17

THE CARBON EVENT MANAGER

Demonstration Programs: CarbonEvents1 and CarbonEvents2

Overview

The Carbon Event Model

The Carbon event model, which was introduced with Carbon as an alternative to what is now termed the Classic event model, reduces the amount of events-related code required by an application and, in addition, facilitates a more efficient allocation of processing time on the preemptive multitasking Mac OS X. Indeed, the Carbon event model is the underlying event model on Mac OS X, the Classic event model being constructed on top of this model and emulated by the Carbon Event Manager.

Event Handling Basics

As will by now be apparent, applications using the Classic event model spend a large amount of time in the `WaitNextEvent` loop handling such user-interface events as mouse-downs and key-downs. In the Carbon event model, this continual and inefficient "polling" for events is avoided, events being dispatched directly to Toolbox objects.

Standard Event Handlers

These dispatched events may be handled automatically by **standard (default) event handlers** provided by the Carbon Event Manager if you so specify. The provision of standard event handlers greatly simplifies the programming task. As an example, and as will be seen in the demonstration program CarbonEvents1, your application requires no code at all to handle basic window dragging, resizing, zooming, activation, and deactivation operations.

Standard event handlers are provided for each type of **event target** (windows, menus, controls, and the application itself).

Overriding and Complementing the Standard Handlers

At the same time, you can override or complement the standard behavior provided by the standard event handlers by writing your own handlers and installing them on the relevant objects. Your application's event handler will override the standard event handler if it returns `noErr`, which defeats the event being passed to the standard handler. Your application's event handler will complement the standard event handler if it returns `eventNotHandledErr`, which causes the event to be further propagated to the standard event handler following handling by your application's event handler. (Event handlers are installed on a stack, the most recently installed on top. The most recently installed handler is called first.)

The Basic Approach

The basic approach to using the Carbon event model API is thus to install the relevant standard event handlers first and then register the types of events your application wishes to receive in order to override or complement the actions of the standard handlers.

Event Propagation Order

Events are propagated in a particular order, that order depending on the type of event. For example, control-related events are sent first to the control, then to the owning window, and then to the application. This means that you can install a handler for the control on either the control, the owning window (as in the demonstration program CarbonEvents1), or the application.

As another example, menu-related events are sent first to the menu, then to the user focus target (that is, the object with current keyboard focus, which can be either a window or a control), then to the application.

RunApplicationEventLoop

At the point where a Classic event model version of your application would call `WaitNextEvent` to enter the main event loop, your Carbon event model application calls the Carbon Event Manager function `RunApplicationEventLoop`. `RunApplicationEventLoop`:

- Moves events from lower-level OS queues into the Carbon queue.
- Dispatches those events from the Carbon queue to the standard event handlers and, for events types that your application has registered, to your application's event handlers.

When an event occurs that requires your program's attention, the Carbon Event Manager calls the handler for that event type. On return from the handler, your program is suspended until the next event requiring its attention is received. Thus your program only uses processor time when processing an event, other programs and processes running in the meantime.

Event Timers

The Carbon Event Manager supports the installation of **timers**, which can be set to fire either once or repeatedly, and which may be used to trigger calls to a specified function at a specified elapsed time or at specified intervals.

Event Reference, Class, and Type

Event Reference

The **event reference** is the core Carbon Event Manager data structure:

```
typedef struct OpaqueEventRef *EventRef;
```

Event Class and Type

As was stated at Chapter 2, the Classic event model is limited to a maximum of 16 event types. By contrast, the Carbon event model can accommodate an unlimited number of **event types**. Event types are grouped by **event class**.

Typical event classes, as represented by constants in CarbonEvents.h, are as follows:

```
kEventClassApplication
kEventClassWindow
kEventClassControl
kEventClassMenu
```

Each event class comprises a number of event types. For example, some of the many event types pertaining to the `kEventClassWindow` event class, as represented by constants in CarbonEvents.h, are as follows:

```
kEventWindowDrawContent
kEventWindowActivated
kEventWindowClickDragRgn
kEventWindowGetIdealSize
```

Given an event reference, your application can ascertain the class and type of a received event by calling `GetEventClass` and `GetEventKind`.

Standard Event Handlers

Standard Application Event Handler

The standard application event handler is installed automatically when your application calls `RunApplicationEventLoop`. Amongst other things, the standard application event handler handles application-activated and application-deactivated events (in Classic event model parlance, resume and suspend events).

Standard Window Event Handler

The standard window event handler handles all of the possible user inter-actions with a window (dragging, resizing, zooming, activation, deactivation, etc.). It must be explicitly installed on the target window by your application. You can cause the standard window event handler to be installed on a window as follows:

- For a window created from a `'WIND'` resource using `GetNewCWindow`, either set the standard handler attribute in a call to the function `ChangeWindowAttributes`, for example:

  ```
  ChangeWindowAttributes(windowRef,kWindowStandardHandlerAttribute,0);
  ```

  or call `InstallStandardEventHandler`, passing in an event target reference (type `EventTargetRef`) obtained using `GetWindowEventTarget`, for example:

  ```
  InstallStandardEventHandler(GetWindowEventTarget(windowRef));
  ```

- For a window created using `CreateNewWindow`, pass `kWindowStandardHandlerAttribute` in the `attributes` parameter.

The Application's Event Handlers

Handlers are Callback Functions

The handlers provided by your application are callback functions. When called, they are passed:

- A reference to the event handler call (type `EventHandlerCallRef`).
- The event reference, from which you can extract the event class and type.
- A pointer to user data (assuming that you passed a pointer to that data when you installed the handler).

Installing the Application's Event Handlers

You can install handlers provided by your application using `InstallEventHandler`:

```
OSStatus  InstallEventHandler(EventTargetRef inTarget,EventHandlerUPP inHandler,
                              UInt32 inNumTypes,const EventTypeSpec *inList,
                              void *inUserData,EventHandlerRef *outRef);
```

`inTarget` An event target reference to the event target the handler is to be registered with. Use one of the following functions to obtain this reference:

```
GetApplicationEventTarget
GetWindowEventTarget
GetControlEventTarget
GetMenuEventTarget
GetUserFocusEventTarget
```

`inHandler`	A universal procedure pointer to the handler function provided by your application.
`inNumTypes`	The number of event types to be registered by this call to `InstallEventHandler`.
`inList`	A pointer to an array of type `EventTypeSpec` structures specifying the event types being registered. The `EventTypeSpec` structure is as follows:

```
struct EventTypeSpec
{
  UInt32 eventClass;  // Event class
  UInt32 eventKind;   // Event type
};
typedef struct EventTypeSpec EventTypeSpec;
```

For example, if you wished to register the `kEventWindowDrawContent` and `kEventWindowActivated` event types, you would define the array as follows:

```
EventTypeSpec myTypes[] = { { kEventClassWindow, kEventWindowDrawContent },
                            { kEventClassWindow, kEventWindowActivated   } };
```

`inUserData`	Optionally, a pointer to data to be passed to your event handler when it is called.
`outRef`	If you will later need to remove the handler, pass a pointer to a variable of type `EventHandlerRef` in this parameter. On return, this variable will receive the event handler reference that will be required by your call to `RemoveEventHandler`.

You can also use the following macros, which are derived from the function `InstallEventHandler`, to install your application's handlers:

```
InstallApplicationEventHandler
InstallWindowEventHandler
InstallControlEventHandler
InstallMenuEventHandler
```

Different object of the same type do not have to have the same handler. For example, you can install separate handlers on each of two windows.

Inside The Application's Event Handlers

Getting Event Parameters

In some circumstances, in order to correctly handle a particular event type, you may need to extract specific data from the event using the function `GetEventParameter`. For example, on receipt of an event in the `kEventClassWindow` class, you will almost invariably need to call `GetEventParameter` to get the window reference required to facilitate the handling of certain event types in that class. Similarly, on receipt of an event of type `kEventMouseDown`, you might need to call `GetEventParameter` to obtain the mouse location.

The `GetEventParameter` prototype is as follows:

```
OSStatus  GetEventParameter(EventRef inEvent,EventParamName inName,
                            EventParamType inDesiredType,EventParamType *outActualType,
                            UInt32 inBufferSize,UInt32 *outActualSize,void *outData);
```

`inEvent`	A reference to the event.
`inName`	The parameter's symbolic name. Symbolic names pertaining to the various event types are listed in CarbonEvents.h immediately after the enumerations for those types. For example, the symbolic name for the mouse location is `kEventParamMouseLocation`.

`inDesiredType`	The type of data. This is listed against the parameter's symbolic name in CarbonEvents.h. For example, the type of data pertaining to the symbolic name `kEventParamMouseLocation` is `typeQDPoint`.
`outActualType`	Actual type of value returned. Specify `NULL` if this information is not needed.
`inBufferSize`	The size of the output buffer.
`outActualSize`	Actual size of value returned. Specify `NULL` if this information is not needed.
`outData`	A pointer to the buffer which will receive the parameter data.

The types of data that can be extracted from an event depends on the type of event. The parameter symbolic names and data types listed in CarbonEvents.h together indicate the type, or types, of data obtainable from an event of a particular type.

Event Parameters and Command Events

The Carbon Event Manager can associate special **command events** with menu items with command IDs. You can, of course, assign your own command IDs to menu items using `SetMenuItemCommandID`; however, note that CarbonEvents.h defines command IDs for many common menu items, for example:

```
kHICommandQuit  = FOUR_CHAR_CODE('quit')
kHICommandUndo  = FOUR_CHAR_CODE('undo')
kHICommandCut   = FOUR_CHAR_CODE('cut ')
kHICommandPaste = FOUR_CHAR_CODE('past')
```

When a menu item with a command ID is chosen by the user, either with the mouse or using a Command-key equivalent, the Carbon Event Manager dispatches the relevant command event (class `kEventClassCommand`, type `kEventProcessCommand`).

When your application's handler receives a `kEventProcessCommand` event type, you pass `kEventParamDirectObject` in the `inName` parameter of your `GetEventParameter` call, `typeHICommand` in the `inDesiredType` parameter, and the address of a structure of type `HICommand` in the `outData` parameter. The `HICommand` structure is as follows:

```
struct HICommand
{
  UInt32 attributes;
  UInt32 commandID;
  struct
  {
    MenuRef menuRef;
    UInt16  menuItemIndex;
  } menu;
};
typedef struct HICommand HICommand;
```

Thus you will be able to extract the menu reference and menu item number, as well as the command ID of the chosen menu item (if any), from the data returned by the call to `GetEventParameter`.

Quit Command Handling

The Quit command event is a special case. When the Quit item is chosen, the standard application event handler calls either the default Quit Application Apple event handler or your application's Quit Application Apple event handler if it has installed its own. (When your application calls `RunApplicationEventLoop`, the default Quit Application Apple event handler is automatically installed if the application has not installed its own.)

Thus the only action required by your application's handler is to ensure that it returns `eventNotHandledErr` when it determines that the `commandID` field of the `HICommand` structure contains `kHICommandQuit`, thereby causing the event to be propagated to the standard application event handler and thence to the relevant Quit Application Apple event handler.

For this to work on Mac OS 8/9, your application must assign the command ID `kHICommandQuit` to the Quit item at program start when the application determines that it is running on Mac OS 8/9.

Setting Event Parameters

In certain circumstances, your handler will need to call SetEventParameter to set a piece of data for a given event. For example, suppose you wish to constrain window resizing to a specified minimum size and, accordingly, register for the kEventWindowGetMinimumSize event type. When this event type is received by your handler (it will be dispatched when a mouse-down occurs in the size box/resize control of a window on which your handler is installed), your handler should call SetEventParameter with kEventParamDimensions passed in the inName parameter and a pointer to a variable of type Point passed in the inDataPtr parameter. (The Point variable should contain the desired minimum window height and width.)

Converting an Event Reference to an Event Record

In certain circumstances, your handler may need to convert the event reference to a Classic event model event structure (type EventRecord) in order to be able to handle the event. You can use the function ConvertEventRefToEventRecord for that purpose.

Menu Adjustment

You can ensure that your application's menu adjustment function is called when appropriate by registering the kEventMenuEnableItems event type (kEventClassMenu event class) and calling your menu adjustment function when that event type is received. The kEventMenuEnableItems event type will be dispatched when a mouse-down occurs in the menu bar and when a menu-related Command-key equivalent is pressed.

Cursor Shape Changing

In Classic event model applications, the application's cursor shape-changing function is typically called when mouse-moved Operating System events are received. An alternative "trigger" is required when using the Carbon event model.

One approach is to install a Carbon events timer set to fire at an appropriate interval and call the cursor shape-changing function when the timer fires. However, this method is not recommended for Mac OS X because it is somewhat like polling for an event, which is more processor-intensive.

The recommended approach is to register for the kEventMouseMoved event type (kEventClassMouse event class) and call the cursor shape-changing function on receipt of that event type.

Window Updating

To accommodate window content region updating (re-drawing) requirements, your application should register for the kEventWindowDrawContent event type (kEventClassWindow event class) and call its update function when that event type is received.

Note that the Window Manager sends an event of type kEventWindowUpdate to all windows that need updating, regardless of whether those windows have the standard window event handler installed or not. If the standard window event handler is installed, then when the standard handler receives the kEventWindowUpdate event, it calls BeginUpdate, sends a kEventWindowDrawContent event, and calls EndUpdate. There is thus no need for your update function to call BeginUpdate and EndUpdate when responding to kEventWindowDrawContent events.

Handler Disposal

All handlers on a target are automatically disposed of when the target is disposed of.

Sending and Explicitly Propagating Events

Sending Events

You can send an event to a specified event target using either the function SendEventToEventTarget or the following macros derived from that function:

```
SendEventToApplication
SendEventToWindow
SendEventToControl
SendEventToMenu
SendEventToUserFocus
```

Explicitly Propagating Events

You can explicitly propagate an event up the propagation chain by calling `CallNextEventHandler` within your event handler. This is useful in circumstances where, for example, you wish to incorporate the standard handler's response into your own pre- or post-processing.

Event Timers

Event timers may be used for many purposes, the most common one being to trigger a call to your application's idle function, perhaps for the purpose of blinking the insertion point caret. You can use `InstallEventLoopTimer` to install an event timer:

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

`inEventLoop`	The event loop on which the timer is to be installed. Usually, this will be the event loop reference returned by a call to `GetCurrentEventLoop`.
`inFireDelay`	The required delay before the timer first fires. This can be `0`.
`inInterval`	A value of type `double` specifying the interval at which the timer is required to fire. For one-shot timers, `0` should be passed in this parameter. For a timer whose purpose is to trigger calls to an idle function which blinks the insertion point caret, pass the value returned by a call to `GetCaretTime` converted to **event time** by the macro `TicksToEventTime`. Note that event time is in seconds since system startup. You can use the macros `TicksToEventTime` and `EventTimeToTicks` to convert between ticks and event time.
`inTimerProc`	A universal procedure pointer to the function to be called when the timer fires.
`inTimerData`	Optionally, a pointer to data to be passed to the function called when the timer fires.
`OutTimer`	A reference to the newly-installed timer. Usually, this will be required only if you intend to remove the timer at some point.

Note that, depending on the parameters passed to this function, the timer can be set to fire either once or repeatedly at a specified interval.

You can remove an installed timer by calling `RemoveEventLoopTimer`.

Getting Event Time

Your application can determine the time an event occurred using the function `GetEventTime`. It can also determine the time from system startup using the function `GetCurrentEventTime`.

Other Aspects of the Carbon Event Model

The Carbon Event Model and Apple Events

Your application requires no code at all to ensure that, when Apple events are dispatched to it, its Apple event handlers are called.

Carbon Event Model and Control Hierarchies

When you establish an embedding hierarchy for controls, you are also establishing an event handling chain. When you click in a given control, the event is sent first to that control. If that control does not handle the event (that is, its handler returns `eventNotHandledErr`), the event is passed up the chain to the control that contains the first control, and so on up the chain.

Carbon Event Model and Event Filter Functions

In Classic event model applications, you must pass a universal procedure pointer to an application-defined event filter(callback) function in the `modalFilter` parameter of `ModalDialog`, and call your application's window updating function within the filter function.

Calling your window updating function from within your event filter function is not necessary in Carbon event model applications provided the application installs the standard window event handler on the relevant windows, registers for the `kEventWindowDrawContent` event type, and calls its window updating function when that event type is received.

Mouse Tracking

The demonstration program QuickDraw (Chapter 12) uses the Event Manager function `StillDown` in the `doDrawWithMouse` function to determine whether the mouse button has been continuously pressed since the most recent `mouseDown` event. The Event Manager function `WaitMouseUp` is often used for similar purposes.

For reasons of efficient use of processor cycles, `TrackMouseLocation` should be used in lieu of `StillDown` and `WaitMouseUp` in applications intended to run on Mac OS X. (`TrackMouseLocation` does not return control to your application until the mouse is moved or the mouse button is released.) When `TrackMouseLocation` returns, the `outResult` parameter contains a value representing the type of mouse activity that occurred (press, release, etc.) and the `outPt` parameter contains the mouse location.

The function `TrackMouseRegion` is similar to `TrackMouseLocation` except that `TrackMouseRegion` only returns when the mouse enters or exits a specified region.

Alternative for Delay Function on Mac OS X

Programs sometimes call the function `Delay` to pause program execution for the number of ticks passed in the `duration` parameter. On Mac OS X, if the delay is more than about two seconds, the cursor will automatically be set to the wait cursor. To avoid this, you can instead call the function `RunCurrentEventLoop` with the required delay in seconds (perhaps converted from ticks using the macro `TicksToEventTime`) passed in the `inTimeout` parameter.

Main Constants, Data Types, and Functions

Constants

Error Codes

```
eventAlreadyPostedErr           = -9860
eventClassInvalidErr            = -9862
eventClassIncorrectErr          = -9864
eventHandlerAlreadyInstalledErr = -9866
eventInternalErr                = -9868
eventKindIncorrectErr           = -9869
eventParameterNotFoundErr       = -9870
eventNotHandledErr              = -9874
eventLoopTimedOutErr            = -9875
eventLoopQuitErr                = -9876
eventNotInQueueErr              = -9877
```

Event Classes

```
kEventClassMouse                = FOUR_CHAR_CODE('mous')
kEventClassKeyboard             = FOUR_CHAR_CODE('keyb')
kEventClassTextInput            = FOUR_CHAR_CODE('text')
kEventClassApplication          = FOUR_CHAR_CODE('appl')
kEventClassMenu                 = FOUR_CHAR_CODE('menu')
kEventClassWindow               = FOUR_CHAR_CODE('wind')
kEventClassControl              = FOUR_CHAR_CODE('cntl')
kEventClassCommand              = FOUR_CHAR_CODE('cmds')
```

Event Types

```
kEventMouseDown                 = 1
kEventMouseUp                   = 2
kEventMouseMoved                = 5
kEventMouseDragged              = 6

kEventRawKeyDown                = 1
kEventRawKeyRepeat              = 2
kEventRawKeyUp                  = 3
kEventRawKeyModifiersChanged    = 4

kEventAppActivated              = 1
kEventAppDeactivated            = 2
kEventAppQuit                   = 3
kEventAppLaunchNotification     = 4

kEventMenuEnableItems           = 8

kEventWindowUpdate              = 1
kEventWindowDrawContent         = 2
kEventWindowActivated           = 5
kEventWindowDeactivated         = 6
kEventWindowGetClickActivation  = 7
kEventWindowShown               = 24
kEventWindowHidden              = 25
kEventWindowBoundsChanging      = 26
kEventWindowBoundsChanged       = 27
kEventWindowClickDragRgn        = 32
kEventWindowClickResizeRgn      = 33
kEventWindowClickCollapseRgn    = 34
kEventWindowClickCloseRgn       = 35
kEventWindowClickZoomRgn        = 36
kEventWindowClickContentRgn     = 37
kEventWindowClickProxyIconRgn   = 38

kEventControlHit

kEventProcessCommand            = 1
```

HI Commands

```
kHICommandOK                = FOUR_CHAR_CODE('ok  ')
kHICommandQuit              = FOUR_CHAR_CODE('quit')
kHICommandCancel            = FOUR_CHAR_CODE('not!')
kHICommandUndo              = FOUR_CHAR_CODE('undo')
kHICommandRedo              = FOUR_CHAR_CODE('redo')
kHICommandCut               = FOUR_CHAR_CODE('cut ')
kHICommandCopy              = FOUR_CHAR_CODE('copy')
kHICommandPaste             = FOUR_CHAR_CODE('past')
kHICommandClear             = FOUR_CHAR_CODE('clea')
kHICommandSelectAll         = FOUR_CHAR_CODE('sall')
kHICommandHide              = FOUR_CHAR_CODE('hide')
kHICommandPreferences       = FOUR_CHAR_CODE('pref')
kHICommandZoomWindow        = FOUR_CHAR_CODE('zoom')
kHICommandMinimizeWindow    = FOUR_CHAR_CODE('mini')
kHICommandArrangeInFront    = FOUR_CHAR_CODE('frnt')
```

Mouse Tracking Result

```
kMouseTrackingMousePressed      = 1
kMouseTrackingMouseReleased     = 2
kMouseTrackingMouseExited       = 3
kMouseTrackingMouseEntered      = 4
kMouseTrackingMouseMoved        = 5
```

Data Types

```
typedef struct OpaqueEventRef *EventRef;
typedef struct OpaqueEventHandlerRef *EventHandlerRef;
typedef struct OpaqueEventHandlerCallRef *EventHandlerCallRef;
typedef struct OpaqueEventLoopRef *EventLoopRef;
typedef double EventTime;
typedef UInt16 MouseTrackingResult;
```

EventTypeSpec

```
struct EventTypeSpec
{
  UInt32 eventClass;
  UInt32 eventKind;
};
typedef struct EventTypeSpec EventTypeSpec;
```

HICommand

```
struct HICommand
{
  UInt32 attributes;
  UInt32 commandID;
  struct
  {
    MenuRef menuRef;
    UInt16  menuItemIndex;
  } menu;
};
typedef struct HICommand HICommand;
```

Functions and Macros

Installing and Removing Event Handlers

```
OSStatus   InstallStandardEventHandler(EventTargetRef inTarget);
OSStatus   InstallEventHandler(EventTargetRef inTarget,EventHandlerUPP inHandler,
                               UInt32 inNumTypes,const EventTypeSpec *inList,
                               void *inUserData,EventHandlerRef *outRef);
#define    InstallApplicationEventHandler(h,n,l,u,r) \
               InstallEventHandler(GetApplicationEventTarget(),(h),(n),(l),(u),(r))
#define    InstallWindowEventHandler(t,h,n,l,u,r) \
               InstallEventHandler(GetWindowEventTarget(t),(h),(n),(l),(u),(r))
#define    InstallControlEventHandler(t,h,n,l,u,r) \
               InstallEventHandler(GetControlEventTarget(t),(h),(n),(l),(u),(r))
```

```
#define    InstallMenuEventHandler(t,h,n,l,u,r) \
               InstallEventHandler(GetMenuEventTarget(t),(h),(n),(l),(u),(r))
OSStatus   RemoveEventHandler(EventHandlerRef inHandlerRef);
OSStatus   AddEventTypesToHandler(EventHandlerRef inHandlerRef,UInt32 inNumTypes,
                                  const EventTypeSpec *inList);
OSStatus   RemoveEventTypesFromHandler(EventHandlerRef inHandlerRef, inNumTypes,
                                       const EventTypeSpec *inList);
```

Creating and Disposing of Event Handler UPPs

```
EventHandlerUPP NewEventHandlerUPP(EventHandlerProcPtr userRoutine);
void            DisposeEventHandlerUPP(EventHandlerUPP userUPP);
```

Running and Quitting Application Event Loop

```
void       RunApplicationEventLoop(void);
void       QuitApplicationEventLoop(void);
```

Getting Event Class and Kind

```
UInt32     GetEventClass(EventRef inEvent);
UInt32     GetEventKind(EventRef inEvent);
```

Testing for User Cancelled

```
Boolean    IsUserCancelEventRef(EventRef event);
```

Getting Data From Events

```
OSStatus   GetEventParameter(EventRef inEvent,EventParamName inName,
                             EventParamType inDesiredType,EventParamType *outActualType,
                             UInt32 inBufferSize,UInt32 *outActualSize,void *ioBuffer);
```

Converting an Event Reference to an EventRecord

```
Boolean    ConvertEventRefToEventRecord(EventRef inEvent,EventRecord *outEvent);
```

Sending Events

```
OSStatus   SendEventToEventTarget(EventRef inEvent,EventTargetRef inTarget);
#define    SendEventToApplication(e) \
               SendEventToEventTarget((e),GetApplicationEventTarget())
#define    SendEventToWindow(e,t) \
               SendEventToEventTarget((e),GetWindowEventTarget(t))
#define    SendEventToControl(e,t) \
               SendEventToEventTarget((e),GetControlEventTarget(t))
#define    SendEventToMenu(e,t) \
               SendEventToEventTarget((e),GetMenuEventTarget(t))
#define    SendEventToUserFocus(e) \
               SendEventToEventTarget((e),GetUserFocusEventTarget())
```

Installing, Resetting, and Removing Timers

```
OSStatus   InstallEventLoopTimer(EventLoopRef inEventLoop, EventTimerInterval inFireDelay,
                                 EventTimerInterval inInterval,
                                 EventLoopTimerUPP inTimerProc,void *inTimerData,
                                 EventLoopTimerRef *outTimer);
OSStatus   SetEventLoopTimerNextFireTime(EventLoopTimerRef inTimer,
                                         EventTimerInterval inNextFire);
OSStatus   RemoveEventLoopTimer(EventLoopTimerRef inTimer);
```

Calling Through to Handlers Below Current Handler

```
OSStatus   CallNextEventHandler(EventHandlerCallRef inCallRef,EventRef inEvent);
```

Getting Event and System Time

```
EventTime  GetEventTime(EventRef inEvent);
EventTime  GetCurrentEventTime(void);
```

Converting Between Ticks and EventTime

```
#define TicksToEventTime(t)  (EventTime) ((t) / 60.0)
#define EventTimeToTicks(t)  (UInt32)    ((t) * 60)
```

Mouse Tracking

```
OSStatus   TrackMouseLocation(GrafPtr inPort,Point *outPt,MouseTrackingResult *outResult);
OSStatus   TrackMouseRegion(GrafPtr inPort,RgnHandle inRegion,Boolean *ioWasInRgn,
          MouseTrackingResult *outResult);
```

Relevant Window Manager Constants and Functions

Constants

```
kWindowStandardHandlerAttribute = (1L << 25)
```

Functions

```
OSStatus   ChangeWindowAttributes(WindowRef window,WindowAttributes setTheseAttributes,
                                  WindowAttributes clearTheseAttributes);
```

Demonstration Program CarbonEvents1 Listing

```
// *******************************************************************************************
// CarbonEvents1.c                                                          CARBON EVENT MODEL
// *******************************************************************************************
//
// This program opens a kWindowFullZoomGrowDocumentProc window, creates a root control for
// the window (on Mac OS 8/9), and adds a pop-up menu button control to the window.
//
// The standard application event handler handles all application events.  The standard
// window event handler is installed on the window.  In addition, the program installs its own
// handler on the window for the purpose of determining which item the user chooses in the
// pop-up menu button's menu.  (Although installed on the window, this handler could just as
// easily be installed on the control.)
//
// The program utilises the following resources:
//
// •  A 'plst' resource.
//
// •  An 'MBAR' resource, and 'MENU' resources for OS9Apple/Application, File, and Edit menus
//    and the pop-up menu (preload, non-purgeable).
//
// •  A 'WIND' resource (purgeable) (initially not visible).
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, canBackground,
//    doesActivateOnFGSwitch, and isHighLevelEventAware flags set.
//
// *******************************************************************************************

// .................................................................................. includes

#include <Carbon.h>

// ................................................................................... defines

#define rMenubar   128
#define rWindow    128
#define mFile      129
#define  iQuit     12
#define mPopupMenu 131
// .......................................................................... global variables

Boolean gRunningOnX = false;

// ....................................................................... function prototypes

void      main               (void);
void      doPreliminaries    (void);
OSStatus  windowEventHandler (EventHandlerCallRef,EventRef,void *);
void      doNewWindow        (void);
void      doGetControls      (WindowRef);

// ************************************************************************************** main

void  main(void)
{
  MenuBarHandle menubarHdl;
  SInt32        response;
  MenuRef       menuRef;

  // ..................................................................... do preliminaries

  doPreliminaries();

  // ............................................................... set up menu bar and menus

  menubarHdl = GetNewMBar(rMenubar);
  if(menubarHdl == NULL)
    ExitToShell();
  SetMenuBar(menubarHdl);
  DrawMenuBar();

  Gestalt(gestaltMenuMgrAttr,&response);
  if(response & gestaltMenuMgrAquaLayoutMask)
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
    {
      DeleteMenuItem(menuRef,iQuit);
      DeleteMenuItem(menuRef,iQuit - 1);
    }

    gRunningOnX = true;
  }
  else
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
      SetMenuItemCommandID(menuRef,iQuit,kHICommandQuit);
  }

  // .............................................................................. open window

  doNewWindow();

  // ................................................................. run application event loop

  RunApplicationEventLoop();
}

// *************************************************************************** doPreliminaries

void  doPreliminaries(void)
{
  MoreMasterPointers(32);
  InitCursor();
}

// ******************************************************************************* doNewWindow

void  doNewWindow(void)
{
  WindowRef     windowRef;
  OSStatus      osError;
  Rect          controlRect = { 42,39,62,235 };
  ControlRef    controlRef;
  EventTypeSpec windowEvents[] = { { kEventClassControl, kEventControlHit } };

  // ............................................................ open window and set attributes

  if(!(windowRef = GetNewCWindow(rWindow,NULL,(WindowRef) -1)))
    ExitToShell();
  SetPortWindowPort(windowRef);

  ChangeWindowAttributes(windowRef,kWindowStandardHandlerAttribute ,0);

  // .............................................................. install window event handler

  InstallWindowEventHandler(windowRef,
                            NewEventHandlerUPP((EventHandlerProcPtr) windowEventHandler),
                            GetEventTypeCount(windowEvents),windowEvents,0,NULL);

  // ............................... create root control for window and get popup button control

  if(!gRunningOnX)
    CreateRootControl(windowRef,&controlRef);
  if((osError = CreatePopupButtonControl(windowRef,&controlRect,CFSTR("Time Zone:"),
                                       mPopupMenu,false,-1,0,0,&controlRef)) != noErr)
    ExitToShell();

  // ........................................................................... show window

  ShowWindow(windowRef);
}

// *************************************************************************** windowEventHandler

OSStatus  windowEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                             void* userData)
{
  OSStatus    result = eventNotHandledErr;
  UInt32      eventKind;
  ControlRef  controlRef;
  MenuRef     menuRef;
  Size        actualSize;
  SInt16      controlValue;
  Str255      menuItemString;
  Rect        theRect = { 0,0,40,293 };
  CFStringRef stringRef;
  Rect        textBoxRect;

  eventKind  = GetEventKind(eventRef);

  if(eventKind == kEventControlHit)
  {
    GetEventParameter(eventRef,kEventParamDirectObject,typeControlRef,NULL,
                      sizeof(controlRef),NULL,&controlRef);

    GetControlData(controlRef,kControlEntireControl,kControlPopupButtonMenuHandleTag,
                   sizeof(menuRef),&menuRef,&actualSize);
    controlValue = GetControlValue(controlRef);
    GetMenuItemText(menuRef,controlValue,menuItemString);

    EraseRect(&theRect);
    stringRef = CFStringCreateWithPascalString(NULL,menuItemString,
                                          kCFStringEncodingMacRoman);
    SetRect(&textBoxRect,theRect.left,7,theRect.right,22);
    DrawThemeTextBox(stringRef,kThemeSmallSystemFont,true,true,&textBoxRect,teJustCenter,
                     NULL);
    if(stringRef != NULL)
      CFRelease(stringRef);

    result = noErr;
  }

  return result;
}

// ********************************************************************************************
```

Demonstration Program CarbonEvents1 Comments

When this program is run, the user should:

- Drag, resize, zoom and, when done, close the window.
- Send the application to the background and bring it to the foreground, noting the activation and deactivation of the pop-up menu button control.
- Choose items in the pop-up menu button's menu.
- Quit the application by choosing the Quit item in the Mac OS 9 File/Mac OS X Application menu and using its Command-key equivalent.

main

If the program is running on OS 8/9, SetMenuItemCommandID is called to assign the command ID 'quit' to the Quit item in the File menu. (This command is assigned to the Mac OS X Quit item by default.) Thus, when the Quit item is chosen on Mac OS 8/9 and Mac OS X, the standard application event handler will call the default Quit Application Apple event handler (automatically installed when RunApplicationEventLoop is called) to close down the program.

The standard application event handler is installed when RunApplicationEventLoop is called. The standard application event handler handles all application events, including, in Classic event model parlance, suspend and resume events (that is, application-deactivated and application-activated events.

doNewWindow

After the window is created, ChangeWindowAttributes is called to set the standard handler attribute, causing the standard window event handler to be installed on the window. The standard window event handles all window dragging, sizing, zooming, collapsing/minimising, and closing operations, attends to control updating, and (provided a root control is created for the window), control deactivation and activation when the program is sent to the back and brought to the. It also calls TrackControl when a mouse-down occurs in the pop-up menu button, thus handling all user interaction with the control.

The call to InstallWindowEventHandler installs the application's window event handler on the window. A single event type (kEventControlHit) is registered. Note that this handler could have been installed on the control itself, but is installed on the window in this program for the purpose of emphasizing the propagation order of events. (No handler is installed on the control, so the event will "fall through" to the window event handler.)

If the program is running on Mac OS 8/9, CreateRootControl is called to create a root control for the window. (This call is not necessary on Mac OS X because, on Mac OS X, a root control is automatically created on windows which have at least one control.)

windowEventHandler

windowEventHandler is a callback function. It is the window event handler installed on the window by the call to InstallWindowEventHandler in main. Its purpose is to determine the control value of the pop-up menu button control, and thus the menu item the user chose.

As previously stated, the standard window event handler calls TrackControl when a mouse-down occurs in the pop-up menu button. The Carbon Event Manager sends the kEventControlHit event type when TrackControl returns (regardless, incidentally, of whether the cursor is still within the control when the mouse button is released).

GetEventType is called to get the event type. If the event type is kEventControlHit, the if block executes and the handler returns noErr, indicating to the Carbon Event Manager that the event has been fully handled and that it should not be propagated further. If the event type is not kEventControlHit, the handler returns eventNotHandledErr, indicating that the event should be propagated further.

Within the if block, GetEventParameter is called to extract certain data from the event, specifically, a reference to the control. This reference is passed in the call to GetControlData, which gets a reference to the control's menu. The call to GetControlValue then gets the control's value, and the call to GetMenuItemText gets the text of the menu item. This text is then drawn at the top of the window to prove that the handler has done its job.

Demonstration Program CarbonEvents2 Listing

```
// *******************************************************************************************
// CarbonEvents2.h                                                        CARBON EVENT MODEL
// *******************************************************************************************
//
// This program allows the user to:
//
// •  Open, close, and interact with kWindowFullZoomGrowDocumentProc windows containing
//    various controls.
//
// •  Open, close and interact with a movable modal dialog and, on Mac OS X only,
//    window-modal (sheet) alerts and window-modal (sheet) dialogs.
//
// The program demonstrates the main aspects of the Carbon Event model, specifically:
//
// •  Events relating to menus, windows and controls, including the detection of mouse-downs
//    in controls in document windows and movable modal dialogs.
//
// •  Keyboard events.
//
// •  Events relating to application activation and deactivation (resume and suspend in
//    Classic event model parlance).
//
// •  The use of mouse-moved events in support of cursor adjustment functions.
//
// •  The installation of event loop timers (used, in this program, to trigger an "idle"
//    function.
//
// The program also demonstrates the implementation of live window resizing.
//
// The window contains a window header frame in which is displayed the menu items chosen from
// pop-up menu buttons, the identity of a push button when that push button is clicked, and
// scroll bar control values when the scroll arrows or gray areas/track of the scroll bars are
// clicked and when the scroll box/scroller is dragged.  (The vertical scroll bar is the
// non-live feedback variant.  The horizontal scroll bar is the live-feedback variant.)  Text
// extracted from the edit text item in the window-modal (sheet) dialog and the identity of
// the button clicked in the window-modal (sheet) alert are also displayed in the window
// header frame.
//
// The movable modal dialog serves the secondary purpose of proving window correct window
// updating even though an event filter function is not used by the dialog.
//
// The program utilises the following resources:
//
// •  A 'plst' resource.
//
// •  An 'MBAR' resource, and 'MENU' resources for OS9Apple/Application, File, Edit, and
//    Typing Target, and Dialogs menus, and the pop-up menus (preload, non-purgeable).
//
// •  A 'WIND' resource (purgeable) (initially not visible).
//
// •  A 'DLOG' resource ((purgeable) (initially not visible), with associated 'DITL', 'dlgx'
//    and 'dfnt' resources, for the window-modal (sheet) dialog.
//
// •  A 'CNTL' resource (purgeable) for an image well control in the window-modal (sheet)
//    dialog.
//
// •  A 'STR#' resource (purgeable) containing text for the window-modal (sheet) alert.
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, canBackground,
//    doesActivateOnFGSwitch, and isHighLevelEventAware flags set.
//
// *******************************************************************************************

// ............................................................................... includes
```

```
#include <Carbon.h>

// ............................................................................................ defines

#define rMenubar                128
#define rWindow                 128
#define rAboutDialog            128
#define mAppleApplication       128
#define  iAbout                 1
#define mFile                   129
#define  iQuit                  12
#define  iNew                   1
#define  iClose                 4
#define mTyping                 131
#define  iDocument              1
#define  iEditTextControl       2
#define  iAllOfTheAbove         3
#define mDialogs                132
#define  iMovableModal          1
#define  iSheetAlert            2
#define  iSheetDialog           3
#define mWindow                 135
#define rSheetDialog            128
#define rSheetStrings           128
#define  sAlertSheetMessage     1
#define  sAlertSheetInformative 2
#define kPopupCountryID         'ctry'
#define kScrollBarWidth         15
#define MIN(a,b)                ((a) < (b) ? (a) : (b))
#define topLeft(r)              (((Point *) &(r))[0])
#define botRight(r)             (((Point *) &(r))[1])

// ........................................................................................... typedefs

typedef struct
{
  ControlRef popupTimeZoneRef;
  ControlRef popupCountryRef;
  ControlRef radiobuttonRedRef;
  ControlRef radiobuttonWhiteRef;
  ControlRef radiobuttonBlueRef;
  ControlRef groupboxColourRef;
  ControlRef groupboxTypingRef;
  ControlRef buttonRef;
  ControlRef buttonDefaultRef;
  ControlRef editTextRef;
  ControlRef scrollbarVertRef;
  ControlRef scrollbarHorizRef;
} docStruc, **docStrucHandle;

// ................................................................................ function prototypes

void            main                  (void);
void            doPreliminaries       (void);
OSStatus        appEventHandler       (EventHandlerCallRef,EventRef,void *);
OSStatus        windowEventHandler    (EventHandlerCallRef,EventRef,void *);
void            doNewWindow           (void);
EventHandlerUPP doGetHandlerUPP       (void);
void            doCloseWindow         (WindowRef);
void            doGetControls         (WindowRef);
void            doIdle                (void);
void            doAdjustMenus         (void);
void            doMenuChoice          (MenuID,MenuItemIndex);
void            doDrawContent         (WindowRef);
void            doActivateDeactivate  (WindowRef,Boolean);
void            doControlHit1         (WindowRef,ControlRef,Point,ControlPartCode);
void            doControlHit2         (void);
void            doPopupMenuChoice     (WindowRef,ControlRef,SInt16);
void            doVertScrollbar       (ControlPartCode,WindowRef,ControlRef,Point);
void            actionFunctionVert    (ControlRef,ControlPartCode);
void            actionFunctionHoriz   (ControlRef,ControlPartCode);
void            doMoveScrollBox       (ControlRef,SInt16);
void            doRadioButtons        (ControlRef,WindowRef);
void            doCheckboxes          (ControlRef);
void            doPushButtons         (ControlRef,WindowRef);
void            doAdjustScrollBars    (WindowRef);
void            doAdjustCursor        (WindowRef);
void            doDrawDocumentTyping  (SInt8,UInt32);
void            doDrawMessage         (WindowRef,Boolean);
void            doConcatPStrings      (Str255,Str255);
void            doCopyPString         (Str255,Str255);

OSStatus        doSheetAlert          (void);
OSStatus        doSheetDialog         (void);
EventHandlerUPP doGetSheetEventHandlerUPP  (void);
OSStatus        sheetEventHandler     (EventHandlerCallRef,EventRef,void *);
OSStatus        doMovableModalDialog  (void);
EventHandlerUPP doGetDialogHandlerUPP (void);
OSStatus        dialogEventHandler    (EventHandlerCallRef,EventRef,void *);

// *******************************************************************************************
// CarbonEvents2.c
// *******************************************************************************************

// ......................................................................................... includes

#include "CarbonEvents2.h"

// ................................................................................. global variables

ControlActionUPP gActionFunctionVertUPP;
ControlActionUPP gActionFunctionHorizUPP;
Boolean          gRunningOnX = false;
SInt16           gNumberOfWindows = 0;
Str255           gCurrentString;
SInt16           gTypingTarget = 3;

// ************************************************************************************** main

void  main(void)
{
  MenuBarHandle     menubarHdl;
  SInt32            response;
  MenuRef           menuRef;
  EventLoopTimerUPP eventLoopTimerUPP;
  EventTypeSpec   applicationEvents[] = { { kEventClassApplication, kEventAppActivated    },
                                          { kEventClassCommand,     kEventProcessCommand  },
                                          { kEventClassMenu,        kEventMenuEnableItems },
                                          { kEventClassMouse,       kEventMouseMoved      } };

  // ............................................................................ do preliminaries

  doPreliminaries();

  // .......................................................... create universal procedure pointers

  gActionFunctionVertUPP  = NewControlActionUPP((ControlActionProcPtr) actionFunctionVert);
  gActionFunctionHorizUPP = NewControlActionUPP((ControlActionProcPtr) actionFunctionHoriz);

  // ..................................................................... set up menu bar and menus

  menubarHdl = GetNewMBar(rMenubar);
  if(menubarHdl == NULL)
    ExitToShell();
  SetMenuBar(menubarHdl);

  CreateStandardWindowMenu(0,&menuRef);
  SetMenuID(menuRef,mWindow);
  InsertMenu(menuRef,0);

  DrawMenuBar();

  Gestalt(gestaltMenuMgrAttr,&response);
  if(response & gestaltMenuMgrAquaLayoutMask)
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
    {
      DeleteMenuItem(menuRef,iQuit);
      DeleteMenuItem(menuRef,iQuit - 1);
    }

    gRunningOnX = true;
  }
  else
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
      SetMenuItemCommandID(menuRef,iQuit,kHICommandQuit);

    menuRef = GetMenuRef(mDialogs);
    if(menuRef != NULL)
    {
      DisableMenuItem(menuRef,iSheetAlert);
      DisableMenuItem(menuRef,iSheetDialog);
    }
  }

  // .................................................... initial advisory text for window header

  doCopyPString("\pManipulate the window and controls.  Do typing.",gCurrentString);

  // ........................................................... install application event handler

  InstallApplicationEventHandler(NewEventHandlerUPP((EventHandlerProcPtr) appEventHandler),
                                 GetEventTypeCount(applicationEvents),applicationEvents,
                                 0,NULL);

  // ..................................................................................... install timer

  eventLoopTimerUPP = NewEventLoopTimerUPP((EventLoopTimerProcPtr) doIdle);

  InstallEventLoopTimer(GetCurrentEventLoop(),0,TicksToEventTime(GetCaretTime()),
                        eventLoopTimerUPP,NULL,NULL);

  // ...................................................................................... open window

  doNewWindow();

  // ............................................................... run application event loop

  RunApplicationEventLoop();
}

// *************************************************************************** doPreliminaries

void  doPreliminaries(void)
{
  MoreMasterPointers(128);
  InitCursor();
}

// *************************************************************************** appEventHandler

OSStatus  appEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                          void * userData)
{
  OSStatus      result = eventNotHandledErr;
  UInt32        eventClass;
  UInt32        eventKind;
  HICommand     hiCommand;
  MenuID        menuID;
  MenuItemIndex menuItem;
  WindowClass   windowClass;

  eventClass = GetEventClass(eventRef);
  eventKind  = GetEventKind(eventRef);

  switch(eventClass)
  {
    case kEventClassApplication:
      if(eventKind == kEventAppActivated)
        SetThemeCursor(kThemeArrowCursor);
      break;

    case kEventClassCommand:
      if(eventKind == kEventProcessCommand)
      {
        GetEventParameter(eventRef,kEventParamDirectObject,typeHICommand,NULL,
                          sizeof(HICommand),NULL,&hiCommand);
        if(hiCommand.commandID == kHICommandQuit)
          result = eventNotHandledErr;
        menuID = GetMenuID(hiCommand.menu.menuRef);
        menuItem = hiCommand.menu.menuItemIndex;
        if((hiCommand.commandID != kHICommandQuit) &&
           (menuID >= mAppleApplication && menuID <= mDialogs))
        {
          doMenuChoice(menuID,menuItem);
          result = noErr;
        }
        if(hiCommand.commandID == kPopupCountryID)
        {
          doControlHit2();
          result = noErr;
        }
      }
      break;

    case kEventClassMenu:
      if(eventKind == kEventMenuEnableItems)
      {
        GetWindowClass(FrontWindow(),&windowClass);
        if(windowClass == kDocumentWindowClass)
          doAdjustMenus();
        result = noErr;
      }
      break;

    case kEventClassMouse:
      if(eventKind == kEventMouseMoved)
      {
        GetWindowClass(FrontWindow(),&windowClass);
        if(windowClass == kDocumentWindowClass)
          doAdjustCursor(FrontWindow());
        result = noErr;
      }
      break;
  }

  return result;
}

// ************************************************************************ windowEventHandler

OSStatus  windowEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                             void* userData)
```

```
{
  OSStatus        result = eventNotHandledErr;
  UInt32          eventClass;
  UInt32          eventKind;
  WindowRef       windowRef;
  Rect            mainScreenRect, portRect;
  BitMap          screenBits;
  Point           idealHeightAndWidth, minimumHeightAndWidth, mouseLocation;
  ControlRef      controlRef;
  ControlPartCode controlPartCode;
  SInt8           charCode;
  UInt32          modifiers;
  HICommand       hiCommand;

  eventClass = GetEventClass(eventRef);
  eventKind  = GetEventKind(eventRef);

  switch(eventClass)
  {
    case kEventClassWindow:                                             // event class window

      GetEventParameter(eventRef,kEventParamDirectObject,typeWindowRef,NULL,sizeof(windowRef),
                        NULL,&windowRef);

      switch(eventKind)
      {
        case kEventWindowDrawContent:
          doDrawContent(windowRef);
          break;

        case kEventWindowActivated:
          doActivateDeactivate(windowRef,true);
          break;

        case kEventWindowDeactivated:
          doActivateDeactivate(windowRef,false);
          break;

        case kEventWindowGetIdealSize:
          mainScreenRect = GetQDGlobalsScreenBits(&screenBits)->bounds;
          idealHeightAndWidth.v = mainScreenRect.bottom - 75;
          idealHeightAndWidth.h = 600;
          SetEventParameter(eventRef,kEventParamDimensions,typeQDPoint,
                            sizeof(idealHeightAndWidth),&idealHeightAndWidth);
           result = noErr;
          break;

        case kEventWindowGetMinimumSize:
          minimumHeightAndWidth.v = 308;
          minimumHeightAndWidth.h = 290;
          SetEventParameter(eventRef,kEventParamDimensions,typeQDPoint,
                            sizeof(minimumHeightAndWidth),&minimumHeightAndWidth);
          result = noErr;
          break;

        case kEventWindowZoomed:
          GetWindowPortBounds(windowRef,&portRect);
          EraseRect(&portRect);
          doAdjustScrollBars(windowRef);
          result = noErr;
          break;

        case kEventWindowBoundsChanged:
          doAdjustScrollBars(windowRef);
          doDrawMessage(windowRef,true);
          result = noErr;
          break;

        case kEventWindowClose:
          doCloseWindow(windowRef);
          break;
      }
      break;

    case kEventClassControl:                                            // event class control
      switch(eventKind)
      {
        case kEventControlClick:
          GetEventParameter(eventRef,kEventParamMouseLocation,typeQDPoint,NULL,
                            sizeof(mouseLocation),NULL,&mouseLocation);
          SetPortWindowPort(FrontWindow());
          GlobalToLocal(&mouseLocation);
          controlRef = FindControlUnderMouse(mouseLocation,FrontWindow(),&controlPartCode);
          if(controlRef)
          {
            doControlHit1(FrontWindow(),controlRef,mouseLocation,controlPartCode);
            result = noErr;
          }
          break;
      }
      break;

    case kEventClassKeyboard:                                          // event class keyboard
      switch(eventKind)
      {
        case kEventRawKeyDown:
          if(gTypingTarget == 1 || gTypingTarget == 3)
          {
            GetEventParameter(eventRef,kEventParamKeyMacCharCodes,typeChar,NULL,
                              sizeof(charCode),NULL,&charCode);
            GetEventParameter(eventRef,kEventParamKeyModifiers,typeUInt32,NULL,
                              sizeof(modifiers),NULL,&modifiers);
            doDrawDocumentTyping(charCode,modifiers);
          }
          if(gTypingTarget == 1)
            result = noErr;
          break;
      }
      break;

    case kEventClassCommand:                                            // event class command
      if(eventKind == kEventProcessCommand)
      {
        GetEventParameter(eventRef,kEventParamDirectObject,typeHICommand,NULL,
                          sizeof(HICommand),NULL,&hiCommand);
        if(hiCommand.commandID == kHICommandOK)
          doCopyPString("\pOK button hit",gCurrentString);
        if(hiCommand.commandID == kHICommandCancel)
          doCopyPString("\pCancel button hit",gCurrentString);
        if(hiCommand.commandID == kHICommandOther)
          doCopyPString("\pOther button hit",gCurrentString);
        GetWindowPortBounds(FrontWindow(),&portRect);
        InvalWindowRect(FrontWindow(),&portRect);
      }
      break;
  }

  return result;
}

// ********************************************************************************** doNewWindow

void  doNewWindow(void)
{
  WindowRef      windowRef;
  Str255         windowTitleString = "\pCarbonEvents2 - ";
  Str255         theString;
  docStrucHandle docStrucHdl;
```

```
SInt16         a;
MenuRef        menuRef;
EventTypeSpec  windowEvents[] = { { kEventClassWindow,   kEventWindowDrawContent    },
                                  { kEventClassWindow,   kEventWindowActivated      },
                                  { kEventClassWindow,   kEventWindowDeactivated    },
                                  { kEventClassWindow,   kEventWindowGetIdealSize   },
                                  { kEventClassWindow,   kEventWindowGetMinimumSize },
                                  { kEventClassWindow,   kEventWindowZoomed         },
                                  { kEventClassWindow,   kEventWindowBoundsChanged  },
                                  { kEventClassWindow,   kEventWindowClose          },
                                  { kEventClassControl,  kEventControlClick         },
                                  { kEventClassKeyboard, kEventRawKeyDown           },
                                  { kEventClassCommand,  kEventProcessCommand       } };

// .......................................................................... open window and set attributes

if(!(windowRef = GetNewCWindow(rWindow,NULL,(WindowRef) -1)))
  ExitToShell();

ChangeWindowAttributes(windowRef,kWindowStandardHandlerAttribute,0);
if(gRunningOnX)
  ChangeWindowAttributes(windowRef,kWindowLiveResizeAttribute,0);

// ............................................................ alternative open window and set attributes

// Rect              contentRect = { 100,100,405,393 };
// WindowAttributes  attributes  = kWindowStandardHandlerAttribute |
//                                 kWindowStandardDocumentAttributes |
//                                 kWindowLiveResizeAttribute;
//
// CreateNewWindow(kDocumentWindowClass,attributes,&contentRect,&windowRef);
// RepositionWindow(windowRef,NULL,kWindowAlertPositionOnMainScreen);

// ............... get block for document structure, assign handle to window record refCon field

if(!(docStrucHdl = (docStrucHandle) NewHandle(sizeof(docStruc))))
  ExitToShell();

// ................................................................................................ set window title

SetWRefCon(windowRef,(SInt32) docStrucHdl);
gNumberOfWindows ++;
NumToString(gNumberOfWindows,theString);
doConcatPStrings(windowTitleString,theString);
SetWTitle(windowRef,windowTitleString);

SetPortWindowPort(windowRef);
TextSize(46);

// ................................... if running on Mac OS 8/9, set theme-compliant colour/pattern

SetThemeWindowBackground(windowRef,kThemeBrushDialogBackgroundActive,false);

// ............................................................................ install window event handler

InstallWindowEventHandler(windowRef,doGetHandlerUPP(),GetEventTypeCount(windowEvents),
                          windowEvents,0,NULL);

// ............................................................. get controls, adjust scroll bars, and show window

doGetControls(windowRef);
doAdjustScrollBars(windowRef);
ShowWindow(windowRef);

// ....................... enable Typing and Window menu, fix typing target and keyboard focus

menuRef = GetMenuRef(mTyping);
EnableMenuItem(menuRef,0);
```

```
  for(a = iDocument;a <= iAllOfTheAbove;a ++)
    CheckMenuItem(menuRef,a,false);
  CheckMenuItem(menuRef,iAllOfTheAbove,true);
  SetKeyboardFocus(windowRef,(*docStrucHdl)->editTextRef,kControlFocusNextPart);
  gTypingTarget = 3;

  EnableMenuItem(GetMenuRef(mWindow),0);
}

// ************************************************************************** doGetHandlerUPP

EventHandlerUPP  doGetHandlerUPP(void)
{
  static EventHandlerUPP windowEventHandlerUPP;

  if(windowEventHandlerUPP == NULL)
    windowEventHandlerUPP = NewEventHandlerUPP((EventHandlerProcPtr) windowEventHandler);

  return windowEventHandlerUPP;
}

// **************************************************************************** doCloseWindow

void  doCloseWindow(WindowRef windowRef)
{
  docStrucHandle docStrucHdl;

  KillControls(windowRef);

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));
  DisposeHandle((Handle) docStrucHdl);

  gNumberOfWindows --;

  if(gNumberOfWindows == 0)
  {
    DisableMenuItem(GetMenuRef(mTyping),0);
    DisableMenuItem(GetMenuRef(mWindow),0);
  }
}

// **************************************************************************** doGetControls

void  doGetControls(WindowRef windowRef)
{
  ControlRef     controlRef;
  docStrucHandle docStrucHdl;
  OSStatus       osError;
  Rect           controlRect;
  Boolean        booleanData = true;

  CreateRootControl(windowRef,&controlRef);

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

  SetRect(&controlRect,40,40,235,60);
  if((osError = CreatePopupButtonControl(windowRef,&controlRect,CFSTR("Time Zone:"),133,false,
                                        -1,0,0,&(*docStrucHdl)->popupTimeZoneRef)) != noErr)
    ExitToShell();

  SetRect(&controlRect,55,73,235,93);
  if((osError = CreatePopupButtonControl(windowRef,&controlRect,CFSTR("Country:"),134,false,
                                        -1,0,0,&(*docStrucHdl)->popupCountryRef)) != noErr)
    ExitToShell();

  SetRect(&controlRect,35,126,91,144);
  if((osError = CreateRadioButtonControl(windowRef,&controlRect,CFSTR("Red"),1,false,
                                        &(*docStrucHdl)->radiobuttonRedRef)) != noErr)
    ExitToShell();
  SetRect(&controlRect,35,148,91,166);
  if((osError = CreateRadioButtonControl(windowRef,&controlRect,CFSTR("White"),0,false,
                                         &(*docStrucHdl)->radiobuttonWhiteRef)) != noErr)
    ExitToShell();

  SetRect(&controlRect,35,170,91,188);
  if((osError = CreateRadioButtonControl(windowRef,&controlRect,CFSTR("Blue"),0,false,
                                         &(*docStrucHdl)->radiobuttonBlueRef)) != noErr)
    ExitToShell();

  SetRect(&controlRect,20,102,107,201);
  if((osError = CreateGroupBoxControl(windowRef,&controlRect,CFSTR("Colour"),true,
                                      &(*docStrucHdl)->groupboxColourRef)) != noErr)
    ExitToShell();

  SetRect(&controlRect,123,102,255,201);
  if((osError = CreateGroupBoxControl(windowRef,&controlRect,CFSTR("Typing"),true,
                                      &(*docStrucHdl)->groupboxTypingRef)) != noErr)
    ExitToShell();

  SetRect(&controlRect,63,213,132,233);
  if((osError = CreatePushButtonControl(windowRef,&controlRect,CFSTR("Cancel"),
                                        &(*docStrucHdl)->buttonRef)) != noErr)
    ExitToShell();

  SetRect(&controlRect,144,213,213,233);
  if((osError = CreatePushButtonControl(windowRef,&controlRect,CFSTR("OK"),
                                        &(*docStrucHdl)->buttonDefaultRef)) != noErr)
    ExitToShell();

  SetRect(&controlRect,26,251,249,267);
  if((osError = CreateEditTextControl(windowRef,&controlRect,NULL,false,true,NULL,
                                      &(*docStrucHdl)->editTextRef)) != noErr)
    ExitToShell();

  SetRect(&controlRect,0,0,16,262);
  if((osError = CreateScrollBarControl(windowRef,&controlRect,0,0,125,0,false,NULL,
                                       &(*docStrucHdl)->scrollbarVertRef)) != noErr)
    ExitToShell();

  SetRect(&controlRect,0,0,245,16);
  if((osError = CreateScrollBarControl(windowRef,&controlRect,0,0,125,0,true,NULL,
                                       &(*docStrucHdl)->scrollbarHorizRef)) != noErr)
    ExitToShell();

  AutoEmbedControl((*docStrucHdl)->radiobuttonRedRef,windowRef);
  AutoEmbedControl((*docStrucHdl)->radiobuttonWhiteRef,windowRef);
  AutoEmbedControl((*docStrucHdl)->radiobuttonBlueRef,windowRef);

  SetControlCommandID((*docStrucHdl)->popupCountryRef,kPopupCountryID);

  SetControlData((*docStrucHdl)->buttonDefaultRef,kControlEntireControl,
                 kControlPushButtonDefaultTag,sizeof(booleanData),&booleanData);
}

// ************************************************************************************ doIdle

void  doIdle(void)
{
  if(!gRunningOnX)
    IdleControls(FrontWindow());
}

// ***************************************************************************** doAdjustMenus

void  doAdjustMenus(void)
{
  MenuRef   menuRef;
  OSStatus  osError;
  WindowRef windowRef;

  if(FrontWindow())
  {
    menuRef = GetMenuRef(mFile);
    EnableMenuItem(menuRef,iClose);
    menuRef = GetMenuRef(mTyping);
    EnableMenuItem(menuRef,0);
  }
  else
  {
    menuRef = GetMenuRef(mFile);
    DisableMenuItem(menuRef,iClose);
    menuRef = GetMenuRef(mTyping);
    DisableMenuItem(menuRef,0);
  }

  if(gRunningOnX)
  {
    if((osError = GetSheetWindowParent(FrontWindow(),&windowRef)) == noErr)
    {
      menuRef = GetMenuRef(mFile);
      DisableMenuItem(menuRef,iClose);
      menuRef = GetMenuRef(mTyping);
      DisableMenuItem(menuRef,0);
    }
    else
    {
      if(FrontWindow())
      {
        menuRef = GetMenuRef(mTyping);
        EnableMenuItem(menuRef,0);
      }
    }

    menuRef = GetMenuRef(mDialogs);
    if(((osError = GetSheetWindowParent(FrontWindow(),&windowRef)) == noErr) ||
       (FrontWindow() == NULL) || IsWindowCollapsed(FrontWindow()))
    {
      DisableMenuItem(menuRef,iSheetAlert);
      DisableMenuItem(menuRef,iSheetDialog);
    }
    else
    {
      EnableMenuItem(menuRef,iSheetAlert);
      EnableMenuItem(menuRef,iSheetDialog);
    }
  }
}

// ******************************************************************************* doMenuChoice

void  doMenuChoice(MenuID menuID,MenuItemIndex menuItem)
{
  WindowRef      windowRef;
  SInt16         a;
  MenuRef        menuRef;
  docStrucHandle docStrucHdl;

  if(menuID == 0)
    return;

  windowRef = FrontWindow();

  switch(menuID)
  {
    case mAppleApplication:
      if(menuItem == iAbout)
```

```
        SysBeep(10);
      break;

    case mFile:
      if(menuItem == iNew)
        doNewWindow();
      else if(menuItem == iClose)
      {
        doCloseWindow(windowRef);
        DisposeWindow(windowRef);
      }
      break;

    case mTyping:
      menuRef = GetMenuRef(mTyping);
      for(a = iDocument;a <= iAllOfTheAbove;a ++)
        CheckMenuItem(menuRef,a,false);
      CheckMenuItem(menuRef,menuItem,true);
      gTypingTarget = menuItem;
      docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));
      if(menuItem == iDocument)
        SetKeyboardFocus(windowRef,(*docStrucHdl)->editTextRef,kControlFocusNoPart);
      else
        SetKeyboardFocus(windowRef,(*docStrucHdl)->editTextRef,kControlFocusNextPart);
      break;

    case mDialogs:
      if(menuItem == iMovableModal)
        if(doMovableModalDialog() != noErr)
          ExitToShell();
      if(menuItem == iSheetAlert)
        if(doSheetAlert() != noErr)
          ExitToShell();
      if(menuItem == iSheetDialog)
        if(doSheetDialog() != noErr)
          ExitToShell();
      break;
  }
}

// ****************************************************************************** doDrawContent

void  doDrawContent(WindowRef windowRef)
{
  SetPortWindowPort(windowRef);
  doDrawMessage(windowRef,windowRef == FrontWindow());
}

// *************************************************************************** doActivateWindow

void  doActivateDeactivate(WindowRef windowRef,Boolean becomingActive)
{
  if(becomingActive)
    doDrawMessage(windowRef,becomingActive);
  else
    doDrawMessage(windowRef,becomingActive);
}

// ****************************************************************************** doControlHit1

void  doControlHit1(WindowRef windowRef,ControlRef controlRef,Point mouseLocation,
                    ControlPartCode controlPartCode)
{
  docStrucHandle docStrucHdl;
  SInt16         controlValue;

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

  if(controlRef == (*docStrucHdl)->popupTimeZoneRef)
  {
    TrackControl(controlRef,mouseLocation,(ControlActionUPP) -1);
    controlValue = GetControlValue(controlRef);
    doPopupMenuChoice(windowRef,controlRef,controlValue);
  }
  else if(controlRef == (*docStrucHdl)->scrollbarVertRef)
  {
    doVertScrollbar(controlPartCode,windowRef,controlRef,mouseLocation);
  }
  else if(controlRef == (*docStrucHdl)->scrollbarHorizRef)
  {
    TrackControl(controlRef,mouseLocation,gActionFunctionHorizUPP);
  }
  else
  {
    if(TrackControl(controlRef,mouseLocation,NULL))
    {
      if(controlRef == (*docStrucHdl)->radiobuttonRedRef ||
         controlRef == (*docStrucHdl)->radiobuttonWhiteRef ||
         controlRef == (*docStrucHdl)->radiobuttonBlueRef)
      {
        doRadioButtons(controlRef,windowRef);
      }
      if(controlRef == (*docStrucHdl)->buttonRef ||
         controlRef == (*docStrucHdl)->buttonDefaultRef)
      {
        doPushButtons(controlRef,windowRef);
      }
    }
  }
}

// **************************************************************************** doControlHit2

void  doControlHit2(void)
{
  docStrucHandle docStrucHdl;
  ControlRef     controlRef;
  SInt16         controlValue;

  docStrucHdl = (docStrucHandle) GetWRefCon(FrontWindow());
  controlRef = (*docStrucHdl)->popupCountryRef;

  controlValue = GetControlValue(controlRef);
  doPopupMenuChoice(FrontWindow(),controlRef,controlValue);
}

// ************************************************************************ doPopupMenuChoice

void  doPopupMenuChoice(WindowRef windowRef,ControlRef controlRef,SInt16 controlValue)
{
  MenuRef menuRef;
  Size    actualSize;

  GetControlData(controlRef,kControlEntireControl,kControlPopupButtonMenuHandleTag,
                 sizeof(menuRef),&menuRef,&actualSize);
  GetMenuItemText(menuRef,controlValue,gCurrentString);
  doDrawMessage(windowRef,true);
}

// ************************************************************************** doVertScrollbar

void  doVertScrollbar(ControlPartCode controlPartCode,WindowRef windowRef,
                      ControlRef controlRef,Point mouseXY)
{
  Str255 valueString;

  doCopyPString("\pScroll Bar Control Value: ",gCurrentString);
  switch(controlPartCode)
  {
    case kControlIndicatorPart:
      if(TrackControl(controlRef,mouseXY,NULL))
      {
        NumToString((SInt32) GetControlValue(controlRef),valueString);
        doConcatPStrings(gCurrentString,valueString);
        doDrawMessage(windowRef,true);
      }
      break;

    case kControlUpButtonPart:
    case kControlDownButtonPart:
    case kControlPageUpPart:
    case kControlPageDownPart:
      TrackControl(controlRef,mouseXY,gActionFunctionVertUPP);
      break;
  }
}

// ********************************************************************** actionFunctionVert

void  actionFunctionVert(ControlRef controlRef,ControlPartCode controlPartCode)
{
  SInt16    scrollDistance, controlValue;
  Str255    valueString;
  WindowRef windowRef;

  doCopyPString("\pScroll Bar Control Value: ",gCurrentString);

  if(controlPartCode)
  {
    switch(controlPartCode)
    {
      case kControlUpButtonPart:
      case kControlDownButtonPart:
        scrollDistance = 2;
        break;

      case kControlPageUpPart:
      case kControlPageDownPart:
        scrollDistance = 55;
        break;
    }

    if((controlPartCode == kControlDownButtonPart) ||
       (controlPartCode == kControlPageDownPart))
      scrollDistance = -scrollDistance;

    controlValue = GetControlValue(controlRef);
    if(((controlValue == GetControlMaximum(controlRef)) && scrollDistance < 0) ||
       ((controlValue == GetControlMinimum(controlRef)) && scrollDistance > 0))
      return;

    doMoveScrollBox(controlRef,scrollDistance);

    NumToString((SInt32) GetControlValue(controlRef),valueString);
    doConcatPStrings(gCurrentString,valueString);
    windowRef = GetControlOwner(controlRef);
    doDrawMessage(windowRef,true);
  }
}

// ********************************************************************* actionFunctionHoriz

void  actionFunctionHoriz(ControlRef controlRef,ControlPartCode controlPartCode)
{
  SInt16    scrollDistance, controlValue;
  Str255    valueString;
  WindowRef windowRef;

  doCopyPString("\pScroll Bar Control Value: ",gCurrentString);

  if(controlPartCode != kControlIndicatorPart)
  {
    switch(controlPartCode)
    {
      case kControlUpButtonPart:
      case kControlDownButtonPart:
        scrollDistance = 2;
        break;

      case kControlPageUpPart:
      case kControlPageDownPart:
        scrollDistance = 55;
        break;
    }

    if((controlPartCode == kControlDownButtonPart) ||
       (controlPartCode == kControlPageDownPart))
      scrollDistance = -scrollDistance;

    controlValue = GetControlValue(controlRef);
    if(((controlValue == GetControlMaximum(controlRef)) && scrollDistance < 0) ||
       ((controlValue == GetControlMinimum(controlRef)) && scrollDistance > 0))
      return;

    doMoveScrollBox(controlRef,scrollDistance);
  }

  NumToString((SInt32) GetControlValue(controlRef),valueString);
  doConcatPStrings(gCurrentString,valueString);
  windowRef = GetControlOwner(controlRef);
  doDrawMessage(windowRef,true);
}

// *************************************************************************** doMoveScrollBox

void doMoveScrollBox(ControlRef controlRef,SInt16 scrollDistance)
{
  SInt16 oldControlValue, controlValue, controlMax;

  oldControlValue = GetControlValue(controlRef);
  controlMax = GetControlMaximum(controlRef);

  controlValue = oldControlValue - scrollDistance;

  if(controlValue < 0)
    controlValue = 0;
  else if(controlValue > controlMax)
    controlValue = controlMax;

  SetControlValue(controlRef,controlValue);
}

// **************************************************************************** doRadioButtons

void  doRadioButtons(ControlRef controlRef,WindowRef windowRef)
{
  docStrucHandle docStrucHdl;

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

  SetControlValue((*docStrucHdl)->radiobuttonRedRef,kControlRadioButtonUncheckedValue);
  SetControlValue((*docStrucHdl)->radiobuttonWhiteRef,kControlRadioButtonUncheckedValue);
  SetControlValue((*docStrucHdl)->radiobuttonBlueRef,kControlRadioButtonUncheckedValue);
  SetControlValue(controlRef,kControlRadioButtonCheckedValue);
}
```

```
// ************************************************************************** doCheckboxes

void  doCheckboxes(ControlRef controlRef)
{
  SetControlValue(controlRef,!GetControlValue(controlRef));
}

// ************************************************************************* doPushButtons

void  doPushButtons(ControlRef controlRef,WindowRef windowRef)
{
  docStrucHandle docStrucHdl;

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

  if(controlRef == (*docStrucHdl)->buttonRef)
  {
    doCopyPString("\pCancel button",gCurrentString);
    doDrawMessage(windowRef,true);
  }
  else if(controlRef == (*docStrucHdl)->buttonDefaultRef)
  {
    doCopyPString("\pDefault button",gCurrentString);
    doDrawMessage(windowRef,true);
  }
}

// ********************************************************************* doAdjustScrollBars

void  doAdjustScrollBars(WindowRef windowRef)
{
  Rect           portRect;
  docStrucHandle docStrucHdl;

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

  GetWindowPortBounds(windowRef,&portRect);

  HideControl((*docStrucHdl)->scrollbarVertRef);
  HideControl((*docStrucHdl)->scrollbarHorizRef);

  MoveControl((*docStrucHdl)->scrollbarVertRef,portRect.right - kScrollBarWidth,
              portRect.top + 25);
  MoveControl((*docStrucHdl)->scrollbarHorizRef,portRect.left -1,
              portRect.bottom - kScrollBarWidth);

  SizeControl((*docStrucHdl)->scrollbarVertRef,16, portRect.bottom - 39);
  SizeControl((*docStrucHdl)->scrollbarHorizRef, portRect.right - 13,16);

  ShowControl((*docStrucHdl)->scrollbarVertRef);
  ShowControl((*docStrucHdl)->scrollbarHorizRef);

  SetControlMaximum((*docStrucHdl)->scrollbarVertRef,portRect.bottom - portRect.top - 25
                    - kScrollBarWidth);
  SetControlMaximum((*docStrucHdl)->scrollbarHorizRef,portRect.right - portRect.left
                    - kScrollBarWidth);
}

// ************************************************************************* doAdjustCursor

void  doAdjustCursor(WindowRef windowRef)
{
  RgnHandle  myArrowRegion;
  RgnHandle  myIBeamRegion;
  Rect       cursorRect;
  Point      mousePt;
  ControlRef controlRef;
  Cursor     arrow;
  myArrowRegion = NewRgn();
  myIBeamRegion = NewRgn();
  SetRectRgn(myArrowRegion,-32768,-32768,32767,32767);

  SetRect(&cursorRect,24,250,254,269);

  SetPortWindowPort(windowRef);
  LocalToGlobal(&topLeft(cursorRect));
  LocalToGlobal(&botRight(cursorRect));

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

  GetGlobalMouse(&mousePt);
  GetKeyboardFocus(FrontWindow(),&controlRef);

  if(PtInRgn(mousePt,myIBeamRegion) && controlRef)
    SetCursor(*(GetCursor(iBeamCursor)));
  else
    SetCursor(GetQDGlobalsArrow(&arrow));

  DisposeRgn(myArrowRegion);
  DisposeRgn(myIBeamRegion);
}

// ******************************************************************** doDrawDocumentTyping

void  doDrawDocumentTyping(SInt8 charCode,UInt32 modifiers)
{
  Rect        typingRect  = { 118,128,194,253 };
  Rect        shiftRect   = { 131,181,139,189 };
  Rect        controlRect = { 144,181,152,189 };
  Rect        optionRect  = { 157,181,165,189 };
  Rect        cmdRect     = { 170,181,178,189 };
  Rect        textBoxRect;
  CFStringRef stringRef;

  EraseRect(&typingRect);

  stringRef = CFStringCreateWithPascalString(NULL,"\pShift",kCFStringEncodingMacRoman);
  SetRect(&textBoxRect,142,132,242,147);
  if((modifiers & shiftKey) != 0)  TextMode(srcOr); else TextMode(grayishTextOr);
  DrawThemeTextBox(stringRef,kThemeSmallSystemFont,0,false,&textBoxRect,teJustLeft,NULL);

  stringRef = CFStringCreateWithPascalString(NULL,"\pControl",kCFStringEncodingMacRoman);
  SetRect(&textBoxRect,142,145,242,160);
  if((modifiers & controlKey) != 0)  TextMode(srcOr); else TextMode(grayishTextOr);
  DrawThemeTextBox(stringRef,kThemeSmallSystemFont,0,false,&textBoxRect,teJustLeft,NULL);

  stringRef = CFStringCreateWithPascalString(NULL,"\pOption",kCFStringEncodingMacRoman);
  SetRect(&textBoxRect,142,158,242,173);
  if((modifiers & optionKey) != 0)  TextMode(srcOr); else TextMode(grayishTextOr);
  DrawThemeTextBox(stringRef,kThemeSmallSystemFont,0,false,&textBoxRect,teJustLeft,NULL);

  stringRef = CFStringCreateWithPascalString(NULL,"\pCmd",kCFStringEncodingMacRoman);
  SetRect(&textBoxRect,142,171,242,186);
  if((modifiers & cmdKey) != 0)  TextMode(srcOr); else TextMode(grayishTextOr);
  DrawThemeTextBox(stringRef,kThemeSmallSystemFont,0,false,&textBoxRect,teJustLeft,NULL);

  if(stringRef != NULL)
    CFRelease(stringRef);

  TextMode(srcOr);
  MoveTo(205,171);
  DrawChar(charCode);
}

// ****************************************************************************** doDrawMessage
```

```
void  doDrawMessage(WindowRef windowRef,Boolean inState)
{
  Rect        portRect, headerRect, textBoxRect;
  CFStringRef stringRef;

  SetPortWindowPort(windowRef);

  GetWindowPortBounds(windowRef,&portRect);

  SetRect(&headerRect,portRect.left - 1,portRect.top - 1,portRect.right + 1,
          portRect.top + 26);
  DrawThemeWindowHeader(&headerRect,inState);

  stringRef = CFStringCreateWithPascalString(NULL,gCurrentString,
                                             kCFStringEncodingMacRoman);
  SetRect(&textBoxRect,portRect.left,5,portRect.right,25);

  if(inState == kThemeStateActive)
    TextMode(srcOr);
  else
    TextMode(grayishTextOr);

  DrawThemeTextBox(stringRef,kThemeSmallSystemFont,inState,false,&textBoxRect,teJustCenter,
                   NULL);
  if(stringRef != NULL)
    CFRelease(stringRef);
}

// ************************************************************************ doConcatPStrings

void  doConcatPStrings(Str255 targetString,Str255 appendString)
{
  SInt16 appendLength;

  appendLength = MIN(appendString[0],255 - targetString[0]);

  if(appendLength > 0)
  {
    BlockMoveData(appendString+1,targetString+targetString[0]+1,(SInt32) appendLength);
    targetString[0] += appendLength;
  }
}

// ************************************************************************** doCopyPString

void  doCopyPString(Str255 sourceString,Str255 destinationString)
{
  SInt16 stringLength;

  stringLength = sourceString[0];
  BlockMove(sourceString + 1,destinationString + 1,stringLength);
  destinationString[0] = stringLength;
}

// ******************************************************************************************
// Dialogs.c
// ******************************************************************************************

// .......................................................................................... includes

#include "CarbonEvents2.h"

// .................................................................................. global variables

Boolean gSound   = 0;
Boolean gVideo   = 0;
Boolean gEffects = 0;
```

```
extern Str255 gCurrentString;

// *************************************************************************** doSheetAlert

OSStatus  doSheetAlert(void)
{
  AlertStdCFStringAlertParamRec paramRec;
  Str255                        messageText, informativeText;
  CFStringRef                   messageTextCF, informativeTextCF;
  OSStatus                      osError = noErr;
  DialogRef                     dialogRef;

  GetStandardAlertDefaultParams(&paramRec,kStdCFStringAlertVersionOne);
  paramRec.cancelText  = CFSTR("Cancel");
  paramRec.otherText  = CFSTR("Other");

  GetIndString(messageText,rSheetStrings,sAlertSheetMessage);
  GetIndString(informativeText,rSheetStrings,sAlertSheetInformative);
  messageTextCF = CFStringCreateWithPascalString(NULL,messageText,
                                                 CFStringGetSystemEncoding());
  informativeTextCF = CFStringCreateWithPascalString(NULL,informativeText,
                                                     CFStringGetSystemEncoding());

  osError = CreateStandardSheet(kAlertCautionAlert,messageTextCF,informativeTextCF,&paramRec,
                                GetWindowEventTarget(FrontWindow()),&dialogRef);
  if(osError == noErr)
    osError = ShowSheetWindow(GetDialogWindow(dialogRef),FrontWindow());

  if(messageTextCF != NULL)
    CFRelease(messageTextCF);
  if(informativeTextCF != NULL)
    CFRelease(informativeTextCF);

  doAdjustMenus();

  return osError;
}

// *************************************************************************** doSheetDialog

OSStatus  doSheetDialog(void)
{
  DialogRef     dialogRef;
  WindowRef     windowRef;
  EventTypeSpec sheetEvents[] = { kEventClassMouse, kEventMouseDown };
  ControlRef    controlRef;
  Str255        stringData = "\pBradman";
  OSStatus      osError = noErr;

  dialogRef = GetNewDialog(rSheetDialog,NULL,(WindowRef) -1);
  windowRef = GetDialogWindow(dialogRef);
  ChangeWindowAttributes(windowRef,kWindowStandardHandlerAttribute,0);

  InstallWindowEventHandler(windowRef,doGetSheetHandlerUPP(),GetEventTypeCount(sheetEvents),
                            sheetEvents,0,NULL);

  SetDialogDefaultItem(dialogRef,kStdOkItemIndex);

  GetDialogItemAsControl(dialogRef,2,&controlRef);
  SetDialogItemText((Handle) controlRef,stringData);
  SelectDialogItemText(dialogRef,2,0,32767);

  osError = ShowSheetWindow(GetDialogWindow(dialogRef),FrontWindow());

  doAdjustMenus();

  return osError;
}
```

```
// ******************************************************************** doGetSheetHandlerUPP

EventHandlerUPP  doGetSheetHandlerUPP(void)
{
  static EventHandlerUPP sheetEventHandlerUPP;

  if(sheetEventHandlerUPP == NULL)
    sheetEventHandlerUPP = NewEventHandlerUPP((EventHandlerProcPtr) sheetEventHandler);

  return sheetEventHandlerUPP;
}

// *********************************************************************** sheetEventHandler

OSStatus  sheetEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                            void* userData)
{
  OSStatus        result = eventNotHandledErr;
  UInt32          eventClass;
  UInt32          eventKind;
  Point           mouseLocation;
  ControlRef      controlRef, controlRefOKButton;
  ControlPartCode controlPartCode;
  DialogRef       dialogRef;
  Rect            portRect;

  eventClass = GetEventClass(eventRef);
  eventKind  = GetEventKind(eventRef);

  if(eventClass == kEventClassMouse)
  {
    if(eventKind == kEventMouseDown)
    {
      GetEventParameter(eventRef,kEventParamMouseLocation,typeQDPoint,NULL,
                        sizeof(mouseLocation),NULL,&mouseLocation);

      SetPortWindowPort(FrontWindow());
      GlobalToLocal(&mouseLocation);
      controlRef = FindControlUnderMouse(mouseLocation,FrontWindow(),&controlPartCode);
      if(controlRef)
      {
        dialogRef = GetDialogFromWindow(FrontWindow());
        GetDialogItemAsControl(dialogRef,1,&controlRefOKButton);

        if(controlRef == controlRefOKButton)
        {
          GetDialogItemAsControl(dialogRef,2,&controlRef);
          GetDialogItemText((Handle) controlRef,gCurrentString);

          HideSheetWindow(FrontWindow());
          DisposeDialog(dialogRef);

          GetWindowPortBounds(FrontWindow(),&portRect);
          InvalWindowRect(FrontWindow(),&portRect);

          return noErr;
        }
      }
    }
  }

  return result;
}

// ******************************************************************** doMovableModalDialog

OSStatus  doMovableModalDialog(void)
{
  OSStatus      osError = noErr;
  Rect          rect = { 0,0,167,148 };
  WindowRef     windowRef;
  Rect          pushButtonRect = { 127,63,147,132 };
  ControlRef    controlRef;
  ControlRef    soundControlRef, videoControlRef, effectsControlRef, boxControlRef;
  Rect          checkboxRect = { 37,32,55,124 };
  ControlID     controlID;
  Rect          groupboxRect = { 10,16,113,132 };
  EventTypeSpec dialogEvents[] = { kEventClassControl, kEventControlHit };

  osError = CreateNewWindow(kMovableModalWindowClass,kWindowStandardHandlerAttribute,&rect,
                            &windowRef);
  if(osError == noErr)
  {
    RepositionWindow(windowRef,FrontWindow(),kWindowAlertPositionOnMainScreen);
    SetThemeWindowBackground(windowRef,kThemeBrushDialogBackgroundActive,false);

    CreateRootControl(windowRef,&controlRef);

    CreatePushButtonControl(windowRef,&pushButtonRect,CFSTR("OK"),&controlRef);
    SetWindowDefaultButton(windowRef,controlRef);
    controlID.id = 'okbt';
    SetControlID(controlRef,&controlID);

    CreateCheckBoxControl(windowRef,&checkboxRect,CFSTR("Sound On"),1,false,&soundControlRef);
    controlID.id = 'chb1';
    SetControlID(soundControlRef,&controlID);
    SetControlValue(soundControlRef,gSound);

    OffsetRect(&checkboxRect,0,22);
    CreateCheckBoxControl(windowRef,&checkboxRect,CFSTR("Video On"),1,false,&videoControlRef);
    controlID.id = 'chb2';
    SetControlID(videoControlRef,&controlID);
    SetControlValue(videoControlRef,gVideo);
    OffsetRect(&checkboxRect,0,22);
    CreateCheckBoxControl(windowRef,&checkboxRect,CFSTR("Effects On"),1,false,
                          &effectsControlRef);
    controlID.id = 'chb3';
    SetControlID(effectsControlRef,&controlID);
    SetControlValue(effectsControlRef,gEffects);

    CreateGroupBoxControl(windowRef,&groupboxRect,CFSTR("Preferences"),true,&boxControlRef);

    AutoEmbedControl(soundControlRef,windowRef);
    AutoEmbedControl(videoControlRef,windowRef);
    AutoEmbedControl(effectsControlRef,windowRef);

    InstallWindowEventHandler(windowRef,doGetDialogHandlerUPP(),
                              GetEventTypeCount(dialogEvents),dialogEvents,windowRef,NULL);
    ShowWindow(windowRef);
    osError = RunAppModalLoopForWindow(windowRef);
  }

  return osError;
}

// ********************************************************************* doGetDialogHandlerUPP

EventHandlerUPP  doGetDialogHandlerUPP(void)
{
  static EventHandlerUPP dialogEventHandlerUPP;

  if(dialogEventHandlerUPP == NULL)
    dialogEventHandlerUPP = NewEventHandlerUPP((EventHandlerProcPtr) dialogEventHandler);

  return dialogEventHandlerUPP;
}

// ************************************************************************ dialogEventHandler
```

```
OSStatus  dialogEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                             void *userData)
{
  OSStatus   result = eventNotHandledErr;
  UInt32     eventClass;
  UInt32     eventKind;
  ControlRef controlRef;
  ControlID  controlID;

  eventClass = GetEventClass(eventRef);
  eventKind  = GetEventKind(eventRef);

  if(eventClass == kEventClassControl)
  {
    if(eventKind == kEventControlHit)
    {
      GetEventParameter(eventRef,kEventParamDirectObject,typeControlRef,NULL,
                        sizeof(ControlRef),NULL,&controlRef);

      GetControlID(controlRef,&controlID);
      if(controlID.id == 'okbt')
      {
        QuitAppModalLoopForWindow(userData);
        DisposeWindow(userData);
        result = noErr;
      }
      else
      {
        SetControlValue(controlRef,!GetControlValue(controlRef));
        if(controlID.id == 'chb1')
          gSound = GetControlValue(controlRef);
        else if(controlID.id == 'chb2')
          gVideo = GetControlValue(controlRef);
        else if(controlID.id == 'chb3')
          gEffects = GetControlValue(controlRef);
        result = noErr;
      }
    }
  }
  return result;
}

// *******************************************************************************************
```

Demonstration Program CarbonEvents2 Comments

When this program is run, the user should:

- Open and close windows, and drag, resize, and zoom open windows, noting particularly the size to which the window zooms in an out.
- Interact with the pop-up menu button, push button and scrollbar controls in open windows.
- Send the application to the background and bring it to the foreground, noting the activation and deactivation of the window controls.
- Type into the edit text control, with and without the Shift, Control, Option, and/or Command keys held down. Then choose Document or Edit Text Control to change the target for keyboard input.
- Choose the items in the Dialogs menu to open, close, and interact with the movable modal dialog and, on Mac OS X only, window-modal (sheet) alerts and window-modal (sheet) dialogs.
- Quit the application by choosing the Quit item in the Mac OS 8/9 File/Mac OS X Application menu and using its Command-key equivalent.

The functions relating to controls in this program, including the action functions for the scroll bars, are similar to those in the demonstration programs Controls1 and Controls2 (Chapter 7).

CarbonEvents2.c

main

If the program is running on OS 8/9, SetMenuItemCommandID is called to assign the command ID 'quit' to the Quit item in the File menu. (This command is assigned to the Mac OS X Quit item by default.) Thus, when the Quit item is chosen on Mac OS 8/9 and Mac OS X, the standard application event handler will call the default Quit Application Apple event handler (automatically installed when RunApplicationEventLoop is called) to close down the program.

The call to InstallApplicationEventHandler installs the program's application event handler.

The call to InstallEventLoopTimer installs a timer set to fire at the interval returned by the call to GetCaretTime, which is converted to event time (seconds) by the macro TicksToEventTime. The timer will be used to trigger a call to the function doIdle, within which IdleControls is called to blink the insertion point caret in the windows' edit text control. A universal procedure pointer to doIdle is passed in the inTimerProc parameter.

When RunApplicationEventLoop is called, registered events will be dispatched to the application.

appEventHandler

appEventHandler is the application's application event handler. It is a callback function.

Firstly, the calls to GetEventClass and GetEventKind get the event class and type. The function then switches on the event class.

If the event class is kEventClassApplication and the event type is kEventAppActivated, the cursor is set to the arrow cursor. eventNotHandledErr is returned by the handler, ensuring that the event will be propagated to the standard application event handler.

If the event class is kEventClassCommand and the event type is kEventProcessCommand, GetEventParameter is called to extract the specified data from the event. This data is returned in a variable of type HICommand.

If an examination of the commandID field of the HICommand structure reveals that the command ID is 'quit', the handler returns eventNotHandledErr, ensuring that the event will be propagated to the standard application event handler. This ensures that the standard handler calls the default Quit Application Apple event handler.

The menu ID and item number are then extracted from the HICommand structure. If the command ID is not 'quit' and the menu ID is that for one of the program's pull-down menus, doMenuChoice is called to further handle the event. Because the event is fully handled by the program, noErr is returned by the handler to ensure that the event is not further propagated.

In this program, each pop-up menu button control is handled in a different manner. As will be seen, the second (Country) pop-up menu button control is assigned the command ID 'ctry' (kPopupCountryID) on creation. Thus, if a mouse-down occurs in this pop-up menu button, the standard window event handler calls TrackControl to handle user action, following which the kEventProcessCommand is dispatched to the application. Within the application's application event handler, if the command ID in the HICommand structure's commandID field is 'ctry', the function doControlHit2 is called, following which noErr is returned by the handler.

If the event class is kEventClassMenu and the event type is kEventMenuEnableItems, the function doAdjustMenus is called. The kEventMenuEnableItems event type is dispatched when a mouse-down occurs in a pull-down menu or a menu-related Command-key equivalent is pressed.

If the event class is kEventClassMouse and the event type is kEventMouseMoved, and if the front window is of the document kind, the function doAdjustCursor is called to adjust the cursor to the I-beam shape if it is over an edit text control with keyboard focus, or to the arrow shape if it is not.

windowEventHandler

windowEventHandler is the application's window event handler. It is a callback function.

Firstly, the calls to GetEventClass and GetEventKind get the event class and type. The function then switches on the event class.

If the event class is kEventClassWindow, GetEventParameter is called to extract the window reference from the event before a switch on the event type is entered.

If the event type is kEventWindowDrawContent, the function doDrawContent (the window update function) is called. (Note that doDrawContent does not call BeginUpdate and EndUpdate because there is no need to call those functions when responding to kEventWindowDrawContent events.) The handler returns eventNotHandledErr, allowing the event to be passed to the standard window event handler which, in turn, attends to its part of the update process, including drawing the controls.

Note that registering the kEventWindowDrawContent event type and responding in this way obviates the necessity for an event filter (callback) function (which calls the window update function) for the movable modal dialog.

If the event type is kEventWindowActivated or kEventWindowDeactivated, the function doActivateDeactivate is called to draw the text in the window header in the appropriate (activated or deactivated) state. The handler returns eventNotHandledErr, allowing the event to be passed to the standard window event handler which, in turn, attends to its part of the activation/deactivation process, including activating/deactivating the controls.

If the event type is kEventWindowGetIdealSize, the handler responds by calling SetEventParameter, which sets the height and width to which the window will be zoomed when it is zoomed out. The handler returns noErr to defeat further propagation of the event.

If the event type is kEventWindowGetMinimumSize, the handler responds by calling SetEventParameter, which sets the minimum height and width to which the window can be resized. The handler returns noErr to defeat further propagation of the event.

If the event type is kEventWindowZoomed, the window's port rectangle is erased, the function doAdjustScrollBars is called to resize and reposition the scroll bars, and the handler returns noErr.

Since the kWindowLiveResizeAttribute attribute is set on the window, the kEventWindowBoundsChanged event type will be received continually as the window is being resized (as opposed to only one kEventWindowBoundsChanged event type being received, when the mouse button is released, when the kWindowLiveResizeAttribute attribute is not set). The function doAdjustScrollBars is continally called to resize and reposition the scroll bars, the function doDrawMessage is continually called to redraw the window header and associated text, and the handler returns noErr.

If the event type is kEventWindowClose, the function doCloseWindow is called to dispose of the window's controls and document structure handle, decrement the global variable holding the current number of open windows, and disable the Typing Target and Window menus if no windows will be open when this window is closed. EventNotHandledErr is returned by the handler to cause the standard window event handler to dispose of the window.

If the event class is kEventClassControl and the event type is kEventControlClick, GetEventParameter is called to extract the mouse location, which will be in global coordinates, from the event. The mouse coordinates are then converted to local coordinates preparatory to a call to FindControlUnderMouse. If there is a control under the mouse cursor, the function doControlHit1 is called to further handle the event. When doControlHit1 returns, the handler returns noErr to defeat further propagation of the event. (Note that, so far as the first (Time Zone) pop-up menu button is concerned, this approach to pop-up menu button control handling differs from that in the demonstration program CarbonEvents1.)

If the event class is kEventClassKeyboard and the event type is kEventRawKeyDown, and if the current typing target (as chosen in the Typing Target menu) is either the "document" or the "document" and edit text control combined, GetEventParameter is called twice to get the character code and the modifier keys that were down (if any), and both of these parameters are passed to the function doDrawDocumentTyping to draw the character and highlight the modifier key (if any) indicator in the ("document") Typing group box in the window. If the current typing target is the "document" only, the handler returns noErr to prevent the edit text control receiving the event, otherwise eventNotHandledErr is returned to allow the event to propagate to the edit text control.

The last block pertains to the window-modal (sheet) alert created by the function doSheetAlert in Dialogs.c. When the user clicks in one of the sheet's buttons, the parent window receives the relevant command ID (kHICommandOK, kHICommandCancel, or kHICommandOther). The identity of the button is drawn in the parent window's window header frame.

doNewWindow

After GetNewCWindow creates the window, ChangeWindowAttributes is called to cause the standard window event handler to be installed on the window.

The next block shows alternative window creation code for windows created programmatically using CreateNewWindow. Note that the standard window event handler will be installed because the kWindowStandardHandlerAttribute is included in the attributes passed in the second parameter of the CreateNewWindow call.

The call to InstallWindowEventHandler installs the application's window event handler on the window. Since more than one window can be opened, the call to InstallWindowEventHandler will be called whenever a new window is opened. Accordingly, to prevent a possible memory leak, the call to doGetHandlerUPP (to get a UPP to the application's window event handler) ensures that only one routine descriptor will be created regardless of how many windows are opened. (Recall from Chapter 5 that universal procedure pointer creation functions always allocate routine descriptors in memory on Mac OS 8/9, and sometimes allocate routine descriptors in memory on Mac OS X (depending on whether the application is compiled as a CFM binary or Mach-O binary.)

Note that registering the kEventProcessCommand event (class kEventClassCommand) is required in order to determine which button is clicked in a window-modal (sheet) alert.

doCloseWindow

Note that doCloseWindow does not dispose of the window. As previously stated, the application's window event handler allows kEventWindowClose events to be passed to the standard window event handler after doCloseWindow is called. The standard window handler disposes of the window.

doGetControls

After the controls have been created, SetControlCommandID is called to assign the command ID 'ctry' (kPopupCountryID) to the second (Country) pop-up menu button control. As previously stated, this will cause a kEventProcessCommand event to be dispatched to the application when a a mouse-down occurs in this pop-up menu button, allowing the application's application event handler to handle the event.

doIdle

doIdle is called when the event timer fires. IdleControls is called to cause the insertion point caret to blink in the edit text control. (This call is not necessary on Mac OS X because Mac OS X controls have built-in timers.)

doDrawContent

doDrawContent is the window update function, which is called when the kEventWindowDrawContent event type is received. As previously stated, there is no need to call BeginUpdate or EndUpdate in this function (though there would be, for Mac OS 8/9 only, if the kEventWindowUpdate event type had been registered rather than the kEventWindowDrawContent event type).

doControlHit1

doControlHit1 is called from the application's window event handler when the kEventMouseDown event type is received and a call to FindControlUnderMouse determines that there is a control under the mouse. Further processing of mouse-downs in the controls in this program is identical to that used in in the demonstration program Controls1 (Chapter 7).

Note the differences in this program's approach to detecting and handling mouse-downs in a pop-up menu button control, as compared with the approach used in the demonstration program CarbonEvents1.

doControlHit2

DoControlHit2 is called from the application's application event handler when the kEventProcessCommand event type is received and the commandID field of the HICommand structure contains the second (Country) pop-up menu button control's command ID. The control's value is determined, allowing the chosen menu item's text to be extracted and drawn in the window header.

Dialogs.c

doSheetAlert

The call to GetStandardAlertDefaultParams initialises a standard CFString alert parameter structure with default values. (The defaults are: no Help button; no Cancel button; no Other button.) The next two lines cause a Cancel and Other button to be included.

The next block get the message and informative strings to be passed in the call to CreateStandardSheet, which creates the sheet. The call to ShowSheetWindow displays the sheet.

Clicks in the sheet's buttons are handled in the parent window's handler (windowEventHandler).

doSheetDialog

The call to GetNewDialog creates the dialog, whose window definition ID is kWindowSheetProc (1088). GetDialogWindow gets a reference to the dialog's window object, allowing ChangeWindowAttributes to be called to cause the standard window event handler to be installed on the window.

The call to InstallWindowEventHandler installs the handler sheetEventHandler on the dialog. Note that the function doGetSheetHandlerUPP is called to get the universal procedure pointer to the handler passed in the second parameter of the InstallWindowEventHandler call.

SetDialogDefaultItem establishes the single push button item in the dialog as the default button. The next block sets some initial text in the dialog's edit text item and selects that text.

The call to ShowSheetWindow displays the sheet.

doGetSheetHandlerUPP

doGetSheetHandler serves the same purpose for window-modal (sheet) dialogs as does doGetHandlerUPP (see above) for document windows.

sheetEventHandler

sheetEventHandler is the event handler for window-modal (sheet) dialogs. It is a callback function.

If the kEventMouseDown event type is received, GetEventParameter is called to get the mouse location, which is then converted to the local coordinates required by FindControlUnderMouse. If the call to FindControlUnderMouse reveals that there is a control under the mouse, GetDialogFromWindow is called to get a reference to the dialog, allowing GetDialogItemAsControl to get a reference to the first item in the dialog's item list (the OK push button).

If the control clicked is the OK push button, the current text in the edit text item is extracted for display in the parent window's window header frame, HideSheetWindow is called to hide the sheet and DisposeDialog disposes of the sheet and releases all related memory.

if the mouse click was in the edit text item, eventNotHandledErr is returned so the event can be handled by the standard handler.

doMovableModalDialog

doMovableModalDialog creates a movable modal dialog containing three checkbox controls, a group box control and an OK push button control.

CreateNewWindow creates an initially invisible window of class kMovableModalWindowClass with the standard window handler installed. The RepositionWindow call ensures that the dialog will appear in the alert position on the main screen. SetThemeWindowBackground sets the dialog's background colour/pattern to the correct colour pattern for dialogs.

CreateRootControl creates a root control for the window, ensuring that activation/deactivation of the controls will be automatic.

The next three blocks create the dialog's controls. In the case of the OK push button and checkbox controls, a control ID is assigned to each control. The initial value of the checkbox controls is set to 0.

InstallWindowEventHandler installs the event handler dialogEventHandler on the window. With the dialog fully prepared, ShowWindow displays the dialog.

RunAppModalLoopForWindow is the Carbon event model equivalent of the Classic event model's ModalDialog. It will exit when QuitAppModalLoopForWindow is called in the dialog's event handler. Although it will block until the modal loop ends, your application's other handlers will still be called.

RunAppModalLoopForWindow attends to the menu deactivation usually associated with the display of a movable modal dialog.

dialogEventHandler

dialogEventHandler is the event handler for the movable modal dialog. It is a callback function.

If the kEventControlHit event type is received, GetEventParameter is called to get a reference to the control, allowing the call to GetControlID to get the control's ID.

If the control is the OK push button, the QuitAppModalLoopForWindow call terminates the modal loop and restores menu activation/deactivation status to that which obtained prior to the call to RunAppModalLoopForWindow.

If the control is one of the checkbox controls, the current value of the control is flipped and the new value is assigned to the relevant global variable which keeps track of the control values between successive invocations of the dialog.