and the window is inactive, `SystemClick` makes it the active window. It does this by sending your application an activate event to deactivate its front window and directing an event to the desk accessory to activate its window.

If the mouse button was pressed while the cursor was in the drag region or go-away region, `SystemClick` calls the Window Manager routine `DragWindow` or `TrackGoAway`, as appropriate. If `TrackGoAway` reports that the user closed the desk accessory, `SystemClick` sends a close message to the desk accessory.

SEE ALSO

See "The Event Record," beginning on page 2-79, for a description of the fields in the event record.

SystemTask

In a multiple-application environment, the `WaitNextEvent` function is responsible for giving time to each open desk accessory or driver to perform any periodic action. You should not call `SystemTask` if your application calls `WaitNextEvent`.

If your application calls `GetNextEvent`, your application should call the `SystemTask` procedure.

```
PROCEDURE SystemTask;
```

DESCRIPTION

The `SystemTask` procedure gives time to each open desk accessory or driver to perform the periodic action defined for it. A desk accessory or device driver specifies how often the periodic action should occur, and `SystemTask` gives time to the desk accessory or device driver at the appropriate interval.

If your application calls `GetNextEvent`, your application should call `SystemTask` at least every sixtieth of a second. This usually corresponds to calling `SystemTask` once

each time through your event loop. If your application does a large amount of processing, you may need to call `SystemTask` more than once in your event loop.

SEE ALSO

For a description of the `WaitNextEvent` function and the `GetNextEvent` function, see page 2-85 and page 2-89, respectively.

SystemEvent

The `WaitNextEvent` and `GetNextEvent` functions call the `SystemEvent` function. In most cases your application should not call the `SystemEvent` function.

The `SystemEvent` function determines if a specific event should be handled by the application or the Operating System.

```
FUNCTION SystemEvent (theEvent: EventRecord): Boolean;
```

`theEvent` The event record for the event.

DESCRIPTION

`SystemEvent` returns `FALSE` as its function result if the event should be handled by the application; otherwise, `SystemEvent` takes any appropriate actions and returns `TRUE`.

For activate, update, mouse-up, and keyboard events (including keyboard equivalents of commands), `SystemEvent` checks to see whether the active window belongs to a desk accessory and whether that desk accessory can handle that type of event. If so, `SystemEvent` sends the event to the desk accessory and returns `TRUE`. Otherwise, `SystemEvent` returns `FALSE`.

For mouse-down events and null events, `SystemEvent` returns `FALSE`.

For disk-inserted events, `SystemEvent` attempts to mount the disk using the `PBMountVol` function but returns `FALSE` so that the application can perform further processing if necessary.

ASSEMBLY-LANGUAGE INFORMATION

If the `SEvtEnb` global variable (a byte) contains 0, `SystemEvent` always returns `FALSE`.

SEE ALSO

See “The Event Record,” beginning on page 2-79, for a description of the fields in the event record. For a description of the `PBMountVol` function, see the chapter “File Manager” in *Inside Macintosh: Files*.

GetOSEvent

The Toolbox Event Manager calls the `GetOSEvent` function to retrieve low-level events stored in the Operating System event queue. In most cases your application should not use this function.

```
FUNCTION GetOSEvent (mask: Integer;
                     VAR theEvent: EventRecord): Boolean;
```

`mask` A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. `GetOSEvent` returns only low-level events stored in the Operating System event queue; it does not return activate, update, operating-system, or high-level events. If no low-level event of any of the designated types is available, `GetOSEvent` returns a null event.

`theEvent` The next available low-level event of the specified type or types in the Operating System event queue. The `GetOSEvent` function removes the returned event from the Operating System event queue and returns the information about the event in an event record. The event record includes the type of event received and other information.

DESCRIPTION

The `GetOSEvent` function retrieves and removes an event from the Operating System event queue. `GetOSEvent` returns `FALSE` as its function result if the event being returned is a null event; otherwise, `GetOSEvent` returns `TRUE`. `GetOSEvent` does not intercept or respond to the event in any way. It also does not process Command–Shift–number key combinations or process any alarms set by the user through the Alarm Clock desk accessory.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with the address of an event record and register D0 with the event mask before invoking `GetOSEvent`. When `GetOSEvent` returns, register D0 indicates whether the returned event is a null event or an event other than a null event and the returned event is accessible through register A0.

Registers on entry

A0 Address of event record

D0 Event mask (low-order word)

Registers on exit

A0 Address of event record

D0 0 if `GetOSEvent` returns any event other than a null event,
or –1 if it returns a null event (low-order byte)

SEE ALSO

See “The Event Record,” beginning on page 2-79, for a description of the fields in the event record. See “Setting the Event Mask,” beginning on page 2-26, for information on how to specify an event mask.

OSEventAvail

The Toolbox Event Manager uses the `OSEventAvail` function to retrieve an event from the Operating System event queue without removing it. In most cases your application does not need to use this function.

```
FUNCTION OSEventAvail (mask: Integer;
                        VAR theEvent: EventRecord): Boolean;
```

`mask` A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. `OSEventAvail` returns only low-level events stored in the Operating System event queue; it does not return activate, update, operating-system, or high-level events. If no low-level event of any of the designated types is available, `OSEventAvail` returns a null event.

`theEvent` The next available event of the specified type or types. The `OSEventAvail` function does not remove the returned event from the Operating System event queue but does return information about the event in an event record. The event record includes the type of event received and other information.

DESCRIPTION

The `OSEventAvail` function retrieves an event from the Operating System event queue without removing it from the queue. The `OSEventAvail` function returns `FALSE` as its function result if the event being returned is a null event; otherwise, `OSEventAvail` returns `TRUE`.

`OSEventAvail` does not intercept or respond to the event in any way. It also does not process Command–Shift–number key combinations or process any alarms set by the user through the Alarm Clock desk accessory.

SPECIAL CONSIDERATIONS

If the `OSEventAvail` function returns a low-level event from the Operating System event queue, the event will not be accessible later if, in the meantime, the event queue becomes full and the event is discarded from it; however, this is not a common occurrence.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with the address of an event record and register D0 with the event mask before invoking `OSEventAvail`. When `OSEventAvail` returns, register D0 indicates whether the returned event is a null event or some other event, and the returned event is accessible through register A0.

Registers on entry

A0	Address of event record
D0	Event mask (low-order word)

Registers on exit

A0	Address of event record
D0	0 if `OSEventAvail` returns any event other than a null event, or –1 if it returns a null event (low-order byte)

SEE ALSO

See “The Event Record,” beginning on page 2-79, for a description of the fields in the event record. See “Setting the Event Mask,” beginning on page 2-26, for information on how to specify an event mask

SetEventMask

The `SetEventMask` procedure sets the system event mask of your application to the specified mask. Your application should not call the `SetEventMask` procedure to disable any event types from being posted. Use `SetEventMask` only to enable key-up events if your application needs to respond to key-up events.

```
PROCEDURE SetEventMask (theMask: Integer);
```

`theMask` An event mask that specifies which events should be posted in the Operating System event queue.

DESCRIPTION

The `SetEventMask` procedure sets the system event mask of your application according to the parameter `theMask`. The Operating System Event Manager posts only low-level events (other than update or activate events) corresponding to bits in the system event mask of the current process when posting events in the Operating System event queue. The system event mask of an application is initially set to post mouse-up, mouse-down, key-down, auto-key, and disk-inserted events into the Operating System event queue.

ASSEMBLY-LANGUAGE INFORMATION

The system event mask of the current application is available in the `SysEvtMask` system global variable.

SEE ALSO

For additional information on event masks, see “Setting the Event Mask” beginning on page 2-26.

GetEvQHdr

The Event Manager uses the `GetEvQHdr` function to get a pointer to the header of the Operating System event queue. In most cases your application should not call the `GetEvQHdr` function.

```
FUNCTION GetEvQHdr: QHdrPtr;
```

DESCRIPTION

The `GetEvQHdr` function returns a pointer to the header of the Operating System event queue.

ASSEMBLY-LANGUAGE NOTE

The `EventQueue` system global variable contains the header of the event queue.

SEE ALSO

See “The Event Queue” on page 2-83 for information on the structure of the Operating System event queue.

Sending Events

You can send events to other applications or processes using the `PostHighLevelEvent` function. To send Apple events to other applications, use the Apple Event Manager function `AESend`. The Operating System Event Manager also provides the `PPostEvent` and `PostEvent` functions for posting low-level events to the Operating System event queue. The `PostEvent` function is used by the Toolbox Event Manager. In most cases your application should not use the `PostEvent` function.

PostHighLevelEvent

You can use the `PostHighLevelEvent` function to send a high-level event to another application.

```
FUNCTION PostHighLevelEvent (theEvent: EventRecord;
                             receiverID: Ptr; msgRefcon: LongInt;
                             msgBuff: Ptr; msgLen: LongInt;
                             postingOptions: LongInt): OSErr;
```


`theEvent` The event to send. Your application should fill out the `what`, `message`, and `where` fields of the event record. Specify the `kHighLevelEvent` constant in the `what` field, the event class of the high-level event in the `message` field, and the event ID in the `where` field. You do not need to fill out the `when` or `modifiers` fields; the Event Manager automatically assigns the appropriate values to these fields when you send the message.

`receiverID`
The recipient of the high-level event. When sending an event to another application on the local computer, you can specify the recipient of the event by session reference number, process serial number, signature, or port name and location name. When sending an event to an application on a remote computer, you can specify the recipient only by the session reference number or by the port name and location name.

To specify a port name and location name, provide the address of a target ID record in the `receiverID` parameter. To specify a process serial number, provide its address in the `receiverID` parameter. To specify a session reference number, or signature, provide the data (cast to the `Ptr` data type) in the `receiverID` parameter.

`msgRefcon` A unique number that identifies the communication associated with this event. Your application can set this field to any value it chooses. If you are replying to a high-level event, you should use the same value in the `msgRefcon` parameter as specified in the high-level event that originated the request.

`msgBuff` A pointer to a data buffer that contains any additional data for the event.

`msgLen` The size (in bytes) of the data buffer pointed to by the `msgBuff` parameter.

`postingOptions`
Options associated with the `receiverID` parameter and delivery options associated with the event. You can specify one or more delivery options to indicate whether you want the other application to receive the event at the next opportunity and to indicate whether you want acknowledgment that the event was received by the other application. You use the options associated with the `receiverID` parameter to indicate how you are specifying the recipient of the event—whether by port name and location name in a target ID record, by session reference number, by process serial number, or by signature.

You can use a combination of these constants in the `postingOptions` parameter:

```
CONST
nAttnMsg
   = $00000001; {give this message priority}
nReturnReceipt
   = $00000200; {return receipt requested}
receiverIDisTargetID
   = $00005000; {ID is port name and location name}
receiverIDisSessionID
   = $00006000; {ID is PPC session reference number}
receiverIDisSignature
   = $00007000; {ID is creator signature}
receiverIDisPSN
   = $00008000; {ID is process serial number}
```

DESCRIPTION

The `PostHighLevelEvent` function posts the high-level event to the specified process.

If the application to which you are sending a high-level event terminates, you receive the result code `sessionClosedErr` the next time your application calls `PostHighLevelEvent` to send another high-level event to the terminated application. If you do not care about any state information about that session, you can just resend your event. Otherwise, you must restart another session and resend your event.

If your application is running in the background and posts a high-level event that requires the network authentication dialog box to be displayed, `PostHighLevelEvent` returns the `noUserInteractionAllowed` result code, does not display the network authentication dialog box, and does not send the event. If your application receives the `noUserInteractionAllowed` result code, you can use the Notification Manager to inform the user that your application needs attention. When the user brings your application to the foreground, you can repost the event. If the reposting is successful, your application can continue to post high-level events without further user interaction. Note that `PostHighLevelEvent` can return `noUserInteractionAllowed` only on the first posting of a high-level event to a remote target.

SPECIAL CONSIDERATIONS

The `PostHighLevelEvent` function may move or purge memory. You should not call this function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PostHighLevelEvent` function are

Trap macro	Selector
`_OSDispatch`	$0034

SEE ALSO

For details on how to send Apple events to other applications using the `AESend` function, see *Inside Macintosh: Interapplication Communication*.

RESULT CODES

`noErr`	0	No error
`connectionInvalid`	–609	Connection is invalid
`noUserInteractionAllowed`	–610	Cannot interact directly with user
`sessionClosedErr`	–917	Session closed

PPostEvent

In most cases your application does not need to post events in the Operating System event queue; however, if you must do so, you can use the `PPostEvent` function.

```
FUNCTION PPostEvent (eventCode: Integer; eventMsg: LongInt;
                     VAR qEl: EvQElPtr): OSErr;
```

`eventCode` A value that indicates the type of event to post into the Operating System event queue. The types of events that can be posted in this queue are represented by these constants: `mouseDown`, `mouseUp`, `keyDown`, `keyUp`, `autoKey`, and `diskEvt`. Do not attempt to post any other type of event in the Operating System event queue.

`eventMsg` A long integer that contains the contents of the `message` field for the event that `PPostEvent` should post in the queue.

`qEl` `PPostEvent` returns a pointer to the event queue entry of the posted event in this parameter.

DESCRIPTION

In the `eventCode` and `eventMsg` parameters, you specify the value for the `what` and `message` fields of the event's event record. The `PPostEvent` function fills out the `when`, `where`, and `modifiers` fields of the event record with the current time, current mouse location, and current state of the modifier keys and mouse button.

The `PPostEvent` function returns a pointer to the event queue entry of the posted event in the `qEl` parameter. You can change any fields of the posted event by changing the fields of its event queue entry. For example, you can change the posted event's modifier keys by changing the value of the `evtQModifiers` field of the event queue entry.

The `PPostEvent` function posts only events that are enabled by the system event mask. If the event queue is full, `PPostEvent` removes the oldest event in the queue and posts the new event.

▲ **WARNING**
Do not post any events other than mouse-down, mouse-up, key-down, key-up, auto-key, and disk-inserted events in the Operating System event queue. Attempting to post other events into the Operating System event queue interferes with the internal operation of the Event Manager. ▲

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 and register D0 before invoking `PPostEvent`. The `PPostEvent` function returns values in registers A0 and D0.

Registers on entry

A0	Event number (low-order word)
D0	Event message (long)

Registers on exit

A0	Pointer to an event queue entry (long)
D0	Result code (low-order word)

RESULT CODES

`evtNotEnb`	1	Event type not valid—event not posted
`noErr`	0	No error

SEE ALSO

For a description of the entries in the event queue, see "The Event Queue" on page 2-83.

PostEvent

The Toolbox Event Manager uses the `PostEvent` function to post events into the Operating System event queue. In most cases your application should not call the `PostEvent` function.

```
FUNCTION PostEvent (eventNum: Integer; eventMsg: LongInt): OSErr;
```

`eventNum` A value that indicates the type of event to post into the Operating System event queue. The types of events that can be posted in this queue are represented by these constants: `mouseDown`, `mouseUp`, `keyDown`, `keyUp`, `autoKey`, and `diskEvt`. Do not attempt to post any other type of event in the Operating System event queue.

`eventMsg` A long integer that contains the contents of the `message` field for the event that `PostEvent` should post in the queue.

DESCRIPTION

In the `eventNum` and `eventMsg` parameters, you specify the value for the `what` and `message` fields of the event's event record. The `PostEvent` function fills out the `when`, `where`, and `modifiers` fields of the event record with the current time, current mouse location, and current state of the modifier keys and mouse button.

The `PostEvent` function posts only events that are enabled by the system event mask. If the event queue is full, `PostEvent` removes the oldest event in the queue and posts the new event.

Note that if you use `PostEvent` to repost an event, the `PostEvent` function fills out the `when`, `where`, and `modifier` fields of the event record, giving these fields of the reposted event different values from the values contained in the original event.

▲ **WARNING**
Do not post any events other than mouse-down, mouse-up, key-down, key-up, auto-key, and disk-inserted events in the Operating System event queue. Attempting to post other events into the Operating System event queue interferes with the internal operation of the Event Manager. ▲

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with the event code and register D0 with the event message before invoking `PostEvent`. When `PostEvent` returns, register D0 contains the result code.

Registers on entry

A0	Event number (low-order word)
D0	Event message (long)

Registers on exit

D0	Result code (low-order word)

RESULT CODES

`evtNotEnb`	1	Event type not valid—event not posted
`noErr`	0	No error

Converting Process Serial Numbers and Port Names

The Event Manager provides two functions to convert between process serial numbers and port names (`GetProcessSerialNumberFromPortName` and `GetPortNameFromProcessSerialNumber`). Both functions are intended to map serial numbers to port names (or vice versa) for applications open on the local computer. They do not return useful results for applications open on remote computers.

GetProcessSerialNumberFromPortName

Use GetProcessSerialNumberFromPortName to get the process serial number of a process.

```
FUNCTION GetProcessSerialNumberFromPortName
             (portName: PPCPortRec;
              VAR PSN: ProcessSerialNumber): OSErr;
```

portName The port name registered to a process whose serial number you want.

PSN Returns the process serial number of the process designated by the portName parameter. You can use the returned process serial number to send a high-level event to that process. Do not interpret the value of the process serial number.

DESCRIPTION

The GetProcessSerialNumberFromPortName function returns the process serial number of the process registered at a specific port.

SPECIAL CONSIDERATIONS

The GetProcessSerialNumberFromPortName function does not move or purge memory but for other reasons should not be called from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the GetProcessSerialNumberFromPortName function are

Trap macro	Selector
_OSDispatch	$0035

RESULT CODES

noErr	0	No error
noPortErr	–903	Invalid port name

SEE ALSO

For a description of the PPCPortRec data type, see the chapter "Program-to-Program Communications Toolbox" in *Inside Macintosh: Interapplication Communication*.

GetPortNameFromProcessSerialNumber

Use `GetPortNameFromProcessSerialNumber` to get the port name of a process.

```
FUNCTION GetPortNameFromProcessSerialNumber
            (VAR portName: PPCPortRec;
             PSN: ProcessSerialNumber): OSErr;
```

`portName` Returns the port name of the process designated by the `PSN` parameter. You can use the returned port name to send a high-level event to that process.

`PSN` The process serial number of the process whose port name you want.

DESCRIPTION

The `GetPortNameFromProcessSerialNumber` function returns the port name registered to a process having a specific process serial number.

SPECIAL CONSIDERATIONS

The `GetPortNameFromProcessSerialNumber` function does not move or purge memory but for other reasons should not be called from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetPortNameFromProcessSerialNumber` function are

Trap macro	Selector
`_OSDispatch`	$0046

RESULT CODES

`noErr`	0	No error
`procNotFound`	–600	No eligible process with specified process serial number

SEE ALSO

For a description of the `PPCPortRec` data type, see the chapter “Program-to-Program Communications Toolbox” in *Inside Macintosh: Interapplication Communication.*

Reading the Mouse

The Event Manager provides routines you can use to get information about the mouse. You can get the current mouse location using the `GetMouse` procedure. You can use the `Button` function to determine whether the user pressed the mouse button. After receiving a mouse-down event, you can use the `StillDown` function to determine whether the mouse button is still down, and you can use `WaitMouseUp` to determine if the user subsequently released the mouse.

GetMouse

You can use the `GetMouse` procedure to obtain the current mouse location.

```
PROCEDURE GetMouse (VAR mouseLoc: Point);
```

`mouseLoc` Returns the current mouse location in local coordinates of the current graphics port (for example, the active window). Note that this value differs from the value of the `where` field of the event record, which specifies the mouse location in global coordinates.

Button

You can use the `Button` function to determine whether the user pressed the mouse button.

```
FUNCTION Button: Boolean;
```

DESCRIPTION

The `Button` function looks in the Operating System event queue for a mouse-down event. If it finds one, the `Button` function returns `TRUE`; otherwise, it returns `FALSE`. To determine whether the mouse button is still down after a mouse-down event, use the `StillDown` function.

SEE ALSO

See “The Event Queue” on page 2-83 for information about the Operating System event queue.

StillDown

After receiving a mouse-down event, you can use the `StillDown` function to determine if the mouse button is still down.

```
FUNCTION StillDown: Boolean;
```


DESCRIPTION

The `StillDown` function looks in the Operating System event queue for a mouse event. If it finds one, the `StillDown` function returns `FALSE`. If it does not find any mouse events pending in the Operating System event queue, the `StillDown` function returns `TRUE`.

SEE ALSO

See "The Event Queue" on page 2-83 for information about the Operating System event queue.

WaitMouseUp

After receiving a mouse-down event, you can use the `WaitMouseUp` function to determine if the user subsequently released the mouse.

```
FUNCTION WaitMouseUp: Boolean;
```

DESCRIPTION

The `WaitMouseUp` function looks in the Operating System event queue for a mouse-up event. If it finds one, the `WaitMouseUp` function removes the mouse-up event from the queue and returns `TRUE`. If it does not find any mouse events pending in the Operating System event queue, the `WaitMouseUp` function returns `FALSE`.

SEE ALSO

See "The Event Queue" on page 2-83 for information about the Operating System event queue.

Reading the Keyboard

The Event Manager reports keyboard events one at a time at your application's request when you use the WaitNextEvent, EventAvail, or GetNextEvent function. In addition to getting keyboard events when the user presses or releases a key, you can directly read the keyboard (and keypad) at any time using the GetKeys procedure.

You can also use the KeyTranslate function to convert virtual key codes to character code values using a specified 'KCHR' resource.

GetKeys

You can use the GetKeys procedure to obtain the current state of the keyboard.

```
PROCEDURE GetKeys (VAR theKeys: KeyMap);
```

theKeys Returns the current state of the keyboard, including the keypad, if any. The GetKeys procedure returns this information using the KeyMap data type.

```
TYPE KeyMap = PACKED ARRAY[0..127] OF Boolean;
```

Each key on the keyboard or keypad corresponds to an element in the KeyMap array. The index for a particular key is the same as the key's virtual key code minus 1. For example, the key with virtual key code 38 (the "J" key on the Apple Keyboard II) can be accessed as KeyMap[37] in the returned array. A KeyMap element is TRUE if the corresponding key is down and FALSE if it isn't. The maximum number of keys that can be down simultaneously is two character keys plus any combination of the five modifier keys.

DESCRIPTION

You can use the GetKeys procedure to determine the current state of the keyboard at any time. For example, you can determine whether one of the modifier keys is down by itself or in combination with another key using the GetKeys procedure.

KeyTranslate

You can use the KeyTranslate function to convert a virtual key code to a character code based on a 'KCHR' resource. The KeyTranslate function is also available as the KeyTrans function.

```
FUNCTION KeyTranslate (transData: Ptr; keycode: Integer;
                       VAR state: LongInt): LongInt;
```

`transData` A pointer to the 'KCHR' resource that you want the `KeyTranslate` function to use when converting the key code to a character code.

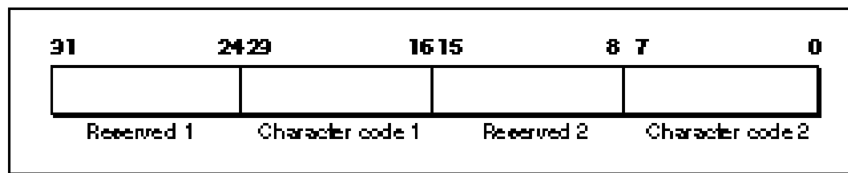
`keycode` A 16-bit value that your application should set so that bits 0–6 contain the virtual key code and bit 7 contains either 1 to indicate an up stroke or 0 to indicate a down stroke of the key. Bits 8–15 have the same interpretation as the high byte of the `modifiers` field of the event record and should be set according to the needs of your application.

`state` A value that your application should set to 0 the first time it calls `KeyTranslate` or any time your application calls `KeyTranslate` with a different 'KCHR' resource. Thereafter, your application should pass the same value for the `state` parameter as `KeyTranslate` returned in the previous call.

DESCRIPTION

The `KeyTranslate` function returns a 32-bit value that gives the character code for the virtual key code specified by the `keycode` parameter. Figure 2-17 shows the structure of the 32-bit number that `KeyTranslate` returns.

Figure 2-17 Structure of the `KeyTranslate` function result



The `KeyTranslate` function returns the values that correspond to one or possibly two characters that are generated by the specified virtual key code. For example, a given virtual key code might correspond to an alphabetic character with a separate accent character. For example, when the user presses Option-E followed by N, you can map this through the `KeyTranslate` function using the U.S. 'KCHR' resource to produce ´n, which `KeyTranslate` returns as two characters in the bytes labeled Character code 1 and Character code 2. If `KeyTranslate` returns only one character code, it is always in the byte labeled Character code 2. However, your application should always check both bytes labeled Character code 1 and Character code 2 in Figure 2-17 for possible values that map to the virtual key code.

SEE ALSO

For additional information on the 'KCHR' resource and the `KeyTranslate` function, see *Inside Macintosh: Text.*

Getting Timing Information

You can get the current number of ticks since the system last started up using the `TickCount` function. You can use this function to compare the number of ticks that have expired since a given event or other action occurred.

By using the `GetDblTime` function, you can get the suggested maximum difference in ticks that should exist to consider two mouse events a double click. The user can adjust this value using the Mouse control panel. Using the `GetCaretTime` function you can get the suggested maximum difference in ticks that should exist between blinks of the caret in editable text. The user can adjust this value using the General Controls panel.

TickCount

You can use the `TickCount` function to get the current number of ticks (a tick is approximately 1/60 of a second) since the system last started up.

```
FUNCTION TickCount: LongInt;
```

DESCRIPTION

The `TickCount` function returns a long integer that indicates the current number of ticks since the system last started up. You can use this value to compare the number of ticks that have elapsed since a given event or other action occurred. For example, you could compare the current value returned by `TickCount` with the value of the `when` field of an event record.

The tick count is incremented during the vertical retrace interrupt, but this interrupt can be disabled. Your application should not rely on the tick count to increment with absolute precision. Your application also should not assume that the tick count always increments by 1; an interrupt task might keep control for more than one tick. If your application keeps track of the previous tick count and then compares this value with the current tick count, your application should compare the two values by checking for a "greater than or equal" condition rather than "equal to previous tick count plus 1."

▲ **WARNING**
Don't rely on the tick count being exact; it's usually accurate to within one tick, but this level of accuracy is not guaranteed. ▲

ASSEMBLY-LANGUAGE NOTE

The value returned by `TickCount` is also accessible in the global variable `Ticks`.

GetDblTime

To determine whether a sequence of mouse events constitutes a double click, your application measures the elapsed time (in ticks) between a mouse-up event and a mouse-down event. If the time between the two mouse events is less than the value returned by `GetDblTime`, your application should interpret the two mouse events as a double click.

```
FUNCTION GetDblTime: LongInt;
```

DESCRIPTION

The `GetDblTime` function returns the suggested maximum elapsed time, in ticks, between a mouse-up event and a mouse-down event. The user can adjust this value using the Mouse control panel.

If your application distinguishes a double click of the mouse from a single click, your application should use the value returned by `GetDblTime` to make this distinction. If your application uses TextEdit, the TextEdit procedures automatically recognize and handle double clicks of text within a TextEdit edit record by appropriately highlighting or unhighlighting the selection.

ASSEMBLY-LANGUAGE NOTE

The value returned by `GetDblTime` is also accessible in the system global variable `DoubleTime`.

GetCaretTime

You can use the `GetCaretTime` function to get the suggested difference in ticks that should exist between blinks of the caret (usually a vertical bar marking the insertion point) in editable text. The user can adjust this value using the General Controls panel.

```
FUNCTION GetCaretTime: LongInt;
```

DESCRIPTION

If your application supports editable text, your application should use the value returned by `GetCaretTime` to determine how often to blink the caret. If your application uses only TextEdit, you can use TextEdit procedures to automatically blink the caret at the time interval that the user specifies in the General Controls panel.

ASSEMBLY-LANGUAGE NOTE

The value returned by `GetCaretTime` is also accessible in the system global variable `CaretTime`.

Application-Defined Routine

When you use `GetSpecificHighLevelEvent`, you supply a filter function so that your application can search for a specific event in the high-level event queue of your application.

Filter Function for Searching the High-Level Event Queue

This section describes the filter function that you can provide to `GetSpecificHighLevelEvent`. For example, you might use a filter function to search for a high-level event sent from a specific application.

MyFilter

When you use `GetSpecificHighLevelEvent` to search the high-level event queue of your application for a specific event, you supply a pointer to a filter function. `GetSpecificHighLevelEvent` calls your filter function once for each event in the high-level event queue until your filter function returns `TRUE` or the end of the queue is reached. Your filter function can examine each event and determine whether that event is the desired event. If so, your filter function should return `TRUE`.

Here's how you declare the filter function `MyFilter`:

```
FUNCTION MyFilter (yourDataPtr: Ptr;
                   msgBuff: HighLevelEventMsgPtr;
                   sender: TargetID): Boolean;
```

`yourDataPtr`
: Specifies the criteria your filter function should use to select a specific event. For example, you can specify the `yourDataPtr` parameter as a reference constant to search for a particular event, as a pointer to a target ID record to search for a specific sender of an event, or as an event class to search for a specific class of event.

`msgBuff`
: Contains a pointer to a record of data type `HighLevelEventMsg`, which provides: the event record for the high-level event and the reference constant of the event. The `HighLevelEventMsg` data type is described in "The High-Level Event Message Record" on page 2-82.

`sender`
: Contains the target ID record of the application that sent the event. The `TargetID` data type is described in "The Target ID Record" on page 2-81.

DESCRIPTION

Your filter function can compare the contents of the `yourDataPtr` parameter with the contents of the `msgBuff` and `sender` parameters. If your filter function finds a match, it can call `AcceptHighLevelEvent`, if necessary, and your filter function should return `TRUE`. If your filter function does not find a match, it should return `FALSE`.

SEE ALSO

For information about how to specify your filter function to the `GetSpecificHighLevelEvent` function, see page 2-92.

Resource

This section explains the structure of a `'SIZE'` resource and the meaning of each of its fields. You are responsible for creating the information in this resource.

The Size Resource

Every application executing in System 7, as well as every application executing under MultiFinder, should contain a size (`'SIZE'`) resource. One of the principal functions of the `'SIZE'` resource is to inform the Operating System about the memory size requirements for the application so that the Operating System can set up an appropriately sized partition for the application. The `'SIZE'` resource is also used to indicate certain scheduling options to the Operating System, such as whether the application can accept suspend and resume events. The `'SIZE'` resource in System 7 contains additional information indicating whether the application is 32-bit clean, whether it supports stationery documents, whether it uses TextEdit's inline input services, whether the application wishes to receive notification of the termination of any applications it has launched, and whether the application wishes to receive high-level events.

A `'SIZE'` resource consists of a 16-bit flags field followed by two 32-bit size fields. The flags field specifies operating characteristics of the application, and the size fields indicate the minimum and preferred partition sizes for the application. The minimum partition size is the actual limit below which your application will not run. The preferred partition size is the memory size at which your application can run most effectively and which the Operating System attempts to secure upon launching the application. If that amount of memory is unavailable, the application is placed into the largest contiguous block available, provided that it is larger than the specified minimum size.

Note

If the amount of available memory is between the minimum and the preferred sizes, the Finder displays a dialog box asking if the user wants to run the application using the amount of memory available. If your application does not have a `'SIZE'` resource, it is assigned a default partition size of 512 KB and the Process Manager uses a default value of `FALSE` for all specifications normally defined by constants in the flags field. ◆

When you define a `'SIZE'` resource, you should give it a resource ID of –1. A user can modify the preferred size in the Finder's information window for your application. If the user does alter the preferred partition size, the Operating System creates a new `'SIZE'` resource having resource ID 0. The Process Manager also creates a new `'SIZE'` resource when the user modifies any of the other settings in the resource.

In system software version 7.1 the user can also modify the minimum size in the Finder's information window for your application. In version 7.1, if the user alters either the minimum or the preferred partition size, the Operating System creates two new 'SIZE' resources, one with resource ID 0 and one with resource ID 1.

At application launch time, the Process Manager looks for a 'SIZE' resource with ID 0 for the preferred partition size; if this resource is not found, it uses your original 'SIZE' resource with ID –1. In version 7.1, the Process Manager looks for a 'SIZE' resource with ID 0 for the preferred size and looks for a 'SIZE' resource with ID 1 for the minimum size; if these resources are not found, it uses your original 'SIZE' resource with ID –1.

Listing 2-19 shows the structure of the 'SIZE' resource in Rez format. See Listing 2-4 in "Creating a Size Resource," beginning on page 2-30 for a sample 'SIZE' resource for an application.

Listing 2-19 A Rez template for a 'SIZE' resource

```
type 'SIZE' {
   boolean  reserved;                   /*reserved*/
   boolean  ignoreSuspendResumeEvents,  /*ignores suspend-resume events*/
            acceptSuspendResumeEvents;  /*accepts suspend-resume events*/
   boolean  reserved;                   /*reserved*/
   boolean  cannotBackground,           /*can't use background null events*/
            canBackground;              /*can use background null events*/
   boolean  needsActivateOnFGSwitch,    /*needs activate event following */
                                        /* major switch*/
            doesActivateOnFGSwitch;     /*activates own windows in */
                                        /* response to OS events*/
   boolean  backgroundAndForeground,    /*app has a user interface*/
            onlyBackground;             /*app has no user interface*/
   boolean  dontGetFrontClicks,         /*don't return mouse events */
                                        /* in front window on resume*/
            getFrontClicks;             /*do return mouse events */
                                        /* in front window on resume*/
   boolean  ignoreAppDiedEvents,        /*applications use this*/
            acceptAppDiedEvents;        /*app launchers use this*/
   boolean  not32BitCompatible,         /*works with 24-bit addr*/
            is32BitCompatible;          /*works with 24- or 32-bit addr*/
   boolean  notHighLevelEventAware,     /*can't use high-level events*/
            isHighLevelEventAware;      /*can use high-level events*/
   boolean  onlyLocalHLEvents,          /*only local high-level events*/
            localAndRemoteHLEvents;     /*also remote high-level events*/
   boolean  notStationeryAware,         /*can't use stationery documents*/
            isStationeryAware;          /*can use stationery documents*/
   boolean  dontUseTextEditServices,    /*can't use inline services*/
            useTextEditServices;        /*can use inline services*/
```

```
   boolean  reserved;                  /*reserved*/
   boolean  reserved;                  /*reserved*/
   boolean  reserved;                  /*reserved*/
                                       /*memory sizes are in bytes*/
   unsigned longint;                   /*preferred memory size*/
   unsigned longint;                   /*minimum memory size*/
};
```


The nonreserved bits in the flags field have the following meanings:

Flag descriptions

`acceptSuspendResumeEvents`
When set, indicates that your application can process suspend and resume events (which the Operating System sends to your application before sending it into the background or when bringing it into the foreground).

Note

If you set the `acceptSuspendResumeEvents` flag, you should also set the `doesActivateOnFGSwitch` flag. ◆

`canBackground` When set, indicates that your application wants to receive null event processing time while in the background. If your application has nothing to do in the background, you should not set this flag.

`doesActivateOnFGSwitch`
When set, indicates that your application takes responsibility for activating and deactivating any windows in response to a suspend or resume event. If the `acceptSuspendResumeEvents` flag is set, if the `doesActivateOnFGSwitch` flag is not set, and if your application is suspended, then your application receives an activate event following the suspend event. However, if you set the `doesActivateOnFGSwitch` flag, then your application won't receive activate events associated with operating-system events, and you must take care of activation and deactivation when it receives the corresponding suspend or resume event. This means that if a window of your application is frontmost, you should treat a suspend event as though a deactivate event were received as well (assuming that both the `doesActivateOnFGSwitch` and `acceptSuspendResumeEvents` flags are set). For example, you should hide scroll bars, hide any caret, and unhighlight any selected text if your application moves to the background. If you do not set this flag, the Process Manager creates an offscreen window to force the activate and deactivate events to occur.

`onlyBackground` When set, indicates that your application runs only in the background. Usually this is because it does not have a user interface and cannot run in the foreground.

`getFrontClicks` When set, indicates that your application is to receive the mouse-down and mouse-up events that are used to bring your application into the foreground when the user clicks in your

application's frontmost window. Typically, the user simply wants to bring your application into the foreground, so it is usually not desirable to receive the mouse events (which would probably move the insertion point or start drawing immediately, depending on the application). The Finder is one application, however, that has the `getFrontClicks` flag set.

When the user clicks in the front window of your application and causes a foreground switch, your application receives a resume event. Your application should activate its front window in response to the resume event. In this case if your application's `getFrontClicks` flag is not set, your application does not receive the associated mouse event that caused the foreground switch. If your application's `getFrontClicks` flag is set, your application does receive the associated mouse event.

Your application always receives the associated mouse event when the user clicks in one of your application's windows other than the front window and causes a foreground switch.

When your application receives a mouse-down event in System 7, your application can examine bit 0 of the `modifiers` field of the event record to determine if the mouse-down event caused a foreground switch. This information can be especially useful if your application sets its `getFrontClicks` flag. For example, your application can examine bit 0 to determine whether to process the mouse-down event (probably depending on whether the clicked item was visible before the foreground switch).

`acceptAppDiedEvents`

When set, indicates that your application is to be notified whenever an application launched by your application terminates or crashes. If the Process Manager is available, your application receives this information as an Apple event, the Application Died event. See the chapter "Process Manager" chapter in *Inside Macintosh: Processes* for more information about launching applications and receiving Application Died events.

Note

Some early versions of MultiFinder do not send application-died events, and your application should not depend on receiving them if it is running in System 6. These events are provided primarily for use by debuggers. ◆

`is32BitCompatible`

When set, indicates that your application can be run with the 32-bit Memory Manager. You should not set this flag unless you have thoroughly tested your application on a 32-bit system (such as a Macintosh IIci computer running System 7 in 32-bit mode or under A/UX).

`isHighLevelEventAware`

When set, indicates that your application can send and receive high-level events. If this flag is not set, the Event Manager does not give your application high-level events when you call

`WaitNextEvent`. There is no way to mask out specific types of high-level events; if this flag is set, your application receives all types of high-level events sent to your application.

Your application must support the four required Apple events if you set the `isHighLevelEventAware` flag. See *Inside Macintosh: Interapplication Communication* for information that describes how to respond to the four required Apple events.

`localAndRemoteHLEvents`
: When set, indicates that your application is to be visible to applications running on other computers on a network (in addition to applications running on the local computer). If this flag is not set, your application does not receive high-level events across a network.

`isStationeryAware`
: When set, indicates that your application can recognize stationery documents. If this flag is not set and the user opens a stationery document, the Finder duplicates the document and prompts the user for a name for the duplicate document. For information about how your application can use stationery documents, see the chapter "Finder Interface" in this book.

`useTextEditServices`
: When set, indicates that your application can use the inline text services provided by TextEdit. See *Inside Macintosh: Text* for information about the inline input capabilities of TextEdit.

The numbers you specify as your application's preferred and minimum memory sizes depend on the particular memory requirements of your application. Your application's memory requirements depend on the size of your application's static heap, dynamic heap, A5 world, and stack. (See "Introduction to Memory Management" in *Inside Macintosh: Memory* for complete details about these areas of your application's partition.)

The static heap size includes objects that are always present during the execution of the application—for example, code segments, Toolbox data structures for window records, and so on.

Dynamic heap requirements depend on how many objects are created on a per-document basis (which may vary in size proportionally with the document itself) and the number of objects that are required for specific commands or functions.

The size of the A5 world depends on the amount of global data and the number of intersegment jumps the application contains.

Finally, the stack contains variables, return addresses, and temporary information. The application stack size varies among computers, so you should base your values for the stack size according to the stack size required on a Macintosh Plus (8 KB). The Process Manager automatically adjusts your requested amount of memory to compensate for the different stack sizes on different machines. For example, if you request 512 KB, more stack space (approximately 16 KB) will be allocated on machines with larger default stack sizes.

Summary of the Event Manager

Pascal Summary

Constants

```
CONST {event codes}
      nullEvent              = 0;        {no other pending events}
      mouseDown              = 1;        {mouse button pressed}
      mouseUp                = 2;        {mouse button released}
      keyDown                = 3;        {key pressed}
      keyUp                  = 4;        {key released}
      autoKey                = 5;        {key repeatedly held down}
      updateEvt              = 6;        {window needs updating}
      diskEvt                = 7;        {disk inserted}
      activateEvt            = 8;        {activate/deactivate window}
      osEvt                  = 15;       {operating-system events }
                                         { (suspend, resume, mouse-moved)}
      kHighLevelEvent        = 23;       {high-level events }
                                         { (includes Apple events)}

      {event masks}
      everyEvent             = -1;       {every event}
      mDownMask              =  2;       {mouse-down event       (bit 1)}
      mUpMask                =  4;       {mouse-up event         (bit 2)}
      keyDownMask            =  8;       {key-down event         (bit 3)}
      keyUpMask              =  16;      {key-up event           (bit 4)}
      autoKeyMask            =  32;      {auto-key event         (bit 5)}
      updateMask             =  64;      {update event           (bit 6)}
      diskMask               =  128;     {disk-inserted event    (bit 7)}
      activMask              =  256;     {activate event         (bit 8)}
      highLevelEventMask     =  1024;    {high-level event       (bit 10)}
      osMask                 = -32768;   {operating-system event (bit 15)}

      {message codes for operating-system events}
      suspendResumeMessage   = $01;      {suspend or resume event}
      mouseMovedMessage      = $FA;      {mouse-moved event}
      osEvtMessageMask       = $FF000000;{can use to extract msg code}
```

```
{flags for suspend and resume events}
resumeFlag               = 1;           {resume event}
convertClipboardFlag     = 2;           {Clipboard conversion }
                                        { required}
{event message masks for keyboard events}
charCodeMask= $000000FF;                {use to get character code}
keyCodeMask = $0000FF00;                {use to get key code}
adbAddrMask = $00FF0000;                {ADB address for ADB keyboard}

{constants corresponding to bits in the modifiers field of event}
activeFlag  = 1;        {bit 0 of low byte--valid only for }
                        { activate and mouse-moved events}
btnState    = 128;      {bit 7 of low byte is mouse button state}
cmdKey      = 256;      {bit 0 of high byte}
shiftKey    = 512;      {bit 1 of high byte}
alphaLock   = 1024;     {bit 2 of high byte}
optionKey   = 2048;     {bit 3 of high byte}
controlKey  = 4096;     {bit 4 of high byte}

{high-level event posting options}
nAttnMsg                 = $00000001;   {give this message priority}
priorityMask             = $000000FF;   {mask for priority options}
nReturnReceipt           = $00000200;   {return receipt requested}
systemOptionsMask        = $00000F00;
receiverIDisTargetID     = $00005000;   {ID is port name & location}
receiverIDisSessionID    = $00006000;   {ID is PPC session ref number}
receiverIDisSignature    = $00007000;   {ID is creator signature}
receiverIDisPSN          = $00008000;   {ID is process serial number}
receiverIDMask           = $0000F000;

{class and ID values for return receipt}
HighLevelEventMsgClass   = 'jaym';      {event class of return receipt}
rtrnReceiptMsgID         = 'rtrn';      {event ID of return receipt}

{modifiers values in return receipt}
msgWasNotAccepted        = 0;           {recipient did not accept }
                                        { the message}
msgWasFullyAccepted      = 1;           {recipient accepted the}
                                        { entire message}
msgWasPartiallyAccepted = 2;            {recipient did not accept }
                                        { the entire message}
```

Data Types

```
TYPE
   EventRecord =
   RECORD
      what:         Integer;         {event code}
      message:      LongInt;         {event message}
      when:         LongInt;         {ticks since startup}
      where:        Point;           {mouse location}
      modifiers:    Integer;         {modifier flags}
   END;

   KeyMap = PACKED ARRAY[0..127] OF Boolean; {records state of keyboard}

   TargetID =
   RECORD
      sessionID:          LongInt;          {session reference number (not }
                                            { used if posting an event)}
      name:               PPCPortRec;       {port name}
      location:           LocationNameRec;  {location name}
      recvrName:          PPCPortRec;       {reserved}
   END;

   TargetIDPtr            = ^TargetID;      {pointer to a target ID record}
   TargetIDHdl            = ^TargetIDPtr;   {handle to a target ID record}

   HighLevelEventMsg =
   RECORD
      HighLevelEventMsgHeaderLength:   Integer;       {reserved}
      version:                         Integer;       {reserved}
      reserved1:                       LongInt;       {reserved}
      theMsgEvent:                     EventRecord;   {event record}
      userRefCon:                      LongInt;       {reference constant}
      postingOptions:                  LongInt;       {reserved}
      msgLength:                       LongInt;       {reserved}
   END;

   HighLevelEventMsgPtr = ^HighLevelEventMsg;
   HighLevelEventMsgHdl = ^HighLevelEventMsgPtr;

   GetSpecificFilterProcPtr = ProcPtr;
```

```
EvQEl =                         {event queue entry}
RECORD
   qLink:          QElemPtr;    {next queue entry}
   qType:          Integer;     {queue type (ORD(evType))}
   evtQWhat:       Integer;     {event code}
   evtQMessage:    LongInt;     {event message}
   evtQWhen:       LongInt;     {ticks since startup}
   evtQWhere:      Point;       {mouse location}
   evtQModifiers:  Integer;     {modifier flags}
END;

EvQElPtr = ^EvQEl;
```


Event Manager Routines

Receiving Events

```
FUNCTION WaitNextEvent       (eventMask: Integer; VAR theEvent: EventRecord;
                              sleep: LongInt; mouseRgn: RgnHandle): Boolean;
FUNCTION EventAvail          (eventMask: Integer; VAR theEvent: EventRecord)
                              : Boolean;
FUNCTION GetNextEvent        (eventMask: Integer; VAR theEvent: EventRecord)
                              : Boolean;
FUNCTION AcceptHighLevelEvent
                             (VAR sender: TargetID; VAR msgRefcon: LongInt;
                              msgBuff: Ptr; VAR msgLen: LongInt): OSErr;
FUNCTION GetSpecificHighLevelEvent
                             (aFilter: GetSpecificFilterProcPtr;
                              yourDataPtr: UNIV Ptr; VAR err: OSErr)
                              : Boolean;
PROCEDURE FlushEvents        (whichMask: Integer; stopMask: Integer);
PROCEDURE SystemClick        (theEvent: EventRecord; theWindow: WindowPtr);
PROCEDURE SystemTask;
FUNCTION SystemEvent         (theEvent: EventRecord): Boolean;
FUNCTION GetOSEvent          (mask: Integer; VAR theEvent: EventRecord)
                              : Boolean;
FUNCTION OSEventAvail        (mask: Integer; VAR theEvent: EventRecord)
                              : Boolean;
PROCEDURE SetEventMask       (theMask: Integer);
FUNCTION GetEvQHdr           : QHdrPtr;
```

Sending Events

```
FUNCTION PostHighLevelEvent  (theEvent: EventRecord; receiverID: Ptr;
                              msgRefcon: LongInt; msgBuff: Ptr;
                              msgLen: LongInt; postingOptions: LongInt)
                              : OSErr;
FUNCTION PPostEvent          (eventCode: Integer; eventMsg: LongInt;
                              VAR qEl: EvQElPtr): OSErr;
FUNCTION PostEvent           (eventNum: Integer; eventMsg: LongInt): OSErr;
```

Converting Process Serial Numbers and Port Names

```
FUNCTION GetProcessSerialNumberFromPortName
                             (portName: PPCPortRec;
                              VAR PSN: ProcessSerialNumber): OSErr;
FUNCTION GetPortNameFromProcessSerialNumber
                             (VAR portName: PPCPortRec;
                              PSN: ProcessSerialNumber): OSErr;
```

Reading the Mouse

```
PROCEDURE GetMouse           (VAR mouseLoc: Point);
FUNCTION Button              : Boolean;
FUNCTION StillDown           : Boolean;
FUNCTION WaitMouseUp         : Boolean;
```

Reading the Keyboard

```
PROCEDURE GetKeys            (VAR theKeys: KeyMap);
{the KeyTranslate function is also available as the KeyTrans function}
FUNCTION KeyTranslate        (transData: Ptr; keycode: Integer;
                              VAR state: LongInt): LongInt;
```

Getting Timing Information

```
FUNCTION TickCount           : LongInt;
FUNCTION GetDblTime          : LongInt;
FUNCTION GetCaretTime        : LongInt;
```

Application-Defined Routine

Filter Function for Searching the High-Level Event Queue

```
FUNCTION MyFilter            (yourDataPtr: Ptr;
                              msgBuff: HighLevelEventMsgPtr;
                              sender: TargetID): Boolean;
```

C Summary

Constants

```
enum {
      /*event codes*/
      nullEvent               = 0,  /*no other pending events*/
      mouseDown               = 1,  /*mouse button pressed*/
      mouseUp                 = 2,  /*mouse button released*/
      keyDown                 = 3,  /*key pressed*/
      keyUp                   = 4,  /*key released*/
      autoKey                 = 5,  /*key repeatedly held down*/
      updateEvt               = 6,  /*window needs updating*/
      diskEvt                 = 7,  /*disk inserted*/
      activateEvt             = 8,  /*activate/deactivate window*/
      osEvt                   = 15, /*operating-system events */
                                    /* (suspend, resume, mouse-moved)*/

      /*event masks*/
      mDownMask               =  2,         /*mouse-down         (bit 1)*/
      mUpMask                 =  4,         /*mouse-up           (bit 2)*/
      keyDownMask             =  8,         /*key-down           (bit 3)*/
      keyUpMask               =  16,        /*key-up             (bit 4)*/
      autoKeyMask             =  32,        /*auto-key           (bit 5)*/
      updateMask              =  64,        /*update             (bit 6)*/
      diskMask                =  128,       /*disk-inserted      (bit 7)*/
      activMask               =  256,       /*activate           (bit 8)*/
      highLevelEventMask      =  1024,      /*high-level         (bit 10)*/
      osMask                  = -32768      /*operating-system   (bit 15)*/
};

enum {
      everyEvent                 = -1,         /*every event*/

      /*event message masks for keyboard events*/
      charCodeMask      = 0x000000FF,       /*use to get character code*/
      keyCodeMask       = 0x0000FF00,       /*use to get key code*/
      adbAddrMask       = 0x00FF0000,       /*ADB address if ADB keyboard*/
      osEvtMessageMask  = 0xFF000000,       /*can use to extract msg code*/
```

```
    /*message codes for operating-system events*/
    mouseMovedMessage       = 0xFA,        /*mouse-moved event*/
    suspendResumeMessage    = 0x01,        /*suspend or resume event*/
    /*flags for suspend and resume events*/
    resumeFlag              = 1,           /*resume event*/
    convertClipboardFlag    = 2,           /*Clipboard conversion */
                                           /* required*/

    /*constants corresponding to bits in the modifiers field of event*/
    activeFlag  = 1,        /*bit 0 of low byte--valid only for */
                            /* activate and mouse-moved events*/
    btnState    = 128,      /*bit 7 of low byte is mouse button state*/
    cmdKey      = 256,      /*bit 0 of high byte*/
    shiftKey    = 512,      /*bit 1 of high byte*/
    alphaLock   = 1024,     /*bit 2 of high byte*/
    optionKey   = 2048,     /*bit 3 of high byte*/
    controlKey  = 4096      /*bit 4 of high byte*/
};
enum {
    kHighLevelEvent         = 23, /*event code for high-level events */
                                  /* (includes Apple events)*/
    /*high-level event posting options*/
    receiverIDMask          = 0x0000F000,  /*mask for receiver ID bits*/
    receiverIDisPSN         = 0x00008000,  /*ID is proc serial number*/
    receiverIDisSignature   = 0x00007000,  /*ID is creator signature*/
    receiverIDisSessionID   = 0x00006000,  /*ID is session ref number*/
    receiverIDisTargetID    = 0x00005000,  /*ID is port name & location*/

    systemOptionsMask       = 0x00000F00,
    nReturnReceipt          = 0x00000200,  /*return receipt requested*/
    priorityMask            = 0x000000FF,  /*mask for priority options*/
    nAttnMsg                = 0x00000001,  /*give this message priority*/

    /*class and ID values for return receipt*/
    #define HighLevelEventMsgClass 'jaym'
    #define rtrnReceiptMsgID 'rtrn'

    /*modifiers values in return receipt*/
    msgWasPartiallyAccepted = 2,
    msgWasFullyAccepted     = 1,
    msgWasNotAccepted       = 0
};
```

Data Types

```
struct EventRecord {
      short    what;                    /*event code*/
      long     message;                 /*event message*/
      long     when;                    /*ticks since startup*/
      Point    where;                   /*mouse location*/
      short    modifiers;               /*modifier flags*/
   };
typedef struct EventRecord EventRecord;

typedef long KeyMap[4];                 /*records state of keyboard*/

struct TargetID {
      long               sessionID;     /*session reference number (not */
                                        /* used if posting an event)*/
      PPCPortRec         name;          /*port name*/
      LocationNameRec    location;      /*location name*/
      PPCPortRec         recvrName;     /*reserved*/
   };

typedef struct TargetID TargetID;
typedef TargetID *TargetIDPtr, **TargetIDHdl;

struct HighLevelEventMsg {
      unsigned short    HighLevelEventMsgHeaderLength;   /*reserved*/
      unsigned short    version;                         /*reserved*/
      unsigned long     reserved1;                       /*reserved*/
      EventRecord       theMsgEvent;                     /*event record*/
      unsigned long     userRefCon;                      /*ref constant*/
      unsigned long     postingOptions;                  /*reserved*/
      unsigned long     msgLength;                       /*reserved*/
   };

typedef struct HighLevelEventMsg HighLevelEventMsg;
typedef HighLevelEventMsg *HighLevelEventMsgPtr, **HighLevelEventMsgHdl;

   struct EvQEl {       /*event queue entry*/
      QElemPtr    qLink;            /*next queue entry*/
      short       qType;            /*queue type (evType)*/
      short       evtQWhat;         /*event code*/
      long        evtQMessage;      /*event message*/
      long        evtQWhen;         /*ticks since startup*/
```

```
      Point       evtQWhere;        /*mouse location*/
      short       evtQModifiers;    /*modifier flags*/
   };
   typedef struct EvQEl EvQEl;
   typedef EvQEl *EvQElPtr;

typedef pascal Boolean (*GetSpecificFilterProcPtr)
                              (void *yourDataPtr,
                              HighLevelEventMsgPtr msgBuff,
                              const TargetID *sender);
```

Event Manager Routines

Receiving Events

```
pascal Boolean WaitNextEvent(short eventMask, EventRecord *theEvent,
                             unsigned long sleep, RgnHandle mouseRgn);
pascal Boolean EventAvail   (short eventMask, EventRecord *theEvent);
pascal Boolean GetNextEvent (short eventMask, EventRecord *theEvent);
pascal OSErr AcceptHighLevelEvent
                            (TargetID *sender, unsigned long *msgRefcon,
                             Ptr msgBuff, unsigned long *msgLen);
pascal Boolean GetSpecificHighLevelEvent
                            (GetSpecificFilterProcPtr aFilter,
                             void *yourDataPtr, OSErr *err);
pascal void FlushEvents     (short whichMask, short stopMask);
pascal void SystemClick     (const EventRecord *theEvent,
                             WindowPtr theWindow);
pascal void SystemTask      (void);
pascal Boolean SystemEvent  (const EventRecord *theEvent);
pascal Boolean GetOSEvent   (short mask, EventRecord *theEvent);
pascal Boolean OSEventAvail (short mask, EventRecord *theEvent);
pascal void SetEventMask    (short theMask);
#define GetEvQHdr()         ((QHdrPtr) 0x014A)
```

Sending Events

```
pascal OSErr PostHighLevelEvent
                            (const EventRecord *theEvent,
                             unsigned long receiverID,
                             unsigned long msgRefcon, Ptr msgBuff,
                             unsigned long msgLen,
                             unsigned long postingOptions);
```

```
pascal OSErr PPostEvent     (short eventCode, long eventMsg, EvQElPtr *qEl)
pascal OSErr PostEvent      (short eventNum, long eventMsg);
```

Converting Process Serial Numbers and Port Names

```
pascal OSErr GetPortNameFromProcessSerialNumber
                            (PPCPortPtr portName,
                             const ProcessSerialNumberPtr pPSN);
pascal OSErr GetProcessSerialNumberFromPortName
                            (const PPCPortPtr portName,
                             ProcessSerialNumberPtr pPSN);
```


Reading the Mouse

```
pascal void GetMouse        (Point *mouseLoc);
pascal Boolean Button       (void);
pascal Boolean StillDown    (void);
pascal Boolean WaitMouseUp  (void);
```

Reading the Keyboard

```
pascal void GetKeys         (KeyMap theKeys);
{the KeyTranslate function is also available as the KeyTrans function}
pascal long KeyTranslate    (const void *transData, short keycode,
                             long *state);
```

Getting Timing Information

```
pascal unsigned long TickCount
                            (void);
#define GetDblTime()        (* (unsigned long*) 0x02F0)
#define GetCaretTime()      (* (unsigned long*) 0x02F4)
```

Application-Defined Routine

Filter Function for Searching the High-Level Event Queue

```
pascal Boolean MyFilter     (void *yourDataPtr,
                             HighLevelEventMsgPtr msgBuff,
                             const TargetID *sender);
```

Assembly-Language Summary

Data Structures

Event Data Structure

0	what	word	event code
2	message	long	event message
6	when	long	ticks since startup
10	where	long	mouse location
14	modifiers	word	modifier flags

Target ID Data Structure

0	sessionID	long	session reference number (not used if posting event)
4	name	68 bytes	port name (specified in a PPCPortRec data structure)
72	location	34 bytes	location name (specified in a LocationNameRec)
106	recvrName	68 bytes	reserved

High-Level Event Message Data Structure

0	HighLevelEventMsgHeaderLength	word	reserved
2	version	word	reserved
4	reserved1	long	reserved
8	theMsgEvent	16 bytes	event record
22	userRefCon	long	reference constant
26	postingOptions	long	reserved
30	msgLength	long	reserved

Event Queue Header Data Structure

0	qLink	long	next queue entry
4	qType	word	queue type
6	evtQWhat	word	event code
8	evtQMessage	long	event message
12	evtQWhen	long	ticks since startup
16	evtQWhere	long	mouse location
20	evtQModifiers	word	modifier flags

Trap Macros

Trap Macros Requiring Routine Selectors

_OSDispatch

Selector	Routine
$0033	AcceptHighLevelEvent
$0034	PostHighLevelEvent

Selector	Routine
$0035	GetProcessSerialNumberFromPortName
$0045	GetSpecificHighLevelEvent
$0046	GetPortNameFromProcessSerialNumber

Trap Macros Requiring Register Setup

Trap macro name	Registers on entry	Registers on exit
_FlushEvents	D0: event mask (low-order word) stop mask (high-order word)	D0: 0 if all events were removed from the queue, or the event code of the event that stopped the search (low-order word)
_GetOSEvent	A0: address of event record D0: event mask (low-order word)	D0: 0 if GetOSEvent returns any event other than a null event, or –1 if it returns a null event (low-order byte)
_OSEventAvail	A0: address of event record D0: event mask (low-order word)	D0: 0 if OSEventAvail returns any event other than a null event, or –1 if it returns a null event (low-order byte)
_PostEvent	A0: event code (low-order word) D0: event message (long word)	D0: result code (low-order word)

Global Variables

CaretTime	The suggested difference in ticks that should exist between blinks of the caret in editable text.
DoubleTime	The suggested maximum difference in ticks that should exist between the time of a mouse-up event and a mouse-down event for your application to consider those two mouse events a double click.
EventQueue	The header of the event queue.
KeyRepThresh	The value of the auto-key rate (the amount of time, in ticks, that must elapse before the Event Manager generates a subsequent auto-key event).
KeyThresh	The value of the auto-key threshold (the amount of time, in ticks, that must elapse before the Event Manager generates an auto-key event).
ScrDmpEnable	A byte that, if set to 0, disables the Event Manager’s processing of Command–Shift–number key combinations with numbers 3 through 9.
SEvtEnb	A byte that, if set to 0, causes the SystemEvent function to always return FALSE.
SysEvtMask	The system event mask of the current application.
Ticks	A long integer that indicates the current number of ticks since the system last started up.

Result Codes

`noErr`	0	No error
`procNotFound`	–600	No eligible process with specified process serial number
`bufferIsSmall`	–607	Buffer is too small
`noOutstandingHLE`	–608	No outstanding high-level event
`connectionInvalid`	–609	Connection is invalid
`noUserInteractionAllowed`	–610	Cannot interact directly with user
`noPortErr`	–903	Invalid port name