Carbon Event Manager Programming Guide

Carbon > Events & Other Input

2005-07-07

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Carbon Event Manager Programming Guide

Note: This document was previously titled *Handling Carbon Events*.

Events are the foundation of all Carbon programming. Each time the user clicks the mouse, types a character from the keyboard, or chooses a command from a menu, you're notified by means of an event. When one of your windows needs to be redrawn, moved, or resized, your application receives an event telling you to perform the operation. When your program becomes the active (foreground) application or moves to the background in favor of another, or when another application starts up or quits, you receive an event informing you of the fact. Just about everything a typical Carbon program does, whether interacting with the user or communicating with the system, takes place in response to an event.

The Carbon Event Manager is the preferred interface for handling events in Carbon applications. You can use this interface to handle events generated in response to user input as well as to create your own custom events.

Some of the types of events that the Carbon Event Manager can handle include the following:

- Window events: resizing, closing, activation, moving, window updates, and so on.
- Menu events: menu tracking and selection, keyboard shortcuts, and so on.
- Control events: activation, selection, dragging, changes in user focus, and so on.
- Mouse events: mouse-up, mouse-down, mouse movement, multiple clicks, multiple buttons, dragging, chording, rollover states, scroll wheel operation, and so on.
- Text and keyboard events: Unicode or Macintosh-encoded text input and raw keyboard presses.
- Application events: application activation, deactivation, requests to quit, and so on.
- Apple events: see Dispatching Apple Events in Your Application in *Apple Events Programming Guide* for details on how to process Apple events using the Carbon Event Manager.
- Volume events: insertion or ejection of CDs and disks.
- Tablet events: tablet proximity and movement.

The Carbon event model is simpler and more efficient than that used by the Classic Event Manager (sometimes referred to as the `WaitNextEvent` event model). Many standard responses to events are handled automatically, which means that you need to write code only when you want to override or augment the default actions. In addition, the Carbon Event Manager also provides replacements for custom definition procedure messages.

System Requirements

The Carbon Event Manager is available on Mac OS X (10.0) and later. It is also available on Mac OS 8.6 through Mac OS 9.1 when CarbonLib 1.1.1 or later is installed. Note that some Carbon Event Manager features are available only on later versions of Mac OS X.

Who Should Read This Document

Because event handling is so fundamental to applications, everyone writing Carbon applications should read this document. You should have some familiarity with Macintosh terminology and understand the basics of creating and manipulating the Mac OS user interface (windows, controls, menus, and so on).

Carbon Event Manager Concepts

This chapter gives an overview of the Carbon event model and how the Carbon Event Manager interacts with your application.

Carbon Event Handling Theory

Carbon event processing is based on a callback mechanism. You define your program's response to various types of event by writing *event handler* functions and installing them in the Carbon Event Manager. Then, each time an event occurs, the Carbon Event Manager will call back the handler routine you've installed for that type of event. By defining how your program responds to events, the event handlers determine everything about the program's appearance and behavior on the screen.

The Event Loop

The heart of an event-driven application is the *main event loop*. After any required initialization, the application enters the main event loop and does not leave it until required to quit. The basic loop operation is as follows:

1. The application sits in a suspended state, waiting for an event. While in this state it uses no processor time, which means that more time is available to other running applications.
2. When an event occurs that requires its attention, the application "wakes up" and processes the event. Typically the Carbon Event Manager calls back the event handler, if any, that the application has installed for that event.
3. After processing, the application returns to its suspended state, waiting for the next event.

The main event loop continues in this manner until it receives a quit event (usually in response to the user's choosing a Quit command from a menu). After leaving the event loop, the application calls any necessary termination routines and then quits.

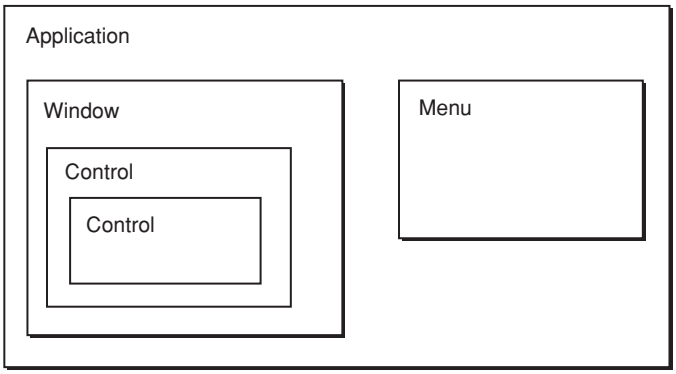
Event Types

Every Carbon event is characterized by an *event type* consisting of two items of information: an *event class* and an *event kind*. The event class denotes a general category of events, such as mouse events or window events; the event kind identifies the specific type of event within that category, such as a mouse-down event (the user pressed the mouse button) or a window-activated event (a window has been brought to the front of the screen).

Event Targets and Containment Hierarchies

Every event handler you create must be associated with a particular object called an *event target*. For example, your handler could be associated with a control, a menu, a window, or even the entire application. The event class and kind do not have to be related to the event target. For example, a handler to process a control event (such as button click) does not have to be attached to a control. You could attach it to the window that contains the control, or even the application itself. How and when your handler gets called for an event is determined by a *containment hierarchy*, as shown in Figure 1-1.

Figure 1-1 Event target containment hierarchy



When an event occurs, it is initially reported to the innermost relevant target in the hierarchy. For example, if the user clicks a button, the event is initially sent to that button control. If the user resizes a window, the event is sent to that window. If that target has no handler for the given event type, the event propagates outward to the next containing target. This makes it possible for an event handler associated with an inner target to override the behavior defined for a given event type by an outer, enclosing target.

For example, you can use this hierarchy to enable or disable your program's menu items according to circumstances. This behavior is controlled via events of type `kEventCommandUpdateStatus`, which ask whether a particular item should be enabled or disabled on the menu. On receiving such an event for, say, the Close command on your program's File menu, you might have an event handler associated with the application event target (the outermost target in the hierarchy) disable the menu, while a handler associated with an individual window enables it. If your program has at least one window open on the screen, the event will be handled by the window's event handler; if not, it will propagate outward to the application event handler. The Close command will thus be enabled if there are any windows present, but disabled if there aren't.

The Handler Stack

Within an event target, you can have multiple event handlers installed. These handlers are arranged in a stack, placed in reverse order of installation (last in is first called). For example, when an event is passed to an event target, it is sent to the top handler in the stack. If that handler doesn't take the event, it is passed to the next handler in the stack, and so on.

Note that this stack design means that you can install more than one handler for a particular event. Plugins, for example, can use this feature to install additional event handlers for existing events. If the plug-in is installed after the application-defined handler, it is first in line to take the event. If it chooses not to handle it, the application-defined handler then has the opportunity to take the event.

If a standard handler is installed, it is placed at the bottom of the stack and will be the last to be called.

Standard Handlers

Carbon provides a standard event handler for the window and application event targets. These handlers define a standard response to each type of event that a window or application target may receive; the one for windows, for instance, implements all the standard behavior for manipulating a window with the mouse—dragging it by its title bar, closing it by clicking the close button, resizing it by dragging the resize control, and so on.

Note: The standard window handler also includes standard responses for control events. The standard application handler includes support for menu events.

By installing the standard handler when you create a window, you automatically inherit all of this standard behavior with no additional effort on your part. You can then proceed to install additional handlers of your own for those aspects of the window's behavior that are specific to your individual application, such as drawing its contents or responding to the user's mouse actions inside it. Events of those specific types will be reported to your own installed handlers for processing; all others will instead be passed through to the standard handler to deal with in the standard way. This frees you from having to provide your own handler for each of the hundred-odd kinds of event that Carbon may throw at you: with the standard event handlers to back you up, you can just focus your attention on those events whose behavior you need to modify or customize in some way and leave the rest to the standard handlers, knowing that they will do something sensible with them.

Note: Some events have a default behavior rather than a standard handler associated with them. The default behavior always occurs if you don't handle the event, whether or not you have a standard event handler installed.

Sometimes the standard event handler's response to a single event can trigger an elaborate cascade of other events. Consider, for example, what happens when the user presses the mouse button in a window's resize control. The mouse press generates an event of type `kEventMouseDown`, reporting such information as the time and location at which the button was pressed, what modifier keys were being held down at the time, and so forth. Responding to this event involves hit-testing the mouse location to determine that it lies in the window's resize control, tracking the mouse's movements for as long as the button is held down, providing appropriate visual feedback on the screen, and finally resizing the window when the button is released. Theoretically, you could provide a handler routine for mouse-down events to do all this yourself, but it's generally more convenient to let the standard window event handler manage all these chores for you in the standard way. It does this by generating a sequence of further events representing various stages in the process of responding to the original mouse press:

1. A hit-test event (`kEventWindowHitTest`) to analyze the mouse location and determine what object on the screen received the mouse press
2. A click-resize-region event (`kEventWindowClickResizeRgn`) indicating that the mouse button was pressed in the resize control of one of your windows
3. A get-minimum-size (`kEventWindowGetMinimumSize`) and a get-maximum-size (`kEventWindowGetMaximumSize`) event requesting the smallest and largest dimensions to which the user should be allowed to resize the window

   a. A mouse-dragged event (`kEventMouseDragged`) reporting the mouse's coordinates

   b. A window bounds-changing event (`kEventWindowBoundsChanging`) indicating that the window's size is about to change

   c. A get-grow-image-region event (`kEventWindowGetGrowImageRegion`) requesting the size and shape of the window outline to be drawn for visual feedback on the screen

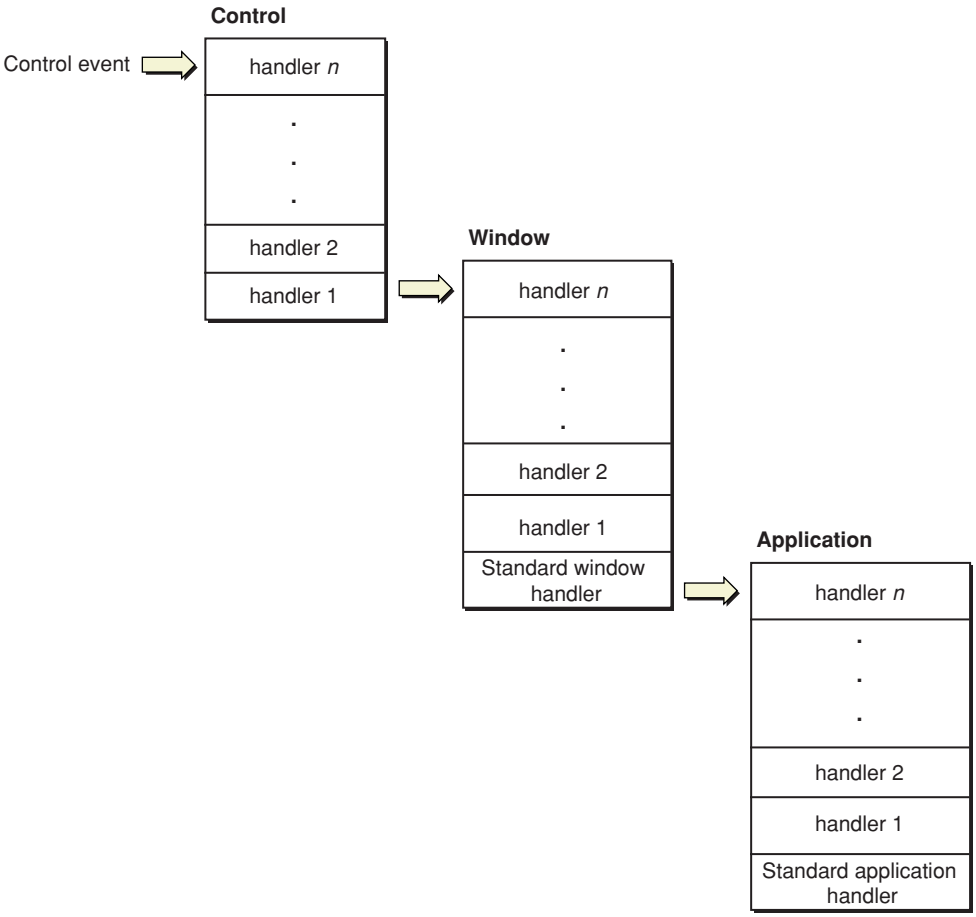
4. A mouse-up event (`kEventMouseUp`) when the mouse button is released
5. A draw-frame event (`kEventWindowDrawFrame`) to redraw the window's structural elements (frame, title bar, and so forth) in the new size
6. A window bounds-changed event (`kEventWindowBoundsChanged`) indicating that the window's size has changed
7. A window update event (`kEventWindowUpdate`) indicating that the portion of the window's contents visible on the screen has changed and must be redrawn
8. A draw-content event (`kEventDrawContent`) to redraw the window's interior contents

This proliferation of events may seem daunting, but most of them are really intended to be processed by the standard window event handler itself, with no active intervention on your part. The only reason for sending all these events is to give you the flexibility to step in at various points in the process and take control yourself if you choose to do so. Maybe you want to reimplement the draw-frame event to change the standard rectangular window frame to an octagonal viewing port for your starship simulation, or intercept mouse-dragged events to play a cool sound effect while the user is dragging the mouse around. Most of the time, you'll just leave these events for the standard handler to manage in its own way.

An Event Propagation Example

Here's a simple example of how an event would propagate through the containment hierarchy when you have the standard handlers installed, as shown in Figure 1-2.

Figure 1-2 Event propagation with standard handlers



Say the user clicks on a button. Doing so generates an event which is sent to the associated event target (the button control). If the event makes its way through the control's handler stack without being processed, it is propagated to the window that contains it. (Currently there are no standard handlers that can be installed on controls.)

If the window event target contains no installed handlers that can take the control event, the standard window handler takes the event. (The standard window handler includes responses for control events.)

Note that if you installed a handler for a control event on the application event target, it would never get called, because the standard window handler will take the event before it can get propagated to the application.

Event Timers

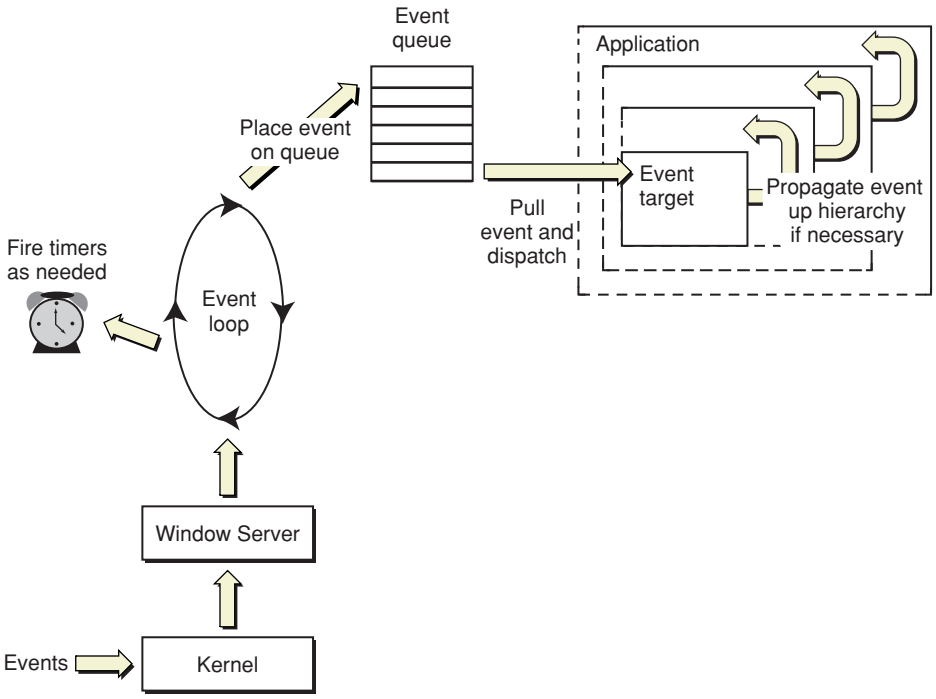
In addition to letting you install event handlers, the Carbon Event Manager also lets you create event timers, which you use to perform some action repeatedly at regular intervals. For example, you may want to use a timer to handle actions such as blinking a text-insertion caret, sounding a repeating beep, or updating a clock display on the screen. The timer fires at specified intervals, calling a timer routine you specified when installing the timer. You can also specify that a timer fire only once. For example, you can install a one-shot timer that dismisses a dialog after two minutes.

The Event Model

In most cases, you can simply write your event handlers and not worry about the details of how events are propagated to your application. However, if you have more sophisticated needs, understanding the event model will make it easier to write your code.

Figure 1-3 diagrams the basic Carbon event model in Mac OS X.

Figure 1-3 The Carbon event model



User events of all kinds are propagated through the kernel to the Window Server. From there, events are sent to your application in a two-step process:

1. A low-level event loop extracts the events that are relevant to your application and places them into the application's event queue. This loop also fires timers as necessary. If neither of these tasks need attention, the loop is blocked.

2. The Carbon Event Manager removes events from the event queue and dispatches them to the appropriate event targets. If the target has registered for the event, the appropriate handler is called. If not, the event propagates up the containment hierarchy until someone handles the event.

3. If no registered handler takes the event, and no standard handlers are installed, the event is discarded (unless `WaitNextEvent` is also installed; see "Carbon Events Versus WaitNextEvent" (page 15) for more details).

Note: While the lower level details differ slightly, the Carbon event loop and dispatching mechanism are identical in older Mac OS systems using CarbonLib.

The standard Carbon Event Manager event loop function, `RunApplicationEventLoop`, automatically handles all of the above operations for you. However, if you want more control over the event-handling mechanism, you may choose to call lower-level functions to explicitly run the event loop and dispatch events. For more information about processing events yourself, see "Processing Events Manually" (page 40).

If you create preemptive threads (using Multiprocessing Services), these will have their own low-level event loops and event queues, but they do not receive user events. Cooperatively-scheduled threads (such as you would create with the Thread Manager) share the main application event loop and queue. For more information about processing events in other threads, see "Carbon Events in Multiple Threads" (page 43).

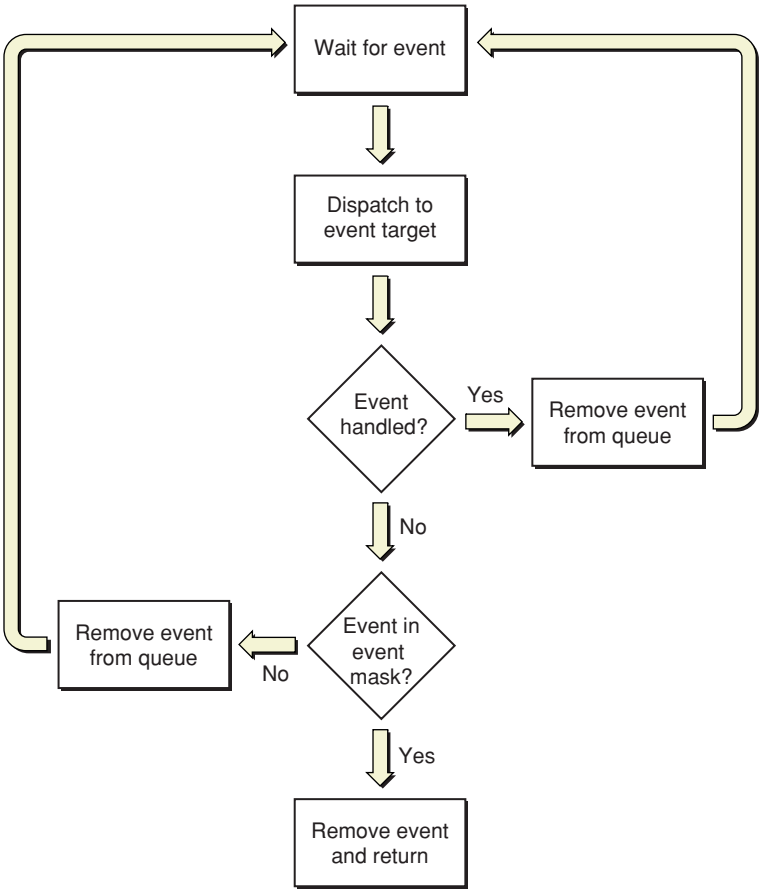
Carbon Events Versus WaitNextEvent

The Carbon Event Manager was designed as a replacement for the Classic Event Manager, which is based around the `WaitNextEvent` loop. If you are writing a new Carbon application, you should use the Carbon Event Manager. If you are porting an existing application to Carbon, here are some reasons why you should adopt the Carbon Event Manager:

- Standard event handlers mean that you don't have to handle most common events.
- No polling. A more efficient event model means that your application uses less processor time, improving overall system performance.
- The Carbon Event Manager can handle any number of event types (not just the 16 available in the Classic Event Manager).
- Defined event types include those that replace defproc messaging.

The Carbon event model is flexible enough that you can make gradual changes to adopt the Carbon Event Manager. In fact, you can have Carbon event handlers installed and still call `WaitNextEvent` to run your event loop. Figure 1-4 shows the modified event handling mechanism used by `WaitNextEvent`.

Figure 1-4 WaitNextEvent execution in the Carbon environment



1. `WaitNextEvent` runs the event loop, placing events into the event queue as they appear. It also fires timers as necessary.
2. When an event appears in the event queue, `WaitNextEvent` dispatches it to the appropriate event target, but does not pull the event off the queue.
3. If a Carbon event handler processed the event, then the event is pulled off the queue and `WaitNextEvent` waits for the next event.
4. If the event wasn't handled, `WaitNextEvent` checks to see if the event is in the event mask specified by the application. If not, it pulls the event off the queue and discards it.
5. It the event is in the event mask, `WaitNextEvent` pulls the event from the queue, packages it as an event specification, and returns.

Note that if you specify that standard event handlers be used for your windows, these will override any `WaitNextEvent` handlers you had written to process window events. Also, the standard handlers for menu events and Apple events are installed only when you call `RunApplicationEventLoop`. Therefore, as long as you use `WaitNextEvent`, you will still have to handle menu tracking, menu selection, and Apple event dispatching yourself.

Here are some simple changes that can improve performance if you are not ready to fully adopt the Carbon Event Manager:

- Maximize the wait time in `WaitNextEvent` to `7FFFFFFF`. Doing so effectively blocks your application so it won't use unnecessary processor time.
- Don't reduce your wait time in order to get null events for idle processing. If you need to perform periodic actions (such as blinking the cursor), install Carbon event timers to do so.

The Carbon Event Manager also provides some utility functions which can be useful for applications using both Carbon events and `WaitNextEvent`. To convert between event references and Classic Event Manager event specifications, use `ConvertEventRefToEventRecord`:

```
EventRef     theRef;
EventRecord  theRecord;

ConvertEventRefToEventRecord (theRef, &theRecord);
```

To determine whether a Carbon event corresponds to a bit in a Classic Event Manager event mask, use `IsEventInMask`:

```
EventRef theRef;
EventMask theMask;
Boolean result;

result = IsEventInMask (theRef, theMask);
```

A pair of convenience macros, `EventTimeToTicks` and `TicksToEventTime`, are available for converting between event times and the older ticks intervals:

```
EventTime  timeInSeconds;
UInt32     timeInTicks;

timeInTicks   = EventTimeToTicks(timeInSeconds);
timeInSeconds = TicksToEventTime(timeInTicks);
```

Carbon Event Manager Tasks

This chapter expands on the basic concepts introduced in "Carbon Event Manager Concepts" (page 9) and shows you how to create and install event handlers using the Carbon Event Manager interface.

Event Classes and Kinds

As introduced in "Event Types" (page 9), each Carbon event is defined by an event class (for example, mouse or window events) as well as an event kind (for example, a mouse-down event).

All of the available event classes and kinds are designated by constants defined in the Universal Interfaces header file `CarbonEvents.h`. Nominally, these values are 32-bit integers; but in practice, the constants denoting event classes are specified as four-character tags—for instance,

```
kEventClassMouse = FOUR_CHAR_CODE('mous');
```

—while those representing event kinds are defined as simple integers:

```
kEventMouseDown = 1;
```

The event class and kind form a unique signature called an *event type*, which is specified in the Carbon Event Manager by the `EventTypeSpec` structure. When you register an event handler, you need to pass one or more `EventTypeSpec` structures to specify which events you want to handle.

The inclusion of standard handlers for many common events means that you can intercept actions only at the level you require. Some examples:

- If the user clicks the zoom box in a window, your handler can intercept the overall action at any of the following levels:
  - When the mouse is pressed (`kEventMouseDown`).
  - When the mouse is determined to be in the zoom region (`kEventWindowClickZoomRgn`).
  - When the mouse is released in the zoom region and the zoom is to take place (`kEventWindowZoom`).
  - When the zoom is completed (`kEventWindowZoomed`).
- When a window needs to be updated (redrawn), you can begin to take action at either of the following times:
  - Immediately (`kEventWindowUpdate`). You must handle all the usual update actions (calling `SetPort`, `BeginUpdate`/`EndUpdate`, drawing) yourself.
  - Only when it is time to draw (`kEventWindowDrawContent`). The standard handler for `kEventWindowUpdate` calls `SetPort` and `BeginUpdate`/`EndUpdate` for you. (It also sends the `kEventWindowDrawContent` event.)

Note: On Mac OS X, you can view all the events that are sent to your application on-the-fly by setting the environment variable EventDebug to 1 in the Terminal application (that is, by entering `setenv EventDebug 1`) and then launching your application from the command line using the LaunchCFMApp tool.

Executing the Event Loop

The Carbon Event Manager provides several ways to execute event loops. The most common method is to simply call the function `RunApplicationEventLoop`, which does the following:

- Installs the standard application event handler
- Puts the application in a suspended state, waiting for events
- Places events into the application event queue as they occur
- Dispatches the events to your handlers or to standard event handlers

Using `RunApplicationEventLoop`, the basic structure of a Carbon application is as shown in Listing 2-1.

Listing 2-1 Structure of a typical Carbon application

```
void main (void)

   // Main function

   {
      Initialize (); // Do one-time-only initialization

         RunApplicationEventLoop (); // Process events until time to quit

      Finalize (); // Do one-time-only finalization

   }  /* end main */
```

You would register your event handlers in the `Initialize` function. Once in the loop, the only actions the application can take are in response to events.

To break out of the event loop, you must call the `QuitApplicationEventLoop` function from whichever event handler handles the quit event.

Note: On Mac OS X, you typically install an Apple event handler to handle the quit event. The `RunApplicationEventLoop` function installs a simple quit Apple event handler for you, but you may want to install your own if you want to take additional actions (such as displaying a "Save changes before quitting?" dialog).

The `RunApplicationEventLoop` function works only on the main event loop; if your application creates preemptive threads, each of them will have its own event loop and queue, and you must retrieve and dispatch these events manually. For more information, see "Processing events manually" (page 41).

Each event loop is represented by an event loop reference, an opaque data object of type `EventLoopRef`. A thread can obtain a reference to its own event loop or to the program's main event loop by calling the functions `GetCurrentEventLoop` or `GetMainEventLoop`, respectively.

The function `RunCurrentEventLoop` runs the event loop belonging to the currently executing thread for a specified time (which can be infinite). This function can be useful if you want your thread to block for a specified time. During execution, it will place events into the queue and fire timers, but will take no other actions (for example, it won't dispatch events to handlers). The function `QuitEventLoop` terminates a designated event loop.

Creating and Registering an Event Handler

The function for installing an event handler is called `InstallEventHandler`:

```
OSStatus InstallEventHandler (EventTargetRef        target,
                              EventHandlerUPP       handlerProc,
                              UInt32                numTypes,
                              const EventTypeSpec*  typeList,
                              void*                 userData,
                              EventHandlerRef*      handlerRef);
```

The second parameter, `handlerProc`, is a universal procedure pointer (UPP) to your handler routine. The conversion function `NewEventHandlerUPP` returns a UPP of the required type; for instance,

```
EventHandlerUPP  handlerUPP;

handlerUPP = NewEventHandlerUPP(ThisHandler);
```

(where `ThisHandler` is the name of your handler routine).

The target parameter to `InstallEventHandler` identifies the event target on which the handler is to be installed. You can obtain a reference to the desired target by calling one of the following functions: `GetApplicationEventTarget`, `GetWindowEventTarget`, `GetMenuEventTarget`, or `GetControlEventTarget`.

For convenience, The Carbon Event Manager also defines a set of specialized macros, `InstallWindowEventHandler`, `InstallMenuEventHandler`, and `InstallControlEventHandler`, which accept the targeted object as a parameter, obtain the corresponding target reference for you, and pass it to `InstallEventHandler`. The remaining parameters to these macros are the same as for the `InstallEventHandler` routine itself. For example, the macro call

```
WindowRef theWindow;

InstallWindowEventHandler (theWindow, handlerUPP,
                           numTypes, typeList,
                           userData, &handlerRef);
```

is equivalent to

```
WindowRef theWindow;
EventTargetRef theTarget;

theTarget = GetWindowEventTarget(theWindow);
InstallEventHandler (theTarget, handlerUPP,
```

```
                    numTypes, typeList,
                    userData, &handlerRef);
```

A similar macro, `InstallApplicationEventHandler`, needs no parameter to identify the application itself as the target; the call

```
InstallApplicationEventHandler (handlerUPP,
                                numTypes, typeList,
                                userData, &handlerRef);
```

is equivalent to

```
theTarget = GetApplicationEventTarget();
InstallEventHandler (theTarget, handlerUPP,
                    numTypes, typeList,
                    userData, &handlerRef);
```

In all of these cases, the `typeList` parameter specifies the event types for which the handler is to be installed. This parameter is nominally declared as a pointer to an event type specifier giving the class and kind of a single event type; but since the C language considers pointers and arrays to be equivalent, it may actually designate an array of such specifiers for more than one type. The `numTypes` parameter tells how many event types are being specified. For example, the following code installs a single handler for both key-down and key-repeat events:

```
EventTypeSpec    eventTypes[2];
EventHandlerUPP  handlerUPP;

eventTypes[0].eventClass = kEventClassKeyboard;
eventTypes[0].eventKind  = kEventRawKeyDown;

eventTypes[1].eventClass = kEventClassKeyboard;
eventTypes[1].eventKind  = kEventRawKeyRepeat;

handlerUPP = NewEventHandlerUPP(KeyboardHandler);

InstallApplicationEventHandler (handlerUPP,
                                2, eventTypes,
                                NULL, NULL);
```

The `userData` parameter to `InstallEventHandler` is a pointer to an arbitrary data value. Any value you supply for this parameter will later be passed back to your handler routine each time it's called. You can use this capability for any purpose that makes sense to your program; for example, you can use it to pass a window reference to the handler for window events.

Finally, `handlerRef` is an output parameter that returns an event handler reference, an opaque object representing the new event handler. The handler reference is needed as a parameter to Carbon routines such as `AddEventTypesToHandler` and `RemoveEventTypesFromHandler`, for dynamically changing the event types to which a handler applies, and `RemoveEventHandler`, for deinstalling it. If you're not going to be using any of these operations, you can simply pass `NULL` for the `handlerRef` parameter, indicating that no handler reference should be returned. In particular, the handler will be disposed of automatically when you dispose of the target object it's associated with, so there's no need to call `RemoveEventHandler` explicitly unless for some reason you want to deinstall the handler while the underlying target object still exists.

Important: Note that if you install an event handler from a plugin, you must explicitly remove your handler before the plugin unloads. Otherwise, the system may attempt to call event handling code that no longer exists.

Listing 2-2 shows an initialization function that installs an event handler for window close events.

Listing 2-2 Installing a Carbon event handler

```
#define kWindowTop 100
#define kWindowLeft 50
#define kWindowRight 250
#define kWindowBottom 250

void Initialize (void)

   // Do one-time-only initialization

   {
      WindowRef         theWindow;                // Reference to window
      WindowAttributes  windowAttrs;              // Window attribute flags
      Rect              contentRect;              // Boundary of content region
      EventTypeSpec     eventType;                // Specifier for event type
      EventHandlerUPP   handlerUPP;               // Pointer to event handler routine

      windowAttrs = kWindowStandardDocumentAttributes      // Standard document window
                  | kWindowStandardHandlerAttribute;  // Use standard event handler
      SetRect (&contentRect, kWindowLeft,  kWindowTop,     // Set content rectangle
                             kWindowRight, kWindowBottom);
      CreateNewWindow (kDocumentWindowClass, windowAttrs,  // Create the window
                       &contentRect, &theWindow);

      SetWindowTitleWithCFString (theWindow, CFSTR("Happy Cows")); // Set title

      eventType.eventClass = kEventClassWindow;          // Set event class
      eventType.eventKind  = kEventWindowClose;          // Set event kind
      handlerUPP = NewEventHandlerUPP(DoWindowClose);    // Point to handler
      InstallWindowEventHandler (theWindow, handlerUPP,  // Install handler
                                 1, &eventType,
                                 NULL, NULL);

      ShowWindow (theWindow);                       // Display window on the screen

      InitCursor ();                                // Set standard arrow cursor

   }  /* end Initialize */
```

By specifying the `kWindowStandardHandlerAttribute` when we call `CreateNewWindow`, we automatically install the standard window handlers. Alternatively, you could call the `InstallStandardEventHandler` function, specifying the window as the event target.

Important: The Carbon Event Manager does not automatically dispose of the event handler UPP, so you should call `DisposeEventHandlerUPP` when you are done with it.

Listing 2-3 shows an event handler that can respond to the window close event registered in Listing 2-2.

Listing 2-3 A window close event handler

```
pascal OSStatus DoWindowClose (EventHandlerCallRef  nextHandler,
                               EventRef             theEvent,
                               void*                userData)

   // Handle window close event

   {
      DoCloseStuff();                // Do any interesting stuff

      return noErr;                                  // Report success

   }  /* end DoWindowClose */
```

You should be aware that the Carbon Event Manager provides a standard handler for the window close event, so this example handler is useful only if you wanted to override the standard close behavior. However, installing your own handler can also be useful if you want to augment the standard behavior. For example, you may want to display a dialog asking if the user wants to save changes before letting the standard handler close the window. To do so, you use the function `CallNextEventHandler`.

The `CallNextEventHandler` function uses `theEventHandlerCallRef` parameter (passed to your event handler), which is a pointer to the next event handler in the calling hierarchy. The Carbon Event Manager will relay the event to the next handler after this one in the hierarchy of handlers for the given type of event, continuing up until it finds a handler willing to accept and process the event. A hander that chooses not to handle the event returns `eventNotHandledErr`, while one that does should return `noErr` after it has finished processing. Assuming that you have not installed any other window close handlers, Listing 2-4 shows how you can use `CallNextEventHandler` to add pre- and post-processing to the standard window close handler.

Listing 2-4 Augmenting the standard window close handler

```
pascal OSStatus DoWindowClose (EventHandlerCallRef  nextHandler,
                              EventRef             theEvent,
                              void*                userData)

   // Example event handler to illustrate explicit event propagation

   {
      OSStatus  result;                              // Function result


      /* Your preprocessing here */                  // Do preprocessing

// Now propagate the event to the next handler (in this case the standard)
      result = CallNextEventHandler (nextHandler, theEvent);

      if (result == noErr)                           // Did it succeed?
         {
            /* Your postprocessing here */          // Do postprocessing

            /* Note that at this point the  */
            /* standard handler has removed */
            /* the window, so any attempts  */
            /* to access it will cause an   */
            /* error.                       */

            return noErr;                            // Report success
```

```
        }  /* end if (result == noErr) */
    else
        return result;                          // Report failure

}  /* end ThisHandler */
```

Important: Your code should not make any assumptions about how the standard handler behavior for an event is implemented, as this may change in the future.

Event Parameters

Many events require more information than just the basic event to be truly useful. For example, knowing that the mouse was clicked is usually not very interesting unless you know where the click occurred. This additional information is embedded in the event reference structure, and you need to call the function `GetEventParameter` to obtain it. These additional parameters are identified by parameter name and type. A mouse-down event, for example, has four event parameters:

- `kEventParamMouseLocation`, a point (parameter type `typeQDPoint`) giving the screen coordinates at which the mouse button was pressed
- `kEventParamMouseButton`, an integer code (parameter type `typeMouseButton`) identifying which button was pressed (allowing support for a one-, two-, or three-button mouse)
- `kEventParamKeyModifiers`, a set of flag bits (parameter type `typeUInt32`) telling which modifier keys, if any, were being held down at the time the button was pressed
- `kEventParamClickCount`, an integer (parameter type `typeUInt32`) telling how many times the button was clicked in the same location (1 for a single click, 2 for a double click, and so on)

To obtain the mouse location from the event reference `mouseDownEvent`, you would make the following call:

```
EventRef mouseDownEvent;
Point wheresMyMouse;

GetEventParameter (mouseDownEvent, kEventParamMouseLocation, typeQDPoint,
                   NULL, sizeof(Point), NULL, &wheresMyMouse);
```

The values `kEventParamMouseLocation` and `typeQDPoint` specify that you want to obtain the mouse location parameter which is of type `Point`. (There are also a pair of arguments for returning the actual parameter type and size of the value returned; you can specify `NULL` for these arguments if you don't need this information or don't expect the actual type and size to differ from those requested.) Obviously, certain parameter values only make sense for certain types of events (for example, you couldn't obtain a mouse location from the event reference for a window update).

Many events specify a `kEventParamDirectObject` parameter, which usually indicates the actual object the event acted upon. For example, the direct object parameter for a window activation event (`kEventWindowActivated`) would be the reference to the window being activated (that is, a `WindowRef`).

In some cases, you can modify the behavior of an event by setting the value of one or more event parameters with the related Carbon function `SetEventParameter`. For example, if you wanted to snap the window to a particular position as it was being dragged, you could install a handler on the `kEventWindowBoundsChanging` event (which indicates that the window bounds are changing because of resizing or movement) and use `SetEventParameter` to set the origin when some condition is met.

The *Carbon Event Manager Reference* lists the permissible parameters (and the associated values to pass to `GetEventParameter`) for many event kinds. Documentation for other event parameters is available in the API reference for each technology (such as the *HIObject Reference*).

Note: In Mac OS X, all events support the `kEventParamPostTarget` parameter (`typeEventTargetRef`), which lets you indicate to the standard event dispatcher where an event should be sent.

Other Event Attributes

In addition to the event parameters, you can obtain other attributes of an event by calling various accessor functions. For example, the functions `GetEventClass` and `GetEventKind` each accept an event reference as a parameter and return a 32-bit integer representing the event's class and kind, respectively:

```
EventRef  theEvent;
UInt32    eventClass;
UInt32    eventKind;

eventClass = GetEventClass(theEvent);
eventKind  = GetEventKind(theEvent);
```

Similarly, the function `GetEventTime` returns the time an event occurred, expressed as a floating point value of type `EventTime` measured in seconds since the system was started up:

```
EventRef   theEvent;
EventTime  timeInSeconds;

timeInSeconds = GetEventTime(theEvent);
```

Another Carbon routine, `GetCurrentEventTime`, returns the current time in seconds since system startup:

```
EventTime  currentTime;

currentTime = GetCurrentEventTime();
```

The Carbon interface defines a set of convenience constants for expressing event times and intervals:

```
#define kEventDurationSecond       1.0
#define kEventDurationMillisecond  ((EventTime)(kEventDurationSecond / 1000))
#define kEventDurationMicrosecond  ((EventTime)(kEventDurationSecond
                                                / 1000000))
#define kEventDurationNanosecond   ((EventTime)(kEventDurationSecond
                                                / 1000000000))
#define kEventDurationMinute       kEventDurationSecond * 60
#define kEventDurationHour         kEventDurationMinute * 60
#define kEventDurationDay          kEventDurationHour * 24
#define kEventDurationNoWait       0.0
#define kEventDurationForever     -1.0
```

These constants are especially useful when creating event timers. See "Installing Timers" (page 38) for more information.

Queue-Synchronized Events (Mac OS X v.10.2 and Later)

Beginning with Mac OS X version 10.2, your application can distinguish between two different user event states. The hardware state is the actual state of an input device. The *queue-synchronized state* is the state of the device according to the events dispatched from the event queue.

Depending on how long it takes to pop the event off the queue and dispatch it, this queue-synchronized user event may be different from the actual state of the hardware. For example, when the user presses the mouse, a mouse down event is placed into the event queue. If the user releases the mouse immediately while the mouse down event is being handled, then the queue-synchronized mouse button state is still down, but the hardware button state is up.

In most cases, you should determine the state of a user input device by checking the queue-synchronized state rather than polling the hardware directly. Not only does this method use less processor time, but it also provides a better user experience when using nonhardware input methods to place events in the queue (for example, when taking input through Apple events or speech recognition).

Obtaining Mouse and Keyboard Modifer States

The queue-synchronized state of the mouse button and any keyboard modifiers is determined by state variables set by the event dispatcher. For example, the "mouse button state" variable is set to "down" when a mouse-down event is dispatched, and it remains in that state until a mouse-up event is dispatched. You can use the following functions to obtain the values of these state variables:

- `GetCurrentEventButtonState` determines the queue-synchronized state of the mouse button(s). You should use this function instead of the `Button` or `GetCurrentButtonState` functions.

  ```
  UInt32 GetCurrentEventButtonState(void);
  ```

  Bit zero indicates the state of the primary button, bit one the state of the secondary button, and so on.

- `GetCurrentEventKeyModifiers` determines the queue-synchronized keyboard modifier state. You should use this function instead of `EventAvail` or `GetCurrentKeyModifiers`.

  ```
  UInt32 GetCurrentEventKeyModifiers(void);
  ```

  See Table 2-1 (page 31) for a listing of keyboard modifier bits.

Note that `GetCurrentEventButtonState` and `GetCurrentEventKeyModifiers` do not return useful values if your application is not active. If your application wants to determine the current mouse or keyboard modifier state while in the background, it must query the hardware using `GetCurrentButtonState` or `GetCurrentKeyModifiers`.

Obtaining the Current User Event

When a user event is dispatched, the Carbon Event Manager caches it as the current event and disposes of it after the event is handled. The `GetCurrentEvent` function retrieves the event currently being dispatched (which could be `NULL` if the event is not a user event):

```
EventRef GetCurrentEvent(void);
```

You should call this function only from an event handler to determine what user event (if any) triggered the call to your handler.

Command Events

The Carbon Event Manager has a special class of events that correspond with the command IDs introduced with Mac OS 8.0. A command ID is a location-independent way to identify an action or command. For example, if you associate a command ID with a menu item, a command event is generated whenever that item is selected, whether by mouse selection or keyboard equivalent. If you associate the same ID with a button, then a menu selection, keyboard equivalent command, or button press will all generate the same event. Because standard handlers can take care of much of the busywork (such as toggling the button, flashing the selected menu or menu item), you only need to handle one event for three different types of selection.

Command events are initially sent to the event target associated with the command. For example, a menu command event is sent to the menu event target, while a command associated with a button is sent to that control.

Command event handlers can be placed anywhere in the containment hierarchy, but it's usually convenient to place them at the application level. Doing so allows you to install one handler that can take the event whether it propagates up the containment hierarchy from a control or from a menu.

Command IDs are 32-bit integers, but they are usually specified as a four-character code. Many common menu items and controls have reserved codes. For example,

```
kHICommandOK        = 'ok '; // the OK button (as in a dialog)
kHICommandCopy      = 'copy';// the Copy menu item
kHICommandAbout     = 'about';// the About item
```

If you want to create custom command IDs, you must define them in your application and register them by calling the Menu Manager function `SetMenuItemCommandID` or the Control Manager function `SetControlCommandID`, depending on the desired target.

Note: If you use the Interface Builder development tool to build your user interface, you can assign command IDs directly to your controls and menus without having to write any code. All you need to do is define and install the proper handlers to process the generated command events.

The event class for commands is `kEventClassCommand`, and to receive commands to process, the event kind is `kEventCommandProcess`. The command ID itself is stored in the event reference and you need to call the `GetEventParameter` function to retrieve it. For example, to handle a (fictitious) menu item Explode, you would need to first define the command ID, register it with the Menu Manager, and then install the handler in your initialization code:

```
const MenuCommand kCommandExplode = FOUR_CHAR_CODE ('Boom');

void MyInitialize()
    {
    …
    SetMenuItemCommandID (GetMenuRef(mFile), iExplode, kCommandExplode);
    …
    EventTypeSpec myEvents = {kEventClassCommand, kEventCommandProcess};
    InstallApplicationEventHandler(NewEventHandlerUPP(MyEventHandler),
                                1, &myEvents, 0, NULL);
    …
    }
```

The constant `iExplode` represents the index value of the Explode item in the File menu.

This example installs the handler `MyEventHandler` on the application target, but you can choose a different target if that better suits your needs.

Your actual event handler needs to obtain the command ID of the Explode command using `GetEventParameter`. The command ID is stored as the direct object parameter:

```
HICommand commandStruct;

GetEventParameter (event, kEventParamDirectObject,
                    typeHICommand, NULL, sizeof(HICommand),
                    NULL, &commandStruct);

if (commandStruct.commandID == kCommandExplode)
    {
        // process explode command
    }
```

Text Events

The Carbon Event Manager provides two ways of obtaining keyboard information: as raw keyboard events, or as text input events. (Text input events are those that have been processed by the Text Services Manager). To avoid conflict with other input methods, you should rely on text input events for handling text.

Note: If your text input needs are modest, you may be able to use the Multilingual Text Engine (MLTE), which does most of the text event handling for you, instead of writing your own handlers.

Text input events are of class `kEventClassTextInput`, and the event kind used to signify text input is `kEventTextInputUnicodeForKeyEvent`. Text returned by a text input event may be a character or a string, depending on the circumstances, so you should not make assumptions about its length. For more information about text input methods as well as about event kinds directly related to the Text Services Manager, see the Text Services Manager documentation.

To obtain the actual text, you need to call the `GetEventParameter` function, specifying the `kEventParamTextInputSendText` parameter, as shown in Listing 2-5

Listing 2-5 Obtaining text from a text input event

```
EventRef     theTextEvent;
UniChar      *text;
UInt32       actualSize;

GetEventParameter (theTextEvent, kEventParamTextInputSendText,
                typeUnicodeText, NULL, 0, &actualSize, NULL);

text = (UniChar*) NewPtr(actualSize);


GetEventParameter (theTextEvent, kEventParamTextInputSendText,
                typeUnicodeText, NULL, actualSize, NULL, text);
```

This example makes two calls to `GetEventParameter`, the first to obtain the size of the string, and the second to actually obtain it. The rationale for doing so is that the string can be arbitrarily large as it may have resulted from an inline input session intended for a nonRoman script.

If your application doesn't support Unicode, you can examine the `kEventParamTextSendKeyboardEvent` parameter to obtain the raw keyboard event that generated the text event and from that event extract the equivalent Macintosh character codes.

In rare cases where your application might need to handle individual key presses (for example, for game controls, or if it will perform its own keyboard translation), you may want to obtain the key presses before the Text Services Manager processes them. In such cases, you should install handlers to obtain raw keyboard events (class `kEventClassKeyboard`).

In any case, all keyboard and text input events are sent to whichever target currently has the user focus (for example, the window, or the text field control). If desired, you can install an event handler on the *user focus event target*, which you obtain by calling `GetUserFocusEventTarget`. All events directed to the current user focus will then be sent to your handler. If you don't handle the event (or if no handler was installed), the event is then propagated to the actual target that has the user focus.

Mouse Events

In today's graphical user interfaces, the mouse provides the user's primary means of controlling and interacting with the system. All of the user's actions with the mouse are reported to your program in the form of mouse events.

All mouse events have parameters named `kEventParamMouseLocation` and `kEventParamKeyModifiers` giving, respectively, the location of the mouse cursor on the screen and the modifier keys that were being held down at the time the event occurred. The value of `kEventParamMouseLocation` is a point giving the horizontal and vertical position of the mouse in global coordinates.

Note: Beginning in Mac OS X, v. 10.1, mouse events also support the `kEventParamWindowMouseLocation` parameter, which returns the mouse position in coordinates local to the window in which the event occurred.

The value of the `kEventParamKeyModifiers` parameter is an unsigned 32-bit integer (type UInt32) containing flag bits corresponding to the various modifier keys. The mask constants shown in Table 2-1 can be used to extract the bit representing any desired modifier key. A bit value of 1 means that the given key was down when the event occurred; 0 means it was not. Thus, for example, you could use the code in Listing 2-6 to determine whether the Caps Lock key was down at the time of a mouse event:

Listing 2-6 Obtaining the modifier key for a mouse event

```
EventRef  theEvent;
UInt32    modifierKeys;

GetEventParameter (theEvent,
                   kEventParamKeyModifiers,
                   typeUInt32, NULL,
                   sizeof(modifierKeys), NULL,
                   &modifierKeys);

if (modifierKeys & alphaLock)
/* Caps Lock down */
else
   /* Caps Lock not down */
```

Table 2-1 Mask constants for modifier keys

Mask constant	Modifier
`cmdKey`	Command
`shiftKey`	Shift
`alphaLock`	Caps Lock
`optionKey`	Option
`controlKey`	Control
`kEventKeyModifierNumLockMask`	Num lock (Mac OS X only)
`kEventKeyModifierFnMask`	Fn (Function) (Mac OS X only)

For a complete listing of modifier constants, see the `EventModifers` enumeration in `Events.h`.

Mouse Button Events

When the user presses or releases the mouse button, it's reported to your program by a mouse-down or mouse-up event (event kind `kEventMouseDown` or `kEventMouseUp`), respectively. Ordinarily, such events are handled by the standard event handler, which analyzes them and converts them into higher-level events representing the meaning of the mouse action, such as `kEventWindowClose` (when the user clicks a window's close button), `kEventWindowClickContentRgn` (when the click is in the window's contents),

kEventControlHit (when it's in a control such as a push button or checkbox), or kEventCommandProcess (when the user chooses a command from a menu). These higher-level events are usually all your program needs to be concerned with. However, you're free to intercept the "raw" mouse events and handle them yourself if necessary.

In addition to the kEventParamMouseLocation and kEventParamKeyModifiers parameters shared by all mouse events, mouse-up and mouse-down events have two additional parameters: kEventParamMouseButton and kEventParamClickCount. The latter is used to identify multiple (for instance, double or triple) mouse clicks, in case your program wishes to assign some special meaning to them. Consecutive presses of the mouse button are considered to constitute a multiple click if they fall within a certain time interval, which is under the user's control via the Mouse pane of System Preferences (on Mac OS X) or the Mouse control panel (on earlier versions). The Classic Event Manager function GetDblTime returns the current value of this interval, expressed in ticks (sixtieths of a second, the time unit used by earlier versions of Mac OS). When a mouse-down event is separated from the previous such event by more than the multiple-click interval, its kEventParamClickCount parameter is set to 1; if it falls within the double-click interval the parameter is incremented by 1 from that of the previous event. Thus the first event in a multiple click has a click count of 1, the second has a click count of 2, the third 3, and so on. (Triple clicks are the most your program should realistically process.)

Unlike earlier versions of Mac OS, which were limited to a one-button mouse, Carbon is designed to support multiple mouse buttons. (Theoretically, it can handle as many as 65,535 buttons, though the most you're likely to encounter in practice is 3.) The kEventParamMouseButton parameter of a mouse-down or mouse-up event identifies which button was pressed or released, using one of the following constants:

```
typedef UInt16  EventMouseButton;
enum
   {
      kEventMouseButtonPrimary   = 1,
      kEventMouseButtonSecondary = 2,
      kEventMouseButtonTertiary  = 3

   }; /* end enum */
```

On a two- or three-button mouse, the left button is normally considered primary and the right button secondary, but left-handed users can reverse these settings as a matter of preference. The middle button on a three-button mouse is always the tertiary button.

Important: The Classic Event Manager includes a number of functions that let you poll the state of the primary mouse button. You should avoid using these functions (`Button`, `GetMouse`, `StillDown`, `WaitMouseUp`) (especially on Mac OS X), as they use excessive processor time and slow down the system. Instead of using `StillDown` or `WaitMouseUp`, you should use `TrackMouseLocation` or `TrackMouseRegion`, which are discussed in "Tracking Mouse Movements" (page 33). On Mac OS X 10.2 and later, if you need the current button state, you should use `GetCurrentEventButtonState` (described in "Queue-Synchronized Events (Mac OS X v.10.2 and Later)" (page 27))instead of `Button`. In most cases you're less interested in the instantaneous state of the button than in its transitions from up to down or vice versa, so it's better to keep track of the button state with mouse-down and mouse-up events than to poll it directly. This is especially true in the common situation where you want to track the mouse's movements and take some repeated action for as long as the button is held down.

Tracking Mouse Movements

The basic task of moving the cursor around on the screen to reflect the physical movements of the mouse is handled for you automatically, with no need for any explicit action on your program's part. In addition, each time the cursor location changes by as much as one pixel horizontally or vertically, a mouse-moved event (`kEventMouseMoved`) is generated. If the user is also holding down the mouse button (or any button on a multiple-button mouse), the result is a mouse-dragged event (`kEventMouseDragged`) instead. Both types of event have the usual `kEventParamMouseLocation` and `kEventParamKeyModifiers` parameters, and the mouse-dragged event also has a `kEventParamMouseButton` parameter to identify the button being held down, as described under "Mouse Button Events" (page 31).

As with the primary mouse button, it's possible to poll the mouse's location on the screen directly by calling the Classic Event Manager function `GetMouse`. However, using this kind of direct polling to track the mouse's movements is usually not a good idea. For instance, as mentioned above, a common reason for tracking the mouse is to provide visual feedback on the screen during a drag by performing some repeated action for as long as the user holds down the button. Doing this with an active polling loop such as

```
while ( WaitMouseUp() )
   {
      GetMouse (&mouseLoc);
      /* Provide feedback based on mouse location */

   } /* end while ( WaitMouseUp() ) */
```

is horribly inefficient, needlessly tying up the processor while spinning the loop waiting for something to happen. Using mouse-dragged events to do the tracking offers some improvement, since the event loop suspends execution except while actively processing an event and hence consumes no extraneous processor cycles. This allows the idle time to be put to better use running other programs or processes in the background—including any periodic timers you may have installed yourself (see "Installing Timers" (page 38)). However, there is an even better way, using the Carbon Event Manager function `TrackMouseLocation`, as shown in Listing 2-7.

Listing 2-7 Tracking the mouse with TrackMouseLocation

```
Point               mouseLoc;
MouseTrackingResult trackingResult;

GetMouse (&mouseLoc);
trackingResult = kMouseTrackingMouseDown;
```

```
while (trackingResult != kMouseTrackingMouseUp)
   {
      /* Provide feedback based on mouse location */
      TrackMouseLocation (NULL, &mouseLoc, &trackingResult);

   } /* end while (trackingResult != kMouseTrackingMouseUp) */
```

The call to `TrackMouseLocation` suspends execution until either the mouse's location or button state changes. It then returns, in its second and third parameters, the coordinates of the new mouse location and a tracking result indicating the nature of the mouse occurrence. (The first parameter specifies a graphics port in whose coordinate system to report the mouse location; passing `NULL` for this parameter designates the current port, which is usually what you want.) The tracking result returned is one of the following values:

```
typedef UInt16  MouseTrackingResult;
enum
   {
        kMouseTrackingMouseDown = 1,
        kMouseTrackingMouseUp = 2,
        kMouseTrackingMouseExited   = 3,
        kMouseTrackingMouseEntered  = 4,
        kMouseTrackingMouseDragged = 5,
        kMouseTrackingKeyModifiersChanged = 6,
        kMouseTrackingUserCancelled = 7,
        kMouseTrackingTimedOut = 8,
        kMouseTrackingMouseMoved = 9

   }; /* end enum */
```

Important: The `kMouseTrackingMouseMoved` constant has been repurposed for Mac OS X v.10.2 and later. In earlier system software versions, `kMouseTrackingMouseMoved` was equivalent to the `kMouseTrackingMouseDragged` result, indicating that the mouse was moved while the mouse button was *down*. In Mac OS X v.10.2. and later, you receive the `kMouseTrackingMouseMoved` tracking result if the user moves the mouse while the mouse button is *up*. If you have code that interprets `kMouseTrackingMouseMoved` in its older sense, you should update it before running it on Mac OS X v.10.2 or later.

The tracking results `kMouseTrackingExited` and `kMouseTrackingEntered` are used by another related Carbon routine, `TrackMouseRegion`. This is typically called after a mouse press in a control (such as a checkbox or a window's close button), to track the mouse's movements in and out of the control so you can provide appropriate visual feedback by highlighting and unhighlighting the control accordingly. The standard event handler ordinarily does all this for you and reports the result with a higher-level event such as `kEventWindowClose`, `kEventWindowZoom`, or `kEventControlHit`; but you may occasionally encounter a situation where you need to make the `TrackMouseRegion` call and process the results yourself.

Listing 2-8 shows a code fragment illustrating how to use `TrackMouseRegion` to respond to a mouse press in a control.

Listing 2-8 Tracking the mouse in a region

```
RgnHandle           controlRegion;              // Region occupied by control
Boolean             isInRegion;                  // Mouse released in region?
MouseTrackingResult  trackingResult;             // Tracking result

/* Set controlRegion to control's region */       // Indicate region
```

```
trackingResult = kMouseTrackingMouseEntered;       // Initialize for first
                                                   // pass of loop

while (trackingResult != kMouseTrackingMouseUp) // Loop until released
   {
      switch (trackingResult)                    // Dispatch on tracking result
         {
            case kMouseTrackingMouseEntered:       // Highlight on entry
               /* Highlight control */
               break;

            case kMouseTrackingMouseExited:        // Unhighlight on exit
               /* Unhighlight control */
               break;

         } /* end switch (trackingResult) */

      TrackMouseRegion (NULL, controlRegion,       // Track mouse in region
                        &isInRegion,
                        &trackingResult);

   } /* end while (trackingResult != kMouseTrackingMouseUp) */

if (isInRegion)                                    // Released in region?
   /* Perform associated action */                 // Take action in response
```

You call `TrackMouseRegion` repeatedly for as long as the mouse button remains down, passing as a parameter a region representing the area the control occupies on the screen. Each time the mouse crosses the boundary in or out of the specified region, `TrackMouseRegion` returns with a tracking result of `kMouseTrackingEntered` or `kMouseTrackingExited`, indicating whether to highlight or unhighlight the control. (Mere mouse movements that don't cross the region boundary are not reported.) When the mouse button is released, `TrackMouseRegion` returns the tracking result `kMouseTrackingMouseReleased` along with a Boolean value indicating whether the button was released inside or outside the region; you can then use this information to determine whether to perform the action associated with the control.

Mouse Tracking Regions (Mac OS X v.10.2 and Later)

In Mac OS X version 10.2 and later, you can designate special mouse tracking regions within windows. When the mouse enters one of these regions, your application receives a `kEventMouseEntered` event (`kEventClassMouse`). When the mouse leaves, the application receives a `kEventMouseExited` event. A window can contain any number of regions; each mouse tracking region has a unique ID, which makes it easy to determine which region was entered. If desired, you can also temporarily disable a region.

Mouse tracking regions make it simple to implement various rollover effects such as highlighting a clickable area, or changing the cursor when it enters a region. If you must use processor-intensive polling for mouse locations (for example, in a drawing program), you can use mouse tracking regions to allow pollling only within particular regions of interest.

To create a mouse tracking region, you call the `CreateMouseTrackingRegion` function:

```
OSStatus CreateMouseTrackingRegion (WindowRef inWindow,
                            RgnHandle inRegion,
                            RgnHandle inClip,
                            MouseTrackingOptions inOptions,
                            MouseTrackingRegionID inID,
```

```
                                    void* inRefCon,
                                    EventTargetRef inTargetToNotify,
                                    MouseTrackingRef* outTrackingRef);
```

- The `inWindow` parameter indicates which window owns this region. Mouse tracking regions are always bound to a particular window.
- The `inRegion` parameter is a standard region handle defining the tracking region.
- The `inClip` parameter specifies an optional clip region. If the clip region is valid (that is, you don't pass `NULL`), the active tracking region is the intersection of the tracking region and the clip region.
- For mouse tracking options you can pass either `kMouseTrackingOptionsLocalClip` or `kMouseTrackingOptionsGlobalClip`:
  - `kMouseTrackingOptionsLocalClip` indicates that the region is defined in local coordinates and that the region is clipped to the owning windows's content region.
  - `kMouseTrackingOptionsGlobalClip` indicates the region is defined in global coordinates and that the region is clipped to the owning window's structure region.
- The `inID` parameter holds a unique mouse tracking ID, which is a combination of an `OSType`, which is a four-character code that uniquely defines your application, and an integer:

  ```
  struct MouseTrackingRegionID {
                      OSType signature;
                      SInt32 id;
      }
  ```

  If you have not already done so, you can register an application signature with Apple Developer Technical Support.
- If you want to associate any application-specific data with this region, you can pass it in the `inRefCon` parameter.
- The `inTargetToNotify` parameter is currently unused; pass `NULL`.

On return, you receive a mouse tracking reference which you can pass to additional mouse tracking region functions. This reference is also included in the event reference for the `kEventMouseEntered` and `kEventMouseExited` events. Use the `GetEventParameter` function to obtain the direct object parameter.

Additional useful mouse tracking region functions include the following:

- To dispose of a tracking region, use the `ReleaseMouseTrackingRegion` function:

  ```
  OSStatus ReleaseMouseTrackingRegion (MouseTrackingRef inMouseRef);
  ```

  Note that because mouse tracking regions are associated with a window, disposing the window will also dispose of its tracking regions.
- To increase the reference count of a tracking region, call the `RetainMouseTrackingRegion` function:

  ```
  OSStatus RetainMouseTrackingRegion (MouseTrackingRef inMouseRef);
  ```

  Calling `ReleaseMouseTrackingRegion` decrements the reference count; if the count reaches 0, the tracking region is disposed.
- To obtain the ID of a mouse tracking region, call the `GetMouseTrackingRegionID` function:

```
OSStatus GetMouseTrackingRegionID (MouseTrackingRef inMouseRef,
                                   MouseTrackingRegionID* outID);
```

- To enable or disable a mouse tracking region, call the `SetMouseTrackingRegionEnabled` function:

```
OSStatus SetMouseTrackingRegionEnabled (MouseTrackingRef inMouseRef,
                                        Boolean inEnabled);
```

  You can use this function to adjust tracking regions that are dependent on the state of your application.

- To obtain the application-specific data associated with the tracking region, call the `GetMouseTrackingRegionRefCon` function:

```
OSStatus GetMouseTrackingRegionRefCon (MouseTrackingRef inMouseRef,
                                       void** outRefCon);
```

Mac OS X v10.4 introduced new HIView-based tracking region functions. These work much like the older mouse tracking regions, with the following exceptions:

- You create these tracking areas on a per-view, rather than per-window basis.
- The tracking areas are described using HIShape objects rather than QuickDraw regions.
- The tracking area is identified by a `UInt64` integer rather than a data structure.
- The area entered and exited events are `kEventControlTrackingAreaEntered` and `kEventControlTrackingAreaExited` respectively. You obtain the tracking area reference in the `kEventParamHIViewTrackingArea` parameter (`typeHIViewTrackingAreaRef`).

To create a view-based tracking area, call the `HIViewNewTrackingArea` function:

```
OSStatus HIViewNewTrackingArea(
  HIViewRef               inView,
  HIShapeRef              inShape,       /* can be NULL */
  HIViewTrackingAreaID    inID,
  HIViewTrackingAreaRef *  outRef)
```

To modify a tracking area, call `HIViewChangeTrackingArea`:

```
OSStatus HIViewChangeTrackingArea(
  HIViewTrackingAreaRef   inArea,
  HIShapeRef              inShape)
```

To obtain an area's ID, call `HIViewGetTrackingAreaID`:

```
 OSStatus HIViewGetTrackingAreaID(
   HIViewTrackingAreaRef   inArea,
   HIViewTrackingAreaID *  outID)
```

To dispose of a tracking area, call `HIViewDisposeTrackingArea`:

```
OSStatus HIViewDisposeTrackingArea (HIViewTrackingAreaRef inArea)
```

If possible, you should use view-based tracking areas in place of the older tracking regions.

Installing Timers

Installing a timer is similar to installing an event handler. Timers are associated with a particular event loop (usually the program's main loop), and they fire as the loop runs. Instead of a list of event types, you specify an initial delay before the timer fires for the first time and a timer interval between subsequent firings, both expressed in seconds. (Setting the timer interval to 0 produces a one-shot timer that will fire only once, at the expiration of the initial delay.) As in installing an event handler, you can also supply an arbitrary item of user data that will be passed back to your timer routine each time it's called. The timer routine itself is identified with a universal procedure pointer of type `EventLoopTimerUPP`, obtained using the conversion function `NewEventLoopTimerUPP`. Listing 2-9 shows how to install a timer routine named `TimerAction` in the program's main event loop with an initial delay of 5 seconds, a timer interval of 1 second, and no user data item

Listing 2-9 Installing a timer

```
EventLoopRef       mainLoop;
EventLoopTimerUPP  timerUPP;
EventLoopTimerRef  theTimer;

mainLoop = GetMainEventLoop();
timerUPP = NewEventLoopTimerUPP(TimerAction);

InstallEventLoopTimer (mainLoop,
                       5*kEventDurationSecond,
                       kEventDurationSecond,
                       timerUPP,
                       NULL,
                       &theTimer);
```

The last parameter of the `InstallEventLoopTimer` function is an output parameter that returns a *timer reference* representing the timer just installed. This value is needed as a parameter to various Carbon Event Manager functions that operate on timers, the most important of which is `RemoveEventLoopTimer`, for uninstalling the timer. This same timer reference will also be passed automatically to your timer routine each time it's called.

Important: As with the `InstallEventHandler` function, if you are installing a timer from a plugin, you must explicitly remove the timer before your plugin is unloaded. Otherwise, the still-existent timer may attempt to call code that no longer exists.

The timer routine itself must have the following prototype:

```
pascal void TimerAction (EventLoopTimerRef  theTimer,
                         void* userData);
```

where `theTimer` is the timer reference identifying the timer and `userData` is the data value you supplied at the time the timer was installed.

Note: A timer fires only when the low-level event loop (which fetches events and places them on the event queue) is actually running. For example, if you start a timer that is set to fire in 1 second, then call a handler that does some computation for 5 seconds, the timer will not fire until you complete the calculation and return to the event loop. The following Carbon Event Manager and Classic Event Manager functions will run an event loop: `RunApplicationEventLoop`, `ReceiveNextEvent`, `RunCurrentEventLoop`, `WaitNextEvent`, `GetNextEvent`, and `EventAvail`.

One more useful function is `SetEventLoopTimerNextFireTime`, which resets the interval until the next time the timer fires. For example, if `theTimer` is the timer installed in the example above, the call

```
SetEventLoopTimerNextFireTime (theTimer, kEventDurationMinute);
```

will cause the timer to "sleep" for one minute and then resume its one-second firing cycle. The effect is equivalent to deinstalling the timer and then reinstalling it with a new initial delay and the same timer interval.

A variant of the basic timer is the idle timer, which is available in Mac OS X v.10.2 and later. The idle timer functions just like a normal timer except that it does not fire until the user has been inactive for the initial delay time. Such timers are useful for letting the application know that it is safe to do some processing without interfering with the user. For example, if the user is entering text into a search field, you should wait for a second or two after the user has stopped typing before beginning the search.

To install an idle timer, you call the `InstallEventLoopIdleTimer` function, which has the same format as the basic `InstallEventLoopTimer` function:

```
OSStatus InstallEventLoopIdleTimer(
                            EventLoopRef         inEventLoop,
                            EventTimerInterval   inFireDelay,
                            EventTimerInterval   inInterval,
                            EventLoopIdleTimerUPP    inTimerProc,
                            void *               inTimerData,
                            EventLoopTimerRef *  outTimer);
```

The only difference is that the timer callback routine takes an additional `EventIdleAction` parameter indicating the idle status:

```
pascal void IdleTimerAction (EventLoopTimerRef inTimer,
                                void *inUserData,
                                EventIdleAction inAction );
```

When your idle timer routine is called, it is passed one of three constants indicating the current idle status:

```
enum
{
    kEventLoopIdleTimerStarted,
    kEventLoopIdleTimerIdling,
    kEventLoopIdleTimerStopped
}
EventIdleAction;
```

- `kEventLoopIdleTimerStarted` indicates that the idle period has just begun (and this is the first time your callback is being called for this period).
- `kEventLoopIdleTimerIdling` indicates that your callback is being called in the middle of an idle period.

- `kEventLoopIdleTimerStopped` is sent to your callback function when the idle period ends (for example, when the user hits a key).

For example, say your application wants to calculate the value of pi. When your callback function receives the `kEventLoopIdleTimerStarted` constant, you create a special pi calculation object. Each time you receive `kEventLoopIdleTimerIdling`, you call an object method to calculate the next digit. When you receive `kEventLoopIdleTimerStopped`, you store the currently calculated value of pi and dispose of the pi calculation object.

Processing Events Manually

In most cases, using the `RunApplicationEventLoop` function to collect and dispatch events is the simplest and most practical way to handle events. However, sometimes you may want more control over the event collection and dispatching mechanism, or you may need to process events that don't occur in the main application thread. In cases like these, you can call other Carbon Event Manager functions to manually collect and dispatch your events.

The `RunApplicationEventLoop` function itself calls several Carbon Event Manager functions to accomplish its task:

- `ReceiveNextEvent` runs the low-level event loop, placing events as they occur into the event queue. The function returns when an event you specified occurs, or when the specified timeout is exceeded.

```
OSStatus ReceiveNextEvent(
            UInt32 inNumTypes,
            const EventTypeSpec *inList,
            EventTimeout inTimeout,
            Boolean inPullEvent,
            EventRef *outEvent);
```

  - The `inNumTypes` parameter specifies the number of events for which `ReceiveNextEvent` should return. Passing 0 indicates you want to return on all events.
  - The `inList` parameter points to the `EventTypeSpec` structure or array containing the class and kind of events to return on. Passing `NULL` indicates that you want to return on all events.
  - The `inTimeout` parameter is the duration to wait before timing out.
  - The `inPull` parameter specifies if whether you want `ReceiveNextEvent` to pull the event off the queue when it returns. Passing `true` causes the event to be pulled. If you pass `false`, `ReceiveNextEvent` only peeks at the event to determine its type. You still can dispatch the event, but it remains on the queue.
  - On return, `outEvent` contains the event that caused `ReceiveNextEvent` to return.

- `GetEventDispatcherTarget` gets the event target reference for the *standard toolbox dispatcher*, which is the default target for all events. The toolbox dispatcher determines the proper target for each event (window, control, and so on) and sends the event there. Note that because the toolbox dispatcher is itself a valid event target, you can actually attach a handler to it. Such a handler can intercept an event before it gets sent on to the actual event target.
- `SendEventToEventTarget` dispatches the event to the appropriate event target.
- `ReleaseEvent` releases the event (disposing of it if necessary).

Listing 2-10 shows how you can use these calls to implement the basic functionality of `RunApplicationEventLoop`.

Listing 2-10 Processing events manually

```
EventRef theEvent;
EventTargetRef theTarget;

theTarget = GetEventDispatcherTarget();

    while  (ReceiveNextEvent(0, NULL,kEventDurationForever,true,
            &theEvent)== noErr)
        {

            SendEventToEventTarget (theEvent, theTarget);
            ReleaseEvent(theEvent);
        }
```

The `ReceiveNextEvent` function is blocked forever (`kEventDurationForever`) until an event occurs. Specifying zero and null for the first two parameters indicates that `ReceiveNextEvent` should return on all events. (Alternatively, you could specify that the function wait only for particular events). Passing `true` in the third parameter indicates that the application should take ownership of the event (which means it is pulled off the event queue).

After an event occurs, we dispatch it to the event dispatcher target, which automatically sends it to the proper event target. Because the application owns the event, the application is then responsible for releasing it by calling `ReleaseEvent`. (There is also a complementary function `RetainEvent`, which you can use to increment the reference count of the event, thus ensuring that it will not get disposed before you are finished with it.)

The only drawback to making your own event loop dispatching calls in the main application thread is that you won't get the standard application event handler installed. Specifically, the `RunApplicationEventLoop` function installs handlers to do the following:

- Allow clicks in the menu bar to begin menu tracking
- Dispatch Apple events by calling `AEProcessAppleEvent`
- Respond to quit Apple events by quitting `RunApplicationEventLoop`.

One way to work around this limitation is by creating a dummy custom event handler. When you are ready to process events, create the dummy event yourself, post it to the queue. and then call `RunApplicationEventLoop` (to install the standard application event handler). The dummy event handler can then process the events manually. For an example of using this method, see Technical Q&A 1061 in Developer Documentation Technical Q&As.

Creating Your Own Events

In addition to processing and dispatching events, the Carbon Event Manager also lets you create your own events. You may want to create your own custom events, or you might want to reproduce standard events.

You create an event using the `CreateEvent` function:

```
OSStatus CreateEvent( CFAllocatorRef inAllocator<null>,
                    UInt32 inClassID, UInt32 kind,EventTime when,
                    EventAttributes flags, EventRef* outEvent);
```

- The `inAllocator` parameter refers to the allocator you want to use to allocate memory for the event. You can pass `NULL` to specify the default allocator.
- The `inClassID` and `kind` parameters indicate the event class and kind. If you are creating custom events, you need to define new values that don't conflict with existing event classes and kinds. And, of course, you must specify this class and kind when your register a handler to process this type of event.
- The `when` parameter indicates when the event occurred. You can pass 0 to specify the current event time (as returned by the `GetCurrentEventTime` function). This value may or may not be useful for custom events.
- The `flags` parameter indicates any event attributes you may want to set. The current choices are `kEventAttributeNone` and `kEventAttributeUserEvent`.
- On return, `outEvent` contains the newly-created event reference.

If your event requires additional information, you can add data by calling `SetEventParameter`. If you are creating custom events, you need to define constants for your parameter names and types if they don't already exist. For example, if you define a parameter for a screen location, you may want to define a new parameter name, but you can probably still use `typeQDPoint` for the parameter type.

Once you create an event, you need to send it to a handler. There are two basic methods for doing so:

- You can post the event to a queue by calling the `PostEventToQueue` function. You need to obtain the queue reference for the queue you want to post to by calling either `GetCurrentEventQueue` (which returns the current thread's queue) or `GetMainEventQueue` (which returns the queue for the main application thread). The event you post will not be processed until it is pulled from the queue and dispatched to the appropriate event target.

  Note that in Mac OS X, you can indicate in the event reference where a posted event should be dispatched by specifying a `kEventParamPostTarget` event parameter.
- You can send it directly to the desired event target by calling `SendEventToEventTarget`. If this is a custom event, the target you choose should be the one to which you attached your custom event handler. Dispatching the event yourself will ensure that your handler is called immediately.

Note that if you send an event to the standard toolbox dispatcher and it does not recognize it (that is, it's a custom event), then it will dispatch the event to the application event target (unless you specified an event target in your custom event using the `kEventParamPostTarget` parameter).

If you want to create and process command events, the Carbon Event Manager provides the function `ProcessHICommand`:

```
OSStatus ProcessHICommand (const HICommand* inCommand);
```

When you pass an Command ID to `ProcessHICommand`, it builds a `kEventCommandProcess` event containing the ID and then dispatches the event to either

- a menu, if the command is defined in a menu, or
- the current user focus

Carbon Events in Multiple Threads

The Carbon Event Manager scales to work with multiple execution threads. If you are creating cooperative threads, each thread shares the main application event loop and queue, so your event handling mechanism does not change. However, the `RunApplicationEventLoop` function never explicitly yields to other threads, so you should create a timer that will call the Thread Manager function `YieldToAnyThread` as necessary.

If you create preemptively-scheduled threads, each such thread contains its own event queue and needs to be processed independently.

Because `RunApplicationEventLoop` works only for the main execution thread, any preemptive threads you create should use `ReceiveNextEvent` to process events, as described in "`Processing Events Manually`" (page 40). Depending on the thread, you can wait for particular events to occur or process every event (much the way `RunApplicationEventLoop` does).

Important: Event loops in preemptive threads you create do not receive user events. The only events your threads receive are those created by your application.

You use the event queues primarily to communicate between threads. For example, if you wanted your preemptive thread to tell the main application it was finished processing data, it could post a custom event on the main application event queue. One advantage of this method is that your application does not have to use extra processing time polling a Multiprocessing Services queue or semaphore.

Depending on the circumstances, either Carbon event queues or Multiprocessing Services notification methods may be suitable for signaling between threads. If you want to use Carbon event queues, here is breakdown of how you might do it:

- After first creating the thread, it should call `GetCurrentEventQueue` to obtain its queue reference. It can then create a custom event signifying that it is ready for use, call `SetEventParameter` to store the queue reference in the event, then post the event to the main thread. It can then call `ReceiveNextEvent`, blocking until someone sends it an event.
- When the main thread receives the ready event, the appropriate handler can call `GetEventParameter` to extract the queue reference. Then, whenever it needs to signal the other thread, it can create a "start processing" event and post it to the proper queue.
- Whenever the preemptive thread receives the process event, it can carry out its particular task. Afterwards, it posts a "processing completed" event to the main event queue and returns to a blocked state in `ReceiveNextEvent`.
- When it comes time to terminate the thread, the main application thread sends a termination event and waits for confirmation from the thread.

For more information about creating cooperatively-scheduled threads, see the Thread Manager documentation and Technical Q&A 1061, "RunApplicationEventLoop and the Thread Manager." For information about creating preemptively-scheduled threads, see the Multiprocessing Services documentation.

Modal Event States

If you need to create application-modal dialogs, you can use several Carbon Event Manager functions to enter and exit the modal state. A modal dialog is a window that allows no other application actions until the window is dismissed. For example, an alert that warns the user about the consequences of some action is typically a modal dialog.

For more information about the proper design and usage of modal dialogs, see *Inside Mac OS X: Aqua Human Interface Guidelines*.

The simplest way to enter the modal state is to call the function `RunAppModalLoopForWindow`, passing the window reference of the window you want to make modal. This function is analogous to the `RunApplicationEventLoop` functions for applications. It runs a sub-event loop, disables the menu bar and dispatches events.

When in a modal state, the standard toolbox dispatcher only processes events for the modal window and any window above it (that is, closer to the front). Typically a modal window is frontmost, but if another window is in front of it, that window will also receive events. This feature was designed to allow stacked modal dialogs. See "Processing Events Manually" (page 40) for more information about the toolbox dispatcher.

Note: You should use `RunAppModalLoopForWindow` instead of the older Dialog Manager function `ModalDialog`.

To leave the modal state, you call the function `QuitAppModalLoopForWindow`.

To make construction of modal dialogs simpler, the Carbon Event Manager also includes some utility functions for setting the default and cancel buttons.

```
OSStatus SetWindowDefaultButton(
                WindowRef    inWindow,
                ControlRef   inControl);      /* can be NULL */

OSStatus SetWindowCancelButton(
                WindowRef    inWindow,
                ControlRef   inControl);      /* can be NULL */

OSStatus GetWindowDefaultButton(
                WindowRef     inWindow,
                ControlRef *  outControl);

OSStatus GetWindowCancelButton(
                WindowRef     inWindow,
                ControlRef *  outControl);
```

Calling the "set" versions of these functions causes the standard event handlers to map keyboard input to the respective controls: pressing the Return or Enter keys will activate the default button, and pressing escape or Command-period will activate the cancel button.

As with the standard event loops, you can also choose to run the modal event loop manually and dispatch events yourself. To do so, you call the low-level function `BeginAppModalStateForWindow` for the desired window. Once in this state you can call the usual low-level event processing functions. (`ReceiveNextEvent`,

`RunCurrentEventLoop`). Note that because the event filtering occurs in the toolbox dispatcher (not the event queue), it is possible to receive and process events that are not related to the window. To leave the modal state, you call `EndAppModalStateForWindow`.

Document Revision History

This table describes the changes to *Carbon Event Manager Programming Guide*.

Date	Notes
2005-07-07	Changed title from "Handling Carbon Events." Made minor content updates and bug fixes.
	Added some information about HIView-based mouse tracking areas to "Mouse Tracking Regions (Mac OS X v.10.2 and Later)" (page 35).
	Described workaround trick for getting standard application event handlers when processing events manually.
	Removed event parameters appendix (now documented with the event kinds in the API references).
	Added link to *Apple Events Programming Guide*.
2003-06-04	Added note to "Standard Handlers" (page 11) describing the difference between default behavior and standard event handling for an event.
	Updated art files.
2003-04-30	Specified in the parameter appendix that `kEventWindowContextualMenuSelect` also contains the parameters available to `kEventMouseDown`.
	Also removed superfluous section head.
2002-10-04	Updated for Mac OS X 10.2.
	Added text to "Carbon Events Versus WaitNextEvent" (page 15) indicating that standard menu event and Apple event handlers are installed only when you call `RunApplicationEventLoop`.
	Added warning to "Creating and Registering an Event Handler" (page 21) that plugins must explictly remove any event handlers they install before being unloaded. Otherwise, the system may attempt to call event handler code that no longer exists. Added similar warning to "Installing Timers" (page 38).
	Corrected error in Listing 3-4 (page 24), adding an `EventRef` parameter to the `CallNextEventHandler` call.
	Added new section, "Queue-Synchronized Events (Mac OS X v.10.2 and Later)" (page 27).

Date	Notes
	Added note about the new `kEventParamWindowMouseLocation` parameter to "Mouse Events" (page 30).
	Removed all suggestions to use Classic Event Manager functions such as `Button` and `WaitMouseUp` in "Mouse Button Events" (page 31), and replaced them with a warning that these functions use excessive processor time (especially on Mac OS X).
	Replaced deprecated mouse tracking result constants (`kMouseTrackingMousePressed/Released/Moved`) in "Tracking Mouse Movements" (page 33) with newer equivalents: `kMouseTrackingMouseDown/Up/Dragged`. Added new tracking result constants as well.
	Added warning about the `kMouseTrackingMouseMoved` constant: The `kMouseTrackingMouseMoved` constant has been repurposed for Mac OS X version 10.2 and later. In earlier system software versions, `kMouseTrackingMouseMoved` was equivalent to the `kMouseTrackingMouseDragged` result, indicating that the mouse was moved while the mouse button was *down*. In Mac OS X 10.2. and later, you receive the `kMouseTrackingMouseMoved` tracking result if the user moves the mouse while the mouse button is *up*. If you have code that interprets `kMouseTrackingMouseMoved` in its older sense, you should update it before running it on Mac OS X version 10.2 or later.
	Removed unnecessary `GetMouse` call in Listing 3-8 (page 34).
	Added new section "Mouse Tracking Regions (Mac OS X v.10.2 and Later)" (page 35).
	Added material about idle timers to "Installing Timers" (page 38), which are available in Mac OS X 10.2 and later.
	Added information about the `kEventParamPostTarget` event parameter in "Creating Your Own Events" (page 41). Also updated information about how the standard event dispatcher handles unrecognized events (it sends them to the application event target).
	Updated available event parameters in "Event Parameters and Types for Common Event Kinds." Added event parameters for application events.
2001-06-22	Added additional sections "The Handler Stack" (page 10) and "An Event Propagation Example" (page 12) to clarify how events are propagated through the containment hierarchy.
	Removed note from "Standard Handlers" (page 11) indicating that the standard handler acts like the outermost object in a containment hierarchy. The actual mechanism is a bit more subtle than that.
	Added third item to the event handling list in "The Event Model" (page 14) indicating that events that are not processed will be discarded.

Date	Notes
	Defined window dimension constants in Listing 3-2 (page 23) to make the code compilable.
	Clarified in "Command Events" (page 28) that command events are first sent to the event target they are associated with (that is, the menu or control).
2001-05-11	WWDC draft.
	Document title changed to *Handling Carbon Events* from the *Carbon Event Manager Handbook*.
	Correction: The Carbon Event Manager does not replace the functionality of the Notification Manager.
	Terminology change: References to the "default event handler" changed to "standard event handler" to better match the API and internal usage.
	System Requirements section moved to the introductory chapter. Also, the Carbon Event Manager is available in CarbonLib 1.1.1 and later, not 1.2 as previously stated.
	Added Note to "Standard Handlers" (page 11) (formerly Default Handlers) indicating that the standard window handler includes support for control events, and the standard application handler includes support for menu events.
	In "Carbon Events Versus WaitNextEvent" (page 15), maximum timeout for `WaitNextEvent` should be 7FFFFFFF. Idle processing information changed to use null events instead of idle events.
	In "Event Classes and Kinds" (page 19), clarified which handler actually calls SetPort and Begin/EndUpdate. It's `kEventWindowUpdate`, not `kEventDrawContent`.
	Indicated that `RunApplicationEventLoop` installs ths standard application event handler, which includes a simple quit Apple event handler in "Executing the Event Loop" (page 20).
	Clarified in "Creating and Registering an Event Handler" (page 21) that the event handler UPP is not automatically released; you must dispose of it yourself.
	Added warning comment in Listing 3-4 (page 24) to indicate that the standard handler for the window close event removes the window, you can't attempt to access it as part of your post-processing.
	Correction in "Other Event Attributes" (page 26): `EventTime` is a floating point number, not an integer.

Date	Notes
	In "Command Events" (page 28), command IDs were introduced with Mac OS 8.0, not 8.5 as stated. Also the `myEvents` parameter in the `InstallApplicationEventHandler` call needs an "&". The `commandID` field of the `commandStruct` variable starts with a lower-case c, not upper case. Added note indicating that Interface Builder makes it easy to assign command IDs to controls and menu items.
	In "Text Events" (page 29), the user focus event target is a valid event target that does not change. If you install a handler on this target, then any events targeted at the current user focus will automatically be sent to your handler before being sent to the actual event target that has the user focus.
	Correction in "Mouse Events" (page 30): The coordinates of the mouse position in `kEventParamMouseLocation` are given in global coordinates, not local as previously stated.
	Added Mac OS X only mask constants `kEventKeyModifierNumLockMask` and `kEventKeyModifierFnMask` to Table 3-1 (page 31).
	Correction in "Installing Timers" (page 38): Event handlers are not associated with a particular event loop. Also added clarification in the Note as to which functons will run the event loop.
	In "Processing Events Manually" (page 40), clarified what handlers `RunApplicationEventLoop` installs: handlers to begin menu tracking, dispatch Apple events, and process the quit Apple event.
	In "Creating Your Own Events" (page 41), indicated that you can pass 0 for the `when` parameter in `CreateEvent`, which specifies the current event time.
	In "Event Parameters and Types for Common Event Kinds" updated text input event constant names to reflect latest header.
2001-04-30	Preliminary review draft. Includes material from the older Carbon Events document.

Control Events Versus Classic Control Messages

The control event constants defined by the Carbon Event Manager generally map to control messages sent to control defprocs (CDEFs) as shown in Table A-1 (page 51).

Table A-1 Control Events versus Control defproc messages

Control event constant	Control defproc message(s)
`kEventControlInitialize`	`initCntlkControlMsgTest-NewMsgSupportkControlMsgGetFeatures` (in that order)
`kEventControlDispose`	`dispCntl`
`kEventControlGetOptimalBounds`	`kControlMsgCalcBestRect`
`kEventControlDefInitialize`	Same as for `kEventControlInitialize`.
`kEventControlDefDispose`	Same as for `kEventControlDispose`
`kEventControlHit`	No equivalent message
`kEventControlSimulateHit`	No equivalent message
`kEventControlHitTest`	`testCntl`
`kEventControlDraw`	`drawCntl`
`kEventControlApplyBackground`	`kControlMsgSetUpBackground`
`kEventControlApplyTextColor`	`kControlMsgApplyTextColor`
`kEventControlSetFocusPart`	`kControlMsgFocus`
`kEventControlGetFocusPart`	No equivalent message.
`kEventControlActivate`	`kControlMsgActivate` (with param = 1)
`kEventControlDeactivate`	`kControlMsgActivate` (with param = 0)
`kEventControlSetCursor`	`kControlMsgSetCursor`
`kEventControlContextualMenuClick`	`kControlMsgContextualMenuClick`
`kEventControlTrack`	`kControlMsgHandleTracking`
`kEventControlGetScrollToHereStartPoint`	No equivalent message.
`kEventControlGetIndicatorDragConstraint`	`thumbCntl`

Control event constant	Control defproc message(s)
kEventControlIndicatorMoved	kControlMsgDrawGhost (for nonlive tracking) or kControlMsgCalcValueFromPos (for live tracking).
kEventControlGhostingFinished	posCntl
kEventControlGetActionProcPart	No equivalent message.
kEventControlGetPartRegion	kControlMsgGetRegion for controls that support GetRegion, calcCntlRgn or calcThumbRgn otherwise.
kEventControlGetPartBounds	No equivalent message.
kEventControlSetData	kControlMsgSetData
kEventControlGetData	kControlMsgGetData
kEventControlValueFieldChanged	drawCntl with param = kControlIndicatorPart
kEventControlAddedSubControl	kControlMsgSubControlAdded
kEventControlRemovingSubControl	kControlMsgSubControlRemoved
kEventControlArbitraryMessage	Anything sent using SendControlMessage
kEventWindowGetClickActivation (of event class kEventClassWindow)	kControlMsgGetClickActivation

Some messages are no longer supported, as shown in Table A-2

Table A-2 Unsupported CDEF messages

Message	Why No Carbon Event Equivalent
calcCRgns	Obsolete. Use kControlMsgGetRegion instead.
autoTrack	Use kControlMsgHandleTracking instead.
dragCntl	Obsolete. No one needs to use this anymore.
drawThumbOutline	Obsolete. Use kControlDrawMsgGhost, kControlMsgCalc-ValueFromPos, and posCntl instead.
kControlMsgKeyDown	Only marginal support available on Mac OS X using kEventControl-ArbitraryMessage. Use kEventTextInput class of Carbon events instead.
kControlMsgIdle	Only marginal support available on Mac OS X using kEventControl-ArbitraryMessage. The CDEF should install a timer instead.

Message	Why No Carbon Event Equivalent
`kControlMsgSubValueChanged`	Only marginal support available on Mac OS X using `kEventControlArbitraryMessage`. CDEF should install `kEventControlValueFieldChanged` on its children instead.
`kControlMsgFlatten`	Obsolete. Never implemented.

Glossary

blocked The state where an application or thread is waiting for some event or action to occur. While blocked, that particular code path uses no processor time.

Classic Event Manager The event handling interface used in Mac OS applications before the Carbon Event Manager. The Classic Event Manager often required a certain amount of polling of the event queue.

containment hierarchy A hierarchy of event targets that determines which handler is to be called to process an event. Events are initially sent to the innermost (or lowest) relevant target in the hierarchy. If the handler associated with that event target does not handle the event (or if no handler exists), then the event is propagated to the next target in the hierarchy. If no handler in the hierarchy processes the event, the default handler is called.

event A constant that notifies an application that some action is occurring, or has occurred.

event class The general category an event belongs to, typically associated with an particular action or user-interface element. Example classes are window events and volume events. Compare event kind (page 55).

event handler A callback procedure that processes one or more events.

event kind A specific type of event within an event class (for example, a mouse-down event). Compare event class (page 55).

event loop In the Carbon Event Manager, an execution loop that obtains events from the Window Server and places them in an event queue. The event loop also fires timers.

event queue A first-in-first-out stack where events pertaining to a thread are stored. Each preemptively-scheduled thread has its own event queue.

event target An object to which an event is sent. An event target is typically a user-interface element, such as a control or a window.

event timer A timer mechanism that fires once, or at periodic intervals, calling a callback procedure when doing so.

event type The combination of event class and event kind that uniquely identifies an event to the Carbon Event Manager. *See also*event class (page 55), event kind (page 55).

main event loop The code loop where the application spends most of its time. The application is blocked while waiting for events. When an event occurs, the application processes it and then returns to the blocked state.

one-shot timer A Carbon event timer that fires only once. *See also*event timer (page 55).

peek To examine an event in an event queue (obtaining its class, kind, parameters and so on) without removing it from the queue. Compare pull (page 55).

pull To remove an event from an event queue. Compare peek (page 55).

queue-synchronized state The state of an input device accordiing to the events that have been dispatched from the event queue. This state may differ from the actual physical state of the input device.

standard event handler The event handler that processes an event if the application did not install one for it.

standard toolbox dispatcher In the Carbon Event Manager, the default event target for events when running under `RunApplicationEventLoop`. Events sent to the standard toolbox dispatcher are automatically routed to the appropriate event targets.

timer *See*event timer (page 55).

toolbox dispatcher *See*standard toolbox dispatcher (page 56).

universal procedure pointer (UPP) A generalized procedure pointer that allows code with different calling conventions to call each other. Some Carbon functions require you to pass UPPs for callbacks because the calling routine doesn't know in advance if your code is Mach-O based or CFM-based.

user focus The window or text field control to which keyboard input is directed. The user can change the user focus by using the mouse or (sometimes) the Tab key.

user focus event target Events sent to this target are automatically sent to the event target that has the current user focus. You can also install a handler on this target to intercept events before they get sent to the current user focus.

WaitNextEvent The function that drove the event loop in older versions of the Mac OS. *See also*Classic Event Manager (page 55).

Index

G

H

I

K

M

O

P

Q

R

S

T

U

W

Y