

Events

This chapter describes how you can use the Event Manager to receive information about user actions and to receive notice of changes in the processing status of your application. One of the key elements of a well-written Macintosh application is its “user-centered” design. This means, among other things, that instead of carrying out a sequence of steps in a predetermined order, the application is driven primarily by user actions (such as moving the mouse, pressing the mouse button, and typing characters) whose order cannot in general be predicted. This chapter describes how the Macintosh system software reports user actions to your application and shows how to structure your application to facilitate the implementation of user-centered design.

This chapter begins by describing some of the features of a good user-centered design and some general ways to implement them. Then it shows how to

- initialize the basic Toolbox managers
- receive information from the Event Manager about user actions
- respond to user actions

For a complete description of the capabilities of the Event Manager, see the chapter “Event Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*. For the complete story on the features of a good user interface, see *Macintosh Human Interface Guidelines*.

About Events

Probably the most distinctive aspect of a well-written Macintosh application is that it puts users in control of the application, not the other way around. To be in control, the user should be able to perform, at any particular time, any of a wide array of actions. These actions might include pulling down one of your application’s menus, choosing a menu command, typing some characters, moving a window, and so forth. A key concept here is that users should feel that your application is always ready to do something for them.

Even when your application is busy performing some lengthy operation (for instance, saving a document to disk) and you need to prevent the user from doing other things, you should provide some safe way for the user to cancel the operation and regain control. Typically you accomplish this by displaying a dialog box indicating that a lengthy operation is underway; the dialog box should indicate some safe way for the user to stop the operation.

The essence of this user-centered design is the use of an *event-driven programming model*. In other words, the system software breaks up the user’s actions into their component *events*, which are passed one by one to your application for handling. For example, when the user presses a key on the keyboard, the system software sends your application information about that event. This information includes which key was pressed, when the key was pressed, whether any modifier keys (for instance, the Command key) were being held down at the time of the keypress, and so forth. Your application responds to the event by performing whatever actions are appropriate.

Events

Your application can receive many types of events. Events are usually divided into three categories:

- low-level events
- operating-system events
- high-level events

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

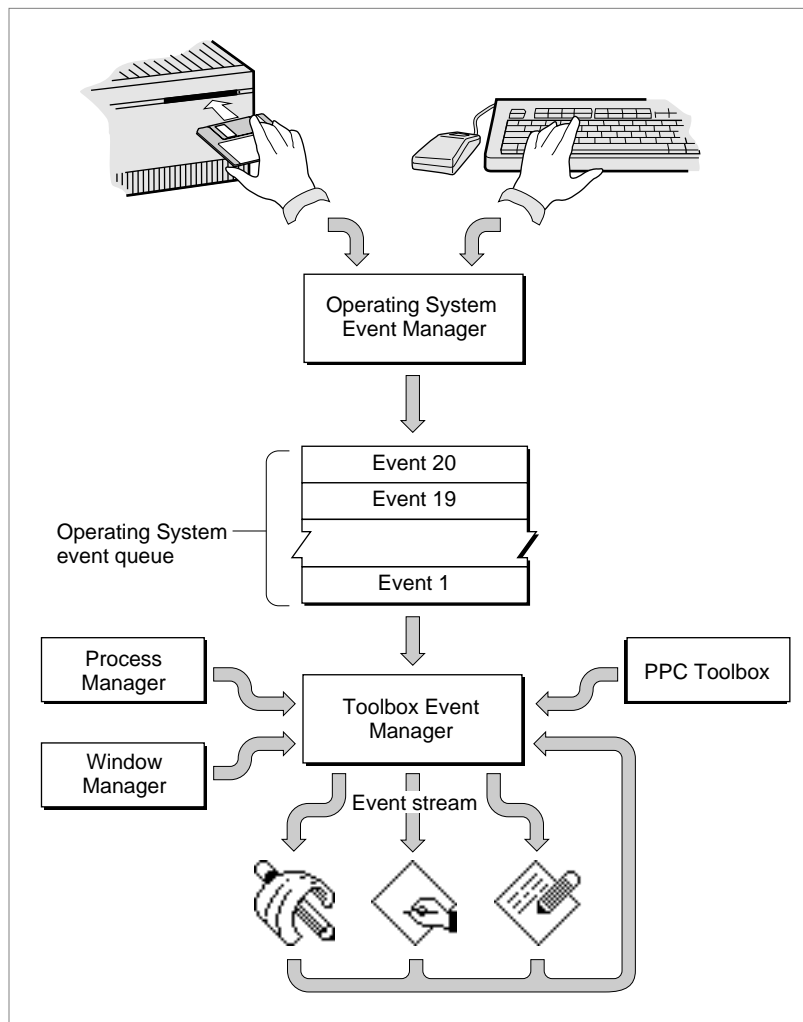
The Event Manager returns *operating-system events* to your application when the processing status of your application is about to change or has changed. For example, if a user brings your application to the foreground, the Process Manager sends an event through the Event Manager to your application.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

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

Note

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

Figure 4-1 illustrates the various sources of events that can be passed to your application. As you can see, events originate from a number of different sources: the Operating System Event Manager, Window Manager, Process Manager, and PPC Toolbox.

Figure 4-1 Sources of events sent to your application

The Event Manager maintains, for each open application, an event stream containing those events that are available to that application. Your general strategy is to retrieve an event, process it, retrieve the next event, process it, and so on indefinitely. You stop this process only when the user elects to quit your application.

Initializing an Application

When your application first starts up, and even before you begin to receive and process events describing the user's actions, you need to do some initial setting up. As you've already seen (page 3), you need to initialize some of the Macintosh Toolbox managers. You also need to set up your menu bar and menus, and perform some other standard initialization. Listing 4-1 shows the code executed by the Venn Diagrammer application when it first starts up.

Listing 4-1 Initializing your application

```
DoInitManagers;           {initialize Toolbox managers}
DoSetupMenus;             {initialize menus}

gDone := FALSE;           {initialize global variables}
gNumDocWindows := 0;      {initialize count of open doc windows}
gPrefsDialog := NIL;      {initialize ptr to Preferences dialog}

gAppsResourceFile := CurResFile; {get refnum of the app's resource file}
gPreferencesFile := -1;    {initialize res ID of preferences file}

DoReadPrefs;              {read the user's preference settings}

DoVennInit;
DoMainEventLoop;          {and then loop forever...}
```

The first thing the Venn Diagrammer application does is call the application-defined routine `DoInitManagers` to set up its application partition and initialize several Toolbox managers. Then it calls `DoSetupMenus` to create its menu bar and menus. (See Listing 8-1 on page 155 for the definition of `DoSetupMenus`.)

After its menu bar has been created, Venn Diagrammer initializes several global variables and reads the user's current preferences from a preferences file. Then the application calls another routine, `DoVennInit`, to handle any other initialization. This includes defining the rectangles and regions in a Venn diagram window and displaying a window.

Note

The `DoVennInit` procedure is not defined in this book. ♦

Events

Once the application has initialized itself, it starts executing its main event loop by calling the `DoMainEventLoop` procedure. In the main event loop, the application calls the Event Manager to get an event, responds to the event, then loops back to repeat the process. See Listing 4-4 on page 77 for a sample event loop.

Listing 4-2 defines the `DoInitManagers` routine. It begins by calling two Memory Manager routines to expand the heap zone to its limit and to create an additional block of master pointers.

Listing 4-2 Initializing the main Toolbox Managers

```
PROCEDURE DoInitManagers;
BEGIN
    MaxApplZone;           {extend heap zone to limit}
    MoreMasters;           {get 64 more master pointers}

    InitGraf(@thePort);    {initialize QuickDraw}
    InitFonts;             {initialize Font Manager}
    InitWindows;           {initialize Window Manager}
    InitMenus;             {initialize Menu Manager}
    TEInit;                {initialize TextEdit}
    InitDialogs(NIL);      {initialize Dialog Manager}

    FlushEvents(everyEvent, 0); {clear event queue}
    InitCursor;            {initialize cursor to arrow}
END;
```

Then `DoInitManagers` calls the standard Toolbox initialization routines. Finally, it clears the event queue and calls the QuickDraw routine `InitCursor` to make sure that the cursor is the standard arrow cursor.

Receiving Events

You receive events by calling an Event Manager routine, usually `WaitNextEvent`. When you ask for an event, the Event Manager returns the next available event according to its *event priority*. The Event Manager returns events in this order of priority:

1. activate events
2. mouse-down, mouse-up, key-down, key-up, and disk-inserted events in FIFO (first-in, first-out) order

Events

3. auto-key events
4. update events (in front-to-back order of windows)
5. operating-system events (suspend, resume, mouse-moved)
6. high-level events
7. null events

To retrieve an event, you pass the `WaitNextEvent` function an *event record*, defined by the `EventRecord` data type:

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

On return from `WaitNextEvent`, the `what` field of the event record contains an integer that specifies the type of event received. The Event Manager uses this set of predefined constants to indicate the event type:

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

The `message` field of the event record contains additional information about the event. The interpretation of this field depends on the type of event you've received. For some events (such as null events, mouse-up, and mouse-down events), the value in the `message` field is undefined. For keyboard events, the `message` field indicates which key was pressed. For activate and update events, the `message` field contains a window pointer to the affected window. For disk-inserted events, the `message` field contains the drive number in the low-order word and the result code of the File Manager's attempt to mount that disk in that drive. Listing 4-3 illustrates how an application reads parts of the `message` field while handling disk-inserted events.

Listing 4-3 Handling disk-inserted events

```

PROCEDURE DoDiskEvent (myEvent: EventRecord);
  VAR
    myResult:   Integer;
    myPoint:    Point;
  BEGIN
    IF HiWord(myEvent.message) <> noErr THEN
      BEGIN
        SetPt(myPoint, 100, 100);
        myResult := DIBadMount(myPoint, myEvent.message);
      END;
    END;
  END;

```

If the disk was not successfully mounted (that is, if the high-order word of the message field does not contain `noErr`), then `DoDiskEvent` calls the system software routine `DIBadMount` to inform the user and allow the disk to be ejected or reformatted. (See the chapter “Disk Initialization Manager” in *Inside Macintosh: Files* for more information about handling disk-inserted events.)

The `where` field of the event record contains, for low-level events, the location of the cursor at the time the event was posted. You can use this information to determine where on the screen a mouse-down event occurred, for instance.

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

To handle an event, you simply take whatever action is appropriate for the kind of event it is. Listing 4-4 shows one way to structure an event-handling routine.

Listing 4-4 An event loop

```

PROCEDURE DoMainEventLoop;
  VAR
    myEvent:   EventRecord;
    gotEvent:  Boolean;           {is returned event for me?}
  BEGIN
    REPEAT
      gotEvent := WaitNextEvent(everyEvent, myEvent, 15, NIL);
      IF NOT DoHandleDialogEvent(myEvent) THEN
        IF gotEvent THEN
          BEGIN
            CASE myEvent.what OF

```

Events

```

        mouseDown:
            DoMouseDown(myEvent);                {see page 120}
        keyDown, autoKey:
            DoKeyDown(myEvent);                  {see page 160}
        updateEvt:
            DoUpdate(WindowPtr(myEvent.message)); {see page 124}
        diskEvt:
            DoDiskEvent(myEvent);                {see page 77}
        activateEvt:
            DoActivate(WindowPtr(myEvent.message),
                        myEvent.modifiers);      {see page 126}
        osEvt:
            DoOSEvent(myEvent);                  {see page 171}
        keyUp, mouseUp:
            ;
        nullEvent:
            DoIdle(myEvent);                     {see page 173}
        OTHERWISE
            ;
    END; {CASE}
END
ELSE
    DoIdle(myEvent);
UNTIL gDone;                                {loop until user quits}
END;

```

The event loop defined in Listing 4-4 repeatedly calls the `WaitNextEvent` function to retrieve the next available event. This function returns a value of `FALSE` if there are no events of the desired type (other than null events) pending for your application. Otherwise, `WaitNextEvent` returns `TRUE`.

After the next available event is retrieved, the `DoMainEventLoop` procedure calls the application-defined function `DoHandleDialogEvent` (defined in Listing 7-5 on page 141) to determine whether the event applies to a dialog box. The `DoHandleDialogEvent` function returns `TRUE` if it handled the event and `FALSE` otherwise.

Note

Dialog boxes receive special treatment because the system software automatically handles many user actions in dialog boxes. For example, the Dialog Manager handles update events for dialog boxes, and it calls the Control Manager to handle user actions affecting any controls in the dialog box. ♦

Events

If the event retrieved does not apply to a dialog box, and if it isn't a null event, then `DoMainEventLoop` branches into a Pascal CASE statement in which the labels are simply the predefined constants for each event type. As you can see, the event loop calls an application-defined routine to handle each particular kind of event. These routines are defined throughout this book.

Handling Events Outside the Main Event Loop

You'll notice that some types of events—for example, `keyUp` and `mouseUp`—are simply ignored by the main event loop defined in Listing 4-4. Key-up events are ignored because most applications don't need to know that a key was released, only that it was pressed. Similarly, you usually don't need to know when the mouse button was released, because you're more interested in knowing whether (and where) the mouse button was pressed. In certain cases, however, you will be interested in a mouse-up event. For example, if the user presses the mouse button while the cursor is in a window's close box but then moves the cursor outside the close box before releasing the mouse button, you don't want to handle the mouse-down event. (This is another good example of user-centered design: allowing users to change their minds.)

It might appear that a problem is lurking, because the main event loop defined in Listing 4-4 ignores mouse-up events. How, then, can your application determine that the user released the mouse button when the cursor was outside of the close box? The answer is simple: the system software provides a routine, `TrackGoAway`, that you call in response to a user click in the close box. The `TrackGoAway` function tracks user actions involving the close box; it returns the Boolean value `TRUE` if the cursor is still inside the close box when the button is released and `FALSE` otherwise. Listing 4-5 illustrates how to call `TrackGoAway`.

Listing 4-5 Tracking mouse events in the close box

```
PROCEDURE DoGoAwayBox (myWindow: WindowPtr; mouseLoc: Point);
BEGIN
    IF TrackGoAway(myWindow, mouseLoc) THEN
        DoCloseWindow(myWindow);
    END;
```

The `TrackGoAway` function exits only when the mouse button is released. Because it determines internally when that happens, your application doesn't need to.

Events

The system software provides routines to handle the three main cases in which you need to track the mouse and determine if the cursor is in a particular location when the button is released. Here are the main routines you'll use:

| Mouse-tracking routine | Action |
|------------------------|--|
| TrackBox | Track the cursor in a window's zoom box |
| TrackControl | Track the cursor within a control |
| TrackGoAway | Track the cursor in a window's close box |

For various purposes, you might need to perform similar tracking on an arbitrary rectangle in a window. The function `DoTrackRect` defined in Listing 4-6 shows one way to define such a function.

Note

Venn Diagrammer calls `DoTrackRect` to handle mouse-down events in the tool icons. See Listing 6-9 beginning on page 121. ♦

Listing 4-6 Tracking the cursor in an arbitrary rectangle

```

FUNCTION DoTrackRect (myWindow: WindowPtr; myRect: Rect): Boolean;
    VAR
        myIgnore:    LongInt;
        myPoint:     Point;
    BEGIN
        InvertRect(myRect);           {invert the rectangle}
        REPEAT
            Delay(kVisualDelay, myIgnore)
        UNTIL NOT StillDown;          {until mouse is released}
        InvertRect(myRect);

        GetMouse(myPoint);           {get mouse location}
        DoTrackRect := PtInRect(myPoint, myRect);
    END;

```

The `DoTrackRect` function inverts the specified rectangle and keeps it inverted until the user releases the mouse button. The Event Manager function `StillDown` looks in your application's event queue for a mouse-up event; if none is found, `StillDown` returns TRUE; otherwise, `StillDown` returns FALSE. Note that `DoTrackRect` loops until `StillDown` returns FALSE, indicating that the corresponding mouse-up event has been found. The call to the `Delay` procedure within the loop is to ensure that the rectangle is inverted for some minimum, user-perceptible amount of time.

Events

```
CONST  
    kVisualDelay          = 6;    {wait 6 ticks (one-tenth second)}
```

The `DoTrackRect` function loops until `StillDown` detects the appropriate mouse-up event and then returns the specified rectangle to its original state by inverting it again. Next, `DoTrackRect` calls the Event Manager function `GetMouse` to determine the current position of the cursor. If, when the mouse button is released, the cursor is still inside the specified rectangle (as determined by the QuickDraw routine `PtInRect`), then `DoTrackRect` returns `TRUE`.

As you can see, you sometimes want to call Event Manager routines from outside your main event loop, most often to monitor mouse movements and button states once the user has clicked in some particular part of a window.

