

# 2

## LOW-LEVEL AND OPERATING SYSTEM EVENTS

*Includes Demonstration Program LowEvents*

### Introduction

All Macintosh applications share one essential characteristic: they are all event-driven. At its most basic level, an application's general strategy is to retrieve an **event** (such as a key press or a mouse click), process it, retrieve the next event, process it, and so on indefinitely until the user quits the application. The core of all Macintosh applications is thus the **main event loop** (see Fig 1).

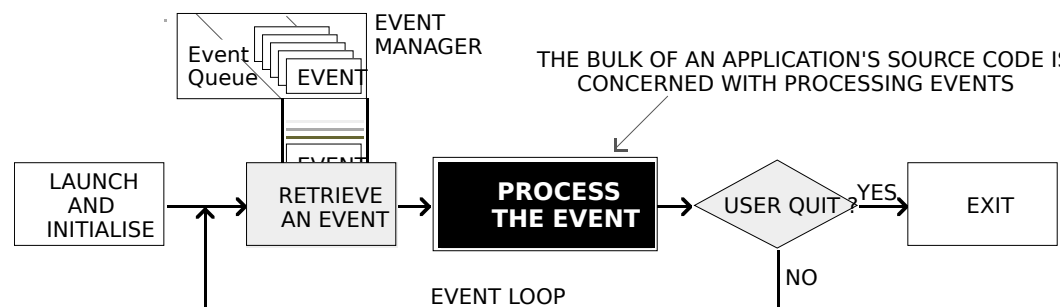


FIG 1 - THE MAIN EVENT LOOP

If no events are pending for the active application at a particular time, that application can choose to relinquish control of the CPU (central processing unit, or microprocessor) for a specified amount of time before again checking to see whether an event has occurred. Events are retrieved, and processor time is relinquished, using the `WaitNextEvent` function.

Information about a received event is placed in an **event structure**. An application may specify which types of events it wants to receive by including an **event mask** as a parameter in certain Event Manager functions.

### Processes and Events

The subject of **processes** is of some relevance to the subject of events.

When multiple applications are open, the user chooses one, and only one, to interact with at any given time. This active application is known as the **foreground process**. The remaining open applications, if any, are known as **background processes**. The user can

bring a background process to the foreground by clicking in one of its windows or by choosing its item in the Application menu. When an application is switched between background and foreground in this way, a **major switch** is said to have occurred.

The foreground process has first priority for accessing the CPU, background processes accessing the CPU only when the foreground process yields time to them. Any application whose 'SIZE' resource (see below) specifies that it should receive null events (see below) when it is in the background is eligible for CPU time when it is not in the foreground. A **minor switch** is said to have occurred when a background process gains a period of CPU access without being brought to the foreground.

## ***Categories of Events***

---

An application can receive many types of events. It can also send certain types of events to other applications. Events are broadly categorised as **low-level** events, **Operating System** events, and **high-level** events. The high-level event is the category of event used to send events to other applications.

Of the three categories, this chapter is concerned only with low-level events and Operating System events. High-level events are addressed at Chapter 10 — Apple Events.

## ***Low Level Events***

---

Low-level events, which are sent to the application by the Toolbox Event Manager, are originated by such low-level occurrences as pressing and releasing a key, pressing and releasing the mouse button and inserting a disk.

The Window Manager also originates low-level events, specifically, two events relating to an application's windows:

- The **activate** event, which has to do with informing the application to make changes to the appearance of a window depending on whether or not it is the frontmost window.
- The **update** event, which has to do with informing the application to re-draw a window's contents.

The event which reports that the Event Manager has no other events to report (the **null** event) is also categorised as a low-level event.

Low-level events, except for update events and null events, are invariably directed to the foreground process only.

## ***Operating System Events***

---

Operating system events are returned to the application when the operating status of an application changes. For example, when an application is about to be switched to the background, the Process Manager sends it a **suspend** event. Then, when the application is switched back to the foreground, the Process Manager sends it a **resume** event. Another Operating System event, called the **mouse-moved** event, is sent when the mouse pointer is moved outside a designated region.

Operating system events are invariably directed to the foreground process only.

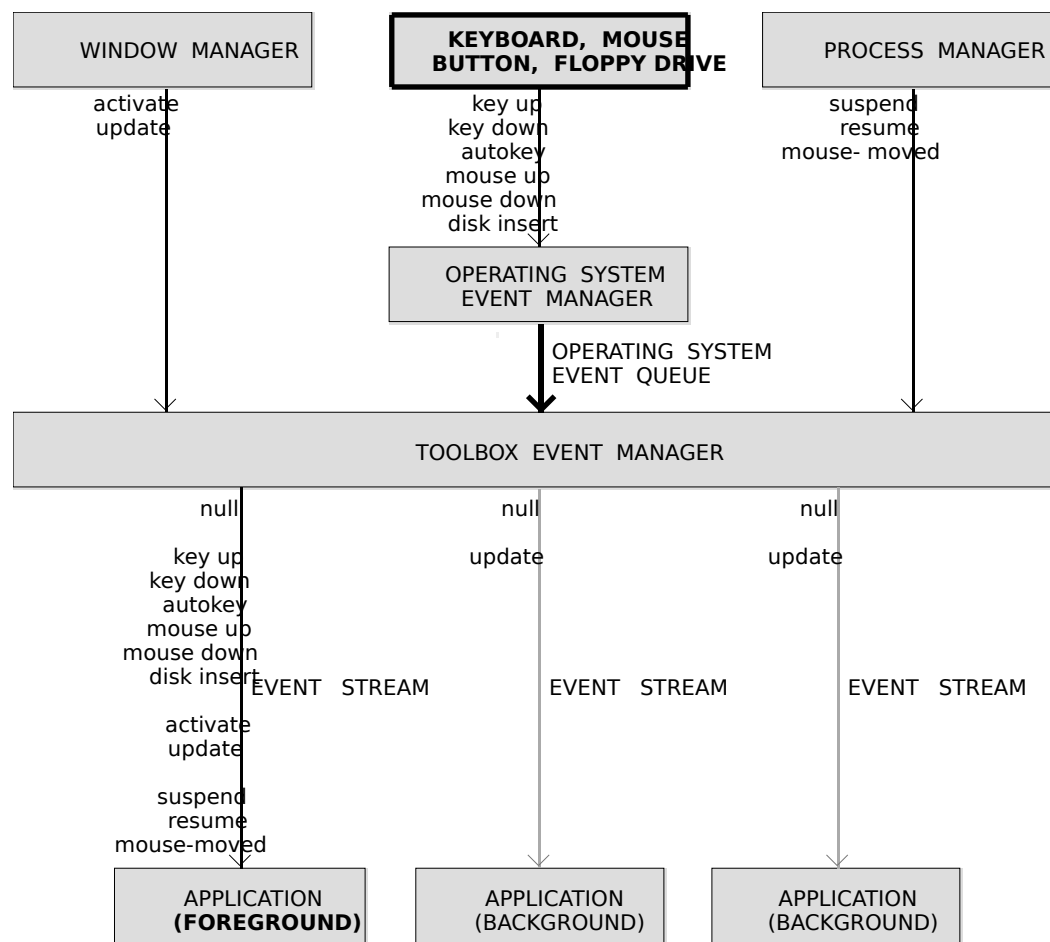
## ***Low-Level and Operating System Events, System Software, and Applications***

---

Fig 2 shows the relationship between low-level and Operating System events, system software managers and open applications.

In Fig 2, note that, in addition to the Operating System event queue created by the Operating System Event Manager, the Toolbox Event Manager maintains a separate event stream for each open application. An event stream contains only those events which are available to the related application. Also note that, when an application is in the background, its event stream can contain only update events and null events, and then only if the application's 'SIZE' resource so specifies.<sup>1</sup>

A maximum of 48 events (Mac OS 8.5 and later) or 20 events (versions of the Mac OS earlier than 8.5) can be pending in the Operating System event queue. If the queue becomes full, the oldest event is discarded to make room for the new.



**FIG 2 - LOW-LEVEL AND OPERATING SYSTEM EVENTS**

## ***Priority of Events***

In general, the Event Manager returns events to the application in the order low-level events, Operating System events, and high-level events. In detail, the order of priority is:

- Activate events.
- Mouse-down, mouse-up, key-down, key-up and disk events in FIFO (first in, first out) order.
- Auto-key events.
- Update events, in front-to-back order of windows.
- Operating system events.
- High-level events.
- Null events.

<sup>1</sup> An application in the background can also receive high-level events. (See Chapter 10 — Apple Events.)

# Obtaining Information About Events

---

## The Event Structure

---

The Event Manager continually captures information about each keystroke, mouse click, etc., and puts information about each event into an **event structure**. As more actions occur, additional event structures are created and joined to the first, forming an event queue.

The EventRecord data type defines the event structure:

```
struct EventRecord
{
    EventKind      what;
    UInt32         message;
    UInt32         when;
    Point          where;
    EventModifiers modifiers;
};

typedef struct EventRecord EventRecord;
```

## Field Descriptions

---

**what** Indicates the type of event received, which may be represented by one of the following constants:

nullEvent	= 0	No other pending events.
mouseDown	= 1	Mouse button pressed.
mouseUp	= 2	Mouse button released.
keyDown	= 3	Character key pressed.
keyUp	= 4	Character key released.
autoKey	= 5	Key held down in excess of autoKey threshold.
updateEvt	= 6	Window needs to be redrawn.
diskEvt	= 7	Disk was inserted.
activateEvt	= 8	Activate/deactivate window.
osEvt	= 15	Operating system event (suspend, resume or mouse moved).

**message** Contains additional information about the event. The content of this field depends on the event type, as follows:

<b>Event Type</b>	<b>Contents of message Field</b>
nullEvent	Undefined.
mouseDown	
mouseUp	
keyDown	Bits 0-7 = character code. Bits 8-15 = virtual key code.
keyUp	Bits 16-23 = For Apple Desktop Bus keyboards, the ADB address of the keyboard where the event occurred.
autoKey	
updateEvt	Pointer to the window to update, activate or deactivate. (For an activateEvt, Bit 0 of the modifiers field indicates whether to activate or deactivate the window.)
activateEvt	
diskEvt	Bits 0-15 = drive number. Bits 16-31 = File Manager result code.
osEvt    resume	Bits 24-31 = suspendResumeMessage constant. Also, a 1 in Bit 0 to indicate that the event is a resume event. Also, a 0 or a 1 in Bit 1 to indicate if clipboard conversion is required.
osEvt    suspend	Bits 24-31 = suspendResumeMessage constant. Also, a 0 in Bit 0 to indicate that the event is a suspend event.
osEvt    mouse-moved	Bits 24-31 = mouseMovedMessage constant.

The following constants may be used to extract certain data from, and to test certain bits in, the message field:

charCodeMask	= 0x000000FF	Mask to extract ASCII character code.
keyCodeMask	= 0x0000FF00	Mask to extract key code.
osEvtMessageMask	= 0xFF000000	Mask to extract OS event message code.
mouseMovedMessage	= 0x00FA	osEvts: mouse-moved event?
suspendResumeMessage	= 0x0001	osEvts: suspend/resume event?
resumeFlag	= 1	osEvts: resume event or suspend event?
convertClipboardFlag	= 2	osEvts: convert clipboard?

For example, the following code example determines whether an event which has previously been determined to be an Operating System event is a resume event, a suspend event, or a mouse-moved event. In this example, the high byte of the message field is examined to determine whether it contains suspendResumeMessage (0x0001) or mouseMovedMessage (0x00FA). If it contains suspendResumeMessage, Bit 0 is then examined to determine whether the event is a suspend event or a resume event.

```
switch((eventRecPtr->message >> 24) & 0x000000FF)
{
    case suspendResumeMessage:
        if((eventRecPtr->message & resumeFlag) == 1)
            // This is a resume event.
        else
            // This is a suspend event.
            break;

    case mouseMovedMessage:
        // This is a mouse-moved event.
        break;
}
```

when Time the event was posted, in ticks since system startup. Typically, this is used to establish the time between mouse clicks.

where Location of cursor, in global coordinates<sup>2</sup>, at the time the event was posted.

modifiers 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 indicates whether the window should be activated or deactivated.

For mouse-down events, this field indicates whether the event caused the application to be switched to the foreground.

### **Bit Description**

Bit 0	activateEvt:	1 if the window pointed to in the message field should be activated.
		0 if the window pointed to in the message field should be deactivated.
	mouseDown:	1 if the event caused the application to be switched to the foreground,
		otherwise 0.
Bit 7		1 if mouse button was up, 0 if not.
Bit 8		1 if Command key down, 0 if not.
Bit 9		1 if Shiftkey down, 0 if not.
Bit 10		1 if Caps Lock key down, 0 if not.
Bit 11		1 if Option key down, 0 if not.
Bit 12		1 if Control key down, 0 if not.
Bit 13		1 if Right Shift Key down, 0 if not.
Bit 14		1 if Right Option Key down, 0 if not.
Bit 15		1 if Right Control Key down, 0 if not.

<sup>2</sup> Global coordinates are explained at Chapter 4 — Windows (see Fig 8).

The following constants may be used as masks to test the setting of the various bits in the modifiers field:

activeFlag	= 0x0001	Window is to be activated? (activateEvt). Foreground switch? (mouseDown).
btnState	= 0x0080	Mouse button up?
cmdKey	= 0x0100	Command key down?
shiftKey	= 0x0200	Shift key down?
alphaLock	= 0x0400	Caps Lock key down?
optionKey	= 0x0800	Option key down?
controlKey	= 0x1000	Control key down?
rightShiftKey	= 0x2000	Right Shift Key down?
rightOptionKey	= 0x4000	Right Option Key down?
rightControlKey	= 0x8000	Right Control Key down?

For example, the following code example determines whether an event which has previously been determined to be an activate event is intended to signal the application to activate or deactivate the window referenced in the message field:

```
Boolean becomingActive;

becomingActive = ((eventRecPtr->modifiers & activeFlag) == activeFlag);

if(becomingActive)
    // Application-defined window activation code here.
else
    // Application-defined window deactivation code here.
```

### Event Structure Examples - Diagrammatic

Fig 3 is a diagrammatic representation of the contents of some typical event structures.

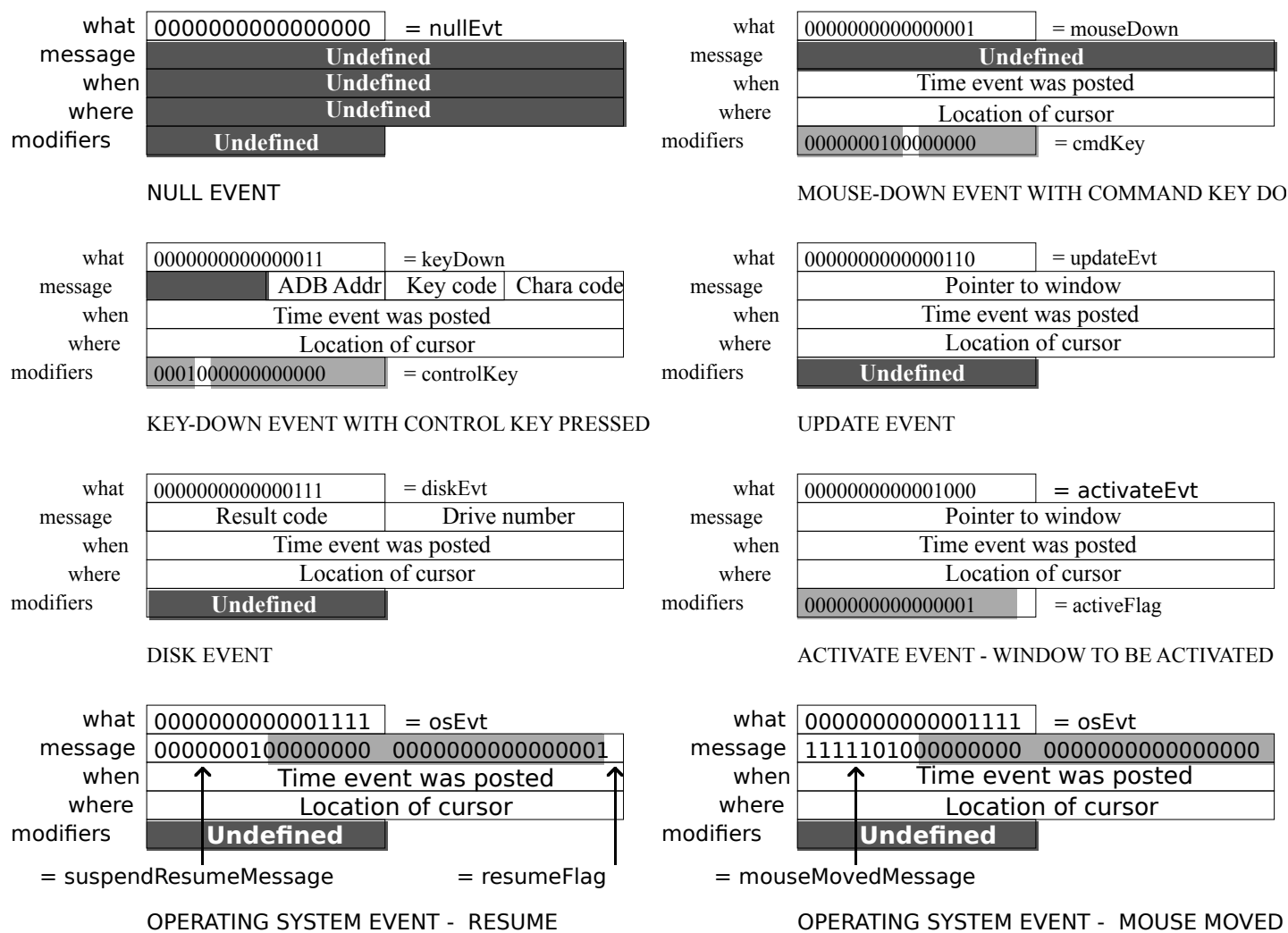


FIG 3 - EXAMPLES OF CONTENTS OF AN EVENT RECORD

## The WaitNextEvent Function

---

The `WaitNextEvent` function retrieves events from the Event Manager. If no events are pending for the application, the `WaitNextEvent` function may allocate processor time to other applications. When `WaitNextEvent` returns, the event structure contains information about the retrieved event, if any.

`WaitNextEvent` returns `true` if it retrieves any event other than a null event. If there are no events of the types specified in the `eventMask` parameter (other than null events), `false` is returned.

Boolean      **WaitNextEvent**(EventMask *eventMask*, EventRecord \**theEvent*, UInt32 *sleep*, RgnHandle *mouseRgn*)

**Returns:** A return code: 0 = null event; 1 = event returned.

*eventMask*      A 16 bit binary mask which may be used to mask out the receipt of certain events.

The following constants are defined in `Events.h`:

<code>mDownMask</code>	= 0x0002	Mouse button pressed.
<code>mUpMask</code>	= 0x0004	Mouse button released.
<code>keyDownMask</code>	= 0x0008	Key pressed.
<code>keyUpMask</code>	= 0x0010	Key released.
<code>autoKeyMask</code>	= 0x0020	Key repeatedly held down.
<code>updateMask</code>	= 0x0040	Window needs updating.
<code>diskMask</code>	= 0x0080	Disk inserted.
<code>activMask</code>	= 0x0100	Activate/deactivate window.
<code>highLevelEventMask</code>	= 0x0400	High-level events (includes <code>AppleEvents</code> ).
<code>osMask</code>	= 0x8000	Operating system events (suspend, resume).
<code>everyEvent</code>	= 0xFFFF	All of the above.

Masked events are not removed from the event stream by the `WaitNextEvent` call. To remove events from the Operating System event queue, call `FlushEvents` with the appropriate mask.

*theEvent*      Address of a 16-byte event structure.

*sleep*          The amount of time, in ticks, the application agrees to relinquish the processor if no events are pending for it. When that time expires, or when an event becomes available for the application, the Process Manager schedules the application for execution.

**Note:** If your application needs to perform small subsidiary tasks at frequent intervals, you must ensure that your application regains control of the CPU at frequent intervals. For example, if the user is editing text and the application needs to blink the caret, the application must specify a value for the `sleep` parameter which is equal to the desired blink rate.

*mouseRgn*      The screen **region** inside which the Event Manager does not generate mouse-moved events. The region should be specified in global coordinates. If the user moves the cursor outside this region and the application is the foreground process, the Event Manager reports mouse-moved events.

If `NULL` is passed as this parameter, the Event Manager does not return mouse-moved events.

Before returning to the application, `WaitNextEvent` performs certain additional processing and may, in fact, intercept the received event so that it is never received by your application. As will be seen, key-up and key-down events are intercepted in this way in certain circumstances.

## Flushing the Operating System Event Queue

---

Immediately after application launch, the `FlushEvents` function should be called to empty the Operating System event queue of any low-level events left unprocessed by another application, for example, any mouse-down or keyboard events that the user may have entered while the Finder launched the application.

## Handling Events

---

### Handling Mouse Events<sup>3</sup>

---

Your application receives mouse-down events only when it is the foreground process and the user clicks in a window belonging to the application or in the menu bar, (If the user clicks in a window belonging to another application, the Event Manager sends your application a suspend event.)

When your application receives a mouse-down event, you need to first determine where the cursor was at the time the mouse button was pressed. A call to `FindWindow` will determine:

- Which of your application's windows, if any, the mouse button was pressed in.
- Which window **part** the mouse button was pressed in. In this context, a window part includes the menu bar as well as various regions within the window.

The following constants, defined in `MacWindows.h`, may be used to test the value returned by `FindWindow`:

<code>inDesk</code>	= 0	In none of the following.
<code>inNoWindow</code>	= 0	In none of the following.
<code>inMenuBar</code>	= 1	In the menu bar.
<code>inSysWindow</code>	= 2	In a desk accessory window.
<code>inContent</code>	= 3	Anywhere in the content region except the grow region if the window is active. Anywhere in the content region including the grow region if the window is inactive.
<code>inDrag</code>	= 4	In the drag (title bar) region.
<code>inGrow</code>	= 5	In the grow region (active window only).
<code>inGoAway</code>	= 6	In the go-away region (active window only).
<code>inZoomIn</code>	= 7	In the zoom-in region (active window only).
<code>inZoomOut</code>	= 8	In the zoom-out region (active window only).
<code>inCollapseBox</code>	= 11	In the collapse region (active window only).
<code>inProxyIcon</code>	= 12	In the window proxy icon (active window only).

### Historical Notes

The `inSysWindow` constant relates to mouse-down events in an old-style desk accessory. Old-style desk accessories were designed for early versions of the Macintosh system software which did not support multitasking, and which were thus launched inside the running application's partition. (Desk accessories written for the multitasking environment are really just small applications which are launched in their own partition and behave in every respect like a normal application.) The application responded to such mouse-downs by calling `SystemClick`, which routed the event to the desk accessory for further handling. In the current day, it is all but inconceivable that a machine running the Appearance Manager would also harbour an old-style desk accessory. Accordingly, the demonstration programs accompanying this version of Macintosh C do not test for `inSysWindow`.

---

<sup>3</sup> Events related to the *movement* 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.



The `inCollapseBox` constant was introduced with Mac OS 8 and the Appearance Manager. Previously, windows did not contain collapse boxes. When the Appearance Manager is not available, `inCollapseBox` will not be returned by `FindWindow`.

The `inProxyIcon` constant was introduced with Mac OS 8.5. (See Chapter 4B — More on Windows — Mac OS 8.5 Window Manager.)

## ***In the Content Region***

---

If the cursor was in the content region<sup>4</sup> of the active window, your application should perform any actions appropriate to the application. If the window has scroll bars, and since scroll bars actually occupy part of the content region, your application should first determine whether the cursor was in the scroll bars — or, indeed, in any other control — and respond appropriately.

## ***In the Drag Bar, Grow Box, Zoom Box, Go Away Box, Collapse Box, or Window Proxy Icon***

---

If the cursor was in one of the non-content regions of the active window, your application should perform the appropriate actions for that region as follows:

- **Drag Bar.** If the cursor was in the drag bar:
  - **Mac OS 8.1 and Earlier.** Under Mac OS 8.1 and earlier, your application should call `DragWindow` to allow the user to drag the window to a new location. `DragWindow` retains control until the mouse button is released.
  - **Mac OS 8.5 and Later.** Under Mac OS 8.5 and later, your application should first call `IsWindowPathSelect` to determine whether the mouse-down event should activate the window path pop-up menu. If `IsWindowPathSelect` returns `true`, your application should then call `WindowPathSelect` to display the menu, otherwise your application should call `DragWindow` to allow the user to drag the window to a new location. (See Chapter 4B — More on Windows — Mac OS 8.5 Window Manager.)
- **Grow Box.** If the cursor was in the grow box:
  - **Mac OS 8.1 and Earlier.** Under Mac OS 8.1 and earlier, your application should first call `GrowWindow` to track user actions while the mouse button remains down. When `GrowWindow` returns (that is, when the mouse button is released), `SizeWindow` should be called to re-draw the window in its new size.
  - **Mac OS 8.5 and Later.** Under Mac OS 8.5 and later, your application should call `ResizeWindow`, which tracks user actions while the mouse button remains down. When the mouse button is released, `ResizeWindow` draws the window in its new size. (See Chapter 4B — More on Windows — Mac OS 8.5 Window Manager.)
- **Zoom Box.** If the cursor was in the zoom box:
  - **Mac OS 8.1 and Earlier.** Under Mac OS 8.1 and earlier, your application should call `TrackBox` to track the mouse while the mouse button remains down. `TrackBox` returns `true` if the cursor is within the zoom box when the button is released, and `false` otherwise. If `true` is returned, the window content region

---

<sup>4</sup> The content region is the part of the window in which an application displays the contents of a document and the window's controls (for example, scroll bars).

should be erased, `ZoomWindow` should be called to redraw the window in its newly zoomed state, and the window's content region should be redrawn.

- **Mac OS 8.5 and Later.** Under Mac OS 8.5 and later, your application should call `IsWindowInStandardState` to determine whether the window is currently in the standard state or the user state. `ZoomWindowIdeal` should then be called to zoom the window to the appropriate state, and the window's content region should be redrawn. (See Chapter 4B — More on Windows — Mac OS 8.5 Window Manager.)
- **Go Away Box.** If the cursor was in the go-away box, your application should call `TrackGoAway` to track user actions while the mouse button remains down. `TrackGoAway`, which returns only when the mouse is released, returns `true` if the cursor is still inside the close box when the mouse button is released, and `false` otherwise.
- **Collapse Box.** If the cursor was in the collapse box, your application should do nothing, because the system will collapse the window for you.
- **Window Proxy Icon (Applicable Under Mac OS 8.5 and Later).** If the cursor was in the window proxy icon, your application should call `TrackWindowProxyDrag`, which handles all aspects of the drag process while the user drags the proxy icon. (See Chapter 4B — More on Windows — Mac OS 8.5 Window Manager.)

### **In the Menu Bar**

If the cursor was in the menu bar, your application should first adjust its menus, that is, enable and disable items and set marks (for, example, checkmarks) based on the context of the active window. It should then call `MenuSelect`, which handles all user action until the mouse button is released.

When the mouse button is released, `MenuSelect` returns a long integer containing, ordinarily, the menu ID in the high word and the chosen menu item in the low word. However, if the cursor was outside the menu when the button was released, the high word contains 0.

### **In an Inactive Application Window**

If the mouse click was in an inactive application window, `FindWindow` can return only the `inContent` or `inDrag` constant. If `inContent` is reported, your application should bring the inactive window to the front using `SelectWindow`.

Ordinarily, the first click in an inactive window should simply activate the window and do nothing more. However, if the mouse click is in the drag bar, for example, you could elect to have your application activate the window *and* call `DragWindow` to allow the user to drag the window to a new location, all on the basis of the first mouse-down.

### **Detecting Mouse Double Clicks**

Double clicks can be detected by comparing the time of a mouse-up event with that of an immediately following mouse-down. `LMGetDoubleClickTime` returns the time difference required for two mouse clicks to be interpreted as a double click.

### **Handling Keyboard Events**

After retrieving a key-down event, an application should determine which key was pressed and which modifier keys (if any) were pressed at the same time. When the user presses a key, or combination of keys, your application should respond appropriately. For

example, your application should allow the user to choose a frequently used menu command by using its keyboard equivalent<sup>5</sup>.

## **Character Code and Virtual Key Code**

The low-order word of the `message` field contains the **character code** and **virtual key code** corresponding to the key pressed by the user. The virtual key code is always the same for a specific key on a particular keyboard. To determine the virtual key code that corresponds to a specific physical key, the system software uses a hardware-specific key-map ('KMAP') resource.

After determining the virtual key code, the system software uses a script-specific keyboard layout ('KCHR') resource to map the virtual keycode to a specific character code. Any given script system (that is, writing system) has one or more 'KCHR' resources (for example, a French 'KCHR' and a U.S. 'KCHR') which determine whether virtual key codes are mapped to, again for example, the French or the U.S. character set.

Usually, your application should use the character code rather than the virtual key code when responding to keyboard events. The following constants may be used as masks to access the virtual key code and character code in the `message` field:

```
keyCodeMask    = 0x0000FF00    Mask to extract key code.  
charCodeMask  = 0x000000FF    Mask to extract ASCII character code.
```

## **Checking for Keyboard Equivalents**

In its initial handling of key-down and auto-key events, the application should first extract the character code from the `message` field and then check the `modifiers` field to determine if the Command key was pressed at the time of the event. If the Command key was down, the menus should be adjusted prior to further processing of the event. This further processing must necessarily accommodate the possibility that one or more of the modifier keys (Shift, Option, and Control) were also down at the same time as the Command key. If the Command key was not down, the appropriate application-defined function should be called to further handle the event.

### ***Historical Note***

Prior to Mac OS 8 and the Appearance Manager, only **Command-key equivalents** for menu items were supported. With the introduction of Mac OS 8 and the Appearance Manager, a menu item can be assigned a **keyboard equivalent**, that is, any combination of the Command key, optionally one or more modifier keys (Shift, Option, Control), and another key. (A Command-key equivalent such as Command-C is thus, by definition, also a keyboard equivalent.) This new capability is generally referred to as "support for extended modifier keys".

## **Checking For a Command-Period Key Combination**

Your application should allow the user to cancel a lengthy operation by using the Command-period combination. This can be implemented by periodically examining the state of the keyboard using `GetKeys` or, alternatively, by scanning the event queue for a Command-period keyboard event. The demonstration program at Chapter 23 — Miscellany contains a demonstration of the latter method.

<sup>5</sup> The term **keyboard equivalent** refers to a keyboard combination, such as Command-C, or any other combination of the Command key, another key and one or more modifier keys (Shift, Option, or Control). The term **Command-key equivalent** refers specifically to a keyboard equivalent comprising the Command key and one other key other than a modifier key.

## Events Not Returned to the Application

---

Certain keyboard events will not, or may not, be returned to your application. These are as follows:

- **Command-Shift-Numeric Key Combinations.** Some keystroke combinations are handled by the Event Manager and not returned to your application. These include certain combinations of Command-Shift-numeric keys, for example, Command-Shift-1 to eject a disk and Command-Shift-3 to take a snapshot of the screen. The action corresponding to such key combinations are implemented as a function that takes no parameters and which is stored in an 'FKEY' resource with a resource ID that corresponds to the number key that activates it. (Note that IDs of 1 to 4 are reserved by Apple.)
- **Key-Up Events.** At application launch, the Operating System initialises another event mask, called the **system event mask**, to exclude key-up messages. If an application needs to receive key-up events, the system event mask must be changed using the `SetEventMask` function.

## Handling Update Events

---

### The Update Region

---

The Window Manager coordinates the display of windows and keeps track of the front-to-back ordering of windows. When one window covers another and the user moves the front window, the Window Manager generates an update event so that the contents of the newly exposed area of the rear window can be updated, that is, redrawn.

The Window Manager maintains an **update region** for each window. It keeps track of all areas of a window's content region that need to be redrawn and accumulates them in this region. When the application calls `WaitNextEvent`, the Event Manager checks to see if any windows have an update region that is not empty. If it finds a non-empty update region, the Event Manager reports an update event to the appropriate application. If more than one window needs updating, the update events are issued for the front window first.

### Updating the Window

---

Upon receiving the update event, your application should first call `BeginUpdate`, which temporarily replaces the **visible region** of the window's graphics port with the intersection of the visible region and the update region and then clears the update region<sup>6</sup>. (If the update region is not cleared, the Event Manager will continue to send an endless stream of update events. Accordingly, it is absolutely essential that `BeginUpdate` be called in response to all update events.)

After the call to `BeginUpdate`, your application should draw the window's contents. (Note that, to prevent the unnecessary drawing of unaffected areas of the window, the system limits redrawing to the visible region, which at this point corresponds to the update region as it was before `BeginUpdate` cleared it.)

`EndUpdate` should then be called to restore the normal visible region.

Application-defined update functions should first determine if the window is a document window or a modeless dialog box and call separate application-defined functions for redrawing the window or the dialog box accordingly. (See Chapter 8 — Dialogs and Alerts.)

---

<sup>6</sup> This process is explained in more detail at Chapter 4 — Windows.

## **Updating Windows in the Background**

Recall that your application will receive update events when it is in the background if the application's 'SIZE' resource so specifies.

## **Automatic Updating - Windows with Static Content**

Your application can allow the Window Manager to automatically update the contents of a window, without sending an update event, by supplying in the window structure a handle to a picture that contains the contents of the window. This technique is generally useful only for windows which contain static information.

## **Handling Activate Events**

Whenever your application receives a mouse-down event, it should first call `FindWindow` to determine if the user clicked in a window other than the active window. If the click was, in fact, in a window other than the active window, `SelectWindow` should be called to begin the process of activating that window and deactivating the currently active window.

`SelectWindow` does some of the activation/deactivation work for you, such as removing the highlighting from the window being deactivated and highlighting the window being activated. It also generates two activate events so that, at your application's next two requests for an event, an activate event is returned for the window being deactivated followed by an activate event for the window being activated. In response, your application must complete the action begun by `SelectWindow`, performing such actions as are necessary to complete the activation or deactivation process. Such actions might include, for example, showing or hiding the scroll bars, restoring or removing highlighting from any selections, adjusting menus, etc.

The message field of the event structure contains a pointer to the window being activated or deactivated and bit 0 of the `modifiers` field indicates whether the window is being activated or deactivated. The `activeFlag` constant may be used to test the state of this bit.

## **Activation/Deactivation in Response to Suspend and Resume Events**

When the user switches between your application and another application, your application is notified of the switch through Operating System (suspend and resume) events. If, in its 'SIZE' resource, your application has the `acceptSuspendResumeEvents` flag set and the `doesActivateOnFGSwitch` flag not set, your application receives an activate event immediately following all suspend and resume events. This means that your application can rely on the receipt of those activate events to trigger calls to its activation/deactivation functions when a major switch occurs.

On the other hand, if your application has both the `acceptSuspendResumeEvents` and the `doesActivateOnFGSwitch` flags set, it does not receive an activate event immediately following suspend and resume events. In this case, your application must call its application-defined window activation/deactivation functions whenever it receives a suspend or resume event, in addition to the usual call made in response to an activate event.

The accepted practise is to set the `doesActivateOnFGSwitch` flag whenever the `acceptSuspendResumeEvents` flag is set.

## **Handling Disk-Inserted Events**

When the user inserts a disk, the Operating System attempts to mount the volume by calling the File Manager function `PBMountVol`. If the volume was successfully mounted, the disk's icon appears on the desktop. The Operating System Event Manager then generates

a disk-inserted event. If the user is, at the time, interacting with a standard file dialog box, the Standard File Package intercepts the event and handles it automatically. Otherwise, the event is left in the event queue for the application to retrieve.

In this latter case, your application should simply check if the disk was successfully mounted. If the disk was not successfully mounted (that is, if the high order word of the message field does not contain `noErr`), then the application should call the Disk Initialisation Manager function `DIBadMount`, which will inform the user via a dialog box. If the user clicks the OK box in this dialog, the disk will be initialised.

Basically, this handling is intended to give the user the chance to initialise or eject an uninitialised or damaged disk. If disk-inserted events are masked out, the event stays in the Operating System event queue until your application calls the Standard File Package or Navigation Services, or until an application which handles disk-inserted events becomes the foreground process. This behaviour can be confusing to the user; accordingly, your application should handle disk inserted events when they occur.

## Handling Null Events

---

The Event Manager reports a null event when the application requests an event and the application's event stream does not contain any of the requested event types. The `WaitNextEvent` function reports a null event by placing `nullEvt` in the `what` field and returning `false`.

When your application receives a null event, and assuming it is the foreground process, it can perform what is known as **idle processing**, such as blinking the caret in the active window of the application.

As previously stated, your application's 'SIZE' resource can specify that the application receive null events while it is in the background. If your application receives a null event while it is in the background, it can perform tasks or do other processing.

In order not to deny a reasonable amount of processor time to other applications, idle processing and background processing should generally be kept to a minimum.

## Handling Suspend and Resume Events

---

When an Operating System event is received, the `message` field of the event structure should be tested with the constants `suspendResumeMessage` and `mouseMovedMessage` to determine what type of event was received. If this test reveals that the event was a suspend or resume event, bit 0 should be tested with the constant `resumeFlag` to ascertain whether the event was a suspend event or a resume event.

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

## Suspend Events

---

When an application receives a suspend event, it does not actually switch to the background until it makes its next request to receive an event from the Event Manager. This gives your application the opportunity to get itself ready for a major switch to the background. On receipt of a suspend event, therefore, your application should hide floating windows<sup>7</sup>, convert any private scrap into global scrap if necessary<sup>8</sup>, call its application-defined window activation/deactivation function if appropriate (see Handling Activate Events, above), and do anything else necessary to get itself ready for the switch.

---

<sup>7</sup> See Chapter 4B — More on Windows — Mac OS 8.5 Window Manager and Chapter 21— Floating Windows. — Mac OS 8.5 and Earlier.

<sup>8</sup> See Chapter 18 — Scrap.

## ***Resume Events***

---

When an application receives a resume event, it should show floating windows, convert any global scrap back to private scrap, and call its application-defined window activation/deactivation function if appropriate (see Handling Activate Events, above).

## ***Handling Mouse-Moved Events***

---

Mouse-moved events are used to trigger a change in the appearance of the cursor according to its position in a window. For example, when the user moves the cursor outside the text area of a document window, applications typically change its shape from the I-beam shape to the standard arrow shape.

The main requirement is to specify a region in the `mouseRgn` parameter of the `WaitNextEvent` function. This causes the Event Manager to report a mouse-moved event if the user moves the cursor outside that region. On receipt of the mouse-moved event, the application can change the shape of the cursor.

An application might define two regions: a region which encloses the text area of a window (the I-beam region) and a region which 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. When the cursor moves outside the 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 non-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 another mouse-moved event.

The application must, of course, recalculate and change the `mouseRgn` parameter immediately it receives a mouse-moved event. Otherwise, mouse-moved events will be continually received as long as the cursor is outside the original region.

The appearance of the cursor may be changed using the QuickDraw function `SetCursor`. The familiar I-beam, crosshairs, plus sign and wristwatch cursors are defined as resources, which can be retrieved by a call to `GetCursor`. (See Chapter 13 — Offscreen Graphics Worlds, Pictures Cursors and Icons). Alternatively, and assuming Mac OS 8.5 or later is present, the appearance of the cursor may be changed using the Appearance Manager function `SetThemeCursor`. (See Chapter 6 — The Appearance Manager.)

Cursor setting functions should account for whether a document window or modeless dialog box is active and set the cursor appropriately.

## ***Handling Events in Alert Boxes and Dialog Boxes***

---

The handling of events in alert boxes and dialog boxes is addressed in detail at Chapter 8 — Dialogs and Alerts. The following is a brief overview only.

### ***Modal and Movable Modal Alert Boxes***

---

The Dialog Manager functions `Alert`, `NoteAlert`, `CautionAlert`, `StopAlert`, and `StandardAlert` are used to invoke modal and movable modal alert boxes and to handle all user interaction while the alert box remains open. The Dialog Manager handles all 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 Dialog Manager closes the alert box and reports the user's action to the application, which is responsible for performing the appropriate subsequent actions.

## **Modal and Movable Modal Dialog Boxes**

---

For modal and movable modal dialog boxes, the Dialog Manager function `ModalDialog` is used to handle all user interaction while the dialog box is open. When the user selects an item, `ModalDialog` reports the selection to the application, in which case the application is responsible for performing the action associated with that item. An application typically calls `ModalDialog` repeatedly, responding to clicks on enabled items as reported by `ModalDialog`, until the user selects the OK or Cancel button.

### **Historical Note**

The capability to use `ModalDialog` to handle all user interaction with movable modal dialog boxes was introduced with Mac OS 8 and the Appearance Manager. Previously, it was necessary to handle events in movable modal dialog boxes in the same manner as for modeless dialog boxes.

## **Modeless Dialog Boxes**

---

For modeless dialog boxes, you can use the function `IsDialogEvent` to determine whether the event occurred while a modeless dialog box was the frontmost window and then, optionally, use the function `DialogSelect` to handle the event if it belongs to a modeless dialog box. `DialogSelect` is similar to `ModalDialog` except that it returns control after every event, not just events relating to an enabled item.

## **The 'SIZE' Resource**

---

Several references have been made in the preceding to the application's 'SIZE' resource because some (though not all) of the flag fields in this resource are relevant to the subject of events.

An application's 'SIZE' resource informs the Operating System:

- About the memory requirements of the application.
- About certain scheduling options (for example, whether the application can accept suspend and resume events).
- Whether the application:
  - Is 32-bit clean.
  - Supports stationery documents.
  - Supports TextEdit's inline input services.
  - Wishes to receive notification of the termination of any application it has launched.
  - Wishes to receive high-level events.

The 'SIZE' resource comprises a 16-bit flags field, which specifies the operating characteristics of your application, followed by two 32-bit size fields, one indicating the minimum size, and one the preferred size, of the application's partition.



## Resource ID

---

The 'SIZE' resource created for your application should have a resource ID of -1. If the user modifies the preferred size in the Finder's Get Info window, the Operating System creates a new 'SIZE' resource having an ID of 0. If it exists, this latter resource will be invoked by the Operating System at application launch. If it does not exist, the Process Manager looks for the original 'SIZE' resource with ID -1.

## Creating a 'SIZE' Resource in CodeWarrior

---

It is possible to create a 'SIZE' resource use Resorcerer; however, it is far more convenient to use the built-in 'SIZE' resource creation facility within CodeWarrior.

**Flags Fields.** In CodeWarrior, the bits of the flags field can be set as desired using the 'SIZE' Flags pop-up menu in the 68K Target or PPC Target sections of the Settings dialog box, which appears when <Project Name> Settings... is chosen from the Edit Menu. The following describes the meanings of the items in the pop-up menu, and thus of the relevant bits of the 16-bit flags field. Those items relevant to low-level and Operating System events appear on a gray background.

<b>'SIZE' Flags Pop-Up Menu</b>	<b>Meaning When Set</b>	<b>Meaning When Not Set</b>
acceptSuspendResumeEvents	Your application can process, and thus wants to receive, suspend and resume events. (When this flag is set, the doesActivateOnFGSwitch flag should also normally be set.)	Your application does not want to receive suspend and resume events.
canBackground	Your application wants to receive null event processing time while in the background.	Your application does no background processing and thus does not want to receive null events when it is in the background.
doesActivateOnFGSwitch	Your application does not want to receive activate events associated with suspend and resume events, and thus activates and deactivates its windows in response to suspend and resume events. (Note that suspend and resume events will be received only if the acceptSuspendResumeEvents flag is set.)	Your application wants to receive activate events associated with suspend and resume events, and will activate and deactivate its windows in response to that activate event. (Note that suspend and resume events will be received only if the acceptSuspendResumeEvents flag is set.)
onlyBackground	Your application runs only in the background. (Usually, this is because it does not have a user interface and cannot run in the foreground.)	Your application runs in the foreground and the background.
getFrontClicks	Your application wants 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.	Your application does not want to receive the mouse-down and mouse-up events that are used to bring your application into the foreground.
acceptAppDiedEvents	Your application wants to be notified whenever an application launched by your application terminates or crashes. (This information is received via an	Your application does not want to be notified whenever an application launched by your application terminates or crashes.

	Apple event.)	
is32BitCompatible	Your application can be run with either the 32-bit Memory Manager or the 24-bit Memory Manager.	Your application cannot be run with the 32-bit Memory Manager.
isHighLevelEventAware	Your application can send and receive high-level events. (Your application must support the four required Apple events (see Chapter 10) if this flag is set.)	The Event Manager does not give your application high-level events when it calls <code>WaitNextEvent</code> .
LocalAndRemoteHLEvents	Your application is to be visible to applications running on other computers on a network.	Your application does not receive high-level events across a network.
isStationeryAware	Your application can recognise stationery documents.	Your application cannot recognise stationery documents. If the user opens a stationery document, the Finder duplicates the document and prompts the user for a name for the duplicate document.
useTextEditServices	Your application can use the inline text services provided by <code>TextEdit</code> .	Your application cannot use the inline text services provided by <code>TextEdit</code> .
isDisplayManagerAware	Your application can handle the Display Notice event, which tells your application to move its windows after the monitor settings have changed.	When the monitor settings are changed, the <code>DisplayManager</code> moves your application's windows so that they do not disappear off the screen.

**Size Fields.** The minimum and preferred sizes of the application's partition may be set in the Preferred Heap Size (k) and MinimumHeap Size (k) sections of the 68K Target section of the Settings dialog box.

## Notes on Multitasking

### Cooperative Multitasking

The yielding of access to the CPU by the foreground process (via the value assigned to the sleep parameter of the `WaitNextEvent` function, is central to the form of multitasking provided by the system software as we know it today. That form of multitasking is known as **cooperative multitasking**.

Under cooperative multitasking, individual applications continue executing until they "decide" to release control, thus allowing the background process of another application to begin executing. Even though this results in a usable form of multitasking, the operating system itself does not control the processor's scheduling. Even under the best of circumstances, an individual application (which has no way of knowing what other applications are running or whether they have a greater "need" to execute) makes inefficient use of the processor, which often results in the processor idling when it could be used for productive work.

Note also that, under this cooperative scheme, the assignment of zero to `WaitNextEvent`'s sleep parameter will cause your application to completely "hog" the CPU whenever it is in the foreground, allowing no CPU time at all to the background processes.

### Preemptive Multitasking

Under **preemptive multitasking**, the operating system itself retains control of which body of code executes, and for how long. No longer does one task have to depend on the

"good will" of another task - that is, the second task's surrender of control - to gain access to the CPU.

# Main Event Manager Constants, Data Types and Functions

---

## Constants

---

### Event Codes

nullEvent	= 0	No other pending events.
mouseDown	= 1	Mouse button pressed.
mouseUp	= 2	Mouse button released.
keyDown	= 3	Character key pressed.
keyUp	= 4	Character key released.
autoKey	= 5	Key held down in excess of autoKey threshold.
updateEvt	= 6	Window needs to be redrawn.
diskEvt	= 7	Disk was inserted.
activateEvt	= 8	Activate/deactivate window.
osEvt	= 15	Operating system event (suspend, resume or mouse moved).

### Event Masks

mDownMask	= 0x0002	Mouse button pressed.
mUpMask	= 0x0004	Mouse button released.
keyDownMask	= 0x0008	Key pressed.
keyUpMask	= 0x0010	Key released.
autoKeyMask	= 0x0020	Key repeatedly held down.
updateMask	= 0x0040	Window needs updating.
diskMask	= 0x0080	Disk inserted.
activMask	= 0x0100	Activate/deactivate window.
highLevelEventMask	= 0x0400	High-level events (includes AppleEvents).
osMask	= 0x8000	Operating system events (suspend, resume).
everyEvent	= 0xFFFF	All of the above.Event Message

### Masks for Keyboard Events

keyCodeMask	= 0x0000FF00	Mask to extract key code.
charCodeMask	= 0x000000FF	Mask to extract ASCII character code.

### Message Codes For Operating System Events

osEvtMessageMask	= 0xFF000000	Mask to extract OS event message code.
mouseMovedMessage	= 0x00FA	For osEvts, test for mouse-moved event.
suspendResumeMessage	= 0x0001	For osEvts, test for suspend/resume event.
resumeFlag	= 1	For osEvts, test Bit 0.
convertClipboardFlag	= 2	For osEvts, test whether to convert clipboard.

### Constants Corresponding to Bits in the modifiers Field

activeFlag	= 0x0001	Set if window being activated (activateEvt). Set if event caused a foreground switch (mouseDown).
btnState	= 0x0080	Set if mouse button up.
cmdKey	= 0x0100	Set if Command key down.
shiftKey	= 0x0200	Set if Shift key down.
alphaLock	= 0x0400	Set if Caps Lock key down.
optionKey	= 0x0800	Set if Option key down.
controlKey	= 0x1000	Set if Control key down.
rightShiftKey	= 0x2000	Set if Right Shift Key down.
rightOptionKey	= 0x4000	Set if Right Option Key down.
rightControlKey	= 0x8000	Set if Right Control Key down.

## Data Types

---

### Event Structure

```
struct EventRecord
{
    EventKind    what;        // Event code.
    UInt32      message;     // Event message.
    UInt32      when;        // Ticks since system startup.
```













The main function calls the application-defined functions for initialising the system software managers and creating the window. It then calls the function containing the main event loop.

## ***doInitManagers***

---

doInitManagers is the standard system software managers initialisation function which will be used in all demonstration programs.

Note that the call to FlushEvents at has now been added to this function. FlushEvents empties the Operating System event queue of any low-level events left unprocessed by another application, for example, any mouse-down or keyboard events that the user may have entered while this program was being launched.

## ***doNewWindow***

---

The function doNewWindow opens the window in which the types of low-level and Operating System events will be printed as they occur. The 'WIND' resource passed as the first parameter specifies that the window has a go-away box and a title (drag) bar. The window's graphics port is set as the current port for drawing and the text size is set to 10 points.

## ***eventLoop***

---

eventLoop is the main event loop.

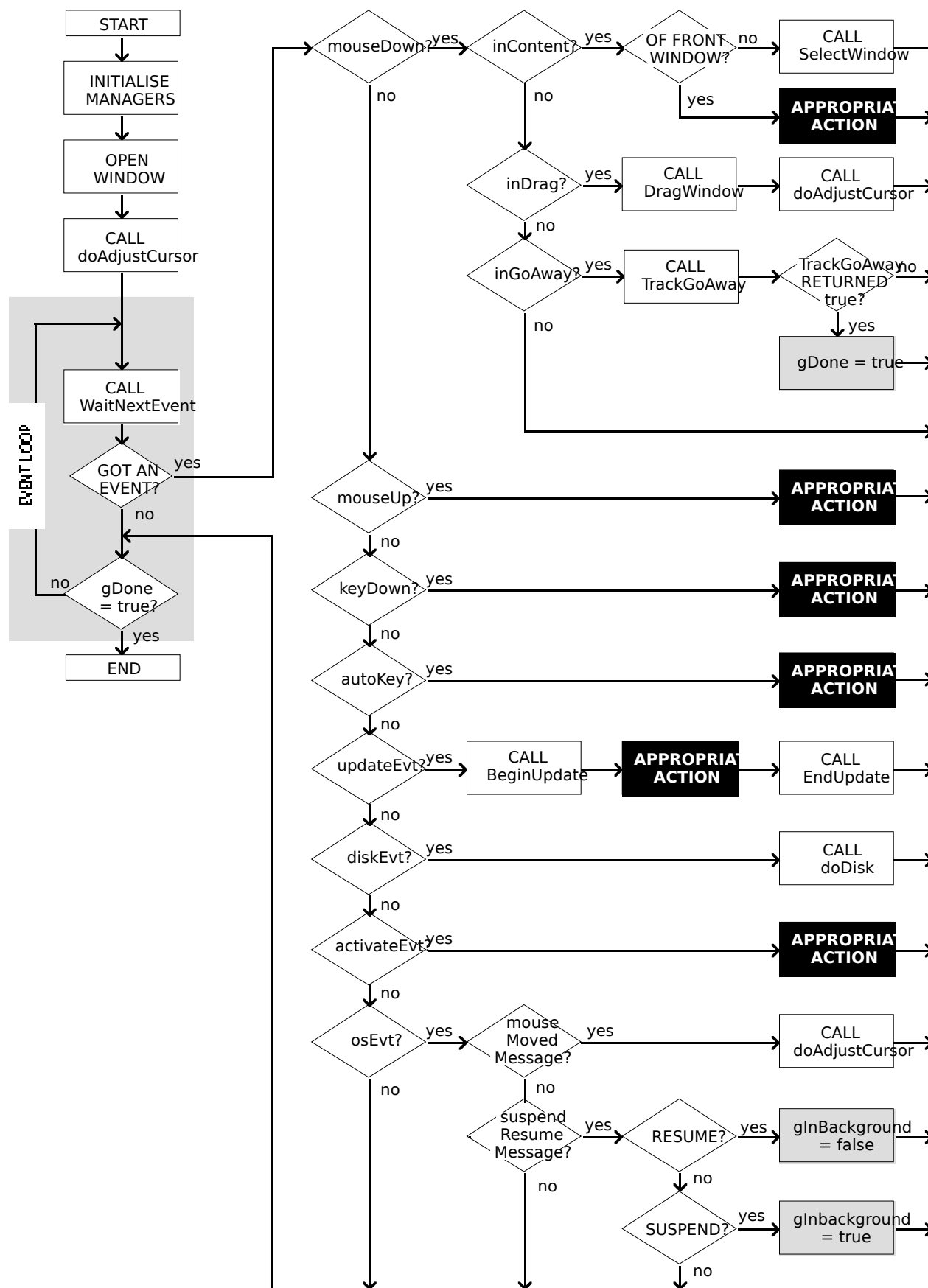
The global variable gDone is set to false before the event loop is entered. This variable will be set to true when the user clicks on the window's go-away box. The event loop (the while loop) terminates when gDone is set to true.

The calls to NewRgn and doAdjustCursor have to do with the generation of mouse-moved events. The NewRgn call allocates storage for a Region structure and initializes the contents of the region to make it an empty region. As will be seen, this first call to doAdjustCursor defines two regions (one for the arrow cursor and one for the I-Beam cursor) and copies the handle to one of them (depending on the current position of the mouse cursor) to the global variable gCursorRegionHandle.

In the call to WaitNextEvent:

- The event mask everyEvent ensures that all types of low-level and Operating System events will be returned to the application (except keyUp events, which are masked out by the system event mask).
- eventStructure is the EventRecord structure which, when WaitNextEvent returns, will contain information about the event.
- 180 represents the number of ticks for which the application agrees to relinquish the processor if no events are pending for it. 180 ticks equates to about three seconds.
- If the cursor is now *not* within the region passed in the cursorRegion parameter, a mouse-moved event will be generated immediately.

WaitNextEvent returns true if an event was pending, otherwise it returns NULL. If an event was pending, the program branches to doEvent to determine the type of event and handle the event according to its type.



**FIG 4 - LowEvents FLOWCHART**

## doEvents

doEvents handles some events to finality and performs initial handling of others.

On return from WaitNextEvent, the what field of the event structure contains an unsigned short integer which indicates the type of event received. The doEvent function isolates the type of event and switches according to that type.

In this demonstration, the action taken in every case is to print the type of event in the window. In addition, and in the case of mouse-down, update, disk and Operating System events only, calls to individual event handling functions are made.

Note that, in the case of an Operating System event, doEvent will only print "osEvt - " in the window. At this stage, the program has not yet established whether the event is a suspend, resume or mouse-moved event.

Note also that:

- The inclusion of the key-up event handling would be pointless, since key-up events are masked out by the Operating System.
- Only one activate event will ever be received when the program is run (that is, when the window opens), the reasons being that only one window is ever open and the doesActivateOnFGSwitch flag in the 'SIZE' resource is set. This latter means that activate events will not accompany suspend and resume events.

## ***doMouseDown***

---

The function doMouseDown handles mouse-down events to completion.

FindWindow is called to get a pointer to the window in which the event occurred and a "part code" which indicates the part of that window in which the mouse-down occurred. The function then switches according to that part code.

The inContent case deals with a mouse-down in a window's content region. FrontWindow returns a pointer to the frontmost window. If this is not the same as the pointer in the event structure's message field, SelectWindow is called to generate activate events and to perform basic window activation and deactivation. (Actually, SelectWindow will never be called in this demonstration because the program only opens one window, which is always the front window.)

The inDrag case deals a mouse-down in the window's drag bar. In this case, control is handed over to DragWindow, which tracks the mouse and drags the window according to mouse movement until the mouse button is released. DragWindow requires a boundary rectangle limiting the area in which the window can be dragged. This is supplied in the third argument which, in this case, is established by the bounds field of the QuickDraw global variable screenBits. screenBits.bounds contains a rectangle which encloses the main screen.

The regions controlling the generation of mouse-moved events are defined in global coordinates. The region for the I-Beam cursor is based on the window's port rectangle. Accordingly, when the window is moved, the new location of the port rectangle, in global coordinates, must be re-calculated so that the arrow cursor and I-Beam cursor regions may be re-defined. The call to doAdjustCursor re-defines these regions for the new window location and copies the handle to one of them, depending on the current location of the mouse cursor, to the global variable gCursorRegionHandle. (Note that this call to doAdjustCursor is also required, for the same reason, when a window is re-sized or zoomed.)

The inGoAway case deals with the case of a mouse-down in the go-away box. In this case, control is handed over to TrackGoAway, which tracks the mouse while the button remains down. When the button is released, TrackGoAway returns true if the cursor is still inside the go-away box, in which case the global variable gDone is set to true, terminating the event loop and the program.

## ***doUpdate***

---

The function doUpdate handles update events to completion.

Although no window updating is performed by this program, it is nonetheless necessary to call BeginUpdate because, amongst other things, BeginUpdate clears the update region, thus preventing the generation of an unending stream of update events. The call to EndUpdate always concludes a call to BeginUpdate, undoing the results of the visible/update region manipulations of the latter.

## ***doDisk***

---

doDisk further processes a disk event. Many applications quite reasonably ignore unexpected disk-inserted events; however, the function doDisk is included in this demonstration to illustrate the basics of dealing with such occurrences.

In the case of a diskEvt event, the message field of the event structure contains the drive number in bits 0-15 and the File Manager result code in bits 16-31. At the first line, the high word is tested. If it indicates that the volume was not successfully mounted, DIBadMount is called to inform the user via dialog box shown at Fig 5.

DIBadMount retains control until the disk is formatted (if the user clicks in the OK box) or until the user clicks in the Cancel box.

The call to SetPt controls the positioning of the top left corner of the dialog box on the screen.



**FIG 5 - DIALOG BOX INVOKED BY DIBadMount**

## ***doOSEvent***

---

doOSEvent first determines whether the Operating System event passed to it is a suspend/resume event or a mouse-moved event by examining bits 24-31 of the message field. It then switches according to that determination.

In the case of a suspend/resume event, a further examination of the message field establishes whether the event was a suspend event or a resume event. The global variable `gInBackground` is set to true or false accordingly. In the case of a resume event, the call to `SetCursor` ensures that the cursor will be set to the arrow cursor shape when the application comes to the foreground.

In the case of a mouse-moved event (which occurs when the mouse cursor has moved outside the region whose handle is currently being passed in `WaitNextEvent`'s `mouseRgn` parameter), `doAdjustCursor` is called to change the handle passed in the `mouseRgn` parameter according to the current location of the mouse.

## ***drawEventString***

---

`drawEventString` is incidental to the demonstration. It simply prints text in the window indicating when the call to `WaitNextEvent` is made and when the various types of events are received. `ScrollRect` scrolls the contents of the current graphics port within the rectangle specified in the first parameter. The second parameter specifies the number of pixels to be scrolled to the right and the third parameter specifies the number of pixels to scroll vertically, in this case 15 up.

## ***doAdjustCursor***

---

`doAdjustCursor`'s primary purpose in this particular demonstration is to force the generation of mouse-moved events. The fact that it also changes the cursor shape simply reflects the fact that changing the cursor shape is usually the sole reason for generating mouse-moved events in the first place.

Basically, the function establishes two regions (the calls to `NewRgn`), one describing the content area of the window (in global coordinates) and the other everything outside that. The location of the cursor is then ascertained by the call to `GetMouse`. If the cursor is in the content area of the window (the I-Beam region), the cursor is set to the I-Beam shape and the handle to the I-Beam region is copied to the global variable passed in the `mouseRgn` parameter in the `WaitNextEvent` call in the `eventLoop` function. If the cursor is in the other region (the arrow region), the cursor is set to the normal arrow shape and the arrow region is copied to the global variable passed in the `mouseRgn` parameter.

`GetCursor` reads in the system 'CURS' resource specified by the constant `iBeamCursor` and returns a handle to the 68-byte `Cursor` structure created by the call. The parameter for a `SetCursor` call is required to be the address of a `Cursor` structure. Dereferencing the handle once provides that address.

`WaitNextEvent`, of course, returns a mouse-moved event only when the cursor moves outside the "current" region, the handle to which is passed in the `mouseRgn` parameter of the `WaitNextEvent` call. Only one mouse-moved event, rather than a stream of mouse-moved events, will be generated when the cursor is moved outside the "current" region because:

- The mouse-moved event will cause `doAdjustCursor` to be called.
- `doAdjustCursor` will thus reset the "current" region to the region in which the cursor is now located.

The cursor and cursor adjustment aspects, as opposed to the region-swapping aspects, of the `doAdjustCursor` function are incidental to the demonstration. These aspects are addressed in more detail at Chapter 13 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons.