Introduction to the Carbon Event Manager

The Carbon Event Manager offers a simple yet flexible approach to event handling that greatly reduces the amount of code needed to write a basic application. Under the Classic event model, every application must include code to handle typical user interface actions such as mouse events and menu tracking. The Carbon Event Manager provides standard handlers for most types of user interaction, so you can concentrate on writing code that's unique to your application. You don't need to write your own event handlers unless you want to override a default behavior.

In addition, the Carbon Event Manager provides a unified event model that replaces Classic events, notifications, and defproc messages with a single method: Carbon events. This simplified system is easier to use, but allows a high degree of control for those who need it. For example, a single callback function, `InstallEventHandler`, is all you need to attach your own event handler to any Toolbox object. And while the Classic event record was limited to 16 event types, the opaque Carbon event object can handle any number of event types.

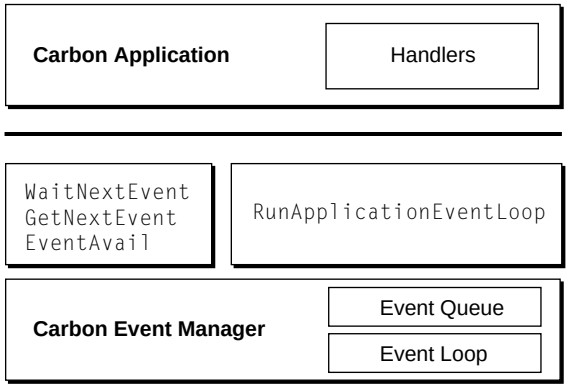
Finally, the Carbon Event Manager's streamlined event handling enhances system performance on Mac OS X through more efficient allocation of processing time. Applications that use the Carbon Event Manager not only run better on Mac OS X, they help improve overall performance and responsiveness, creating a better experience for our customers.

This document introduces the key features of the Carbon Event Manager, and offers some simple steps you can take to improve your application's performance when running on Mac OS X. Although you can port your application to Carbon without adopting the Carbon Event Manager, it will not run as efficiently under Mac OS X, and may actually reduce overall system performance under certain conditions (described later in this document). For this reason, Apple encourages all developers to begin using the Carbon Event Manager.

Overview

The Carbon Event Manager provides the underlying event system for Mac OS X. Classic Event Manager functions such as `WaitNextEvent` are built on top of this foundation and emulated by the Carbon Event Manager. On Mac OS 8 and 9, the Carbon Event Manager is available as an alternative to the Classic Event Manager.

Figure 1-1 Relationship Between Applications and the Carbon Event Manager on Mac OS X



Under the Classic event model, applications call `WaitNextEvent` or `GetNextEvent` to receive events, and then determine what to do with them. A high percentage of time is spent in the event loop simply handling common user interface events such as keystrokes and mouse clicks.

By contrast, the underlying principle of the Carbon Event Manager is that events are dispatched directly to the Toolbox objects in your application, to be handled automatically by standard event handlers unless you choose to override the default behavior. This means your application no longer needs to manage common interface tasks such as handling menu selections, moving and resizing windows, or interacting with control widgets such as buttons and sliders.

Getting Started

The Carbon Event Manager is designed to allow gradual adoption. In fact, you don't need to adopt any of the new Carbon Event Manager functions to port an existing application to Carbon. However, your application will run more efficiently on Mac OS X if you observe the guidelines presented here.

1. Use TrackMouseLocation

If you currently track the mouse using the `WaitMouseUp` or `StillDown` functions, you should use `TrackMouseLocation` instead. The `TrackMouseLocation` function makes more efficient use of processor time because it does not return control to your application until the mouse is moved or a button is released.

Code that uses Classic Event Manager functions to track the mouse location inside a tight loop will needlessly consume processor cycles on Mac OS X. Here is a typical example:

```
GetMouse( &loc );
while ( StillDown() )
{
    ...
    GetMouse( &loc );
}
```

Your application should instead call `TrackMouseLocation` when it receives a mouse down event. Each time the function returns, the `result` parameter indicates what type of mouse activity occurred, and the `outPt` parameter contains the current mouse location.

Here is the equivalent code using `TrackMouseLocation`:

```
TrackMouseLocation( curPort, &loc, &result );
while ( result != kMouseTrackingMouseReleased )
{
    ...
    TrackMouseLocation( curPort, &loc, &result );
}
```

2. Don't Poll the Keyboard

As with mouse tracking, polling the keyboard is a very inefficient use of processor time. The Carbon Event Manager provides a variety of functions that return information about keyboard events. You can obtain high-level information such as the Unicode character represented by a keyboard event, or low-level information such as individual key down and key up events.

A common use of keyboard polling is to determine when a modifier key is pressed. The following code snippet shows how you can install an event handler that will be called whenever the modifier keys change state:

```
// first create an EventTypeSpec to specify the events you want to receive:
const EventTypeSpec kMyKeyModsChanged = { kEventClassKeyboard,
                                          kEventKeyModifiersChanged };
...
InstallEventHandler( GetApplicationEventTarget(),
                     NewEventHandlerUPP( MyKeyModChangedHandler ),
                     1,       // number of event types in your EventTypeSpec
                     kMyKeyModsChanged, // your EventTypeSpec
                     NULL,    // user data (optional)
                     NULL     // event handler reference (optional)
                    );
```

3. Use Timers Instead Of Timeouts

For periodic tasks such as cursor animation or blinking the insertion point, you should use timers instead of calling `WaitNextEvent` with a small `sleep` value. In fact, we recommend that you set the `sleep` value to `MAXINT` and use timers for all periodic tasks. This allows the Carbon Event Manager to allocate processor time more efficiently.

For example, you might use code like this to do some idle processing:

```
gotEvent = WaitNextEvent( everyEvent, theEvent, 60, NULL );
if ( !gotEvent )
    MyDoIdleStuff();
```

Instead, you should install a timer that will be called by the Carbon Event Manager at the specified interval (in this case, every second):

```
InstallEventLoopTimer( GetCurrentEventLoop(), // the target event loop
                       0, // delay before first timer fire
                       1 * kEventDurationSecond, // timer interval
                       NewEventLoopTimerUPP( MyTimerProc ), // timer proc
                       0, // user data (optional)
                       timerRef // returns a reference to the installed timer
                      );
```

Another advantage of this approach is that you can call `InstallEventLoopTimer` from anywhere in your application—you no longer need to be in the main event loop to receive idle time.