4

INTRODUCTION TO WINDOWS

Demonstration Program: Windows1

Scope of This Chapter

On Mac OS 8/9, Mac OS 8.5 introduced a number of significant new features and associated system software functions to the Window Manager. This chapter limits itself to the Window Manager as it existed prior to Mac OS 8.5, as influenced by the Carbon API. The additional features introduced with Mac OS 8.5 and Carbon are addressed at Chapter 16.[1]

Window Basics

Windows, Documents, the Window Manager and Graphics Ports

A **window** is an area on the screen in which the user can enter or view information. Macintosh applications use windows for most communications and interactions with the user, including editing documents and presenting and acknowledging alerts and dialogs.

Users generally enter data in windows and your application typically lets the user save this data to a file. Your application typically creates **document windows** for this purpose. A document window is a view into the document. If the document is larger than the window, the window is a view of a portion of the document.

The Window Manager, which, amongst other things, provides functions for managing windows, itself depends on QuickDraw. QuickDraw supports drawing into **graphics ports**. Each window represents a graphics port.

Standard Window Elements

Note

In the following, Mac OS 8/9 terminology is used. **Size box** equates to **resize control**. **Zoom box** equates to **zoom button**. **Close box** equates to **close button. Collapse box** equates to **minimise button**.

The user can manipulate windows using a set of standard window elements supported by the Window Manager. These standard elements are as follows:

- ***Title Bar.*** The title bar is the bar at the top of a window that displays the window's name, contains the close, zoom, and collapse boxes, and indicates whether a window is active. The name of a newly created window with which no document is yet associated should be "**untitled**". The name of a window containing a saved document should be the document's filename.

[1] By definition, all new functions introduced with Mac OS 8.5 and later are automatically supported by Carbon.

- ***Close Box.*** The close box allows the user to close a window. The close box is sometimes called the **go-away box**.
- ***Zoom Box.*** The zoom box allows the user to choose between two different window sizes, one established by the user and one by the application. On Mac OS 8/9, the zoom box can be full, vertical, or horizontal.
- ***Collapse Box.*** The collapse box allows the user to collapse (minimise on Mac OS X) and uncollapse a window.
- ***Size Box.*** The size box allows the user change the size of a window.
- ***Draggable Area.*** That part of the window's "frame" less the close, zoom,collapse, and size boxes.

Scroll bars, which allow the user to view different parts of a document containing more information than can be displayed in the window at the one time, are not an integral part of a window and must be separately created and managed. By convention, scroll bars are placed on the right and lower edges of those windows that require them.

Active and Inactive Windows

The **active window** is the window in which the user is currently working. It is identified by its general appearance (see Fig 1). All keyboard activity targets the active window and only the active window interacts with the user.

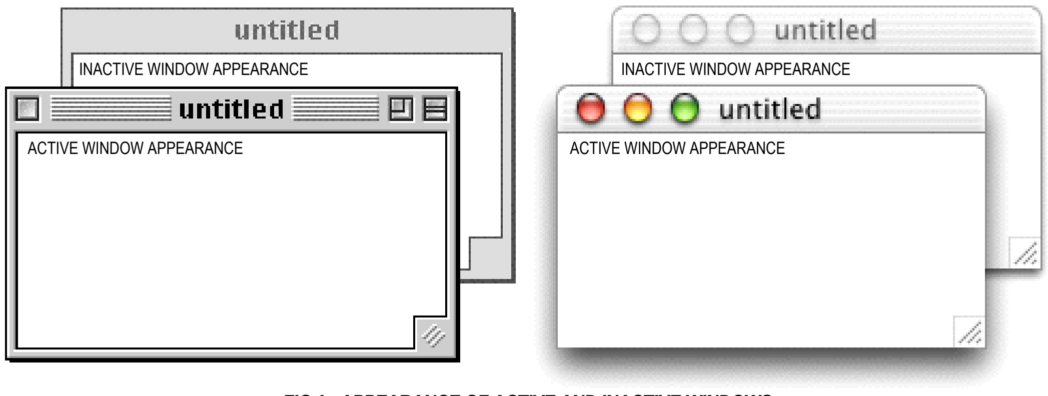


FIG 1 - APPEARANCE OF ACTIVE AND INACTIVE WINDOWS

When the user activates one of your application's windows, the Window Manager redraws the window's title bar and frame/shadow, the close box/button, the title text, the zoom box/button, the collapse box/minimise button, and the size box/resize control as shown at Fig 1. Your application must reinstate the appearance of the rest of the window to its state prior to the deactivation, activating any controls (scroll bars, etc.), drawing the scroll box (scroller, in Mac OS X terminology) in the same position, restoring the insertion point, and highlighting the previous selection, etc.

When a window belonging to your application becomes **inactive**, the Window Manager redraws the title bar and frame/shadow as shown at Fig 1. Your application must deactivate any controls, remove highlighting from selections, and so on.

When the user clicks in an inactive document window, your application should make the window active but should not make any selections in response to the click. To make a selection, the user should be required to click again. This behaviour protects the user from unintentionally losing an existing selection when activating the window.

Window Layering

On Mac OS 8/9, all of an application's document windows are in the one layer. On Mac OS X, document windows from different applications can be interleaved, and clicking on a window to bring it forward does not affect the layering of other windows. That said, if an application has a **Window** menu, choosing **Bring All To Front** will cause all of the application's document windows to be brought in front of all of the document windows of all other applications. Clicking on the application's icon in the Dock has the same effect.

Types of Windows

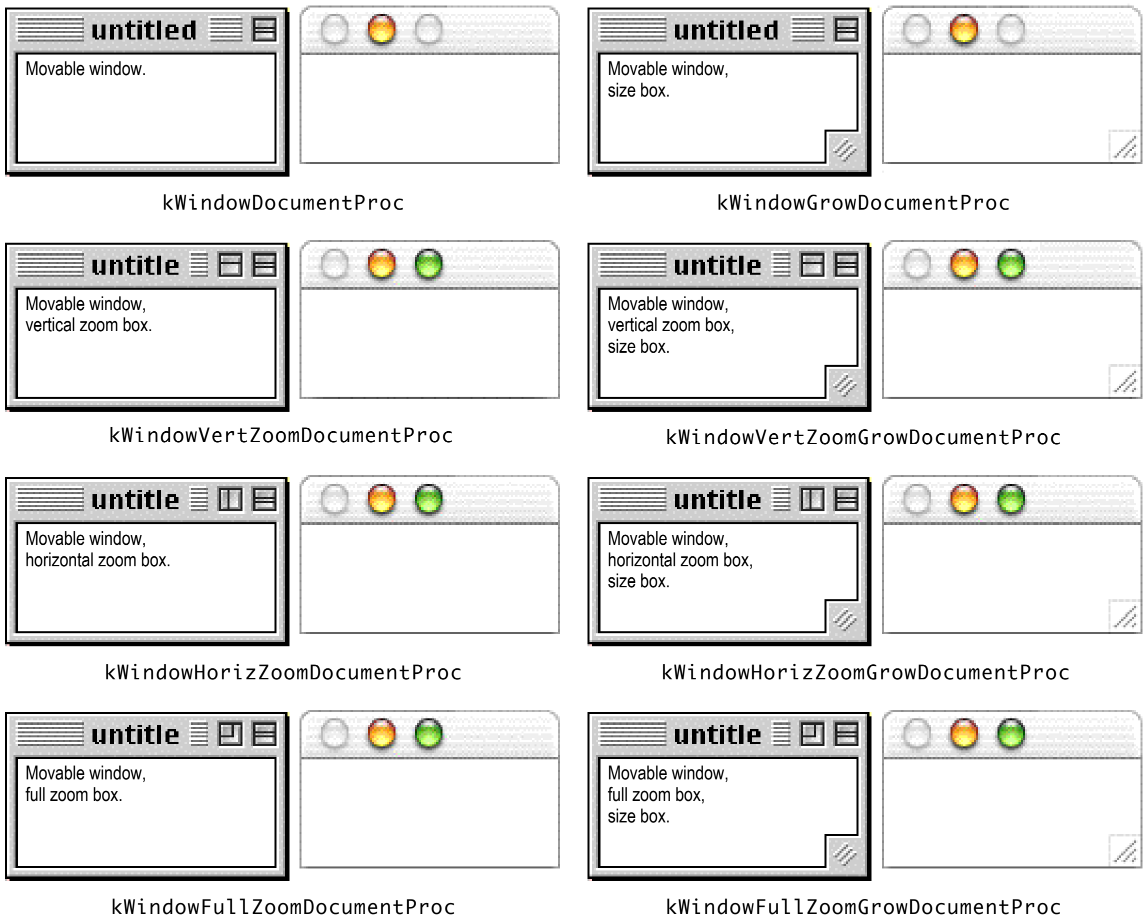
The Window Manager defines a large number of window **types**, which may be classified as follows:

- Document types.
- Dialog and alert types.
- Floating window types.

Window types are often referred to by the constant used in `'WIND'` resources, and by certain Window Manager functions, to specify the type of window required. That constant determines both the visual appearance of the window and its behaviour.

Document Types

Fig 2 shows the eight available window types for documents and the constants that represent those types.



untitled
Movable window.
kWindowDocumentProc
untitled
Movable window, size box.
kWindowGrowDocumentProc
untitle
Movable window, vertical zoom box.
kWindowVertZoomDocumentProc
untitle
Movable window, vertical zoom box, size box.
kWindowVertZoomGrowDocumentProc
untitle
Movable window, horizontal zoom box.
kWindowHorizZoomDocumentProc
untitle
Movable window, horizontal zoom box, size box.
kWindowHorizZoomGrowDocumentProc
untitle
Movable window, full zoom box.
kWindowFullZoomDocumentProc
untitle
Movable window, full zoom box, size box.
kWindowFullZoomGrowDocumentProc

FIG 2 - WINDOW TYPES FOR DOCUMENTS

Dialog and Alert Types

Fig 3 shows the seven available window types for modal and movable modal dialogs and alerts and the constants that represent those types. (The document window type represented by the constant `kWindowDocumentProc` is used for modeless dialogs.)

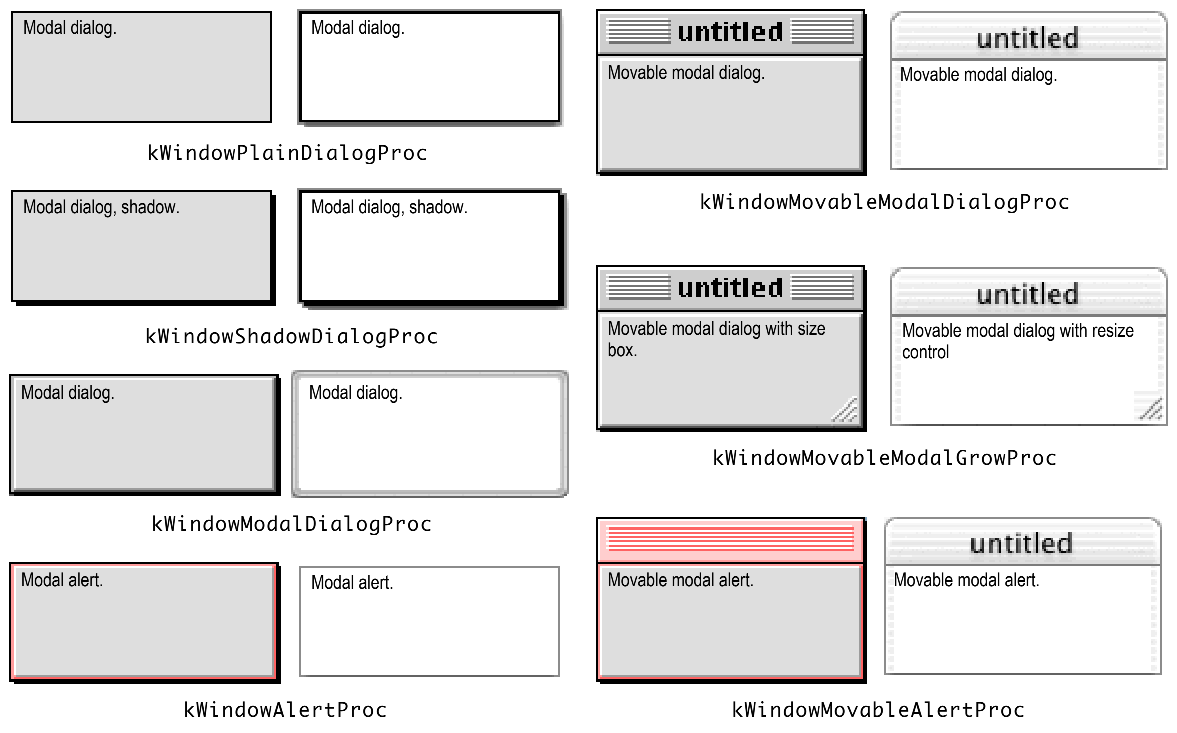


FIG 3 - WINDOW TYPES FOR DIALOGS AND ALERTS

Floating Window Types

Figs 4 and 5 show the sixteen available window types for floating windows and the constants that represent those types.

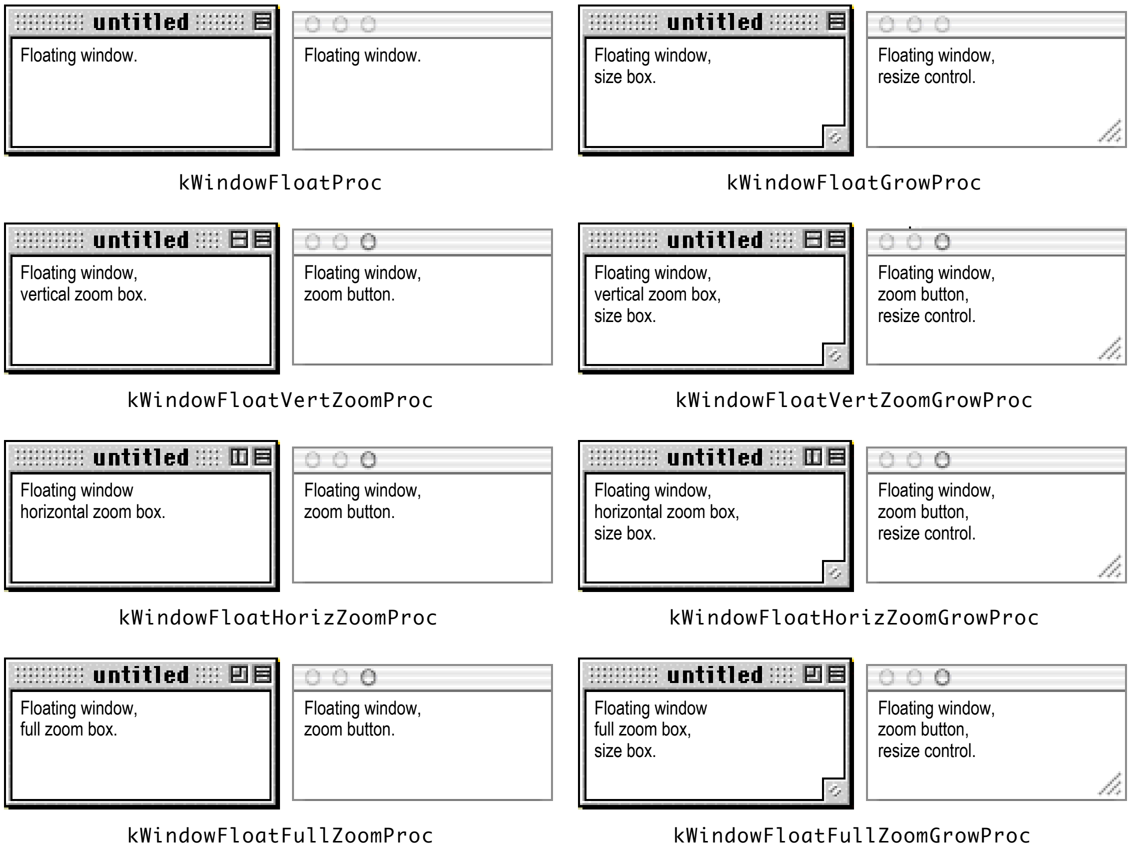

untitled
Floating window.
Floating window.
kWindowFloatProc
untitled
Floating window, size box.
Floating window, resize control.
kWindowFloatGrowProc
untitled
Floating window, vertical zoom box.
Floating window, zoom button.
kWindowFloatVertZoomProc
untitled
Floating window, vertical zoom box, size box.
Floating window, zoom button, resize control.
kWindowFloatVertZoomGrowProc
untitled
Floating window horizontal zoom box.
Floating window, zoom button.
kWindowFloatHorizZoomProc
untitled
Floating window, horizontal zoom box, size box.
Floating window, zoom button, resize control.
kWindowFloatHorizZoomGrowProc
untitled
Floating window, full zoom box.
Floating window, zoom button.
kWindowFloatFullZoomProc
untitled
Floating window full zoom box, size box.
Floating window, zoom button, resize control.
kWindowFloatFullZoomGrowProc

FIG 4 - WINDOW TYPES FOR FLOATING WINDOWS (TITLE BAR AT TOP)

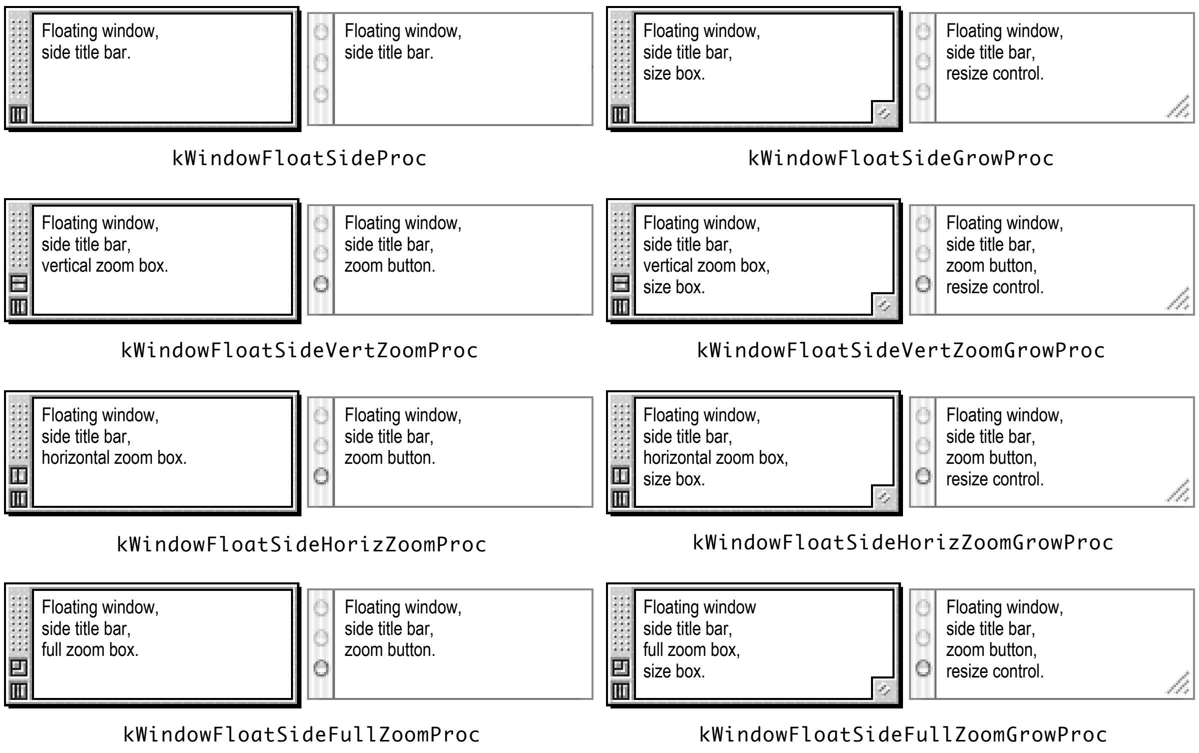

Floating window, side title bar.
Floating window, side title bar.
kWindowFloatSideProc
Floating window, side title bar, size box.
Floating window, side title bar, resize control.
kWindowFloatSideGrowProc
Floating window, side title bar, vertical zoom box.
Floating window, side title bar, zoom button.
kWindowFloatSideVertZoomProc
Floating window, side title bar, vertical zoom box, size box.
Floating window, side title bar, zoom button, resize control.
kWindowFloatSideVertZoomGrowProc
Floating window, side title bar, horizontal zoom box.
Floating window, side title bar, zoom button.
kWindowFloatSideHorizZoomProc
Floating window, side title bar, horizontal zoom box, size box.
Floating window, side title bar, zoom button, resize control.
kWindowFloatSideHorizZoomGrowProc
Floating window, side title bar, full zoom box.
Floating window, side title bar, zoom button.
kWindowFloatSideFullZoomProc
Floating window side title bar, full zoom box, size box.
Floating window, side title bar, zoom button, resize control.
kWindowFloatSideFullZoomGrowProc

FIG 5 - WINDOW TYPES FOR FLOATING WINDOWS (PSUEDO TITLE BAR AT SIDE)

Window Definition IDs

The constants shown at Figs 2, 3, 4, and 5 each represent a specific **window definition ID**. A window definition ID is a 16-bit value which contains the resource ID of the window's **window definition function** in the upper 12 bits and a **variation code** in the lower 4 bits:

- ***Window Definition Function.*** The system software and various Window Manager functions call a window's window definition function (WDEF) when they need to perform certain window-related actions, such as drawing or re-sizing a window's frame.
- ***Variation Code.*** A single WDEF can support up to 16 different window types. The WDEF defines a variation code, an integer from 0 to 15, for each window type it supports.

Four WDEFs (resource IDs 64, 65, 66, and 67) are associated with the three classifications of window types.

The window definition ID is derived by multiplying the resource ID of the WDEF by 16 and adding the variation code to the result, as is shown in the following:

WDEF Resource ID	*Variation Code*	*Window Definition ID (Value)*	*Window Definition ID (Constant)*
64	0	64 * 16 + 0 = 1024	kWindowDocumentProc
64	1	64 * 16 + 1 = 1025	kWindowGrowDocumentProc
64	2	64 * 16 + 2 = 1026	kWindowVertZoomDocumentProc
64	3	64 * 16 + 3 = 1027	kWindowVertZoomGrowDocumentProc
64	4	64 * 16 + 4 = 1028	kWindowHorizZoomDocumentProc
64	5	64 * 16 + 5 = 1029	kWindowHorizZoomGrowDocumentProc
64	6	64 * 16 + 6 = 1030	kWindowFullZoomDocumentProc
64	7	64 * 16 + 7 = 1031	kWindowFullZoomGrowDocumentProc
65	0	65 * 16 + 0 = 1040	kWindowPlainDialogProc
65	1	65 * 16 + 1 = 1041	kWindowShadowDialogProc
65	2	65 * 16 + 2 = 1042	kWindowModalDialogProc
65	3	65 * 16 + 3 = 1043	kWindowMovableModalDialogProc
65	4	65 * 16 + 4 = 1044	kWindowAlertProc
65	5	65 * 16 + 5 = 1045	kWindowMovableAlertProc
65	6	65 * 16 + 6 = 1046	kWindowMovableModalGrowProc
66	1	66 * 16 + 1 = 1057	kWindowFloatProc
66	3	66 * 16 + 3 = 1059	kWindowFloatGrowProc
66	5	66 * 16 + 5 = 1061	kWindowFloatVertZoomProc
66	7	66 * 16 + 7 = 1063	kWindowFloatVertZoomGrowProc
66	9	66 * 16 + 9 = 1065	kWindowFloatHorizZoomProc
66	11	66 * 16 + 11 = 1067	kWindowFloatHorizZoomGrowProc
66	13	66 * 16 + 13 = 1069	kWindowFloatFullZoomProc
66	15	66 * 16 + 15 = 1071	kWindowFloatFullZoomGrowProc
67	1	67 * 16 + 1 = 1073	kWindowFloatSideProc
67	3	67 * 16 + 3 = 1075	kWindowFloatSideGrowProc
67	5	67 * 16 + 5 = 1077	kWindowFloatSideVertZoomProc
67	7	67 * 16 + 7 = 1079	kWindowFloatSideVertZoomGrowProc
67	9	67 * 16 + 9 = 1081	kWindowFloatSideHorizZoomProc
67	11	67 * 16 + 11 = 1083	kWindowFloatSideHorizZoomGrowProc
67	13	67 * 16 + 13 = 1085	kWindowFloatSideFullZoomProc
67	15	67 * 16 + 15 = 1087	kWindowFloatSideFullZoomGrowProc

Window Type Usage

For Documents

A `kWindowFullZoomGrowDocumentProc` window is normally used for document windows because it supports all window manipulation elements. Note that, because you can optionally suppress the close box/button when you create the window, the Window Manager does not necessarily draw/highlight that particular element. Also note that, when the related document contains more data that will fit in the window, you must add scroll bars.

For Modal Alerts and Modal Dialogs

Modal alerts and modal dialogs are simply special-purpose windows. Modal alerts generally use the window type `kWindowAlertProc` and modal dialogs generally use window type `kWindowModalDialogProc`.[2]

For Movable Modal Alerts and Movable Modal Dialogs

Movable modal alerts and movable modal dialogs are used when you want the user to be able to move the alert or dialog window or to bring another application to the foreground before the dialog is dismissed. Movable modal alerts use the window type `kWindowMovableAlertProc` and movable modal dialogs use the window type `kWindowMovableModalDialogProc`.

For Modeless Dialogs

Modeless dialogs allow the user to perform other tasks within the application without first dismissing the dialog. User interface guidelines require that the `kWindowDocumentProc` window type, which can be moved or closed but not resized or zoomed, be used for modeless dialogs.

Window Regions

The Window Manager recognises the special-purpose **regions**[3] shown at Figs 6 and 7.

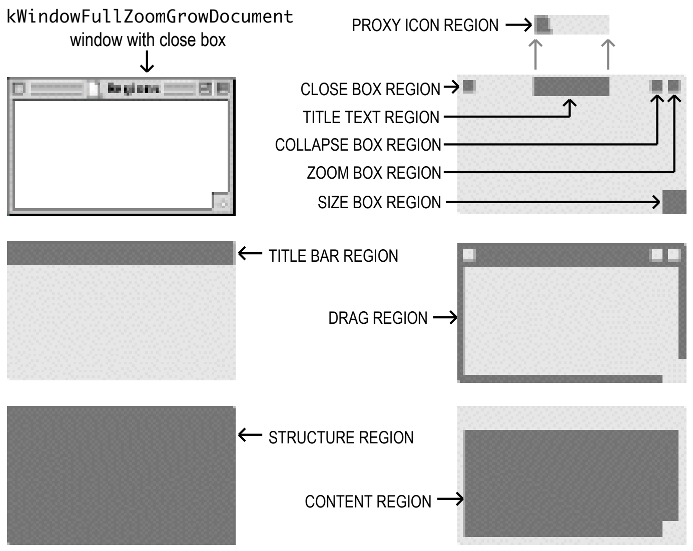


FIG 6 - WINDOW REGIONS - MAC OS 8/9

[2] The creation and handling of alerts and dialogs is addressed in detail at Chapter 8.

[3] A region is an arbitrary area, or set of areas, on the QuickDraw coordinate plane. The outline of a region is one or more closed loops. Regions are explained in more detail at Chapter 12.

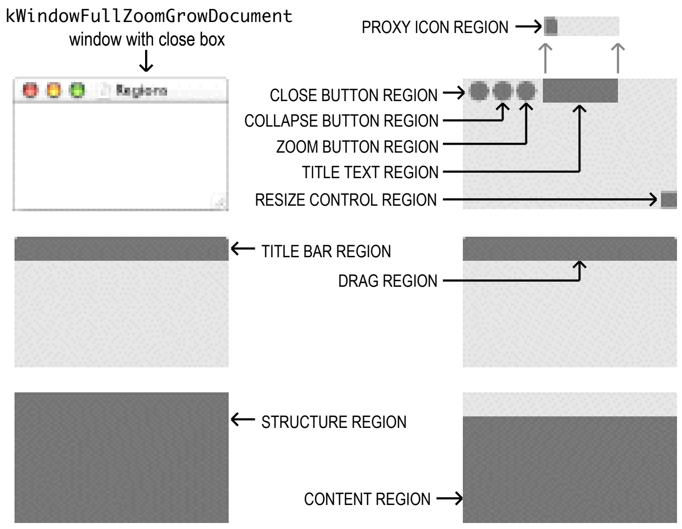


FIG 7 - WINDOW REGIONS - MAC OS X

Handles to these and two other window-related regions, which are represented by constants of type `RegionWindowCode`, may be obtained via a call to `GetWindowRegion`. The definitions of these regions, and the constants which represent them, are as follows:

Region	*Constant*	*Definition*
Title bar region	`kWindowTitleBarRgn`	The entire area occupied by a window's title bar, including the title text region.
Title text region	`kWindowTitleTextRgn`	That portion of a window's title bar that is occupied by the name of the window and the window proxy icon. . (See Chapter 16.)
Close box/button region	`kWindowCloseBoxRgn`	The area occupied by a window's close box/button.
Zoom box/button region	`kWindowZoomBoxRgn`	The area occupied by a window's zoom box/button.
Drag region	`kWindowDragRgn`	On Mac OS X, this equates to the title bar region. On Mac OS 8/9, this includes the window frame, including the title bar and window outline, but excluding the close box, zoom box, collapse box, and size box (if any).
Size box/resize control region	`kWindowGrowRgn`	The area occupied by a window's size box/resize control.
Collapse box/minimise button region	`kWindowCollapseBoxRgn`	The area occupied by a window's collapse box/minimise button.
Proxy icon region	`kWindowTitleProxyIconRgn`	The area occupied by the window proxy icon. (See Chapter 16.)
Structure region	`kWindowStructureRgn`	The entire area occupied by a window, including the frame and content region. (The window may be partially off-screen but its structure region does not change.)
Content region	`kWindowContentRgn`	That part of a window in which the contents of a document and the window's controls (including scroll bars) are displayed.
Update region	`kWindowUpdateRgn`	Contains all areas of a window's content region that need updating (re-drawing).
Global port region	`kWindowGlobalPortRgn`	The bounds of the window's graphics port, in global coordinates, even when the window is collapsed.

The Desktop (Gray) Region

Another region of some relevance to the Window Manager is the **desktop region** (sometimes known as the **gray region**). The desktop region is the region below the menu bar, including all screen real estate in a system equipped with multiple monitors. The Window Manager maintains a pointer to the desktop region in a low-memory global variable named `GrayRgn`. You can get a handle to the desktop region using the function `GetGrayRgn`.

Controls and Control Lists

Windows may contain **controls**. The most common control in a window is the **scroll bar** (see Fig 8), which should be included in the window when there is more data than can be shown at one time in the space available. The Control Manager is used to create, display and manipulate scroll bars.

All controls "belonging" to an individual window and are displayed within the graphics port that represents that window. Entries pointing to the descriptions of a window's controls are stored in a **control list** maintained for that window by the Window Manager.

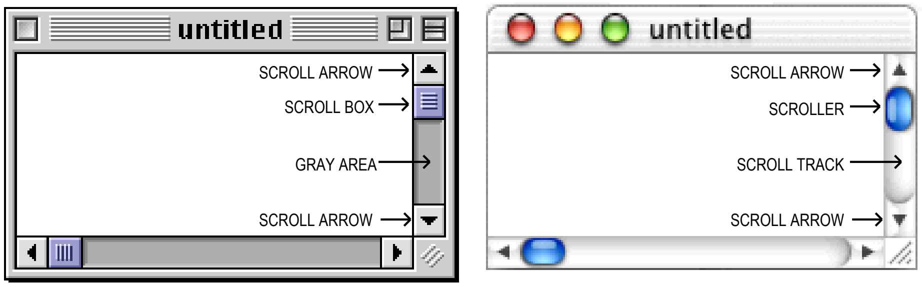


FIG 8 - SCROLL BARS

The Window List

At any one time, many windows, from many applications, may be displayed on the desktop. To track all of the windows on the desktop, the Window Manager maintains a private data structure called the **window list**. The arrangement of the entries in this list reflects the current front-to-back ordering of the windows on the desktop, the frontmost (active) window being the first in the list.

The function `GetWindowList` returns a reference to the window object (see below) for the first window in the window list. The function `GetNextWindow` returns a reference to the window object for the next window in the window list.

The Graphics Port and the Window Object

The Graphics Port

Each window represents a QuickDraw graphics port, an opaque object of type `CGrafPort` which describes a drawing environment with its own coordinate system. The Window Manager creates a graphics port when it creates the window.[4]

The location of a window on the screen is defined in QuickDraw's **global coordinates**. QuickDraw's global coordinates originate at the top left corner of the main screen and extend vertically and horizontally within the range -32768 to 32,767. The positive x-axis extends rightward and the positive y-axis extends downward (see Fig 9).

[4] The graphics port is addressed in detail at Chapter 11.

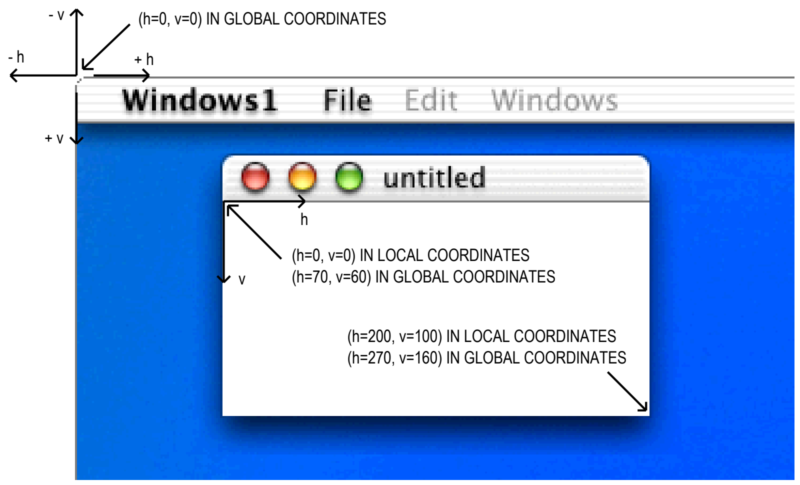

- v
(h=0, v=0) IN GLOBAL COORDINATES
- h
+ h
+ v
Windows1 File Edit Windows
untitled
h
v
(h=0, v=0) IN LOCAL COORDINATES
(h=70, v=60) IN GLOBAL COORDINATES
(h=200, v=100) IN LOCAL COORDINATES
(h=270, v=160) IN GLOBAL COORDINATES

FIG 9 - A WINDOW'S LOCAL AND GLOBAL COORDINATE SYSTEMS

The graphics port object stores a rectangle called the **port rectangle**. In a graphics port representing a window, the port rectangle represents the window's content region. Within the port rectangle, the drawing area is described in **local coordinates**. Fig 9 illustrates the local and global coordinate systems for a window which is 100 pixels high by 200 pixels wide, and which is placed with its content region 70 pixels down and 60 pixels to the right of the upper left corner of the screen.

When the Window Manager creates a window, it places the origin of the local coordinate system at the upper-left corner of the window's port rectangle. Note, however, that the Event Manager describes mouse events in global coordinates, and that you must do most of your window manipulation in global coordinates.

The Window Object

The Window Manager stores information about individual windows in opaque data structures called **window objects**. The data type `WindowPtr` is defined as a pointer to a window object:

```
typedef struct OpaqueWindowPtr* WindowPtr;
```

Note that the data type `WindowPtr` is equivalent to the newer data type `WindowRef`:

```
typedef WindowPtr WindowRef;
```

Carbon Note

One of the fundamental differences between the Classic API and the Carbon API is that, in the Classic API, the data type `WindowPtr` is defined as a pointer to a graphics port structure, not to a window structure. The first field in the Classic API's window structure is a graphics port structure, meaning that the graphics port structure has the same address as the window structure in which it resides. In the Classic API, a `WindowPtr` must be cast to a `WindowPeek` (which is defined as a pointer to a window structure) in order for the fields of the window structure to be directly accessed.

Accessor Functions

Accessor functions are provided to access the information in window objects. The accessor functions are as follows:

Function	**Description**
`GetWindowPort`	Gets a pointer to the specified window's graphics port object.
`GetWindowKind`	Gets the window kind (dialog or application) of the specified window.
`SetWindowKind`	Sets the window kind (dialog or application) of the specified window.
`IsWindowVisible`	Determines whether the specified window is visible.

`ShowWindow`	Makes the specified window visible.
`HideWindow`	Makes the specified window invisible.
`ShowHide`	Makes the specified window visible or invisible without affecting front-to-back ordering.
`IsWindowHilited`	Determines whether the specifed window is highlighted.
`HiliteWindow`	Highlights or unhighlights a window.
`GetWindowRegion`	Gets a handle to the specified window's structure, content, and update regions. **Note:** Handles to the other regions shown at Figs 6 and 7 are not included in the window object. The WDEF determines the location of those particular regions.
`GetWTitle`	Gets the specified window's title.
`SetWTitle`	Sets the specified window's title.
`GetWindowPic`	Gets a handle to the picture stored in the window object by `SetWindowPic`.
`SetWindowPic`	Stores a handle to a picture in the window object, causing the Window Manager to draw the picture instead of generating an update event.
`GetWRefCon`	Gets the reference constant stored in the specified window's window object by `SetWRefCon`.
`SetWRefCon`	Sets a reference constant in the specified window's window object.
`GetWindowStandardState`	Gets the window's standard state (see below) rectangle.
`SetWindowStandardState`	Sets the window's standard state (see below) rectangle.
`GetWindowUserState`	Gets the window's user state (see below) rectangle.
`SetWindowUserState`	Sets the window's user state (see below) rectangle.

Events in Windows

As stated at Chapter 2, the Window Manager itself generates two types of events central to window management, namely, activate events and update events.

One of the Window Manager's main tasks is to report the location of the cursor when the application receives a mouse-down event. As was also stated at Chapter 2, the Window Manager function `FindWindow` may be used to determine whether the cursor is in a window when the mouse-down occurs and, if it is in a window, in exactly which window and which part of that window. `FindWindow` is thus the function which enables you to distinguish between those mouse-down events that affect the window itself and those that affect the document displayed in the window.

Creating Your Application's Windows

You typically create document and floating windows from resources of type `'WIND'`, although you can create them programmatically using the function `NewCWindow`. Additional methods of creating windows, including floating windows, programmatically are described at Chapter 16.

'WIND' Resources

When creating resources with Resorcerer, it is advisable that you refer to a diagram and description of the structure of the resource and relate that to the various items in the Resorcerer editing windows. Accordingly, the following describes the structure of the resource associated with the creation of document and floating windows.

Structure of a Compiled 'WIND' Resource

Fig 10 shows the structure of a compiled `'WIND'` resource and how it "feeds" the window object.

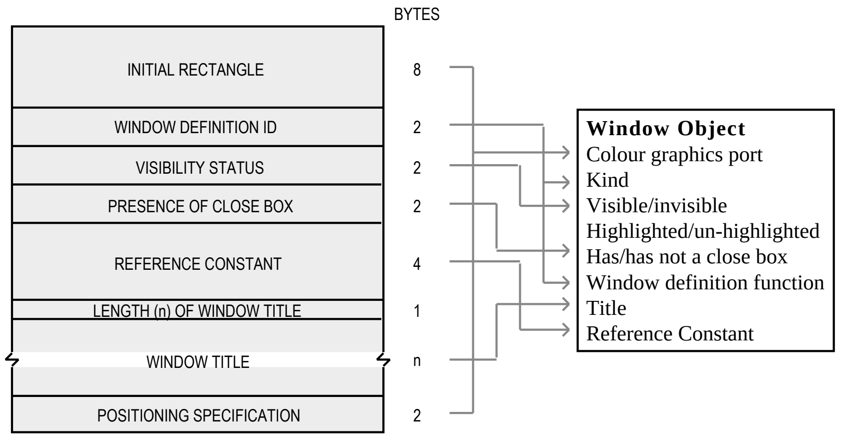


FIG 10 - STRUCTURE OF A COMPILED WINDOW ('WIND') RESOURCE

The following describes the main fields of the `'WIND'` resource:

Field	***Description***
INITIAL RECTANGLE	A rectangle that defines the initial size and placement, in global coordinates, of the window's content region. This rectangle can be changed before displaying the window, either programmatically or by using an optional **positioning specification** (see below).
WINDOW DEFINITION ID	The window's definition ID.
VISIBILITY STATUS	Specifies whether the window is to be visible or invisible. Note that this really means whether the Window Manager displays the window; it does not necessarily mean that the window can be seen on the screen. (A visible window might be completely covered by other windows; nevertheless its visibility status is still "visible".)
PRESENCE OF CLOSE BOX/BUTTON	Specifies whether the window is to have a close box/button.
REFERENCE CONSTANT	A reference constant which your application can use for whatever data it needs to associate with the window. When it builds a new window object, the Window Manager stores in that object whatever value you specify in this field. (You can also store a reference constant in the window object programmatically via a call to `SetWRefCon`.)
WINDOW TITLE	A string that specifies the window's title.
POSITIONING SPECIFICATION	An positioning specification (optional). If this field contains a positioning specification, it overrides the window position established by the rectangle in the first field. The positioning constants (see below) which may be assigned to this field are very useful for specifying the initial position of dialogs, alerts, and windows for new documents. However, the position (and size) of a new window intended to display a previously saved document should be the same as the window position (and size) when the document was last displayed.

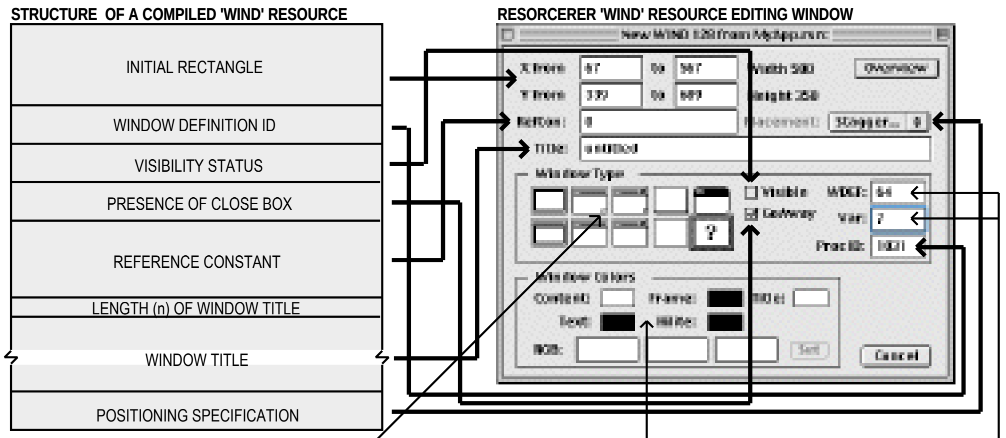
Positioning Specification

The constants for the positioning specification field are as follows:

Constant	***Value***	***Meaning***
`kWindowNoPosition`	`0x0000`	(Use initial rectangle.)
`kWindowDefaultPosition`	`0x0000`	(Use initial rectangle.)
`kWindowCenterMainScreen`	`0x280A`	Centre on main screen.
`kWindowAlertPositionMainScreen`	`0x300A`	Place in alert position on main screen.
`kWindowStaggerMainScreen`	`0x380A`	Stagger on main screen.
`kWindowCenterParentWindow`	`0xA80A`	Center on parent window.
`kWindowAlertPositionParentWindow`	`0xB00A`	Place in alert position on parent window.
`kWindowStaggerParentWindow`	`0xB80A`	Stagger relative to parent window.
`kWindowCenterParentWindowScreen`	`0x680A`	Center on parent window screen.
`kWindowAlertPositionParentWindowScreen`	`0x700A`	Alert position on parent window screen.
`kWindowStaggerParentWindowScreen`	`0x780A`	Stagger on parent window screen.

Creating a 'WIND' Resource Using Resorcerer

Fig 11 shows a `'WIND'` resource being created with Resorcerer.


STRUCTURE OF A COMPILED 'WIND' RESOURCE
INITIAL RECTANGLE
WINDOW DEFINITION ID
VISIBILITY STATUS
PRESENCE OF CLOSE BOX
REFERENCE CONSTANT
LENGTH (n) OF WINDOW TITLE
WINDOW TITLE
POSITIONING SPECIFICATION
RESORCERER 'WIND' RESOURCE EDITING WINDOW

These clickable icons pertain to the old superseded window types. Disregard these icons and simply enter the required window definition ID at the **ProcID:** item.

Creates 'wctb' resources. Not relevant in the Carbon era. Disregard.

The resource ID of the window definition function and the variation code appear automatically when the window definition ID is entered at the **ProcID:** item.

FIG 11 - CREATING A 'WIND' RESOURCE USING RESORCERER

Creating the Window From the 'WIND' Resource

`GetNewCWindow` is used to create a window from a `'WIND'` resource.

Adding Scroll Bars

If a window requires scroll bars, you typically create them from `'CNTL'` resources at the time that you create the document window, and then display them when you make the window visible. (See Chapter 7.)

Window Visibility

If the `'WIND'` resource specifies that the new window is visible, `GetNewCWindow` displays the window immediately. If you are creating a document window, however, it is best to create the window in an invisible state and then make it visible when you are ready to display it. The right time to display a window depends on whether the window is associated with a new or a saved document:

- If you are creating a window because the user is creating a new document, you should display the window immediately (by calling `ShowWindow`).
- If you are creating a new window to display a saved document, you should retrieve the document and draw it in the window before calling `ShowWindow`.

Positioning a New Document Window on the DeskTop

The positioning constants previously described allow you to position new windows automatically. When used, those positioning constants concerned with staggering new window placement will ensure that the Window Manager will use any vacated position for the next new window.

Getting Available Window Positioning Bounds

Carbon introduced the function `GetAvailableWindowPositioningBounds`, which allows your application to determine the available window positioning bounds on a given screen. The function returns the bounds of the screen rectangle less the menu bar and, on Mac OS X, the Dock.

Positioning a Saved Document Window on the DeskTop

For windows created for the purpose of displaying a saved document, you should replicate the size and location of the document's window as it was when the document was last saved. When the user saves a document, you must therefore also save the **user state** rectangle and the current **zoom state** of the window (that is, whether the window is in the user state or the **standard state**).

User State, Standard State, and Zoom State

Some explanation of user state and standard state is necessary. The user state is the last size and position the user, through sizing and dragging actions, established for a window. The standard state is the size and position that your application defines as being best for the display of the data contained in the window.

The user and standard states are stored in the window object and may be set and retrieved using the functions `GetWindowStandardState`, `SetWindowStandardState`, `GetWindowUserState`, and `SetWindowUserState` (see Accessor Functions, above). In addition, the function `IsWindowInStandardState` allows your application to determine the current zoom state, i.e., whether the window is zoomed "out" to the standard state or zoomed "in" to the user state.

Saving the Window State

Returning to the matter of saving the user state and the current zoom state of the window, for windows with zoom boxes/buttons you typically store this data as a custom resource in the resource fork of the document file.

Drawing a Window's Contents

Your application is responsible for drawing a window's contents. It typically uses the Control Manager to draw the window's controls and then draws the user data itself.

Managing Multiple Windows and Associated Data

Your application is likely to have multiple windows open on the desktop at once, each of which will have some form of data, such as text, associated with it. This data, which is external to the window object, may be regarded as the "property" of an individual window.

As previously stated, you can store a reference constant in a window object using the function `SetWRefCon`. Typically, your application will use `SetWRefCon` to store a handle to a structure containing the window's external data. This structure, usually referred to as a **document structure**, might hold a handle to the text being edited, handles to the scroll bars, a file reference number and a file system specification for the document's file, plus a flag indicating whether data has changed since the last save, as shown in the following example:

```
typedef struct
{
  TEHandle      editRec;
  ControlHandle vScrollBar;
  Controlhandle hScrollbar;
  short         fileRefNum;
  FSSpec        fileFSSpec;
  boolean       windowTouched;
} docStructure;
typedef docStructure **docStructureHdl;
```

Your application can retrieve the reference constant using the function `GetWRefCon`.

Handling Events

Handling Mouse Events

Your application, when it is the active application, receives all mouse-down events in its menu bar and its windows. When it receives a mouse-down event, your application should call `FindWindow` to ascertain which window, and which part of that window, the mouse-down occurred in. (In this context, the menu bar is considered to be a window part). The application should then take the appropriate action based on which window, and which part of that window, the mouse-down occurred in.

Mouse-Downs in Inactive Windows

When you receive a mouse-down event in an inactive document window or modeless dialog, and if the active window is a document window or a modeless dialog, you should call `SelectWindow`, passing it the window reference. `SelectWindow` re-layers the windows as necessary, removes highlighting from the previously active window, brings the newly-activated window to the front, highlights it and generates the activate and update events necessary to tell all affected applications which windows must be redrawn.

Note that, if the active window is a modal or movable modal alert or dialog, no action is required by your application. Modal and movable modal alerts and dialogs are handled by the `ModalDialog` function, which does not pass the event to your application.

Handling Keyboard Events

Whenever your application is the active application, it receives all key-down events (except, of course, for the system-defined Command-Shift-number key sequences).

When you receive a key-down event, you should first check whether the user is holding down a modifier key and another key at the same time. Your application should respond to key-down events by inserting data into the document, changing the display or taking other appropriate actions. Typically, your application provides feedback for standard keystrokes by drawing the character on the screen.

Handling Update Events

Handling Update Events — Mac OS 8/9

On Mac OS 8/9, the Window Manager maintains an update region for each window. The update region represents the parts of a window's content region that have been affected by changes to the desktop and need redrawing. The Event Manager continually scans the window objects of all the windows on the desktop, looking for non-empty update regions. If it finds an update region that is not empty, it generates an update event for that window.

When your application receives an update event, it should redraw the content area. When your application redraws the content area, the Window Manager clips all screen drawing to the **visible region** of the window's graphics port. The visible region is that part of a graphics port that is not covered by other windows, that is, the part that is actually visible on screen. The Window Manager stores a handle to the visible region in the graphics port object.

Before redrawing the content area, your application should call `BeginUpdate` and, when it has completed the drawing, it should call `EndUpdate`. As shown at Fig 12, `BeginUpdate` temporarily adjusts the visible region to equate to the intersection of the visible region and the update region. Because QuickDraw limits its drawing to this temporarily modified visible region, only those parts of the window which actually need updating are drawn. `BeginUpdate` also clears the update region, thus ensuring that the Event Manager does not continue sending an endless stream of update events.

When the drawing is completed, and as shown at Fig 12, `EndUpdate` restores the visible region of the graphics port to the full visible region.

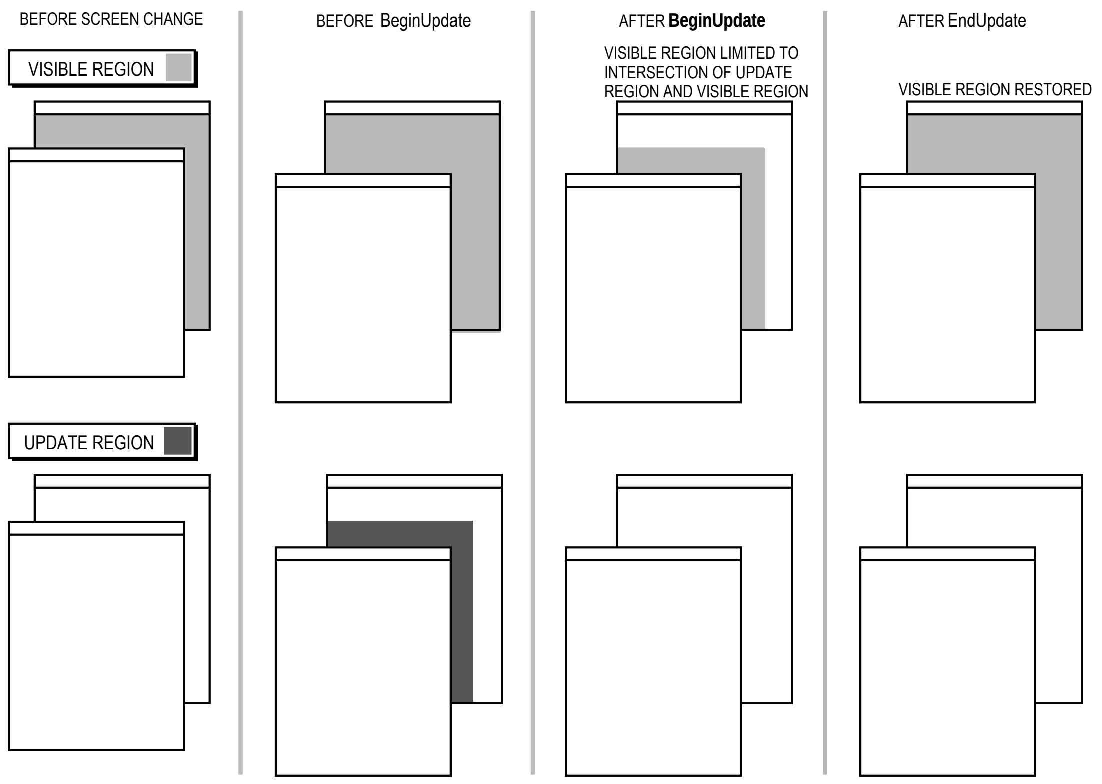


FIG 12 - EFFECTS OF BeginUpdate AND EndUpdate ON VISIBLE AND UPDATE REGIONS

The reason for these update region/visible region machinations is that the handle to the update region is stored in the window object while the handle to the visible region is stored in the graphics port object. QuickDraw knows the graphics port object intimately, but knows nothing about the window object or its contents. QuickDraw needs something it can work with, hence the above process whereby the visible region is temporarily made the equivalent of the update region while QuickDraw does its drawing.

Manipulating the Update Region

Your application can force or suppress update events by manipulating the update region. You can call `InValWindowRect` or `InvalWindowRgn` to add a rectangle or region to the update region, thus causing an update event to be generated and, as a consequence, that area to be redrawn. You can also remove a rectangle or region from the update region by calling `ValidWindowRect` or `ValidWindowRgn` so as to decrease the time spent redrawing. For example, an unaffected text area could be removed from the update region of a window that is being resized.

Handling Update Events — Mac OS X

On Mac OS X, windows are double-buffered, meaning that your application does not draw into the window's graphics port itself but rather into a separate buffer. The Window Manager flushes the buffer to the window's graphics port when your application calls `WaitNextEvent`. On Mac OS X, your application does not require update events to cater for the situation where part, or all, of a window's content region has just been exposed as a result of the user moving an overlaying window.

The receipt of an update event on Mac OS X simply means that your application should draw the required contents of the window. The swapping of visible and update regions required on Mac OS 8/9 is not required, so calls to `BeginUpdate` and `EndUpdate` are irrelevant (and ignored) on Mac OS X.

As is the case on Mac OS 8/9, your application can force the generation of an update event by calling `InValWindowRect` or `InvalWindowRgn`. (Your application can also force the buffer to be flushed to the window's graphics port by calling the QuickDraw function `QDFlushPortBuffer`. This is required, for example, when your application is drawing periodically in a loop that does not call `WaitNextEvent`.)

Type-Dependent Update Functions

An update function should typically first determine whether the type of window being updated is a document window or some other type of window. If the window is a document window, a document window updating function should be called. If the window is a modeless dialog, an updating function for that modeless dialog should be called.

Handling Activate Events

Activate events are generated by the Window Manager to inform your application that a window is becoming active or is about to be made inactive. Each activate event specifies the window to be changed and whether the window is to be activated or deactivated.

Your application typically triggers activate events itself by calling `SelectWindow` following a mouse-down event in a non-active window. `SelectWindow` brings the window in which the mouse-down occurred to the front, adds highlighting to that window, and removes highlighting from the previously active window. `SelectWindow` then generates one activate event to tell your application to perform its part of the deactivation of the previously active window, followed by and another activate event to tell your application to perform its part of the activation of the newly activated window.

When your application receives the event for the window about to be made inactive, it should hide the window's controls and remove any highlighting of selections. When your application receives the event for the newly activated window, it should draw the controls and restore the content area as necessary. This latter might involve, for example, adding the insertion point at its former location or highlighting the previously highlighted selection.

The function for handling activate events should typically first determine whether the window being activated/deactivated is a document window or a modeless dialog. It should then perform the appropriate activation/deactivation actions. The function does not need to check for modal alerts or modal dialogs because the Dialog Manager's `ModalDialog` function automatically handles activate events for those windows.

Manipulating Windows

Dragging a Window

When a mouse-down event occurs in the title bar, your application should call `DragWindow`, which tracks the user's actions until the mouse button is released. `DragWindow` moves an image of the window on the screen as the user moves the mouse. When the user releases the mouse, `DragWindow` redraws the window in its new location. The window is then activated (unless the Command key was pressed during the drag).

The area within which the window may be dragged is specified by a `Rect` variable passed in `DragWindow`'s `boundsRect` parameter.

Carbon Note

In Carbon, assigning `NULL` to the `boundsRect` parameter has the effect of setting the parameter to the bounding rectangle of the desktop region.

Zooming a Window

The zoom box/button allows the user to alternate between two window sizes and positions known as the user state and the standard state. To amplify the previous description of user state and standard state:

- The user state is the window size and location established by the user. (If the user has not yet moved or resized the window, the user state is the size and location of the window when it was created.)

- The standard state is the size and position that your application specifies as being the optimum for the display of the data contained in the window. In a word-processing program, for example, a window in the standard state might show the full width of a page and as much length as will fit on the screen. If the user changes the page size using the Page Setup dialog, the application might adjust the standard state width to reflect the new page width.
- The user and standard states are stored in the window object. The Window Manager sets the initial standard and user states when it fills in the window object on creation, and it updates the user state whenever the user resizes the window.

Mac OS 8.5 introduced new functions for implementing window zooming. Prior to Mac OS 8.5, when `FindWindow` returned either `inZoomIn` or `inZoomOut`, your application would call `TrackBox` to handle highlighting of the zoom box/button and to determine whether the cursor was inside or outside the zoom box/button when the button was released. If `TrackBox` returned `true`, your application would call `ZoomWindow` to resize the window.

The advantage of the `ZoomWindowIdeal` function introduced with Mac OS 8.5 is that, unlike `ZoomWindow`, it zooms the window in accordance with the following Apple human interface guidelines relating to a window's standard state:

- "A window should move as little as possible when zooming between the user state and standard state, to avoid distracting the user."
- "A window in its standard state should be positioned so that it is entirely on one screen."
- "If a window straddles more than one screen in the user state, when it is zoomed to the standard state it should be zoomed to the screen that contains the largest portion of the window's content region."
- "If the ideal size for the standard state is larger than the destination screen, the dimensions of the standard state should be that of the destination screen, minus a few pixels of boundary. If the destination screen is the main screen, space should also be left for the menu bar."
- "When a window is zoomed from the user state to the standard state, the top left corner of the window should remain anchored in place; however, if the standard state of the window cannot fit on the screen with the top left corner anchored, the window should be "nudged" so that the parts of the window in the standard state that would fall offscreen are, instead, just onscreen."

Other advantages of `ZoomWindowIdeal` are that:

- It allows you to specify, in its `ioIdealSize` parameter, the desired height and width of the window's content region in the standard state.
- On Mac OS X, it takes account of the current height of the Dock when zooming to the standard state and constrains the zoom accordingly.

When `ZoomWindowIdeal` is used, another function introduced with Mac OS 8.5 (`IsWindowInStandardState`) must be used to determine the appropriate part code (`inZoomIn` or `InZoomOut`) to pass in `ZoomWindowIdeal`'s `partCode` parameter.

Vertical or Horizontal Zoom Boxes — Mac OS 8/9

Your application should ensure that, when a vertical zoom box is clicked, only the vertical size of the associated window changes. Similarly, when a horizontal zoom box is clicked, your application should ensure that only the horizontal size of the associated window changes.

Re-Sizing a Window

Mac OS 8.5 introduced a new function for implementing window re-sizing. Prior to Mac OS 8.5, when the user pressed the mouse button in the size box, your application would call `GrowWindow`. This function displayed a **grow image**, a dotted outline of the window frame and scroll bar area which expanded and contracted as the user dragged the size box. When the user released the mouse button, `GrowWindow` returned a long integer which described the window's new height (in the high-order word) and width (in the loworder word). A value of zero indicated that the window size did not change. When the mouse-button was released and `GrowWindow` returned a non-zero value, your application called `SizeWindow` to draw the window in its new size.

The function introduced with Mac OS 8.5 is `ResizeWindow`. `ResizeWindow` moves a grow image around the screen, following the user's cursor movements, and handles all user interaction until the mouse button is released. Unlike the function `GrowWindow`, there is no need to follow this call with a call to `SizeWindow`. Once resizing is complete, `ResizeWindow` draws the window in its new size.

If you pass `NULL` in `ResizeWindow`'s `sizeConstraints` parameter, resizing will be constrained by the default minimum size (64 by 64 pixels) and the default maximum size (the bounding rectangle of the desktop region). However, the option is available to supply custom upper and lower re-sizing limits in the fields of a `Rect` variable passed in the `sizeConstraints` parameter. The limits represented by each field are as follows:

- `sizeConstraints.top` represents the minimum vertical measurement.
- `sizeConstraints.left` represents the minimum horizontal measurement.
- `sizeConstraints.bottom` represents the maximum vertical measurement.
- `sizeConstraints.right` represents the maximum horizontal measurement.

Note that the values assigned represent window dimensions, *not* screen coordinates.

Following a window resize, your application should adjust its scroll bars and window contents to accommodate the new size.

Carbon Note

In Carbon, `NULL` may be assigned to `ResizeWindow`'s `newContentRect` parameter if the new dimension of the window's content region is not required.

Closing a Window

The user closes a window by clicking in the close box/button or by choosing **Close** from the **File** menu.

When the user clicks in the close box/button, `TrackGoAway` should be called to track the mouse until the user releases the mouse button. If `TrackGoAway` returns `true`, meaning that the user did not release the mouse button outside the close box/button, your application should call its function for closing down the window.

The actions taken by your window closing function depend on the type of information the window contains and whether the contents need to be saved. The function should cater for different types of windows, that is, modeless dialogs (which may be merely hidden with `HideWindow` rather than closed completely) and standard document windows. In the latter case, the function should check whether any changes have been made to the document since it was opened and, if so, provide the user with an opportunity to save the document to a file before closing the window. (This whole process is explained in detail at Chapter 18.)

As for the window itself, `DisposeWindow` removes a window from the screen, removes it from the window list, and discards all of its data storage, including the window object.

Hiding and Showing a Window

Ordinarily, when the user elects to close a window, you dispose of the window. In some circumstances, however, it may be more appropriate to simply hide the window instead of removing its data structures. It is usually more appropriate, for example, to hide, rather than dispose of, modeless dialogs. That way, when the user next chooses the relevant menu command, the dialog is already available, and in the same location, as when it was last used.

`HideWindow` hides a window. `ShowWindow` will make the window visible and `SelectWindow` will make it the active window.

Providing Help Balloons (Mac OS 8/9)

Help Balloons —'hrct' and 'hwin' Resources

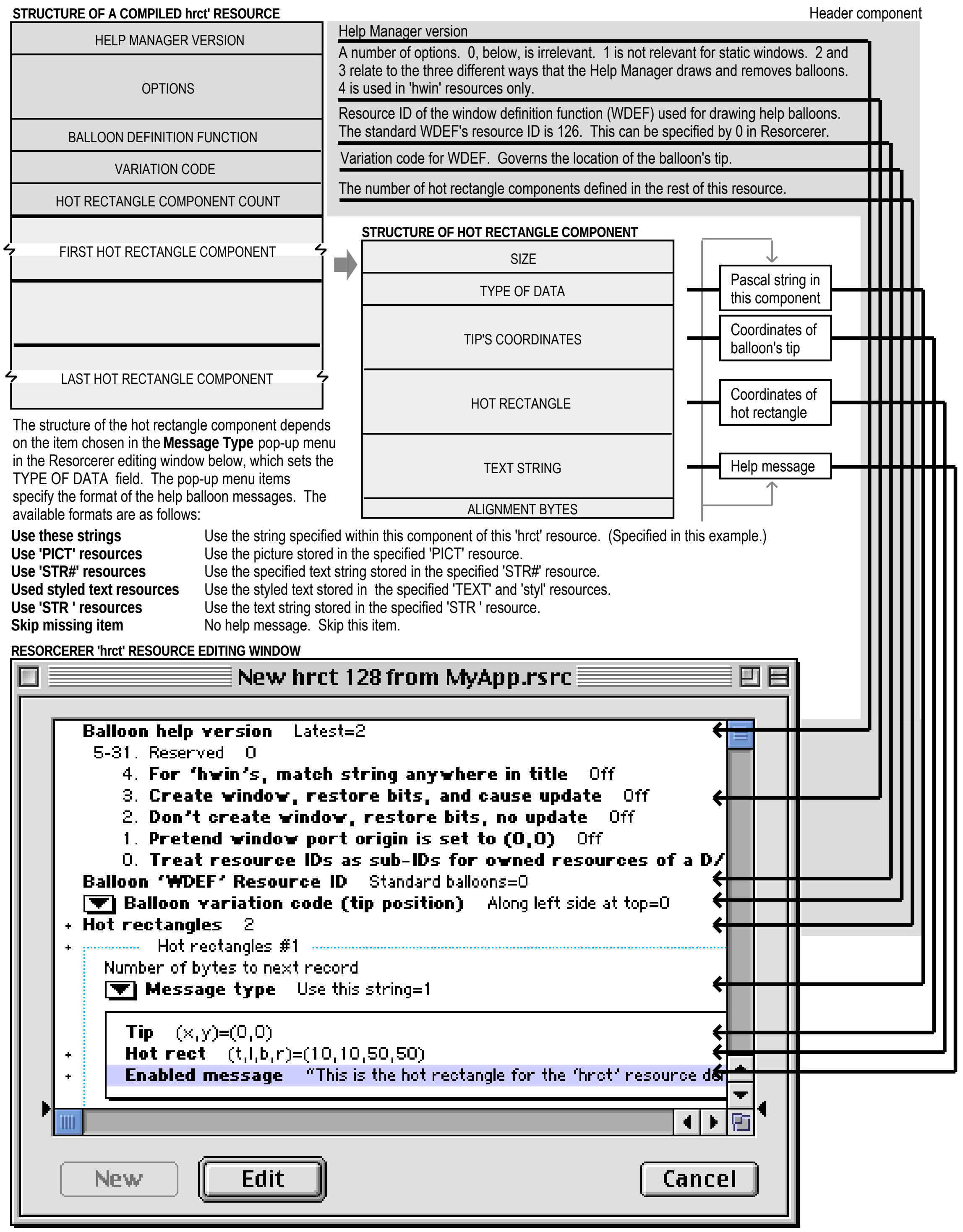
The Mac OS 8/9 system software provides help balloons for the title bar, draggable area, close box, zoom box, and collapse box for windows created with the standard WDEFs. Where applicable, you should provide help balloons for the content area of your windows.

How you choose to provide help balloons for the content area depends mainly on whether your windows are **static** or **dynamic**. A static window does not change its title or reposition any of the objects within its content area. A dynamic window can reposition any of it objects within its content area, or its title may change. For example, any window that scrolls past areas of interest to the user is a dynamic window because the object with associated help balloons can change location as the user scrolls. The following addresses the case of static windows only.

Help balloons for static document and floating windows are defined in `'hrct'` and `'hwin'` resources.

Creating 'hrct' and 'hwin' Resources Using Resorcerer

The `'hrct'` (rectangle help) resource is used to define **hot rectangles** for displaying help balloons in a static window and to specify the help messages for those balloons. All `'hrct'` resources must have resource IDs equal to or greater than 128. Fig 13 shows an `'hrct'` resource being created using Resorcerer.


STRUCTURE OF A COMPILED hrct' RESOURCE
Header component
HELP MANAGER VERSION
OPTIONS
BALLOON DEFINITION FUNCTION
VARIATION CODE
HOT RECTANGLE COMPONENT COUNT
FIRST HOT RECTANGLE COMPONENT
LAST HOT RECTANGLE COMPONENT
Help Manager version
A number of options. 0, below, is irrelevant. 1 is not relevant for static windows. 2 and 3 relate to the three different ways that the Help Manager draws and removes balloons. 4 is used in 'hwin' resources only.
Resource ID of the window definition function (WDEF) used for drawing help balloons. The standard WDEF's resource ID is 126. This can be specified by 0 in Resorcerer.
Variation code for WDEF. Governs the location of the balloon's tip.
The number of hot rectangle components defined in the rest of this resource.
STRUCTURE OF HOT RECTANGLE COMPONENT
SIZE
TYPE OF DATA
TIP'S COORDINATES
HOT RECTANGLE
TEXT STRING
ALIGNMENT BYTES
Pascal string in this component
Coordinates of balloon's tip
Coordinates of hot rectangle
Help message

The structure of the hot rectangle component depends on the item chosen in the **Message Type** pop-up menu in the Resorcerer editing window below, which sets the TYPE OF DATA field. The pop-up menu items specify the format of the help balloon messages. The available formats are as follows:

Use these strings	Use the string specified within this component of this 'hrct' resource. (Specified in this example.)
Use 'PICT' resources	Use the picture stored in the specified 'PICT' resource.
Use 'STR#' resources	Use the specified text string stored in the specified 'STR#' resource.
Used styled text resources	Use the styled text stored in the specified 'TEXT' and 'styl' resources.
Use 'STR ' resources	Use the text string stored in the specified 'STR ' resource.
Skip missing item	No help message. Skip this item.

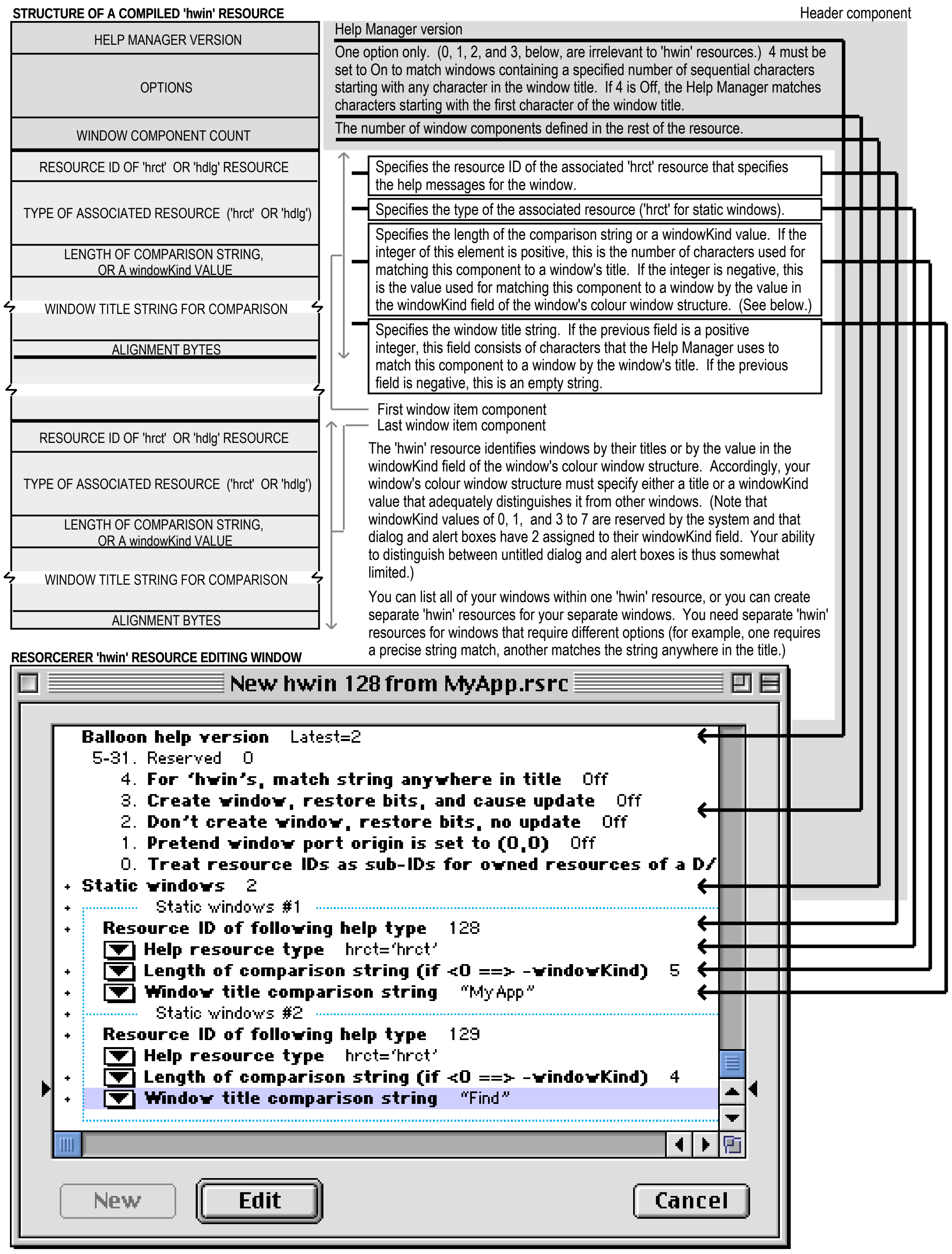
RESORCERER 'hrct' RESOURCE EDITING WINDOW
New hrct 128 from MyApp.rsrc
New
Edit
Cancel

```
Balloon help version   Latest=2
 5-31. Reserved   0
    4. For 'hwin's, match string anywhere in title   Off
    3. Create window, restore bits, and cause update   Off
    2. Don't create window, restore bits, no update   Off
    1. Pretend window port origin is set to (0,0)   Off
    0. Treat resource IDs as sub-IDs for owned resources of a D/
Balloon 'WDEF' Resource ID   Standard balloons=0
Balloon variation code (tip position)   Along left side at top=0
Hot rectangles   2
      Hot rectangles #1
  Number of bytes to next record
  Message type   Use this string=1

    Tip   (x,y)=(0,0)
    Hot rect   (t,l,b,r)=(10,10,50,50)
    Enabled message   "This is the hot rectangle for the 'hrct' resource de
```

FIG 13 - CREATING AN 'hrct' RESOURCE USING RESORCERER

The `'hwin'` (window help) resource is used to associate the help balloons defined in an `'hrct'` resource with a particular window. All `'hrct'` resources must have resource IDs equal to or greater than 128. Fig 14 shows an `'hwin'` resource being created using Resorcerer.


STRUCTURE OF A COMPILED 'hwin' RESOURCE
HELP MANAGER VERSION
OPTIONS
WINDOW COMPONENT COUNT
RESOURCE ID OF 'hrct' OR 'hdlg' RESOURCE
TYPE OF ASSOCIATED RESOURCE ('hrct' OR 'hdlg')
LENGTH OF COMPARISON STRING, OR A windowKind VALUE
WINDOW TITLE STRING FOR COMPARISON
ALIGNMENT BYTES
RESOURCE ID OF 'hrct' OR 'hdlg' RESOURCE
TYPE OF ASSOCIATED RESOURCE ('hrct' OR 'hdlg')
LENGTH OF COMPARISON STRING, OR A windowKind VALUE
WINDOW TITLE STRING FOR COMPARISON
ALIGNMENT BYTES
Header component
Help Manager version
First window item component
Last window item component
RESORCERER 'hwin' RESOURCE EDITING WINDOW
New hwin 128 from MyApp.rsrc
New
Edit
Cancel

One option only. (0, 1, 2, and 3, below, are irrelevant to 'hwin' resources.) 4 must be set to On to match windows containing a specified number of sequential characters starting with any character in the window title. If 4 is Off, the Help Manager matches characters starting with the first character of the window title.

The number of window components defined in the rest of the resource.

Specifies the resource ID of the associated 'hrct' resource that specifies the help messages for the window.

Specifies the type of the associated resource ('hrct' for static windows).

Specifies the length of the comparison string or a windowKind value. If the integer of this element is positive, this is the number of characters used for matching this component to a window's title. If the integer is negative, this is the value used for matching this component to a window by the value in the windowKind field of the window's colour window structure. (See below.)

Specifies the window title string. If the previous field is a positive integer, this field consists of characters that the Help Manager uses to match this component to a window by the window's title. If the previous field is negative, this is an empty string.

The 'hwin' resource identifies windows by their titles or by the value in the windowKind field of the window's colour window structure. Accordingly, your window's colour window structure must specify either a title or a windowKind value that adequately distinguishes it from other windows. (Note that windowKind values of 0, 1, and 3 to 7 are reserved by the system and that dialog and alert boxes have 2 assigned to their windowKind field. Your ability to distinguish between untitled dialog and alert boxes is thus somewhat limited.)

You can list all of your windows within one 'hwin' resource, or you can create separate 'hwin' resources for your separate windows. You need separate 'hwin' resources for windows that require different options (for example, one requires a precise string match, another matches the string anywhere in the title.)

```
Balloon help version   Latest=2
 5-31. Reserved   0
    4. For 'hwin's, match string anywhere in title   Off
    3. Create window, restore bits, and cause update   Off
    2. Don't create window, restore bits, no update   Off
    1. Pretend window port origin is set to (0,0)   Off
    0. Treat resource IDs as sub-IDs for owned resources of a D/
Static windows   2
      Static windows #1
  Resource ID of following help type   128
  Help resource type   hrct='hrct'
  Length of comparison string (if <0 ==> -windowKind)   5
  Window title comparison string   "MyApp"
      Static windows #2
  Resource ID of following help type   129
  Help resource type   hrct='hrct'
  Length of comparison string (if <0 ==> -windowKind)   4
  Window title comparison string   "Find"
```

FIG 14 - CREATING AN 'hwin' RESOURCE USING RESORCERER

Main Window Manager Constants, Data Types, and Functions

Constants

Window Types

```
kWindowDocumentProc               = 1024  Document windows
kWindowGrowDocumentProc           = 1025
kWindowVertZoomDocumentProc       = 1026
kWindowVertZoomGrowDocumentProc   = 1027
kWindowHorizZoomDocumentProc      = 1028
kWindowHorizZoomGrowDocumentProc  = 1029
kWindowFullZoomDocumentProc       = 1030
kWindowFullZoomGrowDocumentProc   = 1031
kWindowPlainDialogProc            = 1040  Dialogs and Alerts
kWindowShadowDialogProc           = 1041
kWindowModalDialogProc            = 1042
kWindowMovableModalDialogProc     = 1043
kWindowAlertProc                  = 1044
kWindowMovableAlertProc           = 1045
kWindowMovableModalGrowProc       = 1046
kWindowFloatProc                  = 1057  Floating windows
kWindowFloatGrowProc              = 1059
kWindowFloatVertZoomProc          = 1061
kWindowFloatVertZoomGrowProc      = 1063
kWindowFloatHorizZoomProc         = 1065
kWindowFloatHorizZoomGrowProc     = 1067
kWindowFloatFullZoomProc          = 1069
kWindowFloatFullZoomGrowProc      = 1071
kWindowFloatSideProc              = 1073
kWindowFloatSideGrowProc          = 1075
kWindowFloatSideVertZoomProc      = 1077
kWindowFloatSideVertZoomGrowProc  = 1079
kWindowFloatSideHorizZoomProc     = 1081
kWindowFloatSideHorizZoomGrowProc = 1083
kWindowFloatSideFullZoomProc      = 1085
kWindowFloatSideFullZoomGrowProc  = 1087
```

Window Kind

```
kDialogWindowKind      = 2
kApplicationWindowKind = 8
```

Window Part Codes Returned by FindWindow

```
inDesk                 = 0
inNoWindow             = 0
inMenuBar              = 1
inSysWindow            = 2
inContent              = 3
inDrag                 = 4
inGrow                 = 5
inGoAway               = 6
inZoomIn               = 7
inZoomOut              = 8
inCollapseBox          = 11
inProxyIcon            = 12
```

Regions Codes Passed to GetWindowRegion

```
kWindowTitleBarRgn     = 0
kWindowTitleTextRgn    = 1
kWindowCloseBoxRgn     = 2
kWindowZoomBoxRgn      = 3
kWindowDragRgn         = 5
kWindowGrowRgn         = 6
kWindowCollapseBoxRgn  = 7
kWindowStructureRgn    = 32
kWindowContentRgn      = 33
```

```
kWindowUpdateRgn      = 34
kWindowGlobalPortRgn  = 40
```

Options For ScrollWindowRect and ScrollWindowRegion

```
kScrollWindowNoOptions              = 0
kScrollWindowInvalidate             = (1L << 0)  Add exposed area to window’s update region.
KScrollWindowEraseToPortBackground  = (1L << 1)  Erase exposed area using background colour/
                                                 pattern of the window’s grafport.
```

Data Types

```
typedef struct OpaqueWindowPtr*  WindowPtr;
typedef WindowPtr  WindowRef;
typedef SInt16     WindowPartCode;
typedef UInt16     WindowRegionCode;
typedef UInt32     OptionBits;
typedef OptionBits ScrollWindowOptions;
```

Functions

Creating Windows

```
WindowRef  GetNewCWindow(short windowID,void *wStorage,WindowRef behind);
WindowRef  NewCWindow(void *wStorage,const Rect *boundsRect,ConstStr255Param title,Boolean
           visible,short procID,WindowRef behind,Boolean goAwayFlag,long refCon);
```

Naming Windows

```
void       GetWTitle(WindowRef theWindow,Str255 title);
void       SetWTitle(WindowRef theWindow,ConstStr255Param title);
OSStatus   CopyWindowTitleAsCFString(WindowRef inWindow,CFStringRef *outString);
OSStatus   SetWindowTitleWithCFString(WindowRef inWindow,CFStringRef inString);
```

Displaying Windows

```
Boolean    IsWindowVisible(WindowRef window);
void       ShowWindow(WindowRef theWindow);
void       HideWindow(WindowRef theWindow);
void       ShowHide(WindowRef theWindow,Boolean showFlag);
Boolean    IsWindowHilited(WindowRef window);
void       HiliteWindow(WindowRef theWindow,Boolean fHilite);
void       SelectWindow(WindowRef theWindow);
void       BringToFront(WindowRef theWindow);
void       SendBehind(WindowRef theWindow,WindowRef behindWindow);
void       DrawGrowIcon(WindowRef theWindow);
```

Moving Windows

```
void       MoveWindow(WindowRef theWindow,short hGlobal,short vGlobal,Boolean front);
void       DragWindow(WindowRef theWindow,Point startPt,const Rect *boundsRect);
```

Resizing Windows

```
long       GrowWindow(WindowRef theWindow,Point startPt,const Rect *bBox);
void       SizeWindow(WindowRef theWindow,short w,short h,Boolean fUpdate);
Boolean    ResizeWindow(WindowRef window,Point startPoint,const Rect *sizeConstraints,
           Rect *newContentRect);
```

Zooming Windows

```
Boolean    TrackBox(WindowRef theWindow,Point thePt,short partCode);
void       ZoomWindow(WindowRef theWindow,short partCode,Boolean front);
OSStatus   ZoomWindowIdeal(WindowRef window,SInt16 partCode,Point *ioIdealSize);
Boolean    IsWindowInStandardState(WindowRef window,Point *idealSize,Rect *idealStandardState);
OSStatus   SetWindowIdealUserState(WindowRef window,Rect *userState);
OSStatus   GetWindowIdealUserState(WindowRef window,Rect *userState);
```

Deallocating Windows

```
Boolean    TrackGoAway(WindowRef theWindow,Point thePt);
void       DisposeWindow(WindowRef theWindow);
```

Collapsing Windows

```
Boolean    IsWindowCollapsable(WindowRef inWindow);
Boolean    IsWindowCollapsed(WindowRef inWindow);
OSStatus   CollapseWindow(WindowRef inWindow,Boolean inCollapseIt);
OSStatus   CollapseAllWindows(Boolean inCollapseEm);
```

Window Kind

```
short      GetWindowKind(WindowRef window);
void       SetWindowKind(WindowRef window,short kind);
```

Window Regions

```
OSStatus   GetWindowRegion(WindowRef inWindow,WindowRegionCode inRegionCode,
           RgnHandle ioWinRgn)
```

Window User State and Standard State

```
Rect*      GetWindowStandardState(WindowRef window,Rect *rect);
Rect*      GetWindowUserState(WindowRef window,Rect *rect);
void       SetWindowStandardState(WindowRef window,const Rect *rect);
void       SetWindowUserState(WindowRef window,const Rect *rect);
Boolean    IsWindowInStandardState(WindowRef window,Point *idealSize,Rect idealStandardState);
```

Getting Available Window Positioning Bounds

```
OSStatus  GetAvailableWindowPositioningBounds(GDHandle inDevice,Rect *availableRect);
```

Maintaining the Update Region

```
void       BeginUpdate(WindowRef theWindow);
void       EndUpdate(WindowRef theWindow);
Boolean    IsWindowUpdatePending(WindowRef window);
OSStatus   InvalWindowRgn(WindowRef window,RgnHandle region);
OSStatus   InvalWindowRect(WindowRef window, const Rect *bounds);
OSStatus   ValidWindowRgn(WindowRef window,RgnHandle region);
OSStatus   ValidWindowRect(WindowRef window, const Rect *bounds);
```

Retrieving Mouse Information

```
short      FindWindow(Point thePoint,WindowRef *theWindow);
WindowRef  FrontWindow(void);
```

Window Reference Constant, Variant, and Picture

```
long       GetWRefCon(WindowRef theWindow);
void       SetWRefCon(WindowRef theWindow,long data);
short      GetWVariant(WindowRef theWindow);
void       SetWindowPic(WindowRef theWindow,PicHandle pic);
PicHandle  GetWindowPic(WindowRef theWindow);
```

Window List

```
WindowRef  GetWindowList(void);
WindowRef  GetNextWindow(WindowRef window);
```

Window's Graphics Port

```
CGrafPtr   GetWindowPort(WindowRef window);
void       SetPortWindowPort(WindowRef window);
WindowRef  GetWindowFromPort(CGrafPtr port);
Rect*      GetWindowPortBounds(WindowRef window,Rect *bounds);
```

Gray Region

```
RgnHandle  GetGrayRgn(void);
```

Scrolling Pixels in the Window Content Region

```
OSStatus   ScrollWindowRect(WindowRef   window,const Rect *scrollRect,SInt16 hPixels,
           SInt16 vPixels,ScrollWindowOptions options,RgnHandle outExposedRgn);
OSStatus   ScrollWindowRegion(WindowRef window,RgnHandle scrollRgn,
           SInt16 hPixels,vPixels,ScrollWindowOptions options,RgnHandle outExposedRgn);
```

Demonstration Program Windows1 Listing

```
// *******************************************************************************************
// Windows1.c                                                             CLASSIC EVENT MODEL
// *******************************************************************************************
//
// This program:
//
// •  Allows the user to open any number of kWindowFullZoomGrowDocumentProc windows, up to the
//    maximum specified by the constant assigned to the symbolic name kMaxWindows, using the
//    File menu Open Command or its keyboard equivalent.
//
// •  Allows the user to close opened windows using the close box/button, the File menu Close
//    command or the Close command's keyboard equivalent.
//
// •  Adds menu items representing each window to a Windows menu as each window is opened.
//    (A keyboard equivalent is included in each menu item for windows 1 to 9.)
//
// •  Deletes menu items from the Windows menu as each window is closed.
//
// •  Fills each window with a plain colour pattern as a means of proving, for demonstration
//    purposes, the window update process.
//
// •  Facilitates activation of a window by mouse selection.
//
// •  Facilitates activation of a window by Windows menu selection.
//
// •  Correctly performs all dragging, zooming and sizing operations.
//
// •  On Mac OS 8/9, demonstrates the provision of balloon help for static windows.
//
// The program utilises the following resources:
//
// •  A 'plst' resource.
//
// •  An 'MBAR' resource, and 'MENU' resources for Apple/Application, File, Edit and Windows
//    menus, (preload non-purgeable).
//
// •  A 'WIND' resource (purgeable) (initially not visible).
//
// •  A 'STR#' resource containing error strings and the window title (purgeable).
//
// •  An 'hrct' resource and an 'hwin' resource for balloon help (both purgeable).
//
// •  Ten 'ppat' (pixel pattern) resources (purgeable), which are used to draw a plain colour
//    pattern in the windows.
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, canBackground,
//    doesActivateOnFGSwitch, and isHighLevelEventAware flags set.
//
// *******************************************************************************************

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

#include <Carbon.h>

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

#define rMenubar          128
#define mAppleApplication 128
#define  iAbout           1
#define mFile             129
#define  iNew             1
#define  iClose           4
#define  iQuit            12
#define mWindows          131
#define rNewWindow        128
#define rStringList       128
```

```
#define  sUntitled        1
#define  eMaxWindows      2
#define  eFailWindow      4
#define  eFailMenus       5
#define  eFailMemory      6
#define rPixelPattern     128
#define kMaxWindows       10
#define kScrollBarWidth   15
#define MAX_UINT32        0xFFFFFFFF
#define MIN(a,b)          ((a) < (b) ? (a) : (b))
#define topLeft(r)        (((Point *) &(r))[0])

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

Boolean   gRunningOnX = false;
Boolean   gDone;
SInt32    gUntitledWindowNumber = 0;
SInt32    gCurrentNumberOfWindows = 0;
WindowRef gWindowRefArray[kMaxWindows + 2];

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

void    main                      (void);
void    doPreliminaries           (void);
OSErr   quitAppEventHandler       (AppleEvent *,AppleEvent *,SInt32);
void    eventLoop                 (void);
void    doEvents                  (EventRecord *);
void    doMouseDown               (EventRecord *);
void    doUpdate                  (EventRecord *);
void    doUpdateWindow            (EventRecord *);
void    doActivate                (EventRecord *);
void    doActivateWindow          (WindowRef,Boolean);
void    doOSEvent                 (EventRecord *);
void    doMenuChoice              (SInt32);
void    doFileMenu                (MenuItemIndex);
void    doWindowsMenu             (MenuItemIndex);
void    doNewWindow               (void);
void    doCloseWindow             (void);
void    doInvalidateScrollBarArea (WindowRef);
void    doConcatPStrings          (Str255,Str255);
void    doErrorAlert              (SInt16);
Boolean eventFilter               (DialogPtr,EventRecord *,SInt16 *);

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

void  main(void)
{
  MenuBarHandle menubarHdl;
  SInt32        response;
  MenuRef       menuRef;
  SInt16        a;

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

  doPreliminaries();

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

  menubarHdl = GetNewMBar(rMenubar);
  if(menubarHdl == NULL)
    doErrorAlert(eFailMenus);
  SetMenuBar(menubarHdl);
  DrawMenuBar();

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

    gRunningOnX = true;
  }

  // ....................................................................... initialize window reference array

  for(a=0;a<kMaxWindows+2;a++)
    gWindowRefArray[a] = NULL;

  // ........................................................................................ enter eventLoop

  eventLoop();
}

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

void  doPreliminaries(void)
{
  OSErr osError;

  MoreMasterPointers(224);
  InitCursor();
  FlushEvents(everyEvent,0);

  osError = AEInstallEventHandler(kCoreEventClass,kAEQuitApplication,
                            NewAEEventHandlerUPP((AEEventHandlerProcPtr) quitAppEventHandler),
                            0L,false);
  if(osError != noErr)
    ExitToShell();
}

// ***************************************************************************** doQuitAppEvent

OSErr  quitAppEventHandler(AppleEvent *appEvent,AppleEvent *reply,SInt32 handlerRefcon)
{
  OSErr    osError;
  DescType returnedType;
  Size     actualSize;

  osError = AEGetAttributePtr(appEvent,keyMissedKeywordAttr,typeWildCard,&returnedType,NULL,0,
                              &actualSize);

  if(osError == errAEDescNotFound)
  {
    gDone = true;
    osError = noErr;
  }
  else if(osError == noErr)
    osError = errAEParamMissed;

  return osError;
}

// ********************************************************************************** eventLoop

void  eventLoop(void)
{
  EventRecord eventStructure;

  gDone = false;

  while(!gDone)
  {
    if(WaitNextEvent(everyEvent,&eventStructure,MAX_UINT32,NULL))
      doEvents(&eventStructure);
  }
}

// ******************************************************************************* doEvents

void  doEvents(EventRecord *eventStrucPtr)
{
  switch(eventStrucPtr->what)
  {
    case kHighLevelEvent:
      AEProcessAppleEvent(eventStrucPtr);
      break;

    case mouseDown:
      doMouseDown(eventStrucPtr);
      break;

    case keyDown:
      if((eventStrucPtr->modifiers & cmdKey) != 0)
        doMenuChoice(MenuEvent(eventStrucPtr));
      break;

    case updateEvt:
      doUpdate(eventStrucPtr);
      break;

    case activateEvt:
      doActivate(eventStrucPtr);
      break;

    case osEvt:
      doOSEvent(eventStrucPtr);
      break;
  }
}

// ****************************************************************************** doMouseDown

void  doMouseDown(EventRecord *eventStrucPtr)
{
  WindowRef      windowRef;
  WindowPartCode partCode, zoomPart;
  BitMap         screenBits;
  Rect           constraintRect, mainScreenRect;
  Point          standardStateHeightAndWidth;

  partCode = FindWindow(eventStrucPtr->where,&windowRef);

  switch(partCode)
  {
    case inMenuBar:
      doMenuChoice(MenuSelect(eventStrucPtr->where));
      break;

    case inContent:
      if(windowRef != FrontWindow())
        SelectWindow(windowRef);
      break;

    case inDrag:
      DragWindow(windowRef,eventStrucPtr->where,NULL);
      break;

    case inGoAway:
      if(TrackGoAway(windowRef,eventStrucPtr->where) == true)
        doCloseWindow();
      break;

    case inGrow:
      constraintRect.top   = 75;
      constraintRect.left = 205;
      constraintRect.bottom = constraintRect.right = 32767;
      doInvalidateScrollBarArea(windowRef);
      ResizeWindow(windowRef,eventStrucPtr->where,&constraintRect,NULL);
      doInvalidateScrollBarArea(windowRef);
      break;

    case inZoomIn:
    case inZoomOut:
      mainScreenRect = GetQDGlobalsScreenBits(&screenBits)->bounds;
      standardStateHeightAndWidth.v = mainScreenRect.bottom;
      standardStateHeightAndWidth.h = 460;

      if(IsWindowInStandardState(windowRef,&standardStateHeightAndWidth,NULL))
        zoomPart = inZoomIn;
      else
        zoomPart = inZoomOut;

      if(TrackBox(windowRef,eventStrucPtr->where,partCode))
        ZoomWindowIdeal(windowRef,zoomPart,&standardStateHeightAndWidth);
      break;
  }
}

// ********************************************************************************* doUpdate

void  doUpdate(EventRecord *eventStrucPtr)
{
  WindowRef windowRef;

  windowRef = (WindowRef) eventStrucPtr->message;

  BeginUpdate(windowRef);

  SetPortWindowPort(windowRef);
  doUpdateWindow(eventStrucPtr);

  EndUpdate(windowRef);
}

// *************************************************************************** doUpdateWindow

void  doUpdateWindow(EventRecord *eventStrucPtr)
{
  WindowRef    windowRef;
  RgnHandle    visibleRegionHdl;
  Rect         theRect;
  SInt32       windowRefCon;
  PixPatHandle pixpatHdl;
  RGBColor     whiteColour = { 0xFFFF,0xFFFF,0xFFFF };
  SInt16       a;

  windowRef = (WindowRef) eventStrucPtr->message;

  visibleRegionHdl = NewRgn();
  GetPortVisibleRegion(GetWindowPort(windowRef),visibleRegionHdl);
  EraseRgn(visibleRegionHdl);
  DisposeRgn(visibleRegionHdl);

  GetWindowPortBounds(windowRef,&theRect);
  theRect.right  -= kScrollBarWidth;
  theRect.bottom -= kScrollBarWidth;

  windowRefCon = GetWRefCon(windowRef);
  pixpatHdl = GetPixPat(rPixelPattern + windowRefCon);
  FillCRect(&theRect,pixpatHdl);
  DisposePixPat(pixpatHdl);
  DrawGrowIcon(windowRef);

  RGBForeColor(&whiteColour);
  TextSize(10);

  if(!gRunningOnX)
  {
    for(a=0;a<2;a++)
    {
      SetRect(&theRect,a*90+10,10,a*90+90,33);
      FrameRect(&theRect);
      MoveTo(a*90+18,25);

      DrawString("\pHot Rectangle");
    }
  }
}

// ******************************************************************************** doActivate

void  doActivate(EventRecord *eventStrucPtr)
{
  WindowRef windowRef;
  Boolean   becomingActive;

  windowRef = (WindowRef) eventStrucPtr->message;

  becomingActive = ((eventStrucPtr->modifiers & activeFlag) == activeFlag);

  doActivateWindow(windowRef,becomingActive);
}

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

void  doActivateWindow(WindowRef windowRef,Boolean becomingActive)
{
  MenuRef windowsMenu;
  SInt16  menuItem, a = 1;

  windowsMenu = GetMenuRef(mWindows);

  while(gWindowRefArray[a] != windowRef)
    a++;
  menuItem = a;

  if(becomingActive)
    CheckMenuItem(windowsMenu,menuItem,true);
  else
    CheckMenuItem(windowsMenu,menuItem,false);

  DrawGrowIcon(windowRef);
}

// ********************************************************************************* doOSEvent

void  doOSEvent(EventRecord *eventStrucPtr)
{
  switch((eventStrucPtr->message >> 24) & 0x000000FF)
  {
    case suspendResumeMessage:
      if((eventStrucPtr->message & resumeFlag) == 1)
        SetThemeCursor(kThemeArrowCursor);
      break;
  }
}

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

void  doMenuChoice(SInt32 menuChoice)
```

```
{
  MenuID        menuID;
  MenuItemIndex menuItem;

  menuID = HiWord(menuChoice);
  menuItem = LoWord(menuChoice);

  if(menuID == 0)
    return;

  switch(menuID)
  {
    case mAppleApplication:
      if(menuItem == iAbout)
        SysBeep(10);
      break;

    case mFile:
      doFileMenu(menuItem);
      break;

    case mWindows:
      doWindowsMenu(menuItem);
      break;
  }

  HiliteMenu(0);
}

// ********************************************************************************* doFileMenu

void  doFileMenu(MenuItemIndex menuItem)
{
  switch(menuItem)
  {
    case iNew:
      doNewWindow();
      break;

    case iClose:
      doCloseWindow();
      break;

    case iQuit:
      gDone = true;
      break;
  }
}

// ****************************************************************************** doWindowsMenu

void  doWindowsMenu(MenuItemIndex menuItem)
{
  WindowRef windowRef;

  windowRef = gWindowRefArray[menuItem];
  SelectWindow(windowRef);
}

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

void  doNewWindow(void)
{
  WindowRef windowRef;
  Str255    untitledString;
  Str255    numberAsString = "\p1";
  Rect      availableBoundsRect, portRect;
  SInt16    windowHeight;
  MenuRef   windowsMenu;
  if(gCurrentNumberOfWindows == kMaxWindows)
  {
    doErrorAlert(eMaxWindows);
    return;
  }

  if(!(windowRef = GetNewCWindow(rNewWindow,NULL,(WindowRef) -1)))
    doErrorAlert(eFailWindow);

  GetIndString(untitledString,rStringList,sUntitled);
  gUntitledWindowNumber += 1;
  if(gUntitledWindowNumber > 1)
  {
    NumToString(gUntitledWindowNumber,numberAsString);
    doConcatPStrings(untitledString,numberAsString);
  }

  SetWTitle(windowRef,untitledString);

  GetAvailableWindowPositioningBounds(GetMainDevice(),&availableBoundsRect);
  GetWindowPortBounds(windowRef,&portRect);
  SetPortWindowPort(windowRef);
  LocalToGlobal(&topLeft(portRect));
  windowHeight = (availableBoundsRect.bottom - portRect.top) - 3;
  if(!gRunningOnX)
    windowHeight -= 27;
  SizeWindow(windowRef,460,windowHeight,false);

  ShowWindow(windowRef);

  if(gUntitledWindowNumber < 10)
  {
    doConcatPStrings(untitledString,"\p/");
    doConcatPStrings(untitledString,numberAsString);
  }
  windowsMenu = GetMenuRef(mWindows);
  InsertMenuItem(windowsMenu,untitledString,CountMenuItems(windowsMenu));

  SetWRefCon(windowRef,gCurrentNumberOfWindows);

  gCurrentNumberOfWindows ++;
  gWindowRefArray[gCurrentNumberOfWindows] = windowRef;

  if(gCurrentNumberOfWindows == 1)
  {
    EnableMenuItem(GetMenuRef(mFile),iClose);
    EnableMenuItem(GetMenuRef(mWindows),0);
    DrawMenuBar();
  }
}

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

void  doCloseWindow(void)
{
  WindowRef windowRef;
  MenuRef   windowsMenu;
  SInt16    a = 1;

  windowRef = FrontWindow();
  DisposeWindow(windowRef);
  gCurrentNumberOfWindows --;

  windowsMenu = GetMenuRef(mWindows);
  while(gWindowRefArray[a] != windowRef)
    a++;
  gWindowRefArray[a] = NULL;
  DeleteMenuItem(windowsMenu,a);
  for(a=1;a<kMaxWindows+1;a++)
  {
    if(gWindowRefArray[a] == NULL)
    {
      gWindowRefArray[a] = gWindowRefArray[a+1];
      gWindowRefArray[a+1] = NULL;
    }
  }

  if(gCurrentNumberOfWindows == 0)
  {
    DisableMenuItem(GetMenuRef(mFile),iClose);
    DisableMenuItem(GetMenuRef(mWindows),0);
    DrawMenuBar();
  }
}

// ******************************************************************* doInvalidateScrollBarArea

void  doInvalidateScrollBarArea(WindowRef windowRef)
{
  Rect tempRect;

  SetPortWindowPort(windowRef);

  GetWindowPortBounds(windowRef,&tempRect);
  tempRect.left = tempRect.right - kScrollBarWidth;
  InvalWindowRect(windowRef,&tempRect);

  GetWindowPortBounds(windowRef,&tempRect);
  tempRect.top = tempRect.bottom - kScrollBarWidth;
  InvalWindowRect(windowRef,&tempRect);
}

// ************************************************************************** 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;
  }
}

// ****************************************************************************** doErrorAlert

void  doErrorAlert(SInt16 errorType)
{
  AlertStdAlertParamRec paramRec;
  ModalFilterUPP        eventFilterUPP;
  Str255                labelText;
  Str255                narrativeText;
  SInt16                itemHit;

  eventFilterUPP = NewModalFilterUPP((ModalFilterProcPtr) eventFilter);

  paramRec.movable       = true;
  paramRec.helpButton    = false;
  paramRec.filterProc    = eventFilterUPP;
  paramRec.defaultText   = (StringPtr) kAlertDefaultOKText;
  paramRec.cancelText    = NULL;
  paramRec.otherText     = NULL;
  paramRec.defaultButton = kAlertStdAlertOKButton;
  paramRec.cancelButton  = 0;
  paramRec.position      = kWindowAlertPositionMainScreen;

  GetIndString(labelText,rStringList,errorType);

  if(errorType == eMaxWindows)
  {
    GetIndString(narrativeText,rStringList,errorType + 1);
    StandardAlert(kAlertCautionAlert,labelText,narrativeText,&paramRec,&itemHit);
    DisposeModalFilterUPP(eventFilterUPP);
  }
  else
  {
    StandardAlert(kAlertStopAlert,labelText,0,&paramRec,&itemHit);
    ExitToShell();
  }
}

// ***************************************************************************** eventFilter

Boolean  eventFilter(DialogPtr dialogPtr,EventRecord *eventStrucPtr,SInt16 *itemHit)
{
  Boolean handledEvent = false;

  if((eventStrucPtr->what == updateEvt) &&
     ((WindowRef) eventStrucPtr->message != GetDialogWindow(dialogPtr)))
  {
    doUpdate(eventStrucPtr);
  }

  return handledEvent;
}

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

Demonstration Program Windows1 Comments

When this program is run, the user should:

- Open and close windows using both the Open and Close commands from the File menu and their keyboard equivalents, noting that, whenever a window is opened or closed, a menu item representing that window is added to, or deleted from, the Windows menu.

- Note that keyboard equivalents are added to the menu items in the Windows menu for the first nine windows opened.

- Activate individual windows by both clicking the content region and pressing the keyboard equivalent for the window.

- Send the application to the background and bring it to the foreground, noting window activation/deactivation.

- Zoom, close, and resize windows using the zoom box/button, close box/button and size box/resize control, noting window updating and activation.

- On Mac OS X, note that, when a window is zoomed to the standard state, the zoom is constrained by the current height of the Dock.

- On Mac OS 8/9, choose Show Balloons from the Help menu and move the cursor over the hot rectangles in the frontmost window.

If an attempt is made to open more than 10 windows, a movable modal alert appears.

defines

The first nine #defines establish constants representing menu IDs and resources, and window and menu bar resources. The next six establish constants representing the resource ID of a 'STR#' resource and the various strings in that resource. rPixelPattern represents the resource ID of the first of ten 'ppat' (pixel pattern) resources. kMaxWindows controls the maximum number of windows allowed to be open at one time.

MAX_UINT32 is defined as the maximum possible UInt32 value. (This will be assigned to WaitNextEvent's sleep parameter.) The two fairly common macros which follow are required by, respectively, the string concatenation function doConcatPStrings and the window creation function doNewWindow.

Global Variables

The global variable gRunningOnX will be set to true if the program is running on Mac OS X. gDone, when set to true, causes the main event loop to be exited and the program to terminate.

gUntitledWindowNumber keeps track of the window numbers to be inserted into the window's title bar. This number is incremented each time a new window is opened. gCurrentNumberOfWindows keeps track of how many windows are open at any one time.

In this program, CreateStandardWindowMenu is not used to create the Window menu. The Window menu is created in the same way as the other menus, and managed ny the program. gWindowRefArray[] is central to the matter of maintaining an association between item numbers in the Window menu and the windows to which they refer, regardless of how many windows are opened and closed, and in what sequence. When, for example, a Window menu item is chosen, the program must be able to locate the window object for the window represented by that menu item number so as to activate the correct window.

The strategy adopted by this program is to assign the references for each opened window to the elements of gWindowRefArray[], starting with gWindowRefArray[1] and leaving gWindowRefArray[0] unused. If, for example, six windows are opened in sequence, gWindowRefArray[1] to gWindowRefArray[6] are assigned the references to the window objects for each of those six windows. (At the same time, menu items representing each of those windows are progressively added to the Window menu.)

If, say, the third window opened is then closed, gWindowRefArray[3] is set to NULL and the window object references in gWindowRefArray[4] to gWindowRefArray[6] are moved down in the array to occupy gWindowRefArray[3] to gWindowRefArray[5]. Since the Window menu item for the third window is deleted from the menu when the window is closed, there remains five windows and their associated menu items, the "compaction" of the array having maintained a direct relationship between the number of the array element to which each window reference is assigned and the number of the menu item for that window.

main

The first action is to call doPreliminaries, which performs certain preliminary actions common to most applications.

The next block sets up the menus. Note that error handling involving the invocation of alerts is introduced in this program. If an error occurs, the function doErrorAlert is called to display either a stop or caution movable modal alert advising of the nature of the error.

In the final three lines, gWindowRefArray[] is initialised and the main event loop is entered.

doPreliminaries

Note that MoreMasterPointers is called with 224 passed in the inCount parameter to provide sufficient master pointers for this program.

eventLoop

eventLoop will exit when gDone is set to true, which occurs when the user selects Quit from the File menu. (As an aside, note that the sleep parameter in the WaitNextEvent call is set to MAX_UINT32, which is defined as the maximum possible UInt32 value.)

doEvents

doEvents switches according to the event type received.

mouseDown, updateEvt, activateEvt and osEvt events are of significance to the windows aspects of this demonstration. keyDown events are significant only with regard to File and Window menu keyboard equivalents.

doMouseDown

doMouseDown continues the processing of mouseDown events, switching according to the part code.

The inContent case results in a call to SelectWindow if the window in which the mouse-down occurred is not the front window. SelectWindow:

- Unhighlights the currently active window, brings the specified window to the front and highlights it.

- Generates activate events for the two windows.

- Moves the previously active window to a position immediately behind the specified window.

The inDrag case results in a call to DragWindow, which retains control until the user releases the mouse button. The third parameter in the DragWindow call establishes the limits, in global screen coordinates, within which the user is allowed to drag the window. In Carbon, NULL may be passed in this parameter. This has the effect of setting the third parameter to the bounding rectangle of the desktop region (also known as the "gray region").

The inGoAway case results in a call to TrackGoAway, which retains control until the user releases the mouse button. If the pointer is still within the close box/button when the mouse button is released, the function doCloseWindow is called.

At the inGrow case, the first three lines establish the resizing constraints. The top and left fields of the Rect variable constraintRect are assigned values representing the minimum height and width to which the window may be resized. The bottom and right fields, which establish the maximum height and width, are assigned the maximum possible SInt16 value. (Since the mouse cursor cannot be moved beyond the edges of the screen (or screens in a multi-monitor system), these values mean that the window can be resized larger to the limits of mouse cursor movement.)

ResizeWindow retains control until the user releases the mouse button. When the mouse button is released, ResizeWindow redraws the window frame (that is, all but the content region) in the new size and, where window height and/or width has been increased, adds the newly-exposed areas of the content region to update region (on Mac OS 8/9). (Note that, in Carbon, the fourth (newContentRect) parameter may be set to NULL if the new dimension of the window's content region is not required.)

The call to ResizeWindow is bracketed by two calls to the function doInvalidateScrollBarArea. In this program, scroll bars are not used but it has been decided to, firstly, limit update drawing to the window's content region less the areas normally occupied by scroll bars and, secondly, to use DrawGrowIcon to draw the draw scroll bar delimiting lines. (For Mac OS 8/9, this is the usual practice for windows with a size box but no scroll bars. The DrawGrowIcon call is ignored on Mac OS X.)

The first call to doInvalidateScrollBarArea is necessary to cater for the case where the window is resized larger. If this call is not made, the scroll bar areas prior to the resize will not be redrawn by the window updating function unless these areas are programmatically added to the new update region created by the Window Manager as a result of the resizing action.

The second call to doInvalidateScrollBarArea is necessary to cater for the case where the window is resized smaller. This call works in conjunction with the EraseRgn call in the function doUpdateWindow. The call to doInvalidateScrollBarArea results in an update event being generated, and the call to EraseRgn in the doUpdateWindow function causes the update region (that is, in this case, the scroll bar areas) to be erased. (Remember that, on Mac OS 8/9, between the calls to BeginUpdate and EndUpdate, the visible region equates to the update region and that QuickDraw limits its drawing to the update region.)

At the inZoomIn/inZoomOut case, the first action is to assign the desired height and width of the windows's standard state content region to the fields of a Point variable. This variable is then passed in the second parameter of a call to IsWindowInStandardState, which sets the variable zoomPart to either true or false depending on whether the window is currently in the standard state or the user state. TrackBox is then called, taking control until the user releases the mouse button. If the mouse button is released while the pointer is still within the zoom box, ZoomWindowIdeal is called to zoom the window in accordance with human interface guidelines. The second parameter tells ZoomWindow whether to zoom out to the standard state or in to the user state.

doUpdate

On Mac OS 8/9 and Mac OS X, an update event will be received:

- When the window is created.
- When the window is resized larger.
- When the window is resized smaller (because of calls to InvalWindowRect in the function doInvalidateScrollBarArea).
- When the window is zoomed.

On Mac OS 8/9, update events will also be received when a window has a non-empty update region.

doUpdate attends to basic window updating. On Mac OS 8/9, the call to BeginUpdate clips the visible region to the intersection of the visible region and the update region. The visible region is now a sort of proxy for the update region. The graphics port is then set before the function doUpdateWindow is called to redraw the content region. On Mac OS 8/9, the EndUpdate call restores the window's true visible region. (The calls to BeginUpdate and EndUpdate are ignored on Mac OS X.)

doUpdateWindow

doUpdateWindow is concerned with redrawing the window's contents less the scroll bar areas.

The first action is to retrieve the window object reference from the message field of the event structure.

The next block retrieves the handle to the window's visible region, following which EraseRgn is called for reasons explained at doMouseDown, above.

The window's graphics port's bounding rectangle is then retrieved, following which the right and bottom fields are adjusted to exclude the scroll bar areas. The next four lines fill this rectangle with a plain colour pattern provided by a 'ppat' resource, simply as a means of proving the correctness of the window updating process.

Note the call to GetWRefCon, which retrieves the window's reference constant stored in the window object. As will be seen, whenever a new window is opened, a value between 1 and kMaxWindows is stored as a reference constant in the window object. In this function, this is just a convenient number to be added to the base resource ID (128) in the single parameter of the GetPixPat call, ensuring that FillCRect has a different pixel pattern to draw in each window.

The call to DrawGrowIcon draws the scroll bar delimiting lines (on Mac OS 8/9). Note that this call, the preceding EraseRgn call, and the calls to doInvalidateScrollbarArea are made for "cosmetic" purposes only and would not be required if the window contained scroll bars.

If the program is running on Mac OS 8/9, the remaining lines draw two rectangles and some text in the windows to visually represent to the user the otherwise invisible "hot rectangles" defined in the 'hrct' resource and associated with the window by the 'hwin' resource. When Show Balloons is chosen from the Help menu, the help balloons will be invoked when the cursor moves over these rectangles.

doActivate

doActivate attends to those aspects of window activation not handled by the Window Manager.

The modifiers field of the event structure is tested to determine whether the window in question is being activated or deactivated. The result of this test is passed as a parameter in the call to the function doActivateWindow.

doActivateWindow

In this demonstration, the remaining actions carried out in response to an activateEvt are limited to placing/removing checkmarks in/from items in the Window menu.

The first step is to associate the received WindowRef with its item number in the Window menu. At the while loop, the array maintained for that purpose is searched until a match is found. The array element number at which the match is found correlates directly with the menu item number; accordingly, this is assigned to the variable menuItem, which is used in the following CheckMenuItem calls. Whether the checkmark is added or removed depends on whether the window in question is being activated or deactivated, a condition passed to the call to doActivateWindow as its second parameter.

The call to DrawGrowIcon ensures that the scroll bar area delimiting lines will be drawn in gray when the window is deactivated (on Mac OS 8/9).

doOSEvent

doOSEvent handles operating system events. In this demonstration, action is taken only in the case of resume events. If the event is a resume event, the cursor is set to the arrow shape.

doMenuChoice and doFileMenu

doMenuChoice switches according to the menu choices of the user. doFileMenu switches according to the File menu item choice of the user.

doWindowsMenu

doWindowsMenu takes the item number of the selected Window menu item and, since this equates to the number of the array element in which the associated window object reference is stored, extracts the window object reference associated with the menu item. This is used in the call to SelectWindow, which generates the activateEvts required to activate and deactivate the appropriate windows.

doNewWindow

doNewWindow opens a new window and attends to associated tasks.

In the first block, if the current number of open windows equals the maximum allowable specified by kMaxWindows, a caution movable modal alert is called up via the function doErrorAlert (with the string represented by eMaxWindows displayed) and an immediate return is executed when the user clicks the alert's OK button.

At the next block, the new window is created by the call to GetNewCWindow. The third parameter specifies that the window is to be opened in front of all other windows. If the call is not successful for any reason, a stop movable modal alert is called up via the function doErrorAlert (with the string represented by eFailWindow displayed) and the program terminates when the user clicks the alert's OK button.

The next seven lines create the string which will be used to set the window's title. The code reflects the fact that Aqua Human Interface Guidelines require that a number only be appended to "untitled" for the second and later windows. Accordingly, concatenating a number to the string "untitled" is not effected for the first window created.

GetIndString retrieves the string "untitled " from the specified 'STR#' resource and the global variable which keeps track of the numbers for the title bar is incremented. If this is not the first window to be created, NumToString converts that number to a Pascal string and this string is concatenated to the "untitled " string. The SetWTitle call then sets the window's title.

The next block sets adjusts the size of the window before it is displayed. The width is set to 460 pixels and the height is adjusted according to the available screen real estate.

The call to GetAvailableWindowPositioningBounds returns, in global coordinates, the available real estate on the main screen (device). This excludes the menu bar and, on Mac OS X, the Dock. The call to SetPortWindowPort sets the window's graphics port as the current port, a necessary precursor to the call to LocalToGlobal, which converts the top-left (local) coordinates of the port rectangle to global coordinates. The height of the rectangle returned by GetAvailableWindowPositioningBounds is then reduced by the distance of the top of the port rectangle from the top of the screen, and then further reduced by three. On Mac OS X, this will cause the bottom of the window to be just above the top of the Dock. If the program is running on Mac OS 8/9, the height is reduced by a further 27 pixels to accommodate the height of the control strip. The call to SizeWindow sets the window's size. (The window's location is determined by the positioning specification in the window's 'WIND' resource.)

The ShowWindow call makes the window visible.

The next block adds the metacharacter \ and the window number to the string used to set the window title (thus setting up the Command key equivalent) before InsertMenuItem is called to create a new menu item to the Window menu. Note that the Command-key equivalent is only added for the first nine windows opened.)

The SetWRefCon call stores the value represented by gCurrentNumberOfWindows in the window object as the window's reference constant. As previously stated, in this demonstration this is used to select a pixel pattern to draw in each window's content region.

At the next two lines, the variable which keeps track of the current number of opened windows is incremented and the appropriate element of the window reference array is assigned the reference to newly opened window's window object.

The last block enables the Window menu and the Close item in the File menu when the first window is opened.

doCloseWindow

The function doCloseWindow closes an open window and attends to associated tasks.

At the first two lines, a reference to the frontmost window's window object is retrieved and passed in the call to DisposeWindow. DisposeWindow removes the window from the screen, removes it from the window list, and discards all its data storage. With the window closed, the global variable that keeps track of the number of windows currently open is decremented.

The next block deletes the associated menu item from the Window menu. At the first four lines, the array element in which the window object reference in question is located is searched out, the element number (which correlates directly with the menu item number) is noted and the element is set to NULL. The call to DeleteMenuItem then deletes the menu item.

The for loop "compacts" the array, that is, it moves the contents of all elements above the NULLed element down by one, maintaining the correlation with the Windows menu.

The last block disables the Windows menu and the Close item in the File menu if no windows remain open as a result of this closure.

doInvalidateScrollBarArea

doInvalidateScrollBarArea invalidates that part of the window's content area which would ordinarily be occupied by scroll bars. The function simply retrieves the coordinates of the content area into a local Rect and reduces this Rect to the relevant scroll bar area before invalidating that area, that is, adding it to the window's update region.

doConcatPStrings

The function doConcatPStrings concatenates two Pascal strings.

doErrorAlert and eventFilter

doErrorAlert displays either a caution alert or a stop alert with a specified string (two strings in the case of the eMaxWindows error) extracted from the 'STR#' resource identified by rStringList. eventFilter supports doErrorAlert.

The creation of alerts using the StandardAlert function, and event filter functions, are addressed at Chapter 8.