2 Version 1.2 (Frozen)

LOW-LEVEL AND OPERATING SYSTEM EVENTS

Includes Demonstration Program LowEventsPascal

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).

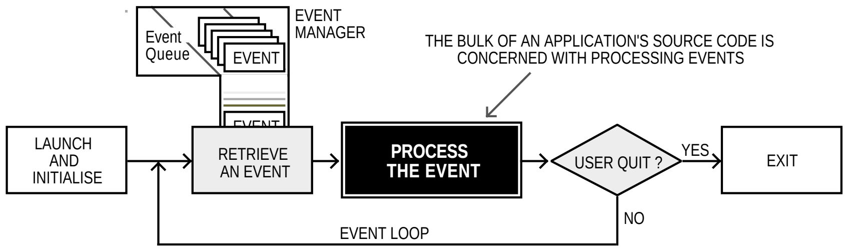


FIG 1 - THE MAIN EVENT LOOP

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 record**. An application may specify which types of events it wants to receive by including an **event mask** as a parameter in certain Event Manager routines.

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 8 — Required 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.[1]

[1]An application in the background can also receive high-level events. (See Chapter 8 — Required Apple Events.)

A maximum of 20 events 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.

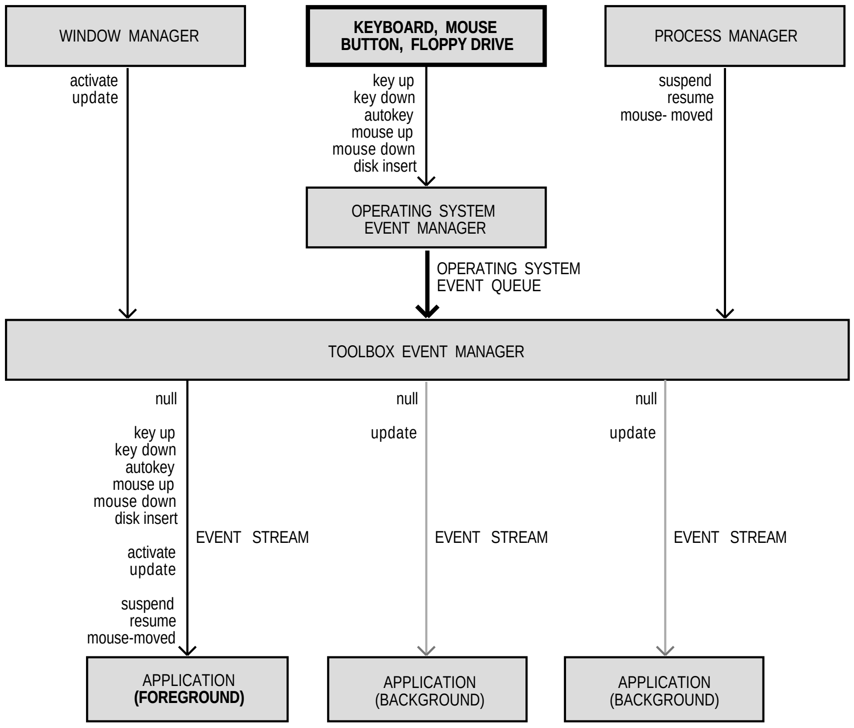


FIG 2 - LOW-LEVEL AND OPERATING SYSTEM EVENTS

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.

Obtaining Information About Events

The Event Record

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

The `EventRecord` data type in `Events.p` defines the event record:

```
type
EventRecord = record
  what:       EventKind;
  message:  UInt32;
  when:       UInt32;
  where:    Point;
  modifiers:EventModifiers;
end;
```

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:

Event Type	Contents of message Field

`nullEvent`
`mouseDown`
`mouseUp`
	Undefined.

`keyDown`
`keyUp`
`autoKey`
	Bits 0-7 = character code. Bits 8-15 = virtual key code.
	Bits 16-23 = For Apple Desktop Bus keyboards, the ADB address of the keyboard where the event occurred.

`updateEvt` `activateEvt`	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.)
`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.

```
begin
case (BAnd(CHR(BSR(eventRec.message, 24)), mouseMovedMessage)) of

  suspendResumeMessage:
    begin
    if ((BAnd(eventRec.message, resumeFlag)) = 1)
      then begin
        {This is a resume event.}
        end
      else begin
        {This is a suspend event.}
        end;
    end;

  mouseMovedMessage:
    {This is a mouse-moved event.}
    end;

  end;
```

`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[2], 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.

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?
```

[2] Global coordinates are explained at Chapter 4 — Windows (see Fig 3).

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:

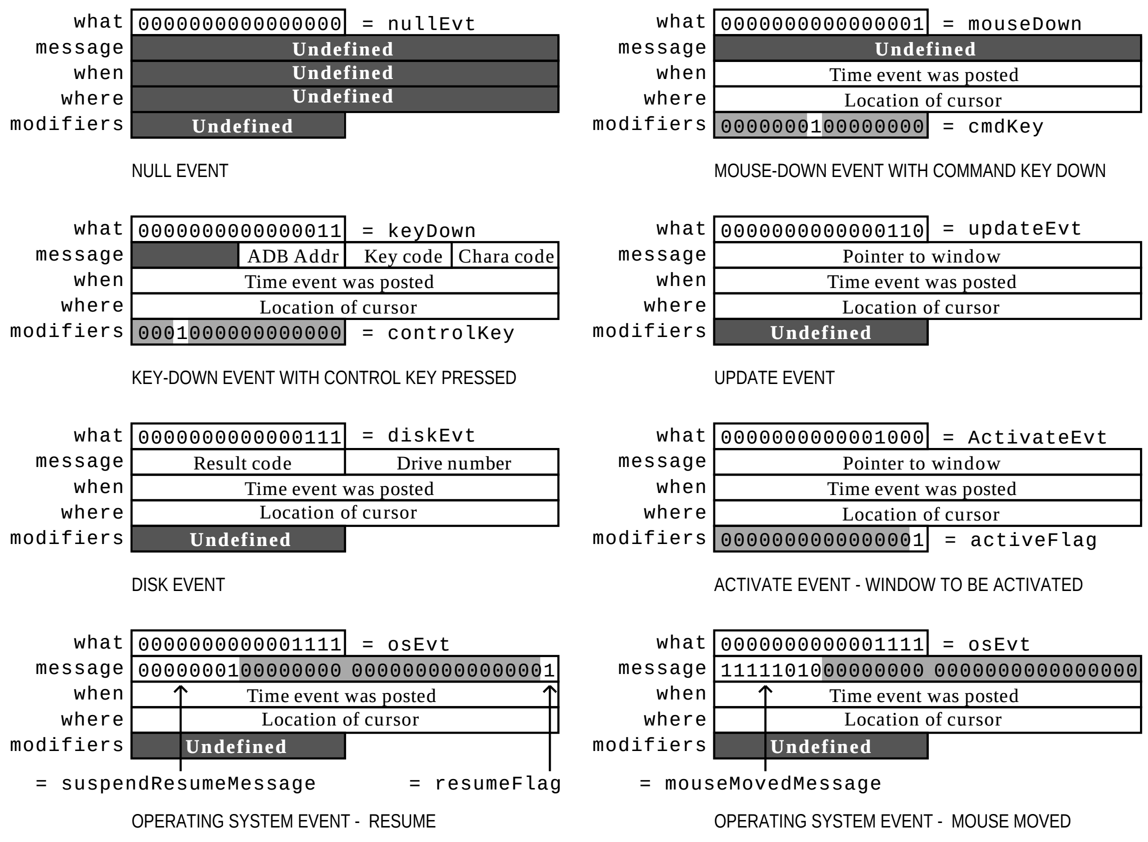
```
becomingActive : Boolean;

becomingActive := boolean(BAnd(eventRec.modifiers, activeFlag) <> 0);

if (becomingActive)
  then  {Application-defined window activation code here.}
  else  {Application-defined window deactivation code here.}
```

Event Record Examples - Diagrammatic

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



what 0000000000000000 = nullEvt
message Undefined
when Undefined
where Undefined
modifiers Undefined
NULL EVENT
what 0000000000000001 = mouseDown
message Undefined
when Time event was posted
where Location of cursor
modifiers 0000000100000000 = cmdKey
MOUSE-DOWN EVENT WITH COMMAND KEY DOWN
what 0000000000000011 = keyDown
message ADB Addr Key code Chara code
when Time event was posted
where Location of cursor
modifiers 0001000000000000 = controlKey
KEY-DOWN EVENT WITH CONTROL KEY PRESSED
what 0000000000000110 = updateEvt
message Pointer to window
when Time event was posted
where Location of cursor
modifiers Undefined
UPDATE EVENT
what 0000000000000111 = diskEvt
message Result code Drive number
when Time event was posted
where Location of cursor
modifiers Undefined
DISK EVENT
what 0000000000001000 = ActivateEvt
message Pointer to window
when Time event was posted
where Location of cursor
modifiers 0000000000000001 = activeFlag
ACTIVATE EVENT - WINDOW TO BE ACTIVATED
what 0000000000001111 = osEvt
message 0000000100000000 0000000000000001
when Time event was posted
where Location of cursor
modifiers Undefined
= suspendResumeMessage = resumeFlag
OPERATING SYSTEM EVENT - RESUME
what 0000000000001111 = osEvt
message 1111101000000000 0000000000000000
when Time event was posted
where Location of cursor
modifiers Undefined
= mouseMovedMessage
OPERATING SYSTEM EVENT - MOUSE MOVED

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 record 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.

```
WaitNextEvent  (eventMask: EventMask; var theEvent: EventRecord; sleep: UInt32;
                mouseRgn: RgnHandle): boolean;

            Returns:  A return code: false = null event; true = 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.p`:

```
mDownMask          = $0002  Mouse button pressed.
mUpMask            = $0004  Mouse button released.
keyDownMask        = $0008  Key pressed.
keyUpMask          = $0010  Key released.
autoKeyMask        = $0020  Key repeatedly held down.
updateMask         = $0040  Window needs updating.
diskMask           = $0080  Disk inserted.
activMask          = $0100  Activate/deactivate window.
highLevelEventMask = $0400  High-level events (includes AppleEvents).
osMask             = $8000  Operating system events (suspend, resume).
everyEvent         = $FFFF  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 record.

`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 `NIL` 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[3]

Your application receives mouse-down events only when it is the foreground process and the user clicks in a window belonging to the application, in a window belonging to a desk accessory that was

[3]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.

launched in your application's partition[4], 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.
- Whether the event occurred in a desk accessory launched in your application's partition.

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

```
inDesk       = 0  In none of the following.
inMenuBar    = 1  In the menu bar.
inSysWindow  = 2  In a desk accessory window.
inContent    = 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.
inDrag       = 4  In the drag (title bar) region.
inGrow       = 5  In the grow region (active window only).
inGoAway     = 6  In the go-away region (active window only).
inZoomIn     = 7  In the zoom-in region (active window only).
inZoomOut    = 8  In the zoom-out region (active window only).
```

In the Content Region

If the cursor was in the content region[5] 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, Go Away Box, or Zoom Box

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, 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.
- **Grow Box**. If the cursor was in the grow box, 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.
- **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.
- **Zoom Box**. If the cursor was in the zoom box, 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

[4]Desk accessories were originally designed for early versions of the Macintosh system software which did not support multitasking, hence the requirement to be launched inside the 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.

[5]The content region is the part of the window in which an application displays the contents of a document, the size box (window inactive), and the window's controls (for example, scroll bars).

content region 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.

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 a Desk Accessory Window

The `inSysWindow` constant is returned only when a mouse-down event occurs in an old-style desk accessory that was launched in your application's partition. If this occurs, your application should call `SystemClick`, which routes the event to the desk accessory for further handling.

Note that, if the mouse button was pressed while the cursor was in the content region of an old-style desk accessory window and the window was inactive, `SystemClick` makes it the active window, sending your application an activate event to deactivate its front window.

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`. Note, however, that if the active window is a movable modal dialog box, your application should instead call `SysBeep` to play the system alert sound rather than activate the selected window. (See Chapter 6 — Dialogs and Alerts.)

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. `GetDblTime` 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.

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   = $0000FF00; Mask to extract key code.
charCodeMask  = $000000FF; Mask to extract ASCII character code.
```

Checking for Command Key Menu 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 and the menu command executed. Otherwise, the appropriate application-defined function should be called to further handle the event.

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 21 — Miscellany contains a demonstration of the latter method.

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 routine 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[6]. (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 routines for redrawing the window or the dialog box accordingly. (See Chapter 6 — Dialogs and Alerts.)

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 record 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 record 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,

[6]This process is explained in more detail at Chapter 4 — Windows.

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 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 record 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[7], convert any private scrap into global scrap if necessary[8], 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.

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 `SetCursor` or other cursor handling routines. The familiar I-beam, crosshairs, plus sign and wristwatch cursors are defined as resources, which can be retrieved by a call to `GetCursor`. The following constants specify the resource IDs:

```
iBeamCursor = 1
crossCursor = 2
plusCursor  = 3
watchCursor = 4
```

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 covered in detail at Chapter 6 — Dialogs and Alerts. The following is a brief overview only.

[7]See Chapter 20— Floating Windows and Window Definition Functions.

[8]See Chapter 16 — Scrap.

Alert Boxes

The Dialog Manager functions `Alert`, `NoteAlert`, `CautionAlert` and `StopAlert` used to invoke alert boxes are capable of handling 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 Dialog Boxes

For modal dialog boxes, the Dialog Manager function `ModalDialog` may be called 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.

Movable Modal Dialog Boxes

`IsDialogEvent` or `DialogSelect` may be used to handle events in movable modal dialog boxes.

Modeless Dialog Boxes

Events in modeless dialog boxes may be handled using an approach similar to that used to handle events in windows, that is, when an event is received, the type of event is ascertained and appropriate action is taken based on the type of window that is in front. If a modeless dialog box is in front, the application's code can take any actions specific to that modeless dialog box and `DialogSelect` can be called to handle events that the application code does not handle.

Alternatively, the `IsDialogEvent` function may be called in the event loop, in which case that function can be used to determine whether the event is for a modeless dialog that belongs to the application. If the event involves a modeless dialog box (including null events), and a modeless dialog box is active, `IsDialogEvent` returns `true`. If the return is `true`, the application can check to see what type of event occurred and, depending on the type of event, choose to handle the event itself.

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

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 stationary 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.

Example `'SIZE'` Resource

The following is an example of a `'SIZE'` resource in Rez input format:

```
resource 'SIZE' (-1)
{
  reserved,                  /* Reserved. */
  acceptSuspendResumeEvents, /* Accepts suspend and 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,        /* No mouse events in front window on resume. */
  ignoreAppDiedEvents,       /* Does not want app-died events. */
  is32BitCompatible,         /* Works with 24- or 32-bit addressing. */
  isHighLevelEventAware,     /* Supports high-level events. */
  localAndRemoteHLEvents,    /* Also remote high-level events. */
  isStationeryAware,         /* Can use stationery documents. */
  dontUseTextEditServices,   /* Cannot use in-line input. */
  reserved,                  /* Reserved. */
  reserved,                  /* Reserved. */
  reserved,                  /* Reserved. */
  kPrefSize * 1024,          /* Preferred memory size. */
  kMinSize * 1024            /* Minimum memory size. */
};
```

Creating a `'SIZE'` Resource in CodeWarrior

A `'SIZE'` resource for your application is created automatically by CodeWarrior. The bits of the flags field can be set as desired using the **'SIZE' Flags** pop-up menu in the Project preferences panel, which is available when **Preferences** is chosen from the **Edit** Menu. The following relates the **'SIZE' Flags** pop-up menu items to the 16 bits of the flags field of the resource. Those bits relevant to low-level and Operating System events are outlined in bold.

Bit	'SIZE' Flags Pop-Up Menu	Rez Flags	Meaning
3	**useTextEditServices**	`dontUseTextEditServices` `useTextEditServices`	Cannot use inline services Can use inline services
4	**isStationeryAware**	`notStationeryAware` `isStationeryAware`	Cannot use stationery documents Can use stationery documents
5	**LocalAndRemoteHLEvents**	`onlyLocalHLEvents` `localAndRemoteHLEvents`	Can use only local high-level events Can use local and remote high-level events
6	**isHighLevelEventAware**	`notHighLevelEventAware` `isHighLevelEventAware`	Cannot use high-level events Can use high-level events
7	**is32BitCompatible**	`not32BitCompatible` `is32BitCompatible`	Works with 24-bit addressing Works with 24- and 32-bit addressing
8	**acceptAppDiedEvents**	`ignoreAppDiedEvents` `acceptAppDiedEvents`	Ignore application died events Accept application died events
9	**getFrontClicks**	`dontGetFrontClicks` `getFrontClicks`	Don't return mouse events on resume Do return mouse events on resume

10	**onlyBackground**	`backgroundAndForeground` `onlyBackground`	Application has a user interface Application has no user interface
11	**doesActivateOnFGSwitch**	`needsActivateOnFGSwitch` `doesActivateOnFGSwitch`	Needs activate event on foreground switch Needs no activate event on foreground switch
12	**canBackground**	`cannotBackground` `canBackground`	Does no background processing Can use background null events
14	**acceptSuspendResumeEvents**	`ignoreSuspendResumeEvents` `acceptSuspendResumeEvents`	Ignores suspend-resume events Accepts suspend-resume events

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` routine, is central to the form of multitasking provided by the system software as we know it today.[9] 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

Apple's projected new operating system, currently code-named Rhapsody, will introduce a new form of multitasking in which the operating system itself retains control of which body of code executes, and for how long. No longer will 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. This arrangement is known as **preemptive multitasking**.

Main Event Manager Constants, Data Types and Routines

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             = $0002      Mouse button pressed.
mUpMask               = $0004      Mouse button released.
```

[9] At the time of writing, the latest version of the system software was 7.5.5.

```
keyDownMask           = $0008      Key pressed.
keyUpMask             = $0010      Key released.
autoKeyMask           = $0020      Key repeatedly held down.
updateMask            = $0040      Window needs updating.
diskMask              = $0080      Disk inserted.
activMask             = $0100      Activate/deactivate window.
highLevelEventMask    = $0400      High-level events (includes AppleEvents).
osMask                = $8000      Operating system events (suspend, resume).
everyEvent            = $FFFF      All of the above.Event Message
```

Masks for Keyboard Events

```
keyCodeMask           = $0000FF00  Mask to extract key code.
charCodeMask          = $000000FF  Mask to extract ASCII character code.
```

Message Codes For Operating System Events

```
osEvtMessageMask      = $FF000000  Mask to extract OS event message code.
mouseMovedMessage     = $00FA      For osEvts, test for mouse-moved event.
suspendResumeMessage  = $0001      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            = $0001      Set if window being activated (activateEvt).
                                   Set if event caused a foreground switch (mouseDown).
btnState              = $0080      Set if mouse button up.
cmdKey                = $0100      Set if Command key down.
shiftKey              = $0200      Set if Shift key down.
alphaLock             = $0400      Set if Caps Lock key down.
optionKey             = $0800      Set if Option key down.
controlKey            = $1000      Set if Control key down.
rightShiftKey         = $2000      Set if Right Shift Key down.
rightOptionKey        = $4000      Set if Right Option Key down.
rightControlKey       = $8000      Set if Right Control Key down.
activeFlagBit         = 0          Activate? (activateEvt and mouseDown)
```

Data Types

Event Record

```
type
  EventRecord = record
    what:       EventKind;
    message:  UInt32;
    when:       UInt32;
    where:    Point;
    modifiers:EventModifiers;
  end;
```

Routines

Receiving Events

```
function   WaitNextEvent(eventMask: EventMask; var theEvent: EventRecord; sleep: UInt32;
           mouseRgn: RgnHandle): boolean;
function   EventAvail(eventMask: EventMask; var theEvent: EventRecord): boolean;
procedure  FlushEvents(whichMask: EventMask; stopMask: EventMask);
procedure  SystemClick(var theEvent: EventRecord; theWindow: WindowPtr);
procedure  SystemTask;
function   GetOSEvent(mask: EventMask; var theEvent: EventRecord): boolean;
function   OSEventAvail(mask: EventMask; var theEvent: EventRecord): boolean;
procedure  SetEventMask(value: EventMask);
```

Reading the Mouse

```
procedure  GetMouse(var mouseLoc: Point);
function   Button: boolean;
function   StillDown: boolean;
function   WaitMouseUp: boolean;
```

Reading the KeyBoard

```
procedure  GetKeys(KeyMap theKeys);
function   KeyTranslate(transData: UNIV Ptr; keycode: UInt16; VAR state: UInt32): UInt32;
```

Getting Timing Information

```
function   TickCount: UInt32;
function   GetDblTime : UInt32;
function   GetCaretTime : UInt32;
```

Demonstration Program

```
{ ####################################################################################
LowEventsPascal.p
// ###################################################################################
//
This program:
//
// •  Contains a main event loop function, together with subsidiary functions which
//    perform nominal handling only of low-level and Operating System events.
//
// •  Opens a window in which the types of all received low-level and Operating System
//    events are displayed.
//
// •  Terminates when the user clicks the window's close box.
//
// Event handling is only nominal in this program because its main purpose is to
// demonstrate the basics of an application's main event loop.
//
// Programs in later chapters demonstrate the full gamut of individual event handling.
//
// The program utilizes the following resources:
//
// •  A 'WIND' resource (purgeable).

// •  A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateonFGSwitch, and
//    is32BitCompatible flags set.
//
// ################################################################################### }

program LowEventsPascal(input, output);

{ ………………………………………………………………………………………………… include the following Universal Interfaces }

uses

  Windows, Fonts, Menus, TextEdit, Dialogs, Memory, Types, Events, Quickdraw,
  QuickdrawText, Processes, OSUtils, Sound, ToolUtils, DiskInit, Segload;

{ ……………………………………………………………………………………………………………………… define the following constants }

const

rWindowResource = 128;

{ …………………………………………………………………………………………………………………………………………………… global variables }

var

gDone : Boolean;
gInBackground : Boolean;
gMouseMoved : Boolean;

{ ######################################################################### DoInitManagers }

procedure DoInitManagers;

  begin
  MaxApplZone;
  MoreMasters;

  InitGraf(@qd.thePort);
  InitFonts;
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(nil);

  InitCursor;
  FlushEvents(everyEvent, 0);
  end;
    {of procedure DoInitManagers}

{ ########################################################################## DoNewWindow }

procedure DoNewWindow;

  var
  myWindowPtr : WindowPtr;

  begin
  myWindowPtr := GetNewWindow (rWindowResource, nil, WindowPtr(-1));
  if (myWindowPtr = nil) then
    begin
    SysBeep(10);
    ExitToShell;
    end;

  SetPort(myWindowPtr);
  TextSize(10);
  end;
    {of procedure DoNewWindow}

{ ########################################################################## DoMouseDown }

procedure DoMouseDown(var eventRec : EventRecord);

  var
  partCode : integer;
  myWindowPtr : WindowPtr;

  begin
  partCode := FindWindow(eventRec.where, myWindowPtr);

  case partCode of

    inSysWindow:
      begin
      SystemClick(eventRec, myWindowPtr);
      end;

    inContent:
      begin
      if (myWindowPtr <> FrontWindow) then
        SelectWindow(myWindowPtr);
      end;

    inDrag:
      begin
      DragWindow(myWindowPtr, eventRec.where, qd.screenBits.bounds);
      end;

    inGoAway:
      begin
      if (TrackGoAway(myWindowPtr, eventRec.where)) then
        gDone := true;
      end;

    end;
      {of case statement}
  end;
    {of procedure DoMouseDown}

{ ############################################################################# DoUpdate }

procedure DoUpdate (var eventRec : EventRecord);

  begin
  BeginUpdate(WindowPtr(eventRec.message));
  EndUpdate(WindowPtr(eventRec.message));
  end;
    {of procedure DoUpdate}

{ ############################################################################# DoDisk }

procedure DoDisk (var eventRec : EventRecord);

  var
  thePoint : Point;
  theErr : OSErr;

  begin
  if (HiWord(eventRec.message) <> noErr)
    thenbegin
      SetPt(thePoint, 120, 120);
      theErr := DIBadMount(thePoint, eventRec.message);
      end

    elsebegin
      {Attempt to mount was successful.  Record drive number for accessing
      the disk, etc.}
      end;
  end;
    {of procedure DoDisk}

{ ########################################################################### DoOSEvent }

procedure DoOSEvent (var eventRec : EventRecord);

  begin
  case BAnd(BSR(eventRec.message, 24), $000000FF) of
    suspendResumeMessage:
        begin
        if (BAnd(eventRec.message, resumeFlag) = 1)
          thenbegin
            gInBackground := false;
            DrawString('Resume event');
            end

          elsebegin
            gInBackground := true;
            DrawString('Suspend event');
            end;
        end;

    mouseMovedMessage:
        begin
        gMouseMoved := true;
        DrawString('Mouse-moved event');
        end;
    end;
      {of case statement}
  end;
    {of procedure DoOSEvent}

{ ###################################################################### DrawEventString }

procedure DrawEventString(eventString : string);

  var
  tempRegion : RgnHandle;
  myWindowPtr : WindowPtr;

  begin
  myWindowPtr := FrontWindow;
  tempRegion := NewRgn;

  ScrollRect(myWindowPtr^.portRect, 0, -15, tempRegion);
  DisposeRgn(tempRegion);

  MoveTo(8, 291);
  DrawString(eventString);
  end;
    {of procedure DrawEventString}

{ ####################################################################### DoAdjustCursor }

```

```
procedure DoAdjustCursor(frontWindow : WindowPtr; mouseRegion : RgnHandle);

  var
  myArrowRegion : RgnHandle;
  myIBeamRegion : RgnHandle;
  cursorRect : Rect;
  mousePt : Point;

  begin
  myArrowRegion := NewRgn;
  myIBeamRegion := NewRgn;
  SetRectRgn(myArrowRegion, -32768, -32768, 32766, 32766);

  cursorRect := frontWindow^.portRect;
  LocalToGlobal(cursorRect.topLeft);
  LocalToGlobal(cursorRect.botRight);

  RectRgn(myIBeamRegion, cursorRect);
  DiffRgn(myArrowRegion, myIBeamRegion, myArrowRegion);

  GetMouse(mousePt);
  LocalToGlobal(mousePt);

  if (PtInRgn(mousePt, myIBeamRegion))
    thenbegin
      SetCursor(GetCursor(iBeamCursor)^^);
      CopyRgn(myIBeamRegion, mouseRegion);
      end

    elsebegin
      SetCursor(qd.arrow);
      CopyRgn(myArrowRegion, mouseRegion);
      end;

  DisposeRgn(myArrowRegion);
  DisposeRgn(myIBeamRegion);
  end;
    {of procedure DoAdjustCursor}

{ ############################################################################ DoEvents }

procedure DoEvents(var eventRec : EventRecord);

  begin
  case eventRec.what of

    mouseDown:
      begin
      DrawEventString('   • mouseDown');
      DoMouseDown(eventRec);
      end;

    mouseUp:
      begin
      DrawEventString('   • mouseUp');
      end;

    keyDown:
      begin
      DrawEventString('   • keyDown');
      end;

    autoKey:
      begin
      DrawEventString('   • autoKey');
      end;

    updateEvt:
      begin
      DrawEventString('   • updateEvt');
      DoUpdate(eventRec);
      end;

    diskEvt:
      begin
      DrawEventString('   • diskEvt');
      DoDisk(eventRec);
      end;

    activateEvt:
      begin
      DrawEventString('   • activateEvt');
      end;

    osEvt:
      begin
      DrawEventString('   • osEvt - ');
      DoOSEvent(eventRec);
      end;

    otherwise
      begin
      end
    end;
      {of case statement}
  end;
    {of procedure DoEvents}

{ ############################################################################ EventLoop }

procedure EventLoop;

  var
  cursorRegion : RgnHandle;
  eventRec : EventRecord;
  gotEvent : Boolean;

  begin
  gDone := false;
  cursorRegion := NewRgn;

  while (not gDone) do
    begin
    if (not gInBackground & gMouseMoved) then
      begin
      DoAdjustCursor(FrontWindow, cursorRegion);
      gMouseMoved := false;
      end;

    gotEvent := WaitNextEvent(everyEvent, eventRec, 180, cursorRegion);
    if (gotEvent) then
      DoEvents(eventRec);
    end;
  end;
    {of procedure EventLoop}

{ ################################################################ start of main program }

begin

  DoInitManagers;
  DoNewWindow;
  EventLoop;

end.

{ ######################################################################################## }
```

Demonstration Program Comments

When the program is run, the user should move the mouse cursor inside and outside the window, click the mouse inside and outside the window, drag the window, press and release keyboard keys, and insert initialised and uninitialised disks, noting the types of events generated by these actions as printed on the scrolling display inside the window.

The user should also note the basic window deactivation and activation which occurs when the mouse is clicked outside, and then inside the window.

The program may be terminated by a click in the window's go-away box.

The general"flow" of the program is illustrated in the flow chart at Fig 4.

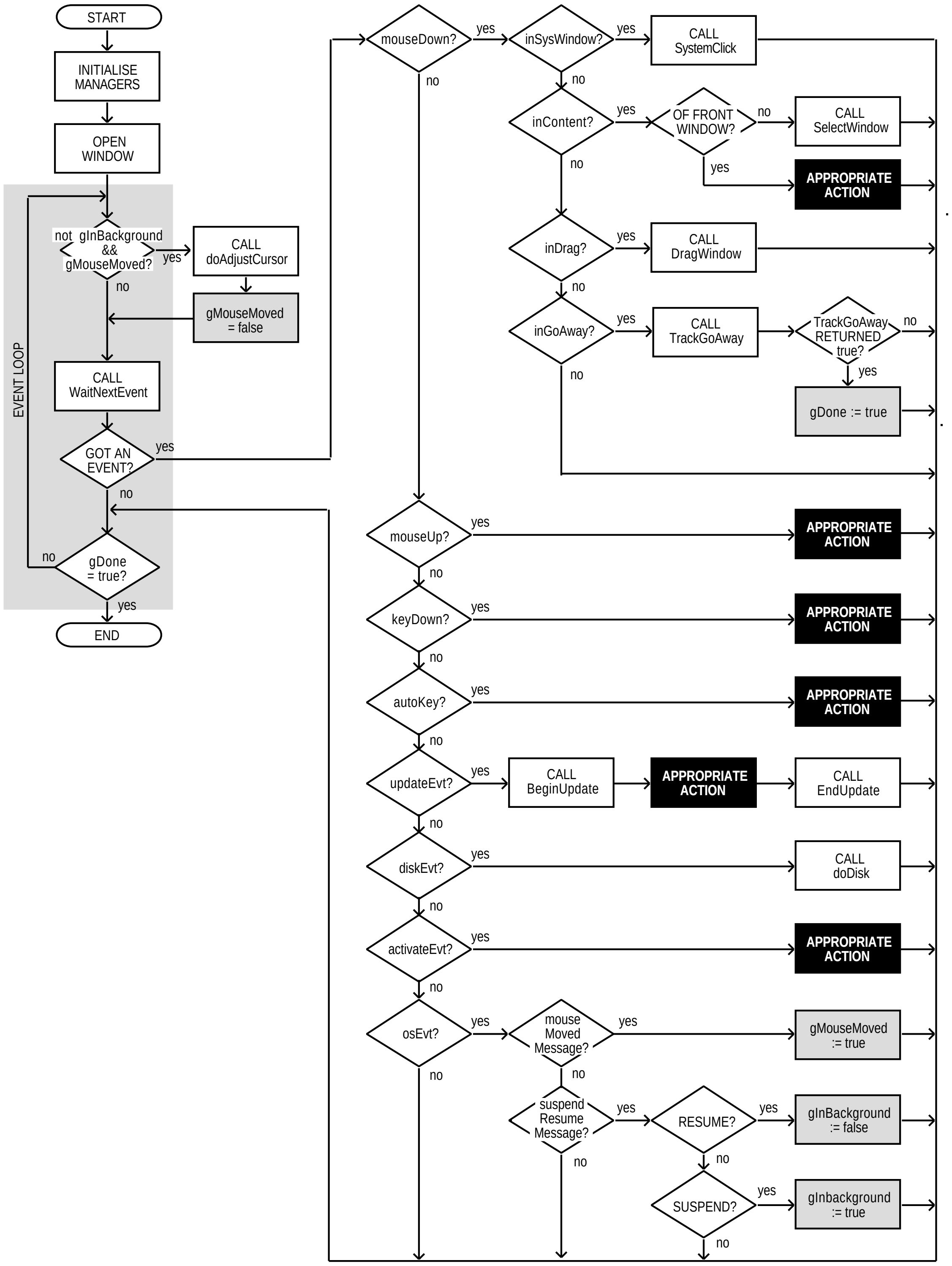


FIG 4 - LowEvents FLOWCHART

The constant declaration block

Line 42 establishes a constant for the ID of the 'WIND' resource.

The variable declaration block

The global variable gDone controls the termination of the main event loop and thus of the program. gInBackground will be set to true when the application is about to move to the background and to false when the application returns to the foreground. gMouseMoved will be set to true when a mouse-moved event is received.

The procedure DoInitManagers

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

Note that the call to FlushEvents at Line 68 has now been added to this procedure. 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.

The procedure DoNewWindow

The procedure DoNewWindow opens the window (Line 80) 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. Line 87 sets this window's graphics port as the current port for drawing and Line 88 sets the text size to 10 points.

The procedure DoMouseDown

The DoMouseDown procedure handles mouse-down events to completion.

At line 101, 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 procedure then switches according to that part code.

Lines 105-108 deal with the case of a mouse-down in a system window, for example, an old-style desk accessory launched in the application's partition. The call to SystemClick simply routes the event to the desk accessory for further handling.

Lines 110-114 deal with the case of 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 record'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.)

Lines 116-119 deal with the case of 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.

Lines 121-125 deal 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.

The procedure DoUpdate

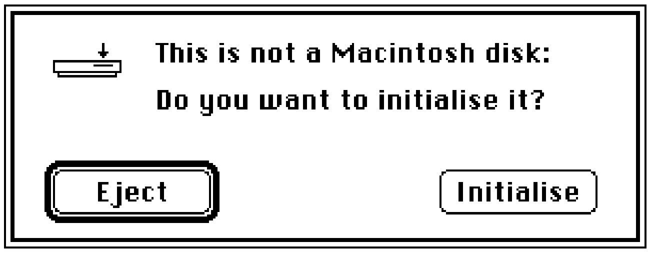
The procedure DoUpdate handles update events to completion.

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.

The procedure DoDisk

DoDisk further processes a disk event. Many applications quite reasonably ignore unexpected disk-inserted events; however, the procedure 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 EventRecord contains the drive number in bits 0-15 and the File Manager result code in bits 16-31. At line 151, the high word is tested. If it indicates that the volume was not successfully mounted, DIBadMount is called to inform the user via this system-supplied dialog box:


This is not a Macintosh disk:
Do you want to initialise it?

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.

Line 153 controls the positioning of the top left corner of the dialog box on the screen.

The procedure 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 (Line 169). It then switches according to that determination.

In the case of a suspend/resume event, a further examination of the message field (Line 172) establishes whether the event was a suspend event or a resume event. The global variable gInBackground is set to true or false accordingly (Lines 173-181).

In the case of a mouse-moved event, the global variable gMouseMoved is set to true so that doAdjustCursor will be called next time though the main event loop.

The procedure 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 (Line 206) 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.

The procedure 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, one describing the content area of the window and the other everything outside that (Lines 225-234). The location of the cursor is then ascertained (Line 236). 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 I-Beam region is copied to the region used as the fourth parameter in the WaitNextEvent call (Lines 239-243). 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 region used as the fourth parameter in the WaitNextEvent call (Lines 245-248).

At Line 241, 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 contained in the fourth parameter to 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 just before the call to WaitNextEvent in the main event (except when the application is in the background).

- doAdjustCursor will thus reset the "current" region to the region in which the cursor is now located before the call to WaitNextEvent.

The cursor adjustment aspects, as opposed to the region-swapping aspects, of the doAdjustCursor function are incidental to the demonstration. This aspect is addressed in more detail at Chapter 12 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons.

The procedure DoEvents

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

On return from WaitNextEvent, the what field of the EventRecord structure contains an unsigned short integer which indicates the type of event received. The DoEvent function isolates the type of event (Line 260) and branches 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.

The procedure EventLoop

EventLoop is the main event loop.

The global variable gDone is set to false before the event loop is entered (Line 324). This variable will be set to true when the user clicks on the window's go-away box. The event loop (the while loop initiated at Line 327) terminates when gDone is set to true.

Line 325 has to do with the generation of mouse-moved events. The call to NewRgn sets up a region (cursorRegion) which will receive cursor region information from the application-defined function doAdjustCursor and which will be used as a parameter in the WaitNextEvent function call at Line 335. Mouse-moved events will be generated from the WaitNextEvent call only when the cursor is not within the region held in the cursorRegion variable.

Line 329 ensures that the call to DoAdjustCursor (Line 331) will not be made unless the application is in the foreground and a mouse-moved event has been received. (Only foreground processes are responsible for setting cursor appearance.) If doAdjustCursor is called, the global variable which is set to true by the receipt of a mouse-moved event is reset to false (Line 332).

In the call to WaitNextEvent at Line 335:

- 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).
- eventRec 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 not within the region held 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 (Lines 336-337) to determine the type of event and handle the event according to its type.

The main program block

The main function calls the application-defined functions for initialising the system software managers and creating the window (Lines 346-347). It then calls the function containing the main event loop (Line 348).